



MCUXpresso SDK Documentation

Release 25.12.00



NXP
Dec 18, 2025



Table of contents

1	FRDM-KE02Z40M	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	58
1.3.1	Getting Started with MCUXpresso SDK Repository	58
1.4	Release Notes	65
1.4.1	MCUXpresso SDK Release Notes	65
1.5	ChangeLog	67
1.5.1	MCUXpresso SDK Changelog	67
1.6	Driver API Reference Manual	93
1.7	Middleware Documentation	93
1.7.1	FreeMASTER	93
2	MKE02Z4	95
2.1	ACMP: Analog Comparator Driver	95
2.2	ADC: 12-bit Analog to Digital Converter Driver	98
2.3	Clock Driver	104
2.4	CRC: Cyclic Redundancy Check Driver	115
2.5	FGPIO Driver	118
2.6	FTMRx Flash Driver	120
2.7	FTM: FlexTimer Driver	138
2.8	GPIO: General-Purpose Input/Output Driver	160
2.9	GPIO Driver	161
2.10	I2C: Inter-Integrated Circuit Driver	163
2.11	I2C Driver	163
2.12	Irq	177
2.13	IRQ: external interrupt (IRQ) module	180
2.14	KBI: Keyboard interrupt Driver	180
2.15	Common Driver	181
2.16	MCM: Miscellaneous Control Module	194
2.17	PIT: Periodic Interrupt Timer	199
2.18	PORT Driver	203
2.19	RTC: Real Time Clock	209
2.20	SPI: Serial Peripheral Interface Driver	214
2.21	SPI Driver	214
2.22	TPM: Timer PWM Module	227
2.23	UART: Universal Asynchronous Receiver/Transmitter Driver	238
2.24	UART Driver	238
2.25	WDOG8: 8-bit Watchdog Timer	253
3	Middleware	257
3.1	Motor Control	257
3.1.1	FreeMASTER	257
4	RTOS	295
4.1	FreeRTOS	295

4.1.1	FreeRTOS kernel	295
4.1.2	FreeRTOS drivers	295
4.1.3	backoffalgorithm	295
4.1.4	corehttp	295
4.1.5	corejson	295
4.1.6	coremqtt	296
4.1.7	corepkcs11	296
4.1.8	freertos-plus-tcp	296

This documentation contains information specific to the frdmke02z40m board.

Chapter 1

FRDM-KE02Z40M

1.1 Overview

The Freedom-KE02Z40M is an ultra-low-cost development platform for Kinetis KE02 MCUs



MCU device and part on board is shown below:

- Device: MKE02Z4
- PartNumber: MKE02Z64VQH4

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.

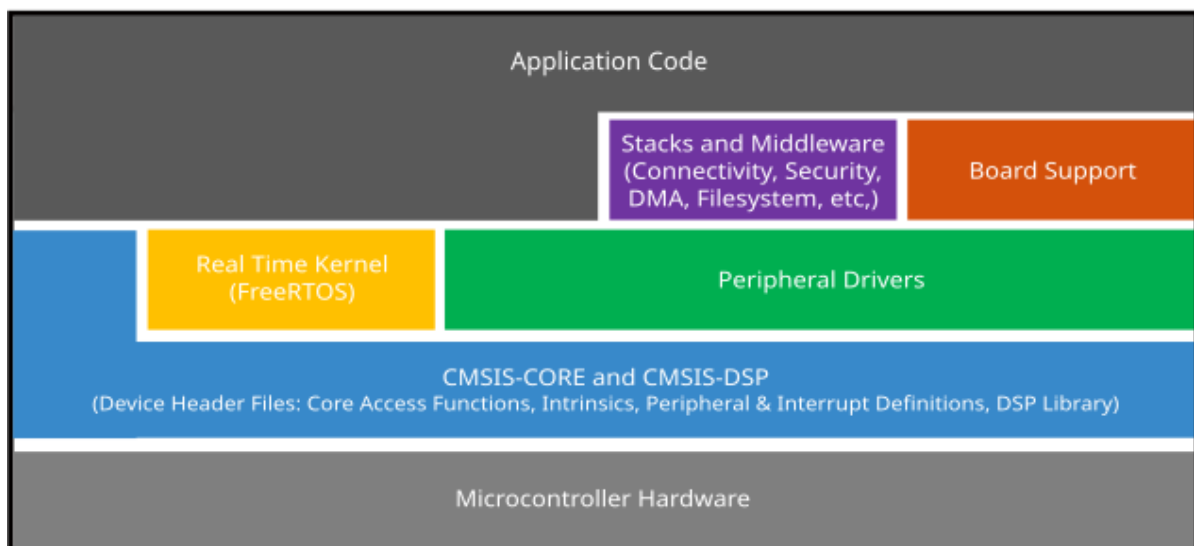
Note: The Repository-Layout SDK package was first introduced in version 25.09.00, but initially only for MCXW23x platforms.

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):

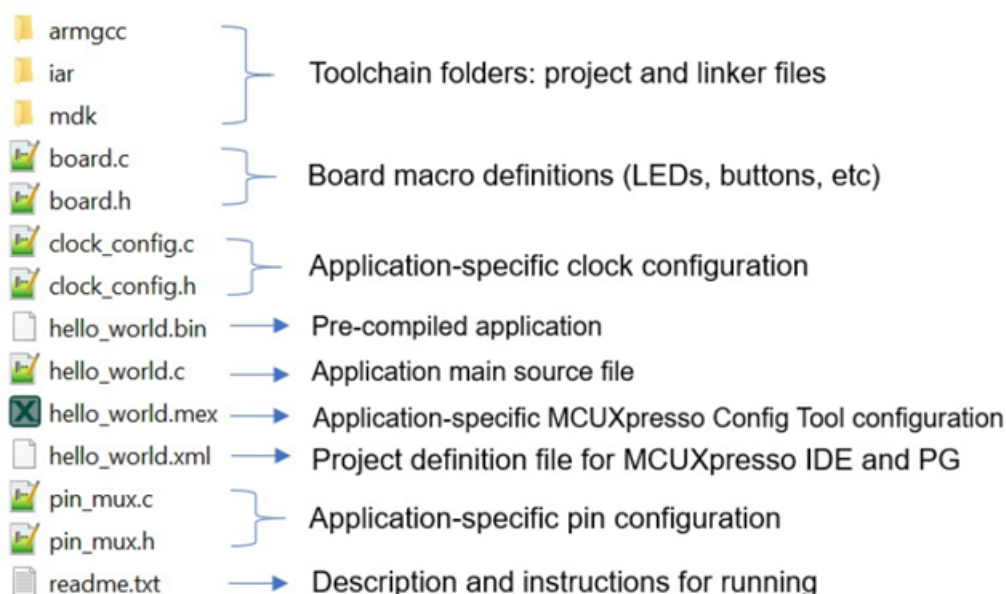
- `cmsis_driver_examples`: Simple applications intended to show how to use CMSIS drivers.
- `demo_apps`: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may leverage stacks and middleware.
- `driver_examples`: 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/<devices_name>/project: Project template used in CMSIS PACK new project creation

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

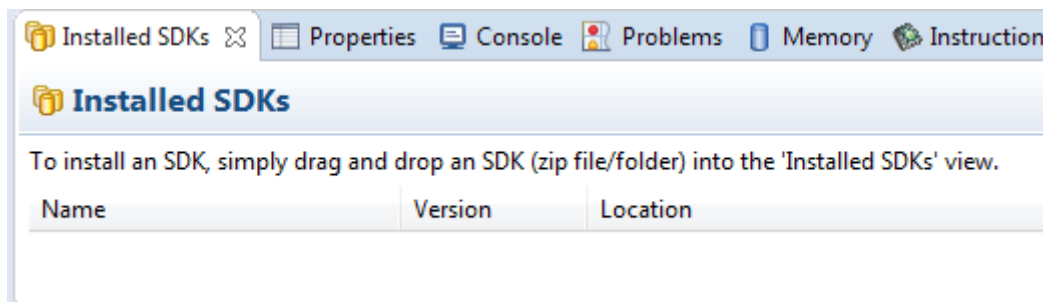
Run a demo using MCUXpresso IDE **Note:** Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

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

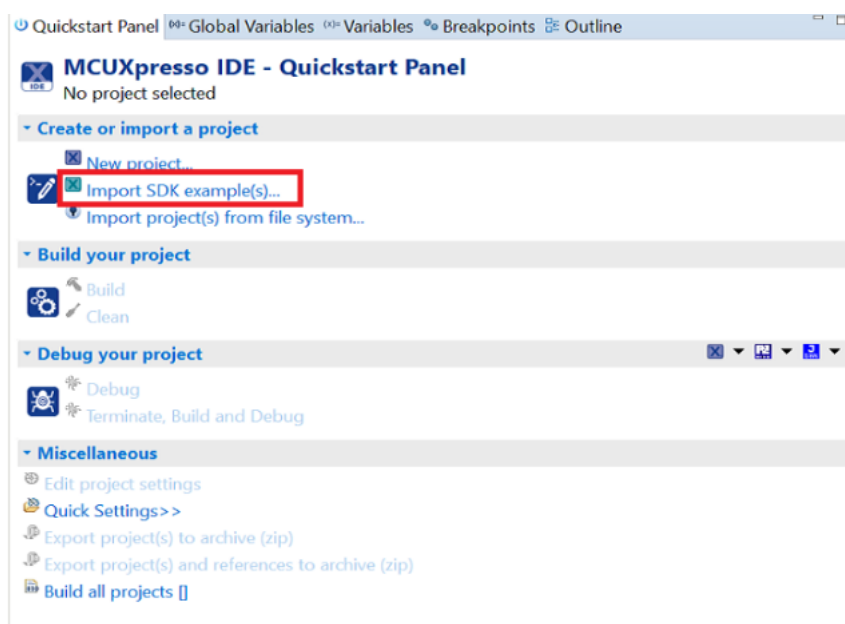
Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace.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

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

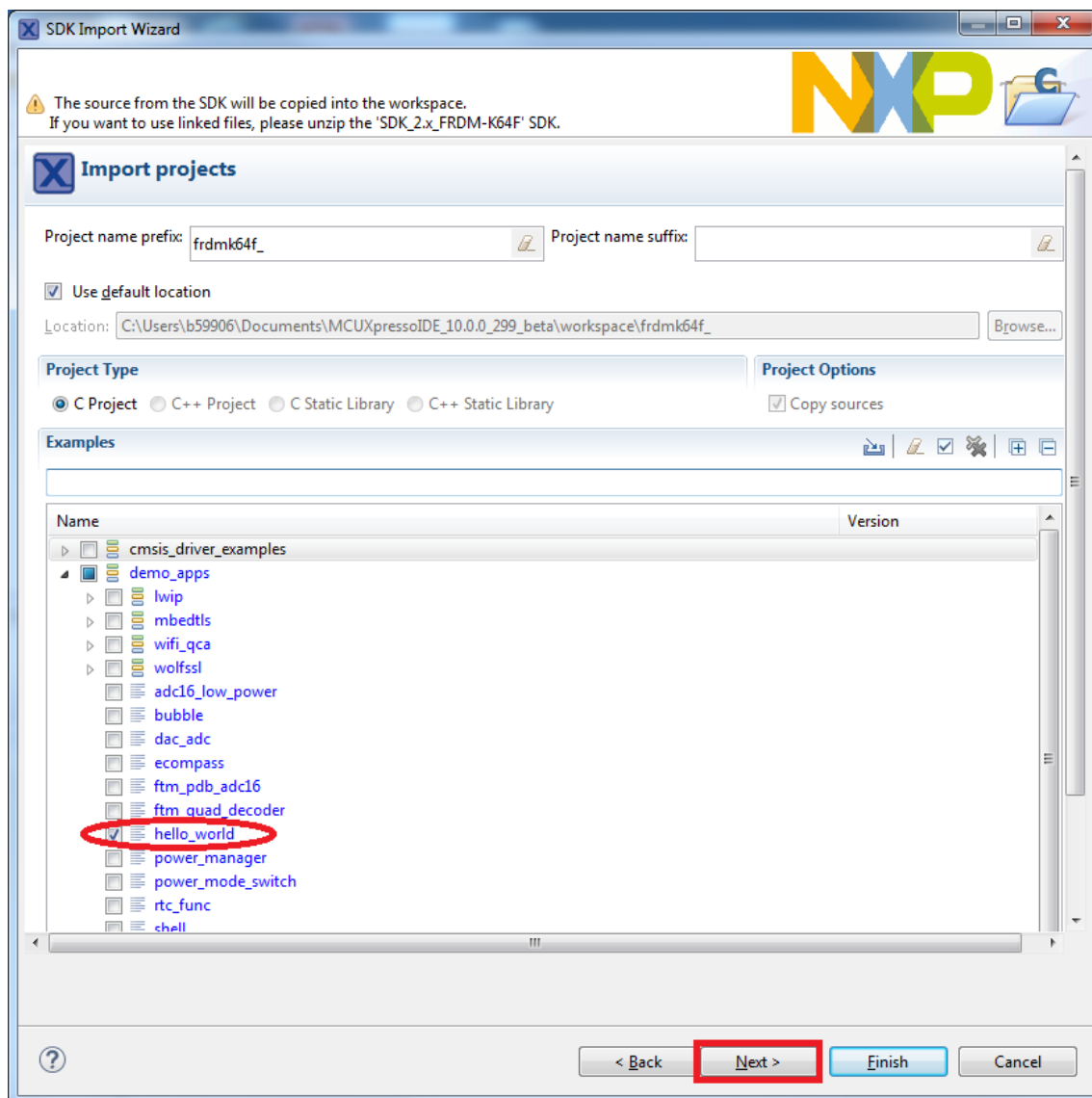
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



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



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



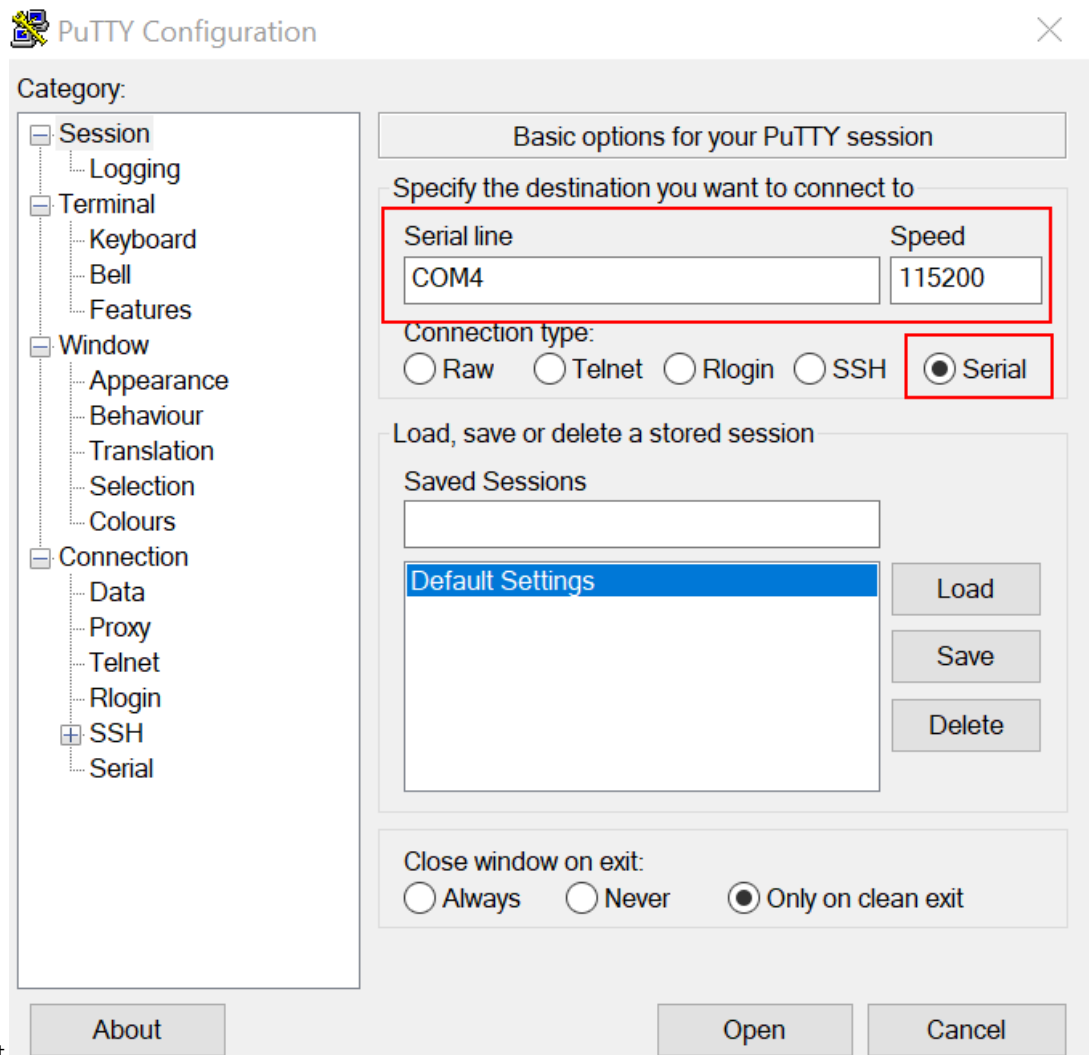
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminal for demo applications such as `adc_basic`, `adc_burst`, `adc_dma`, and `adc_interrupt`. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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

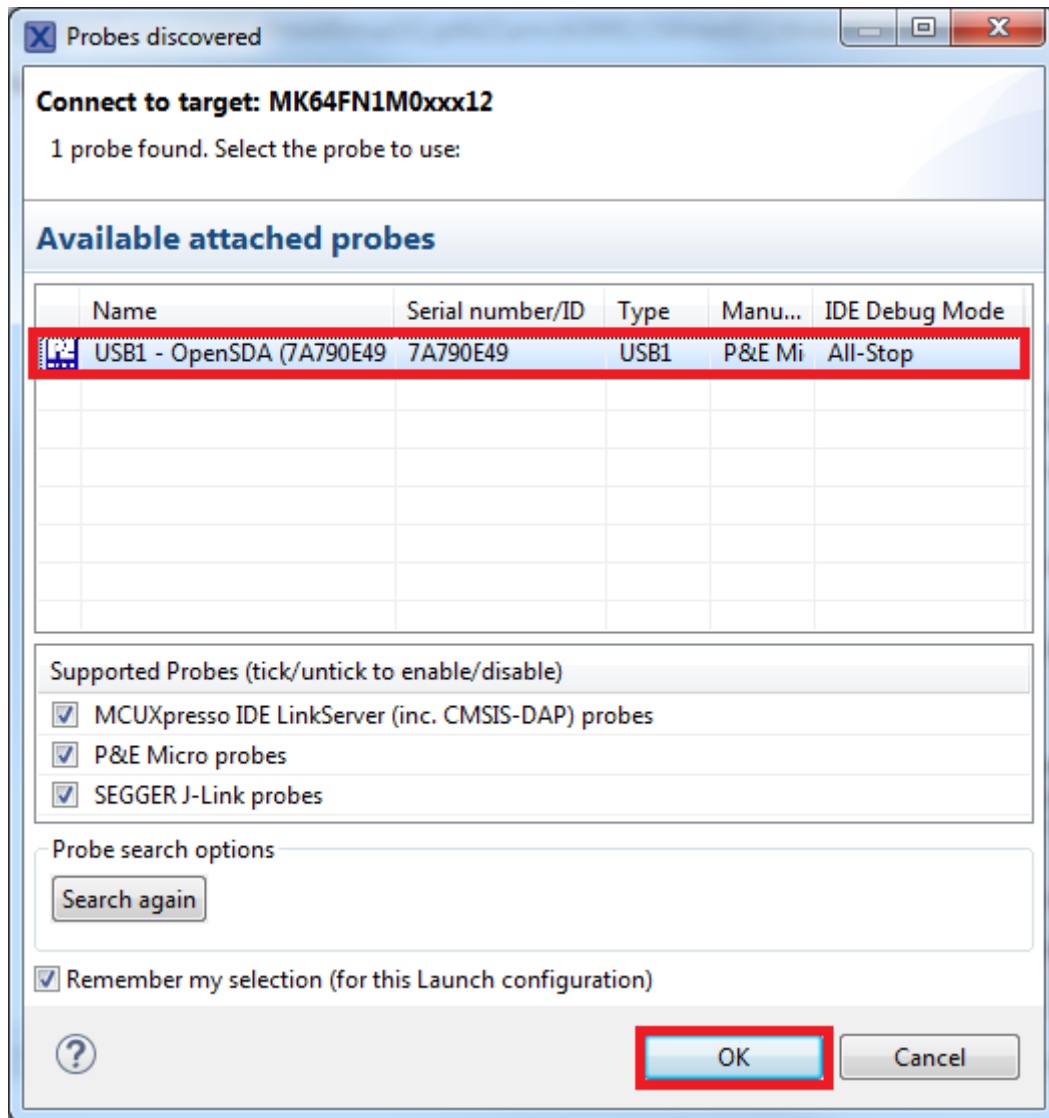
To download and run the application, perform the following steps:

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

1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
2. No parity
3. 8 data bits



4. 1 stop bit
4. On the **Quickstart Panel**, click **Debug** to launch the debug session.
5. The first time you debug a project, the **Debug Emulator Selection** dialog is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)



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

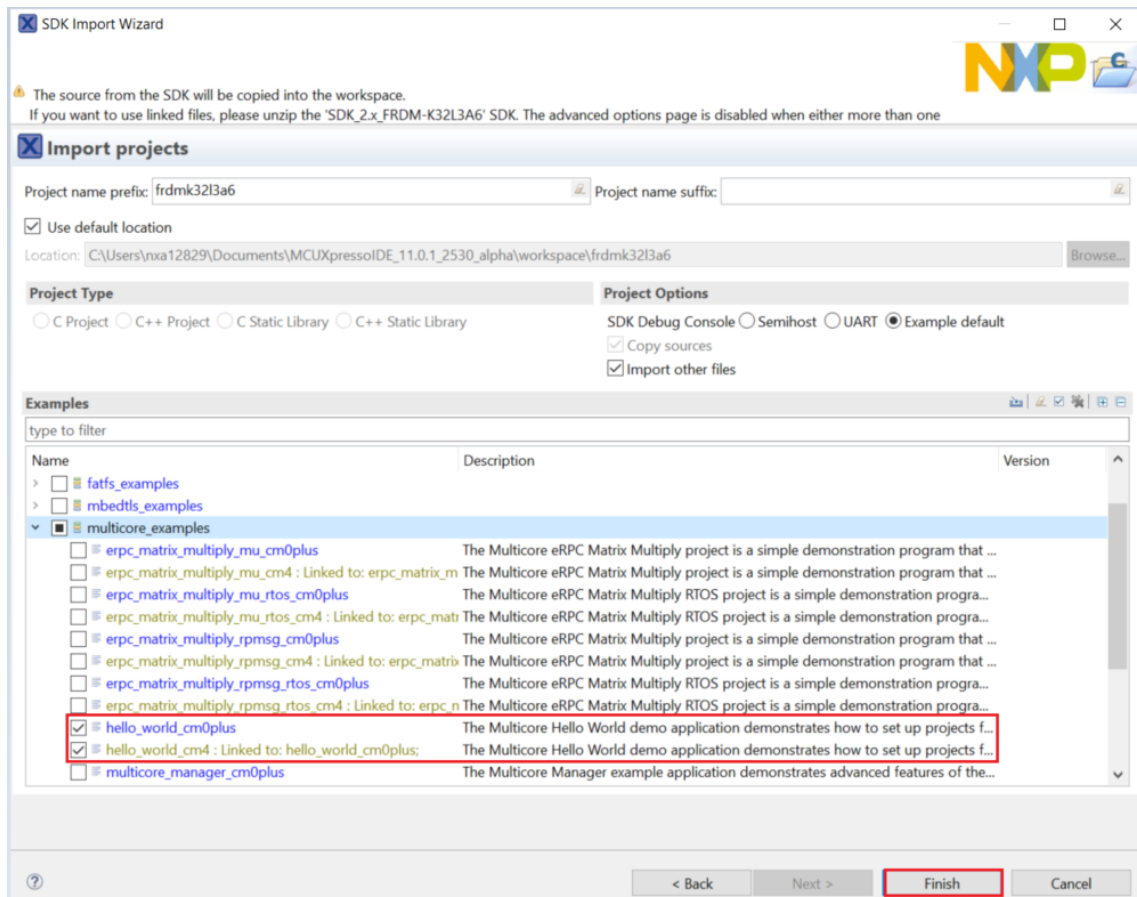


The `hello_world` application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.

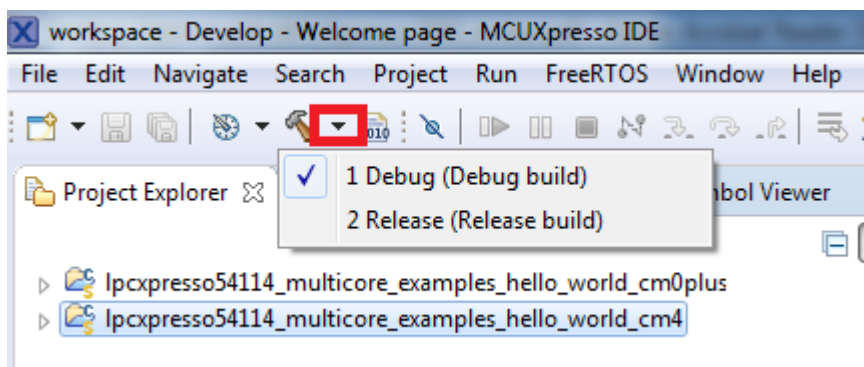


Build a multicore example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello_world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

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

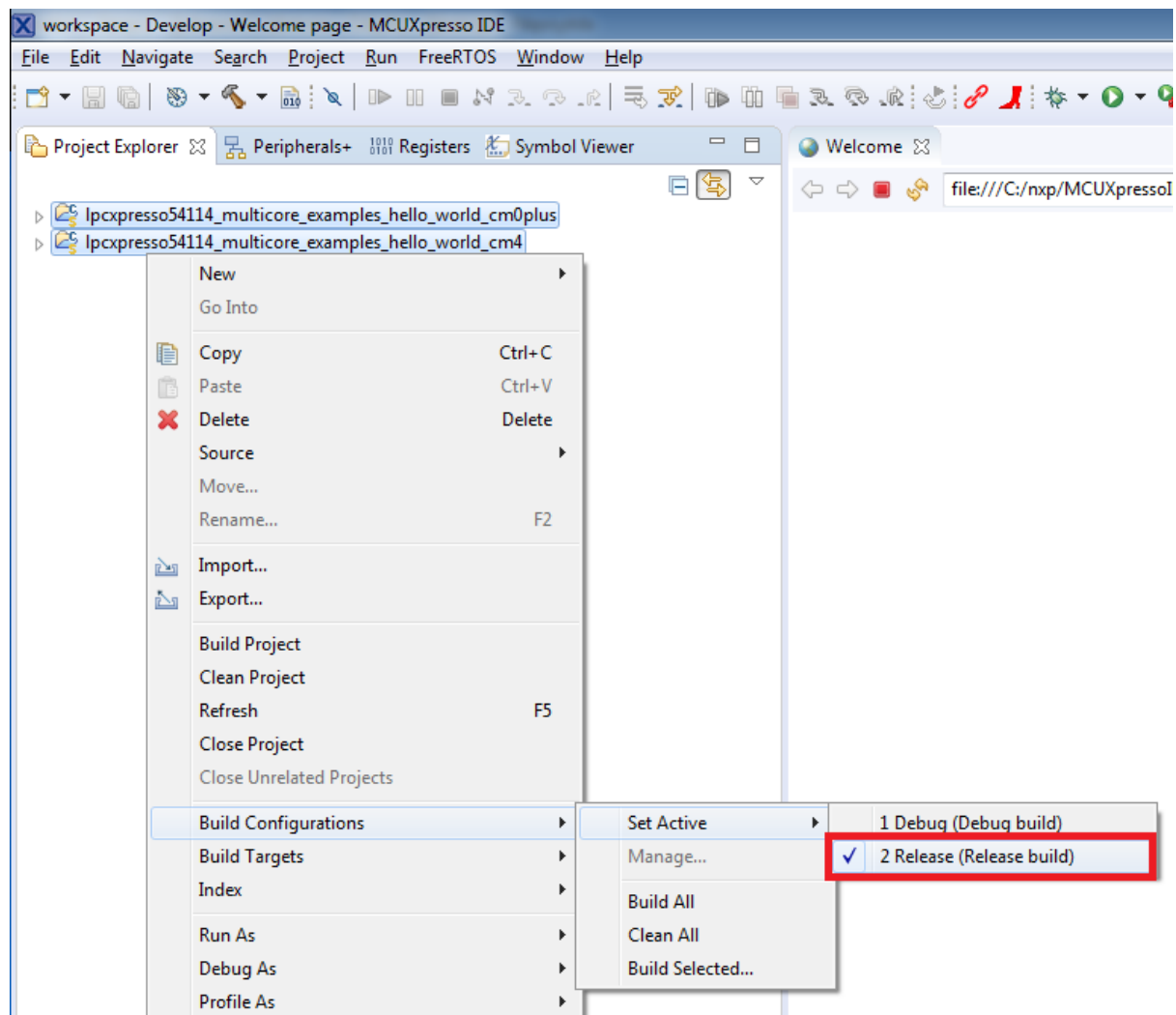


3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the `lpcxpresso54114_multicore_examples_hello_world_cm4` project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by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

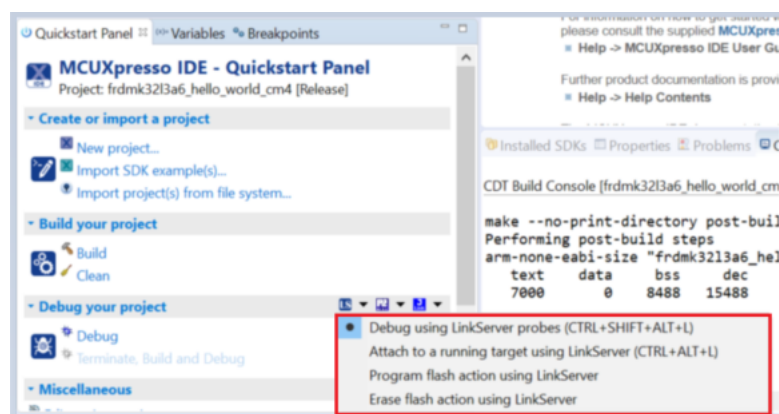


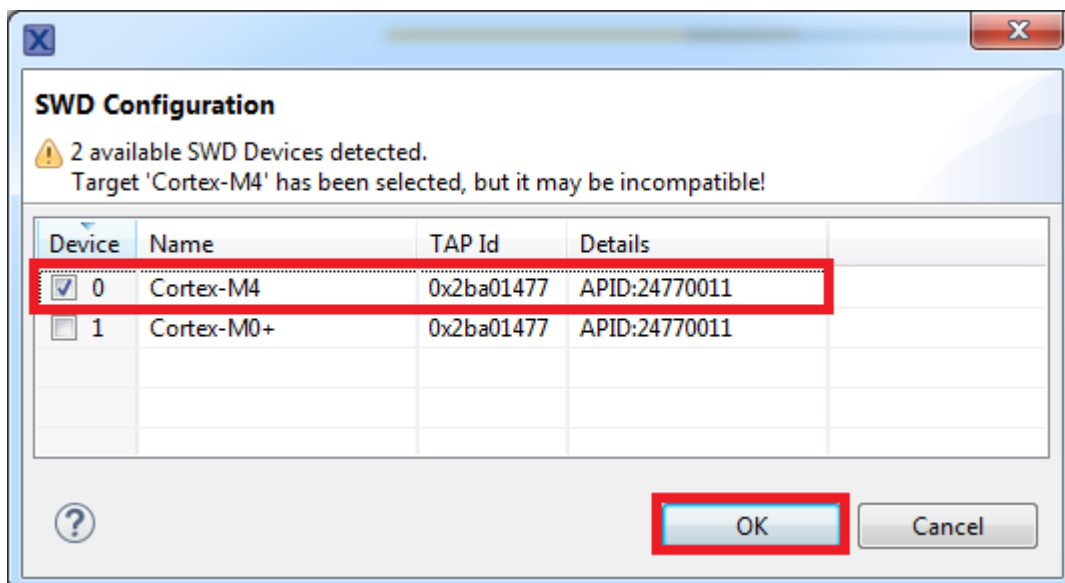
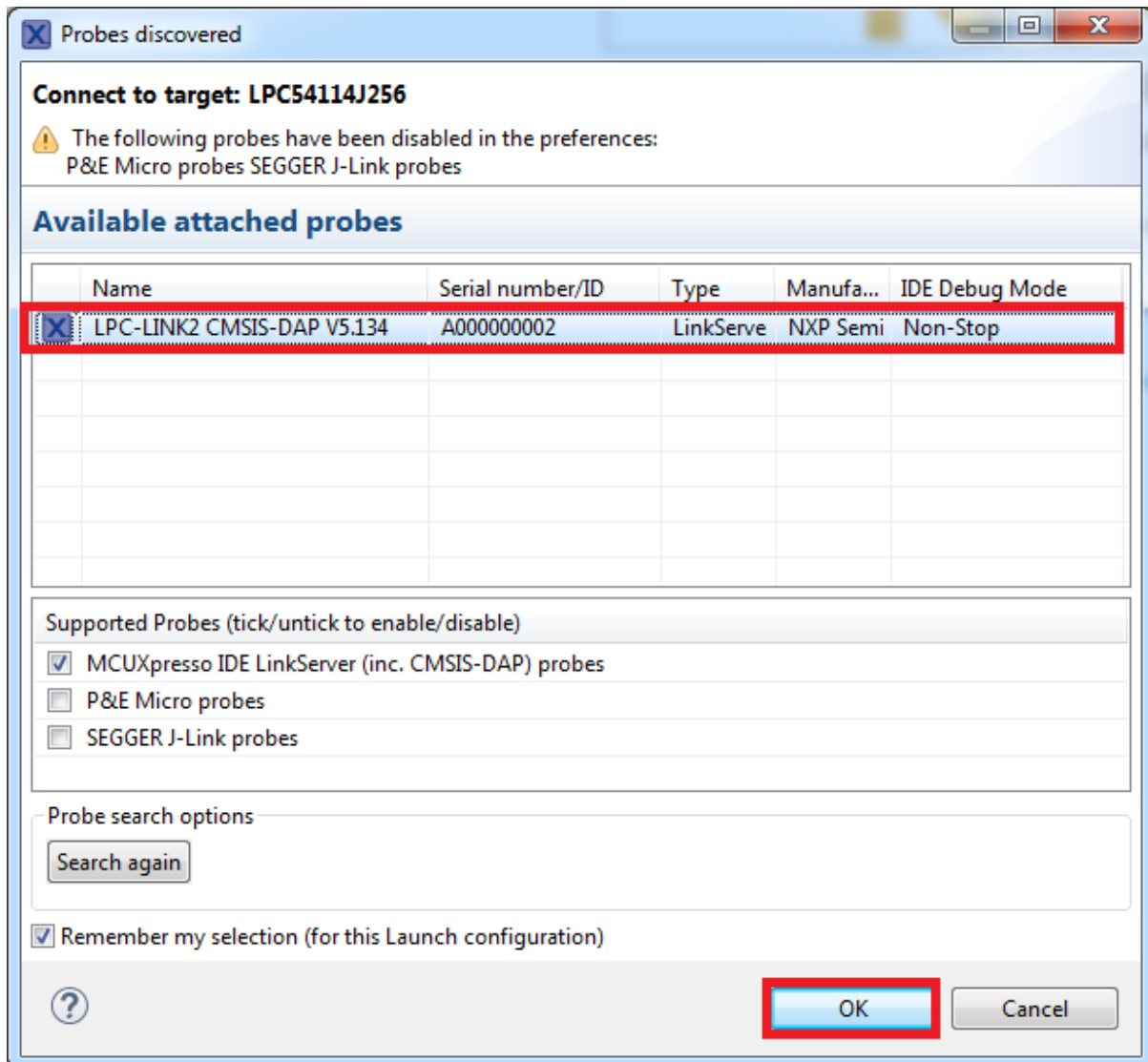
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

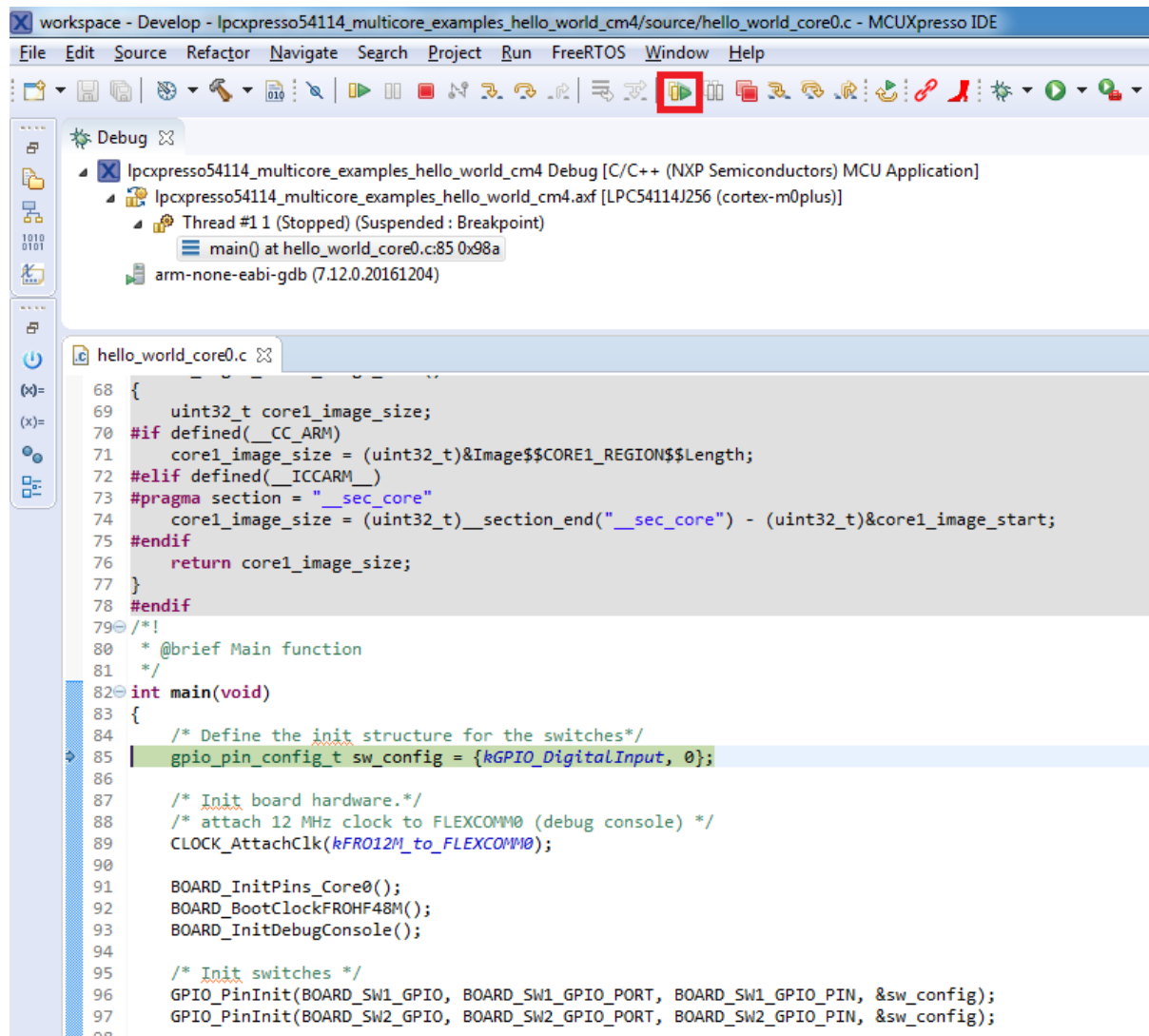
Note: When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **Build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



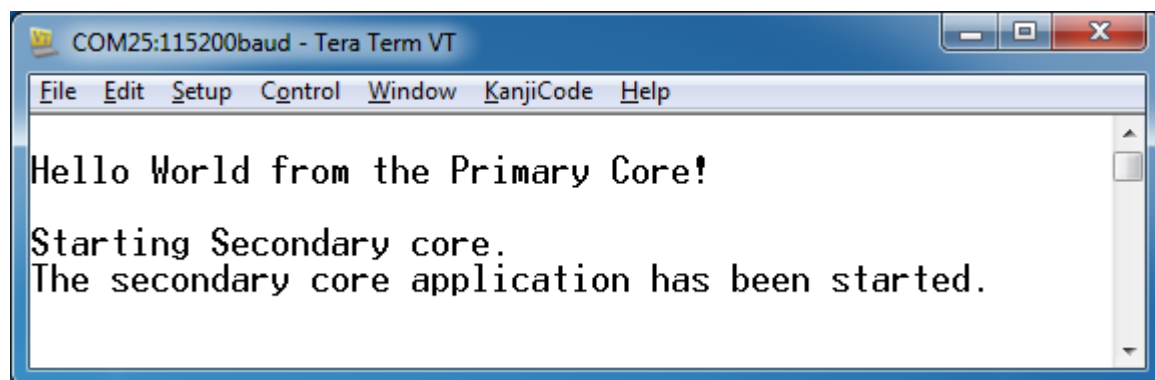
Run a multicore example application The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run an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





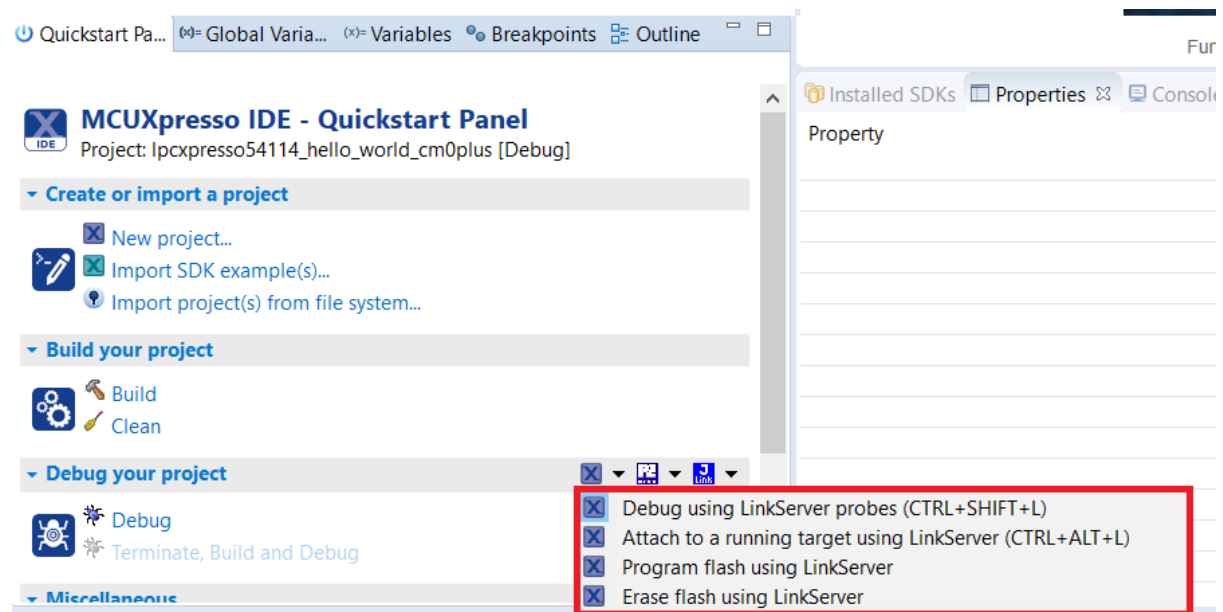


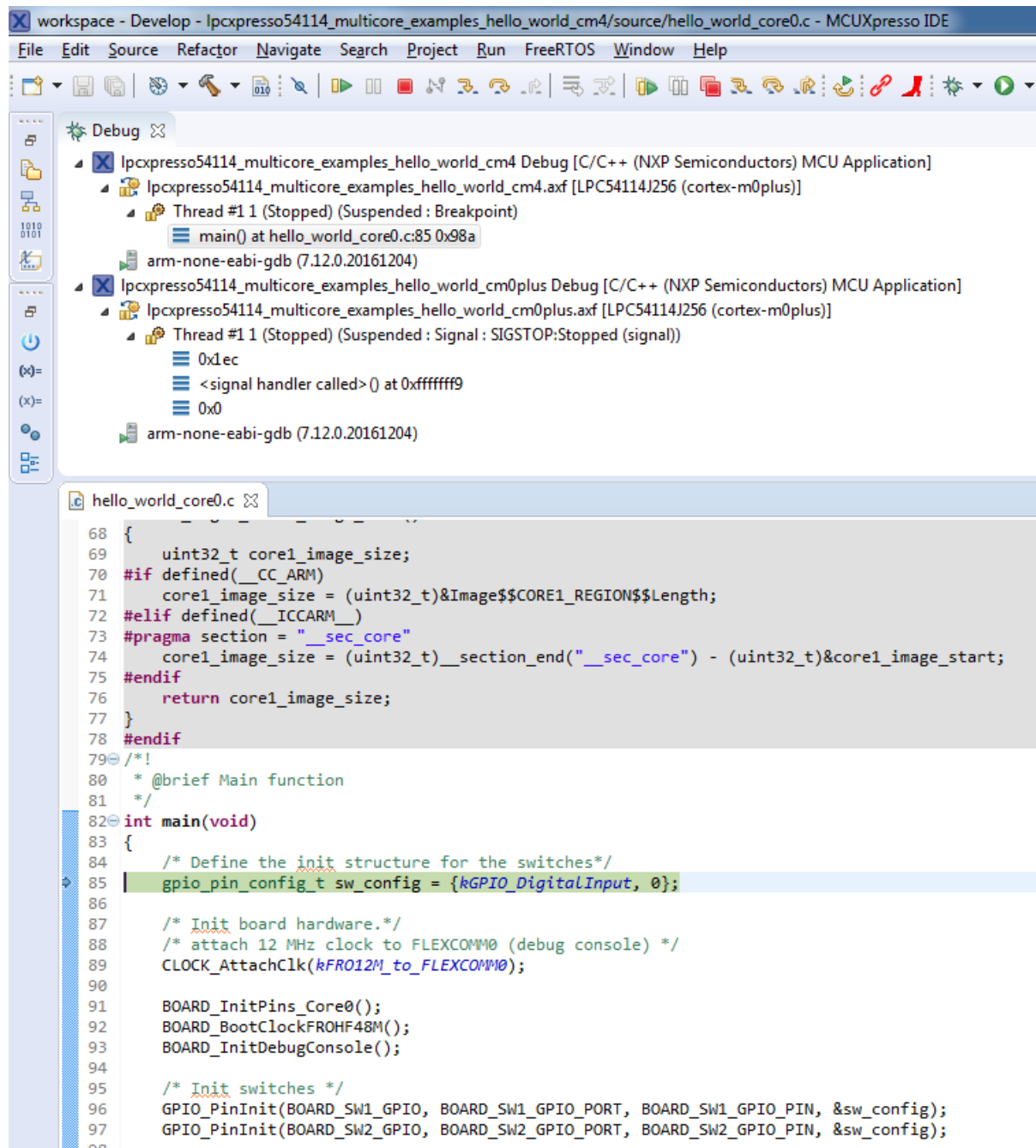
After clicking the “Resume All Debug sessions” button, the hello_world multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



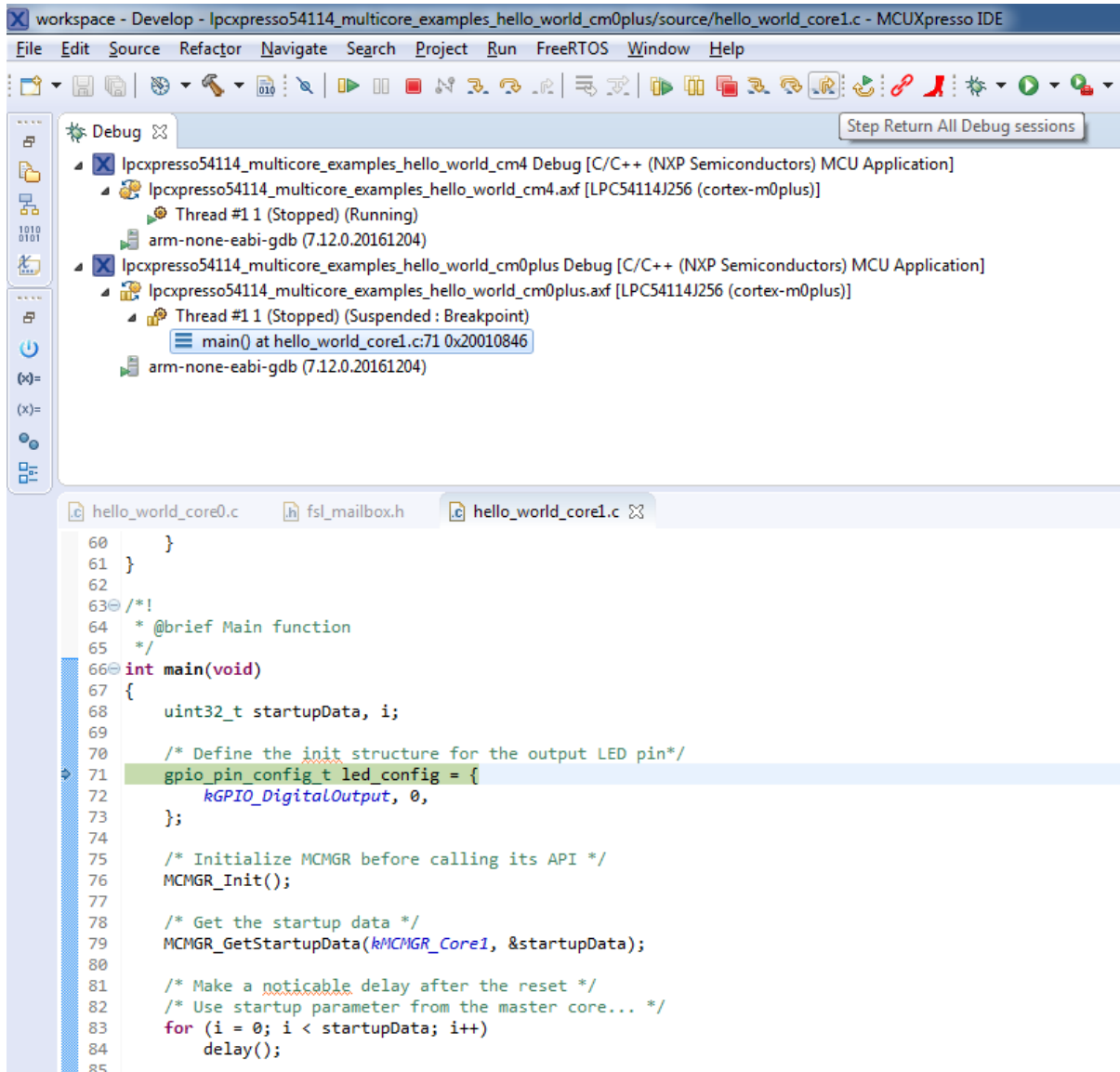
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps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

session.

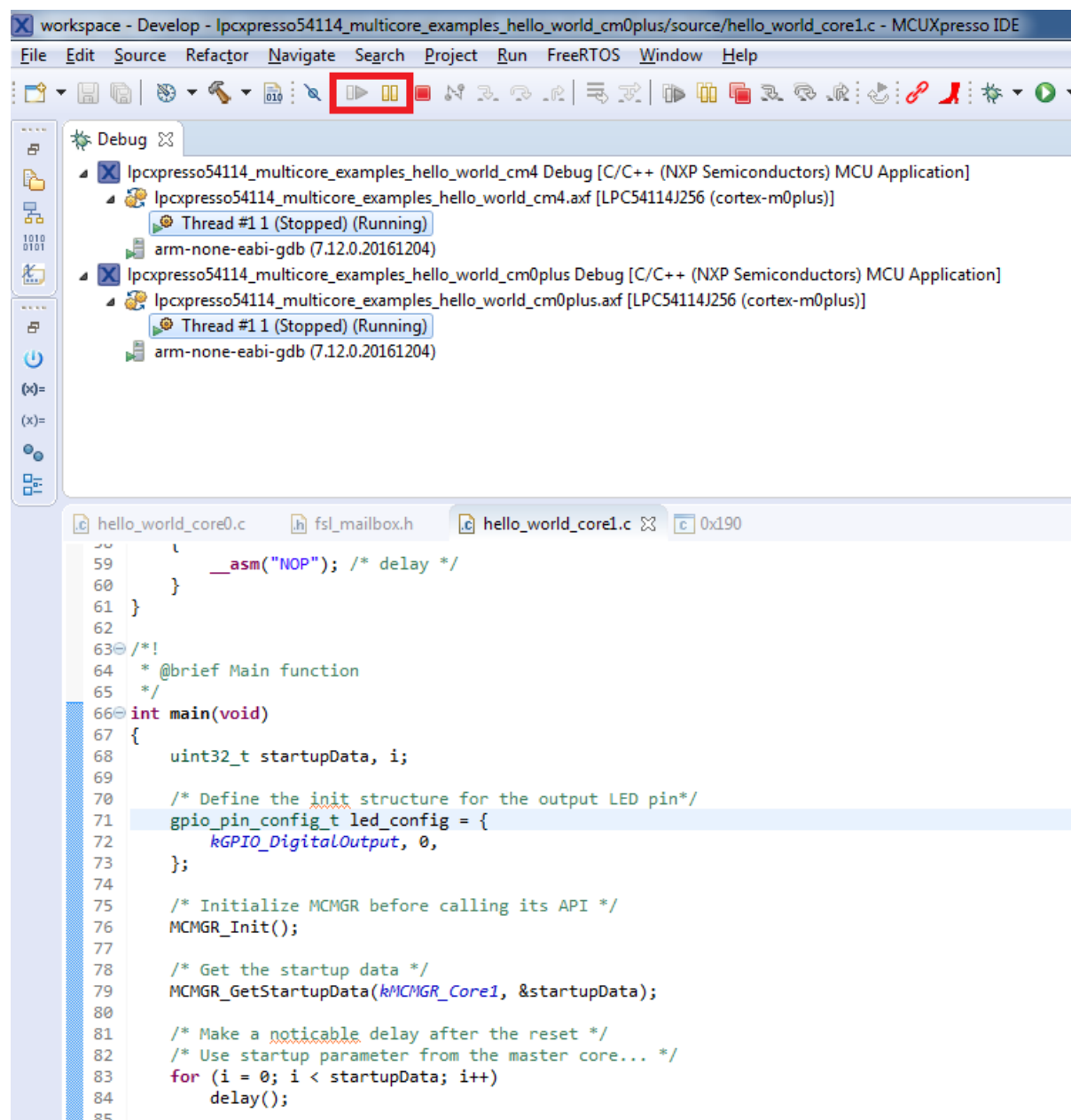


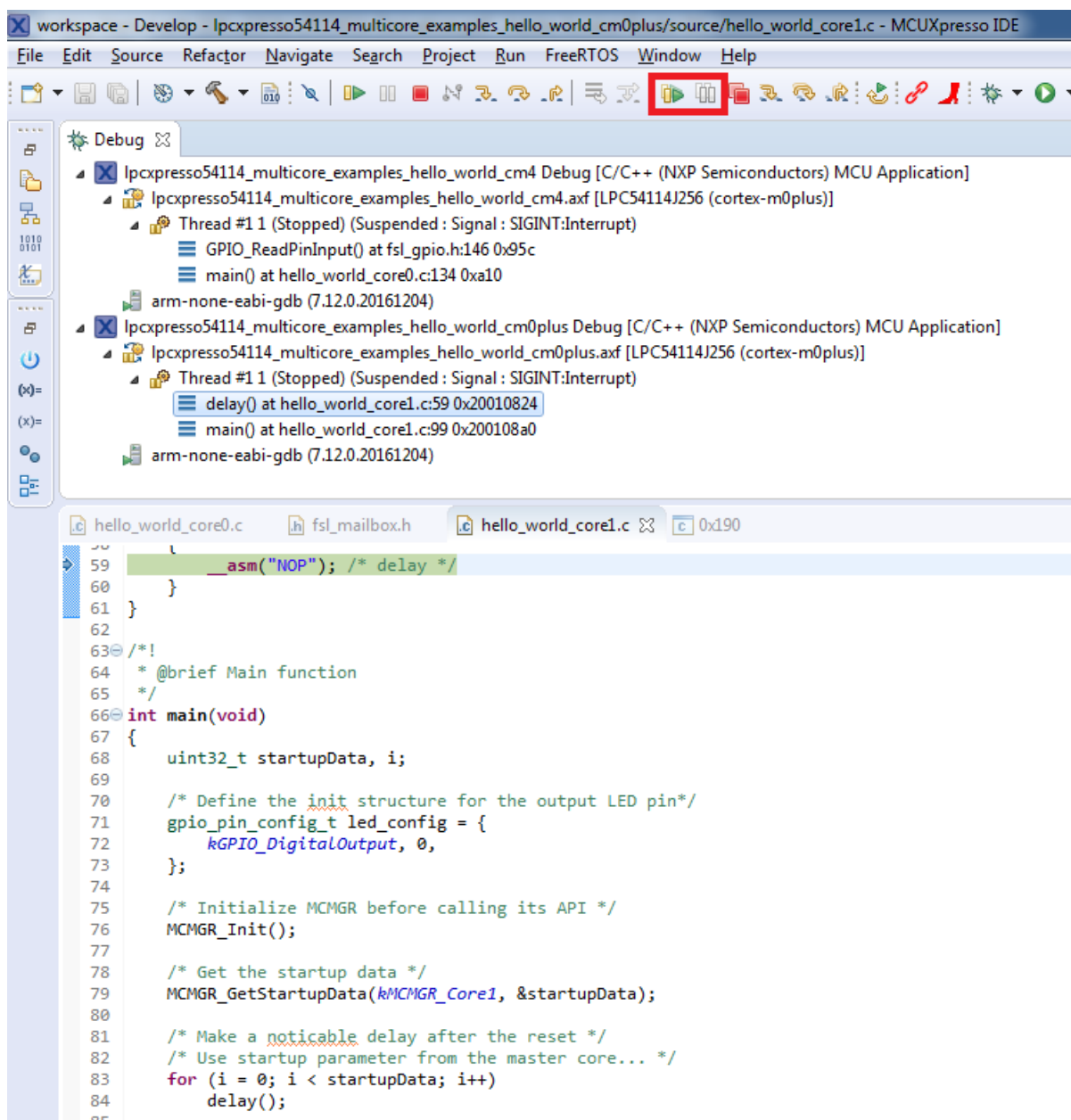


Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the “Resume” button. The hello_world multicore application then starts running. The primary core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



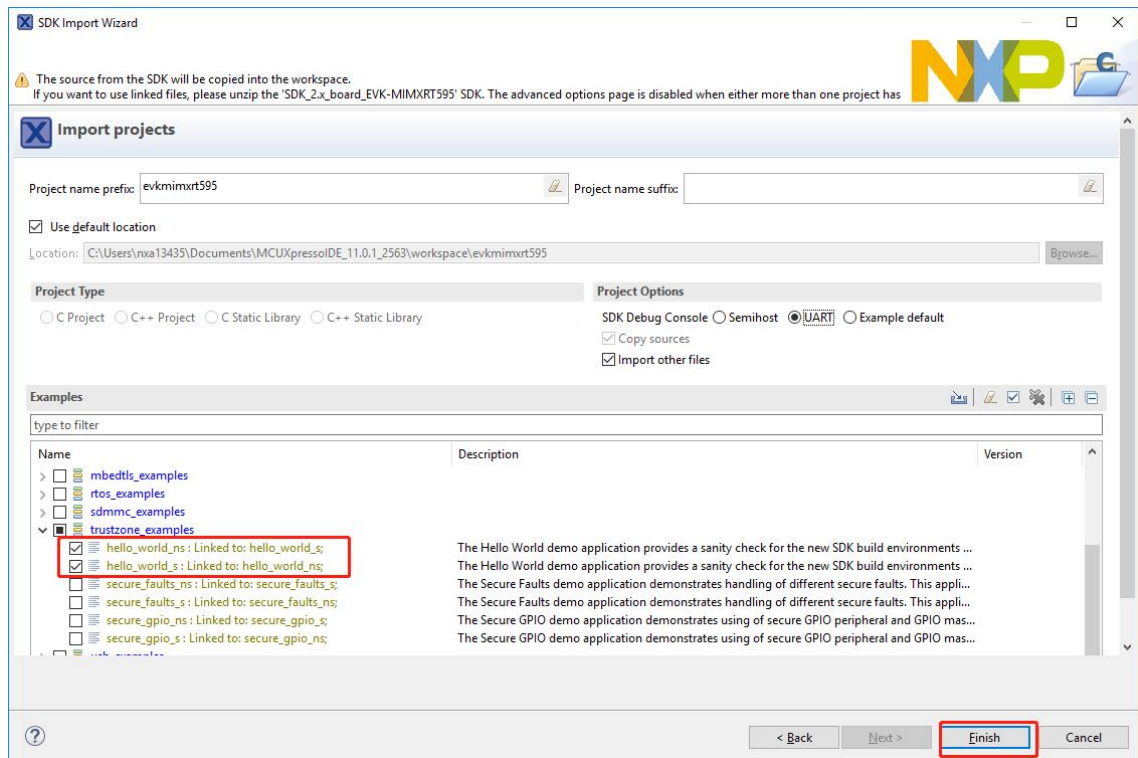
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the “Suspend” / “Resume” control button,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



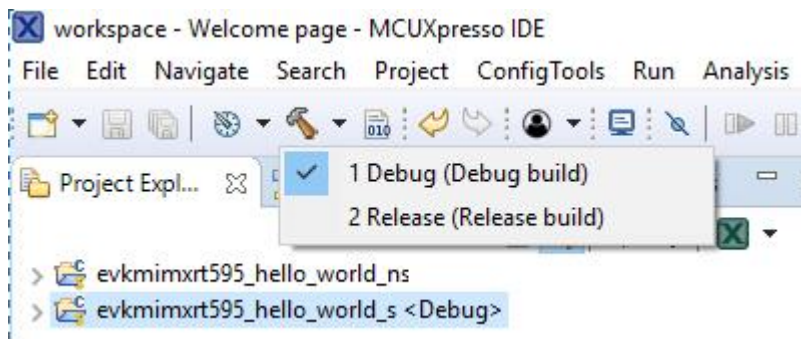


Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the hello_world example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though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

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

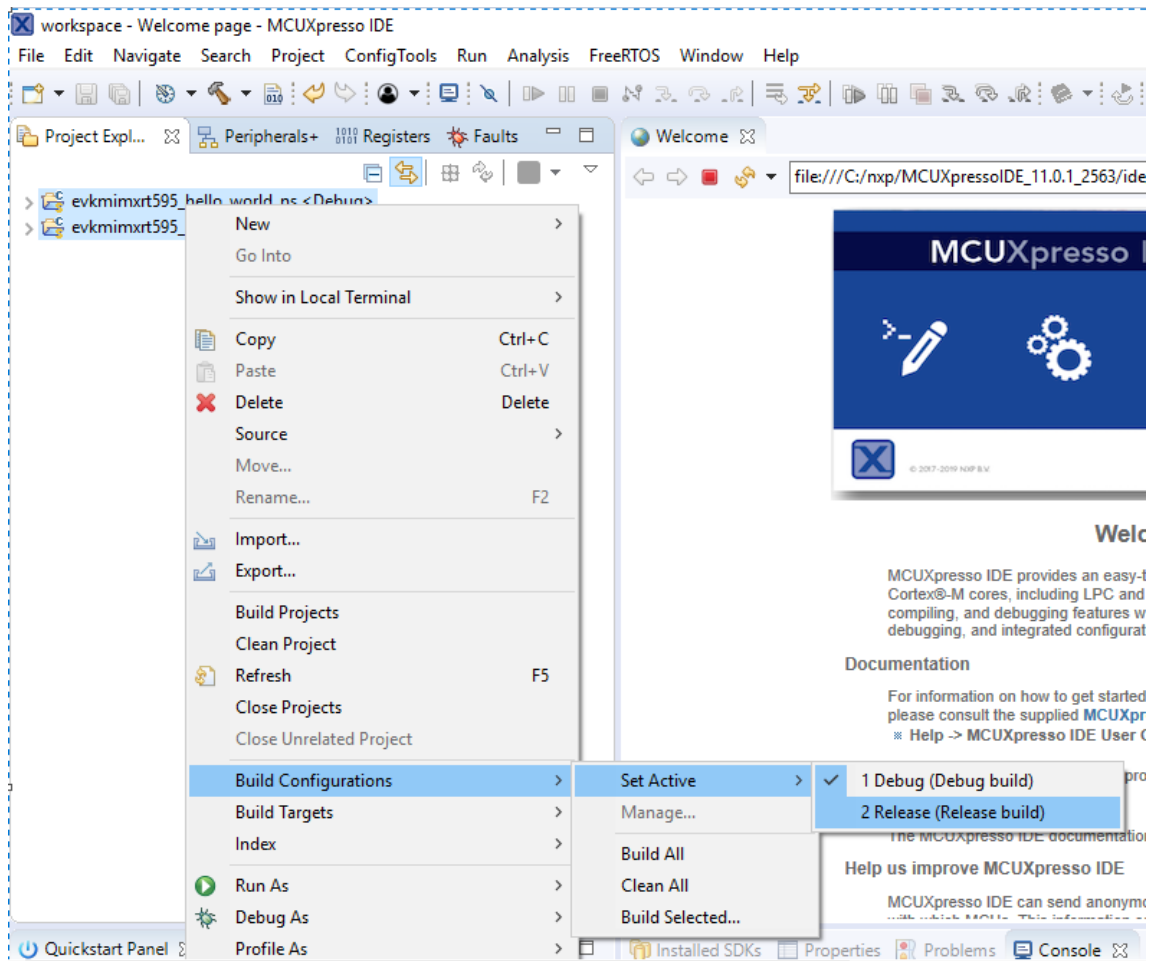


3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the `evkmimxrt595_hello_world_s` project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward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



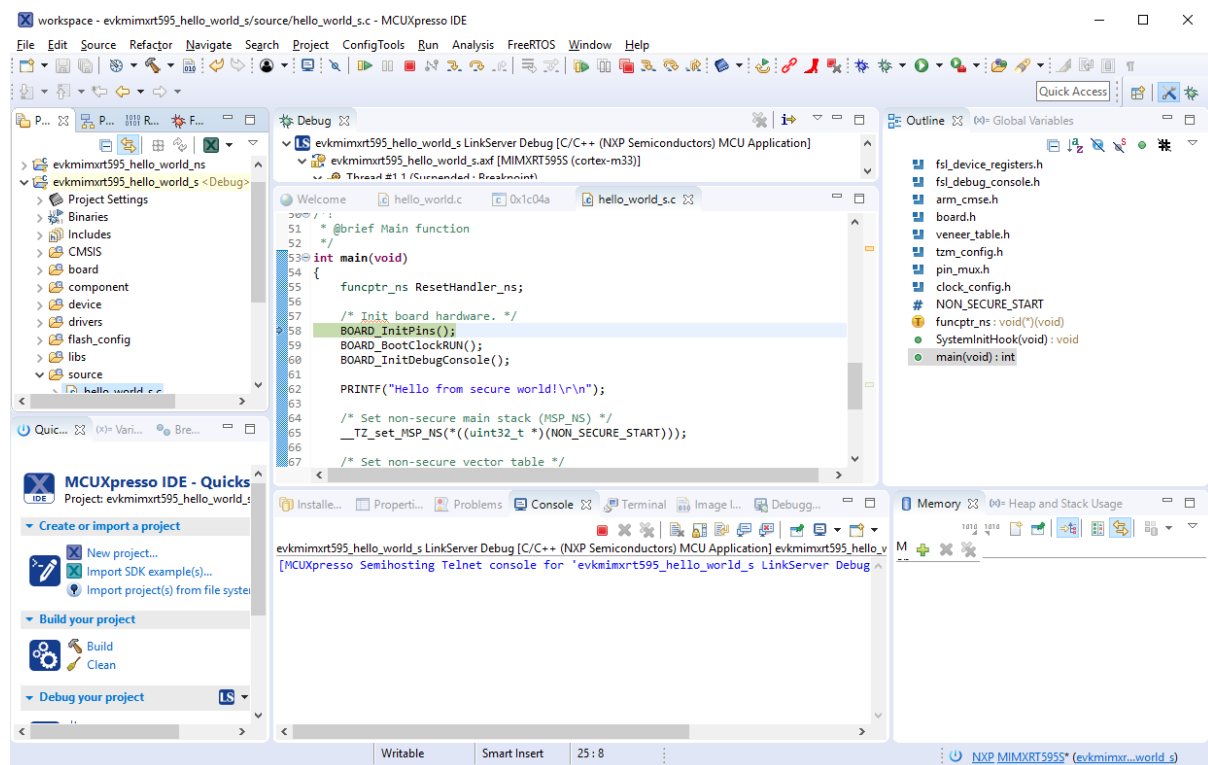
The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when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

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



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

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



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

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

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

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

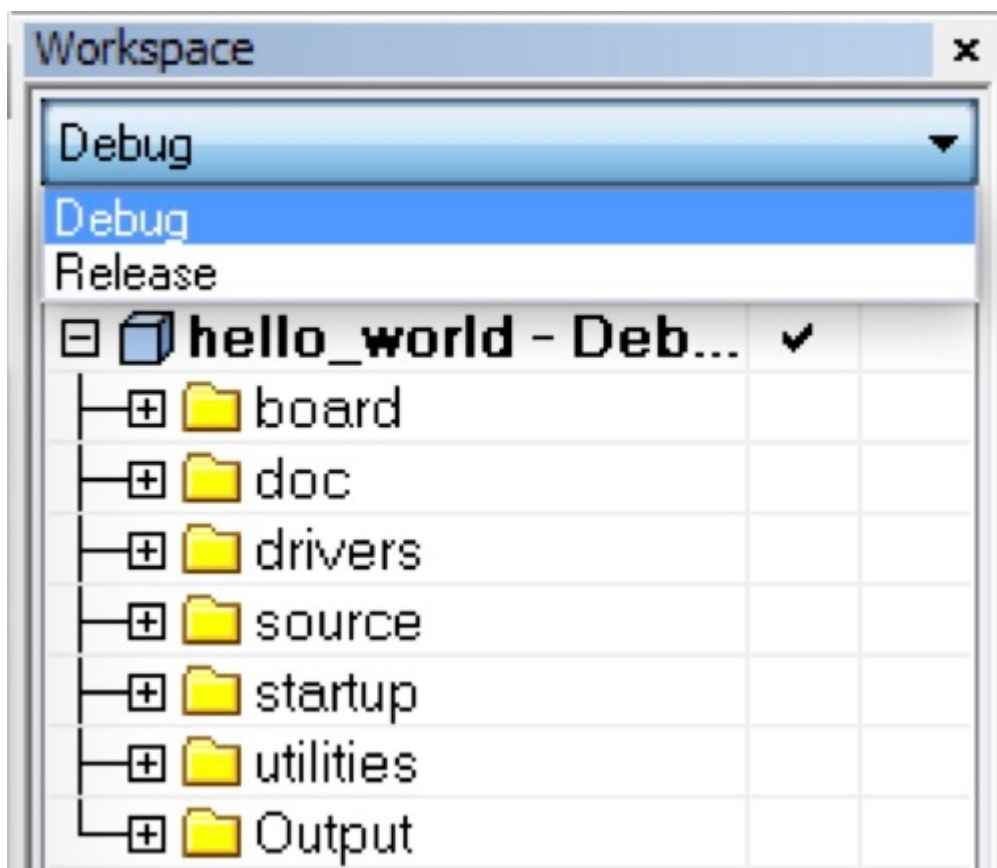
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

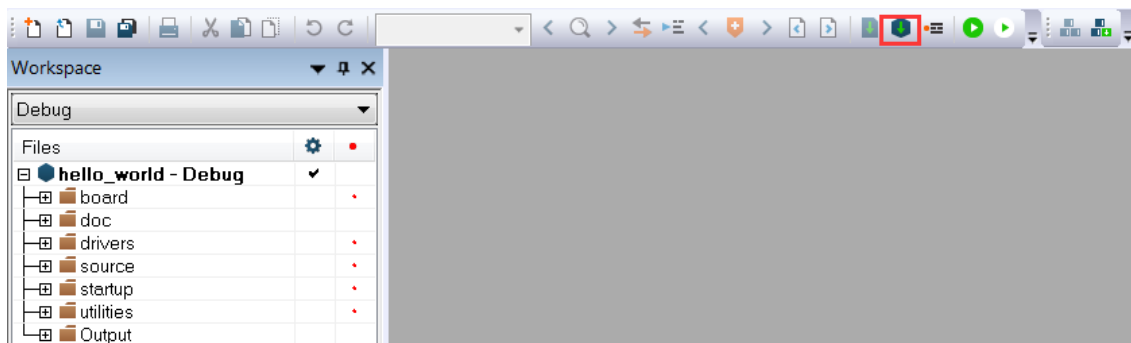
Other example applications may have additional folders in their path.

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

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



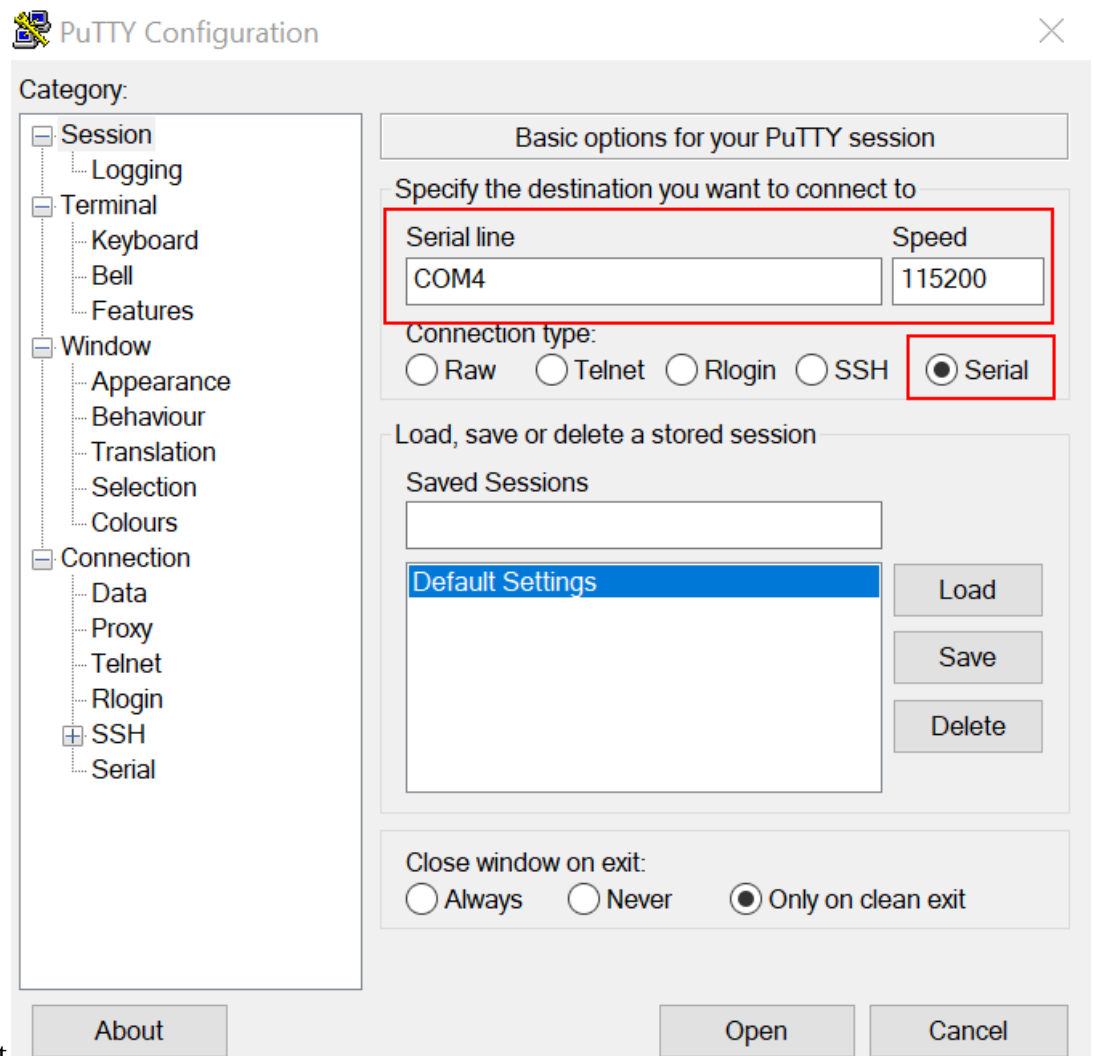
3. To build the demo application, click **Make**, highlighted in red in following figure.



4. The build completes without errors.

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

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

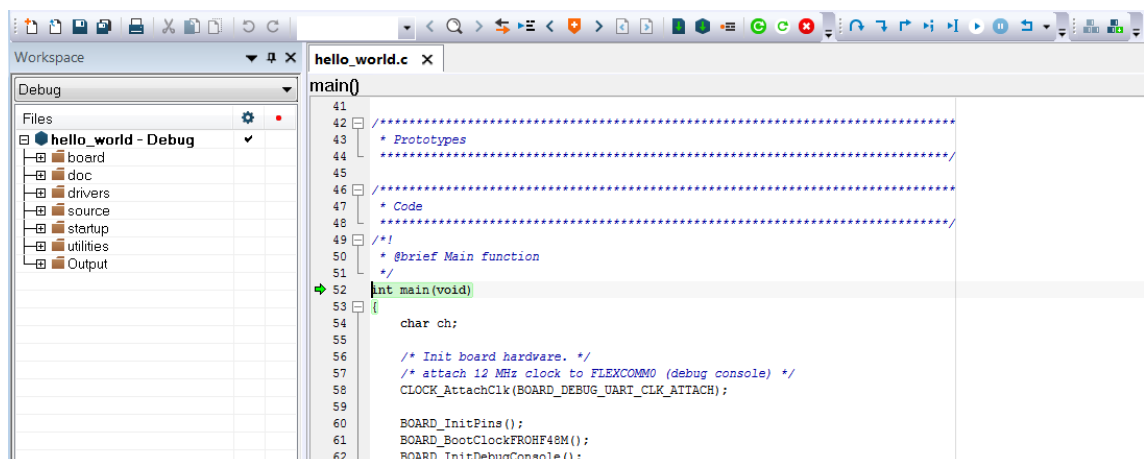


4. 1 stop bit

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



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



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

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



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

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

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

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

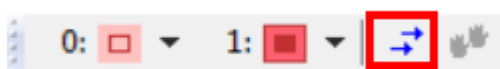
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/iar/hello_world_cm4.eww
```

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

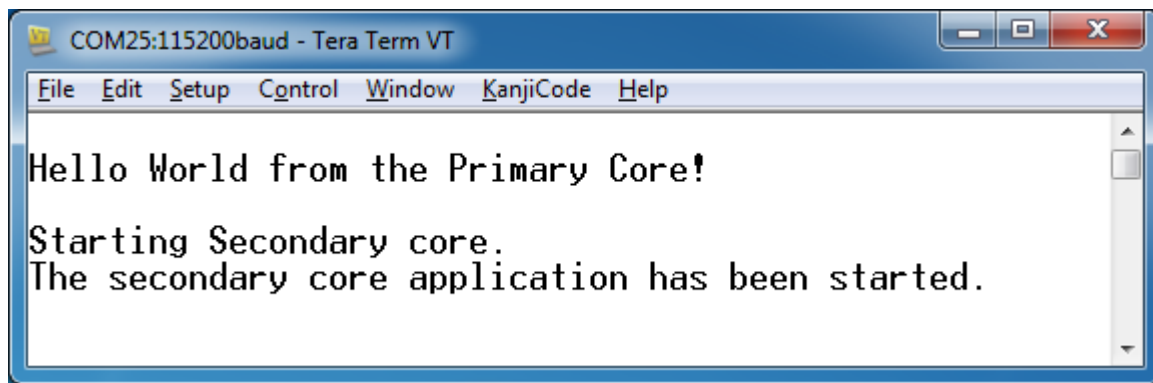
Run a multicore example application The primary core debugger handles flashing both primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 4 as described in **Run an example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

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

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



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



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

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

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

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

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

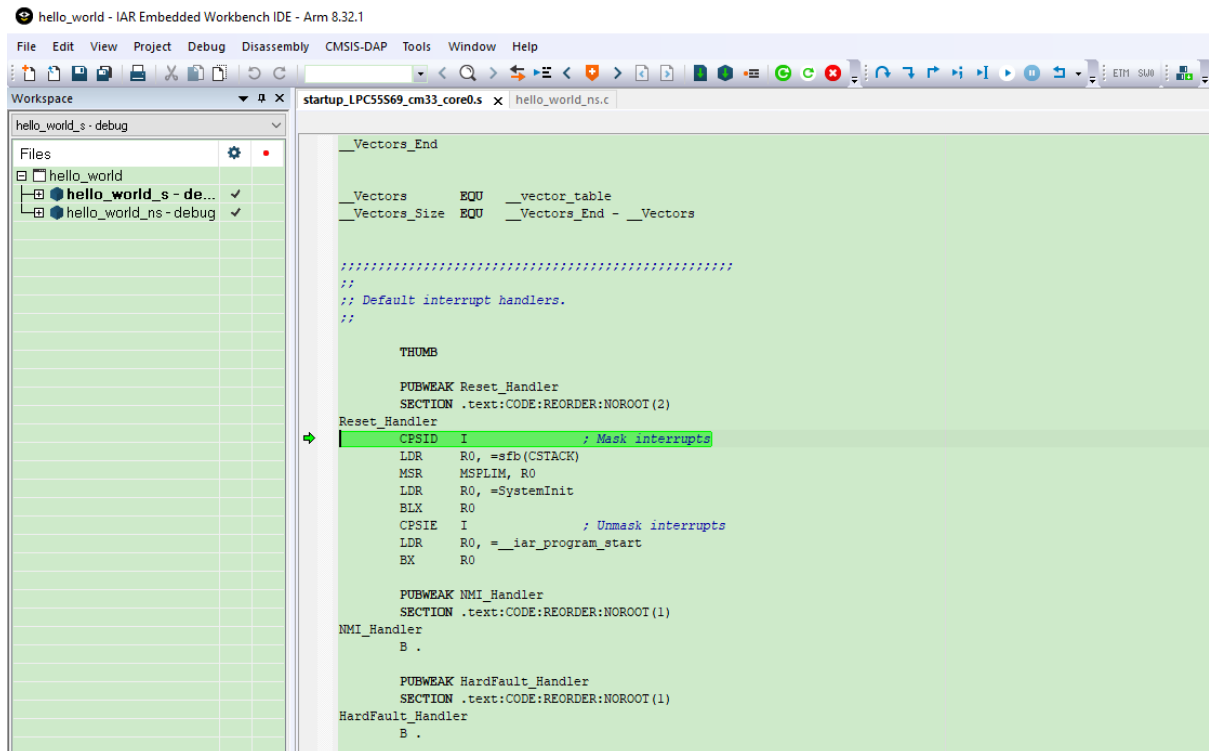
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_  
↪ns.eww
```

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

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

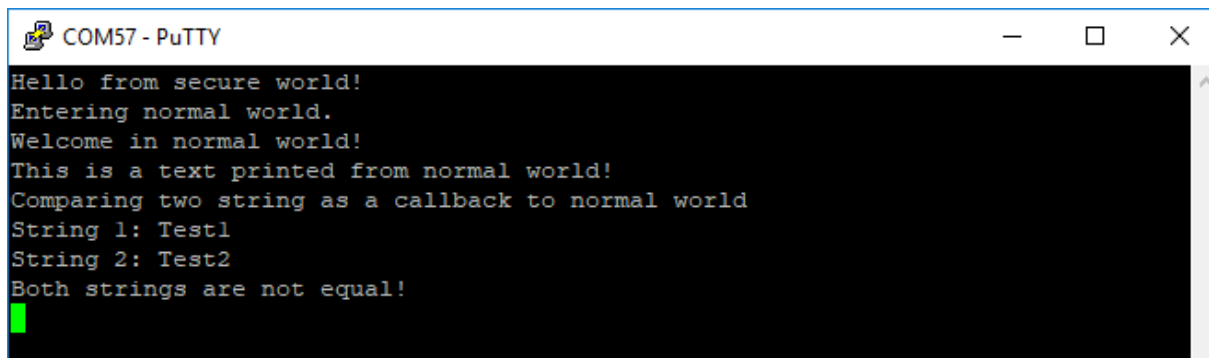
This project `hello_world.eww` contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because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

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

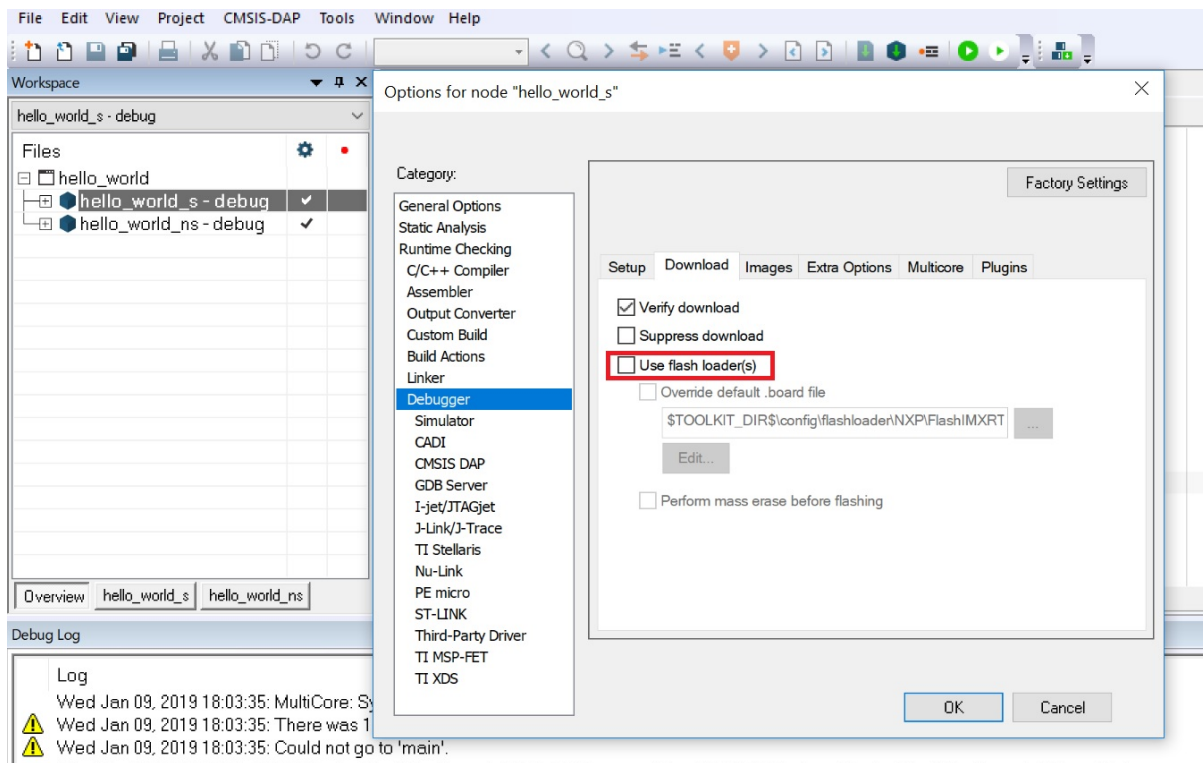


Run the code by clicking **Go** to start the application.

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



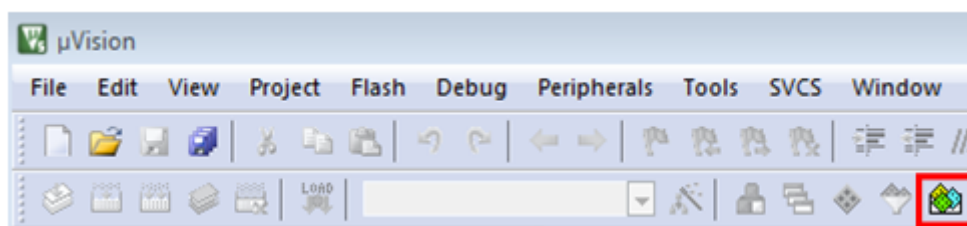
Note: If the application is running in RAM (debug/release build target), in **Options**>**Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the `_ns` download issue on i.MXRT500.



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

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

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



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

Build an example application

1. Open the desired example application workspace in:

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

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

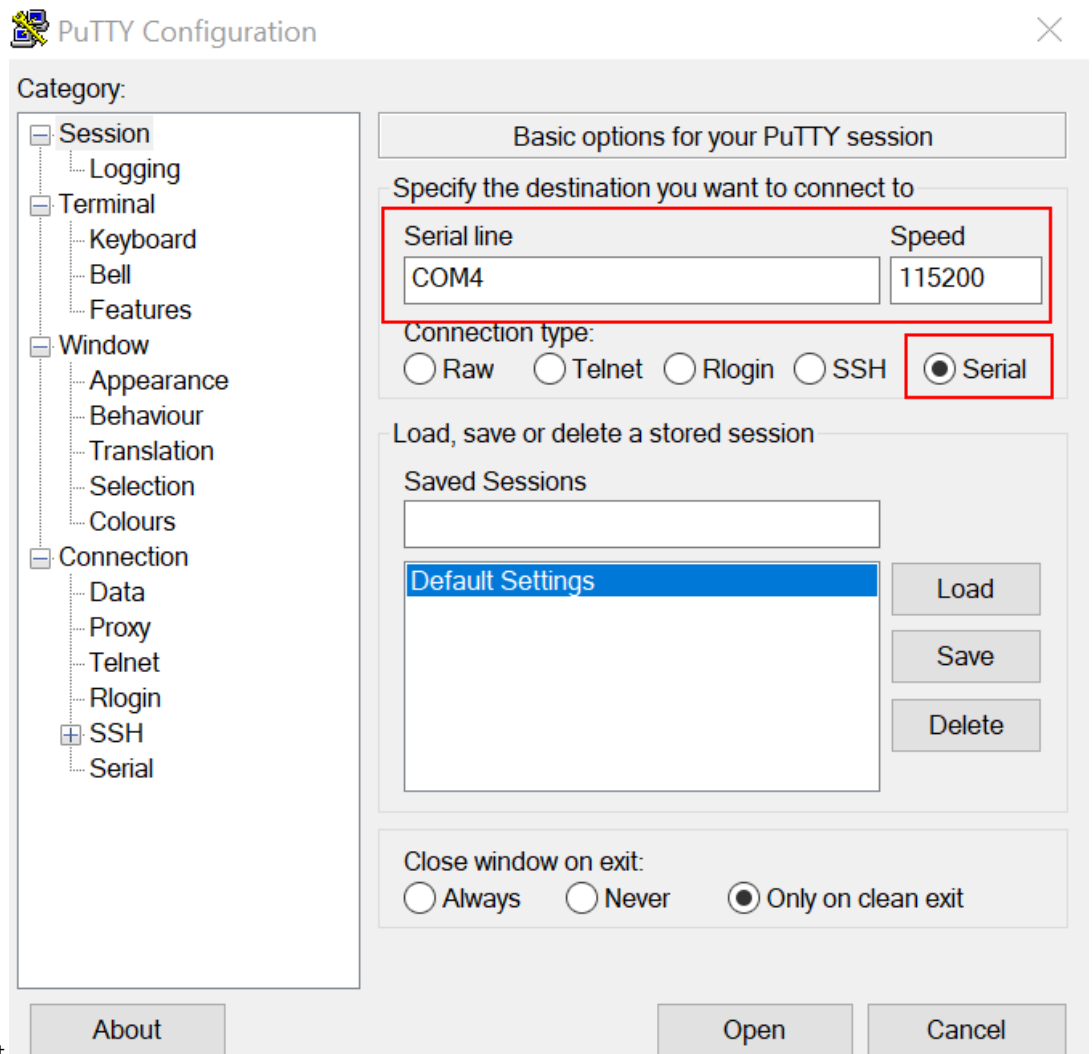
- To build the demo project, select **Rebuild**, highlighted in red.



- The build completes without errors.

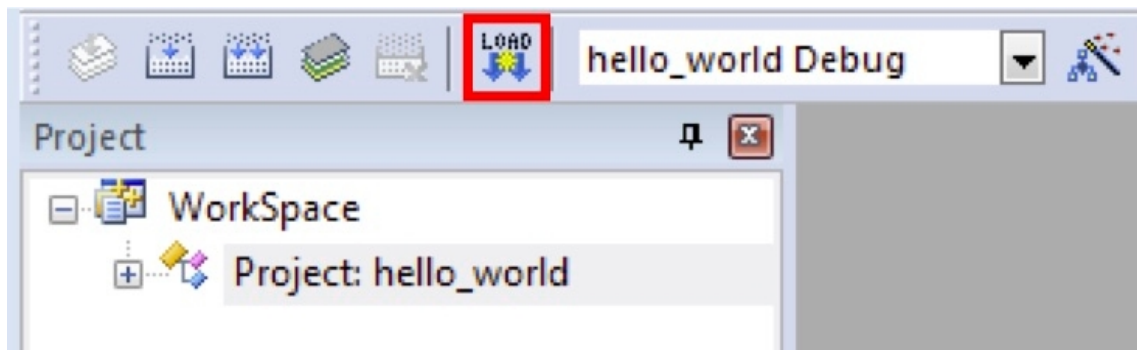
Run an example application To download and run the application, perform these steps:

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

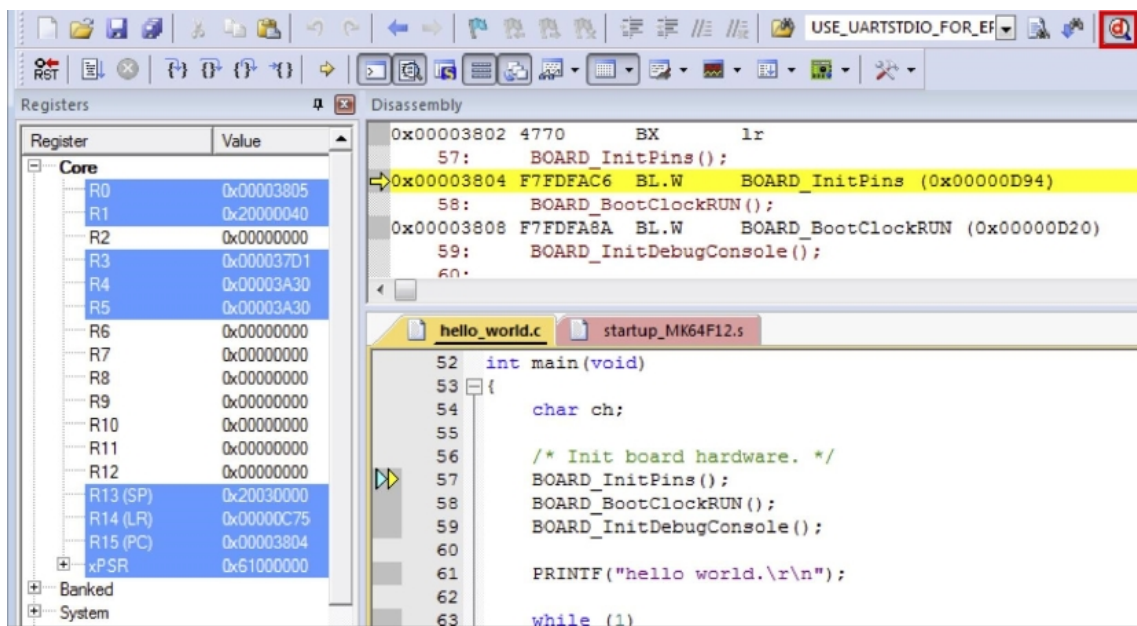


- 1 stop bit

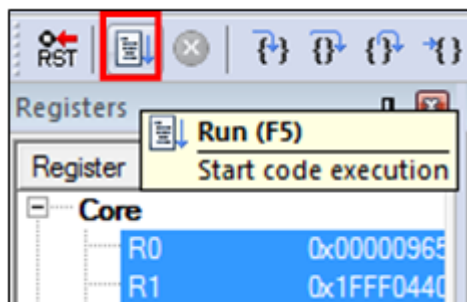
- In μ Vision, after the application is built, click the **Download** button to download the application to the target.



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



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



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



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

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

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

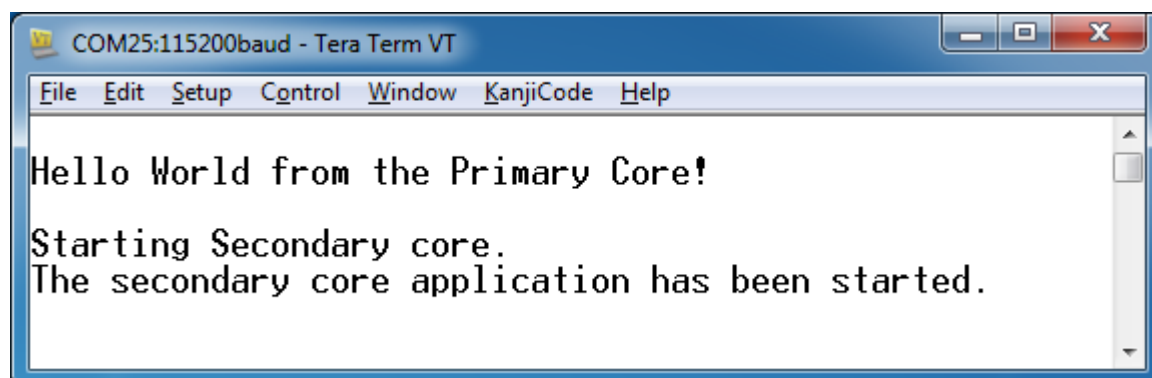
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
→cm0plus.uvmpw
```

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

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

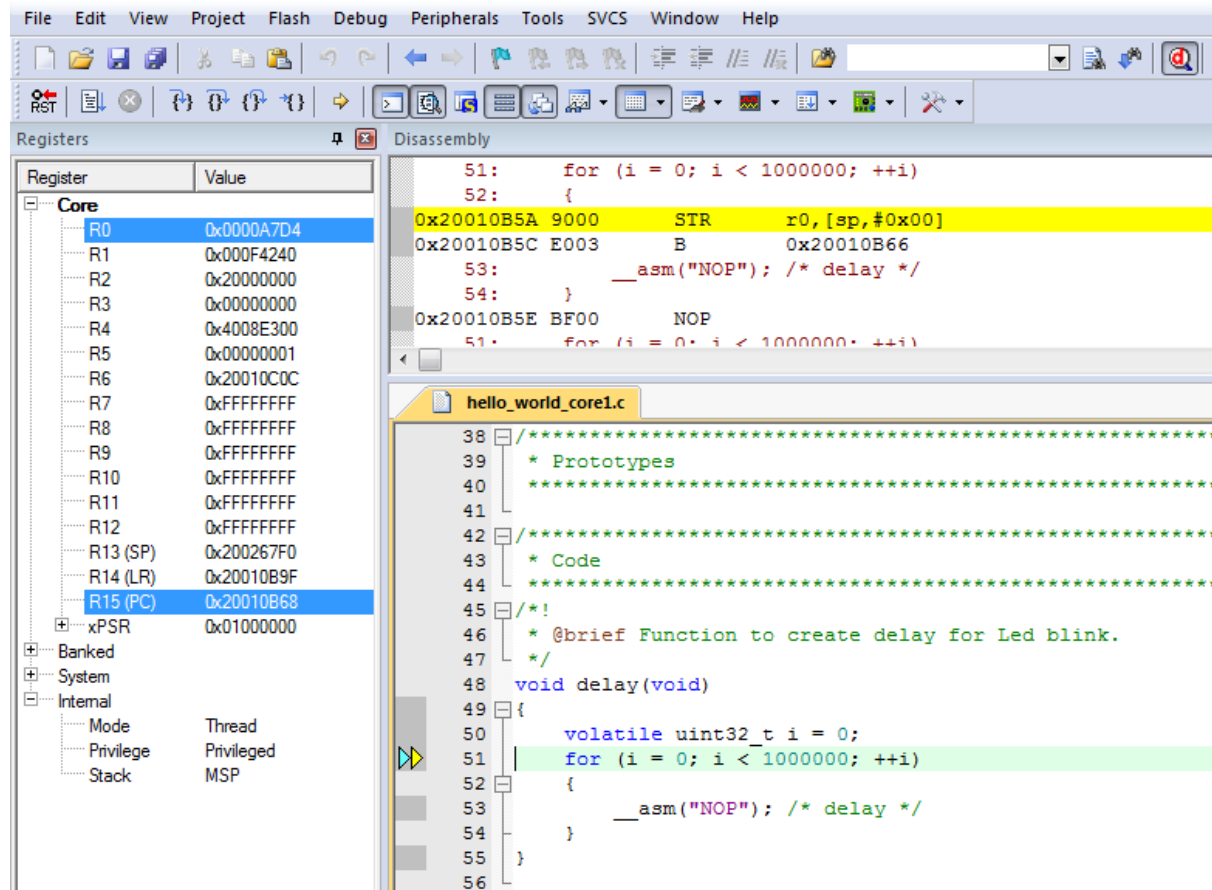
Run a multicore example application The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example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

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



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

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



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

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

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

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

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

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

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

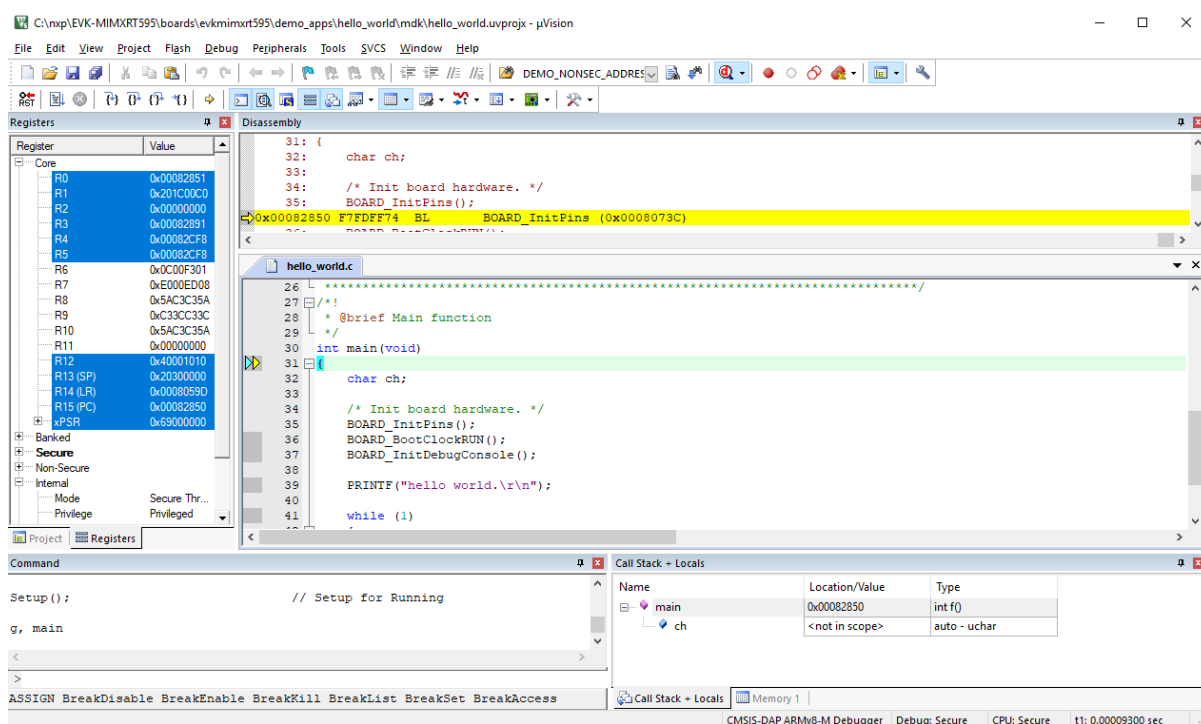
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.  
→ uvmpw
```

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

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

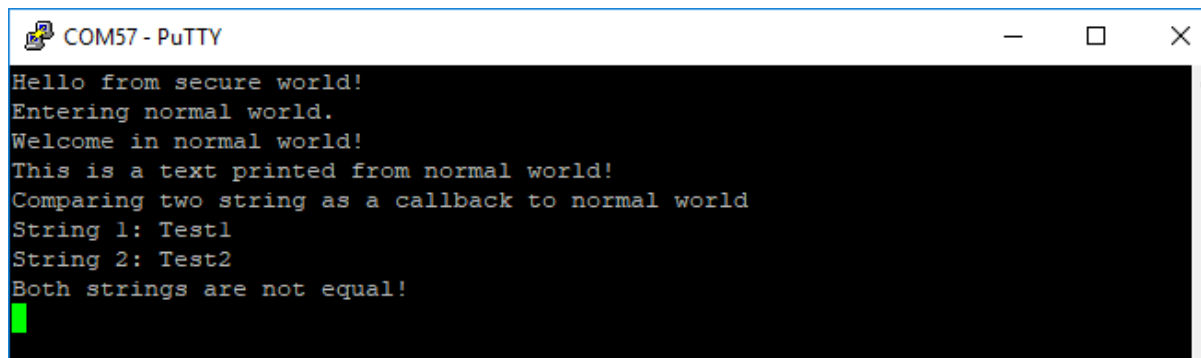
Run a TrustZone example application The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

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



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

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



Run a demo using 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 µVision, 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.

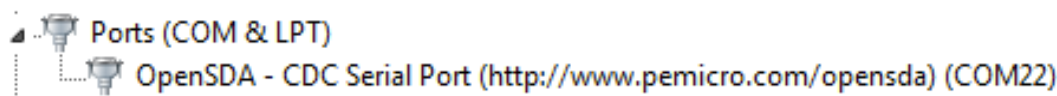
2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

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.

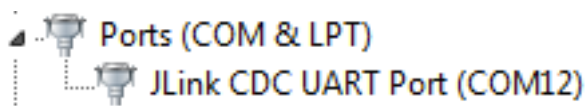
1. **CMSIS-DAP/mbed/DAPLink** interface:



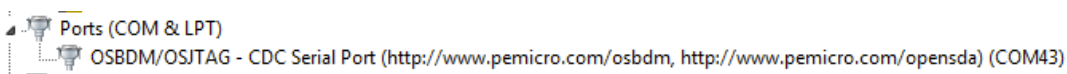
2. **P&E Micro:**



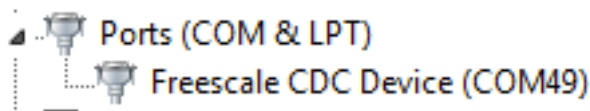
3. **J-Link:**



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.

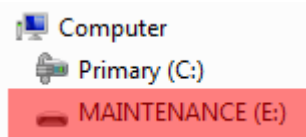
- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
- For boards with a P&E Micro interface, see [PE micro](#) to download 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-MCXA266	CMSIS-DAP	MCU-Link
FRDM-MCXA344	CMSIS-DAP	MCU-Link
FRDM-MCXA346	CMSIS-DAP	MCU-Link
FRDM-MCXA366	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
FRDM-MCXN947	CMSIS-DAP	MCU-Link
FRDM-MCXW23	CMSIS-DAP	MCU-Link

continues on next page

Table 1 – continued from previous page

Hardware platform	Default debugger firmware	On-board debugger probe
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		Professional development	ARM Toolchain	GNU	ARMGCC_DIR
	IAR EWARM			IAR Systems	IAR_DIR	
Keil MDK			ARM ecosystem	ARM Developer		MDK_DIR
ARM Compiler			Advanced optimization	ARM Developer		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

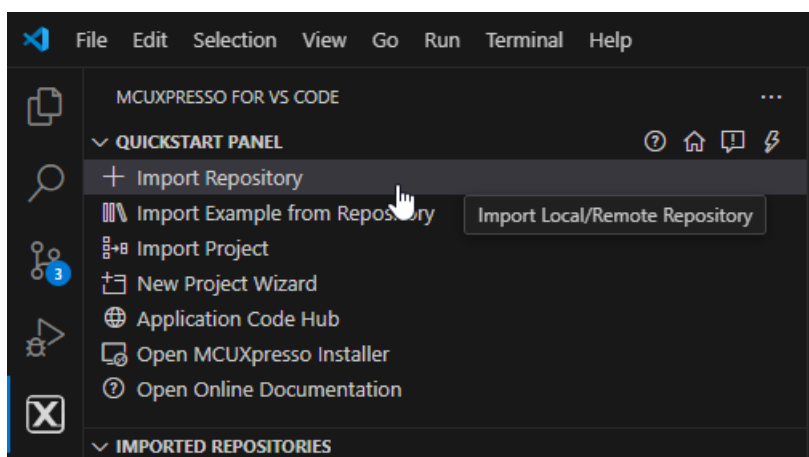
Extension Installation

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

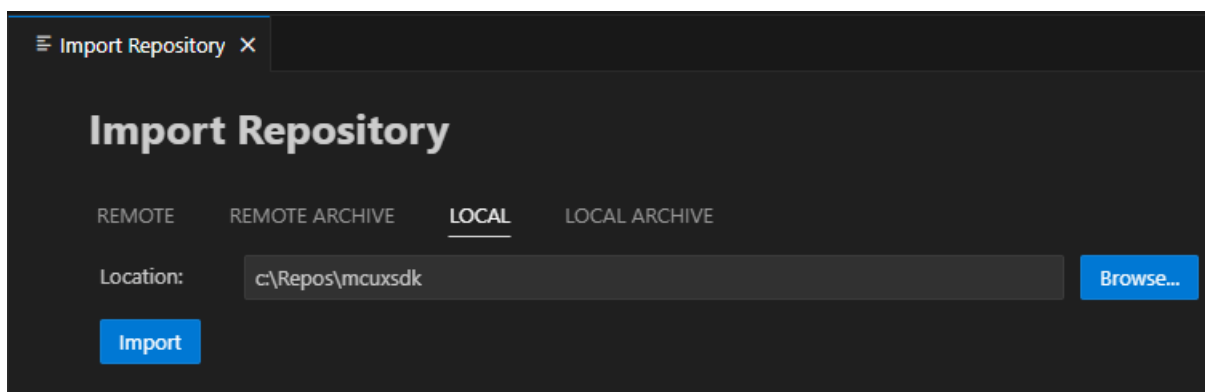
Import Methods The SDK can be imported in several ways. The MCUXpresso for VS Code extension supports both GitHub Repository SDK and Repository-Layout SDK Package distributions.

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

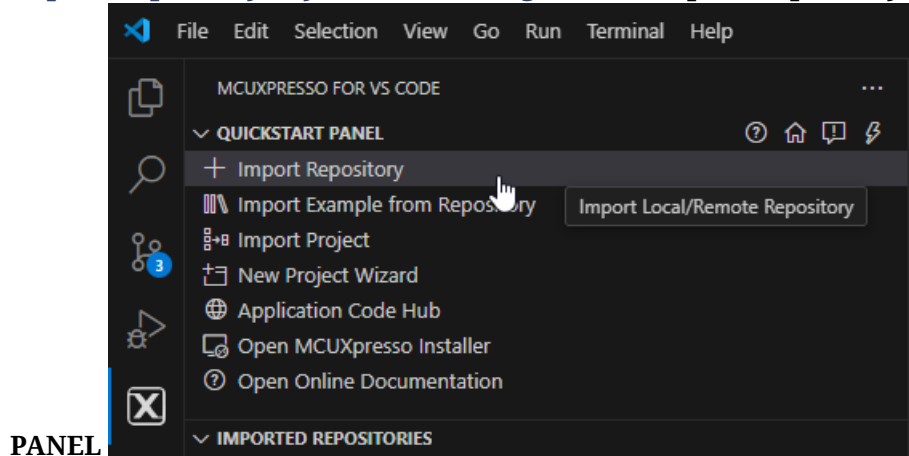


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

Select **Local** if you've already obtained the SDK according to [setting up the repo](#). Select your location and click **Import**.

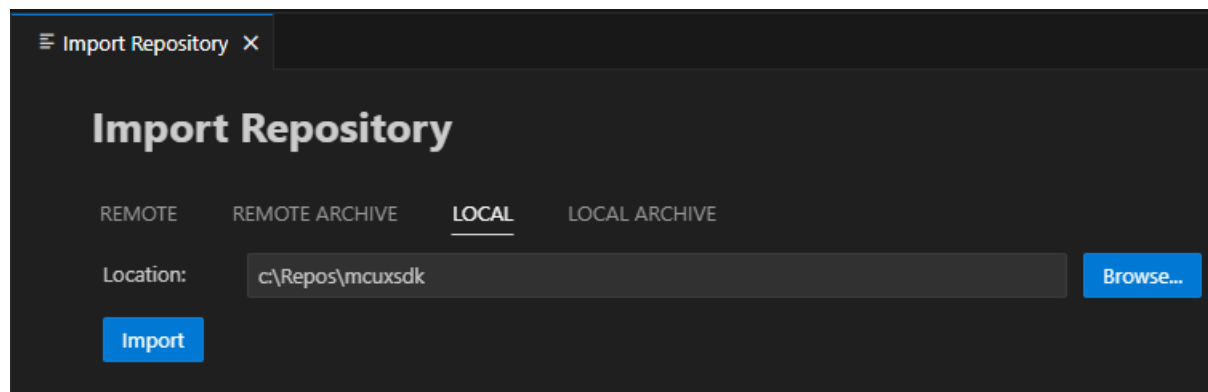


Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART**

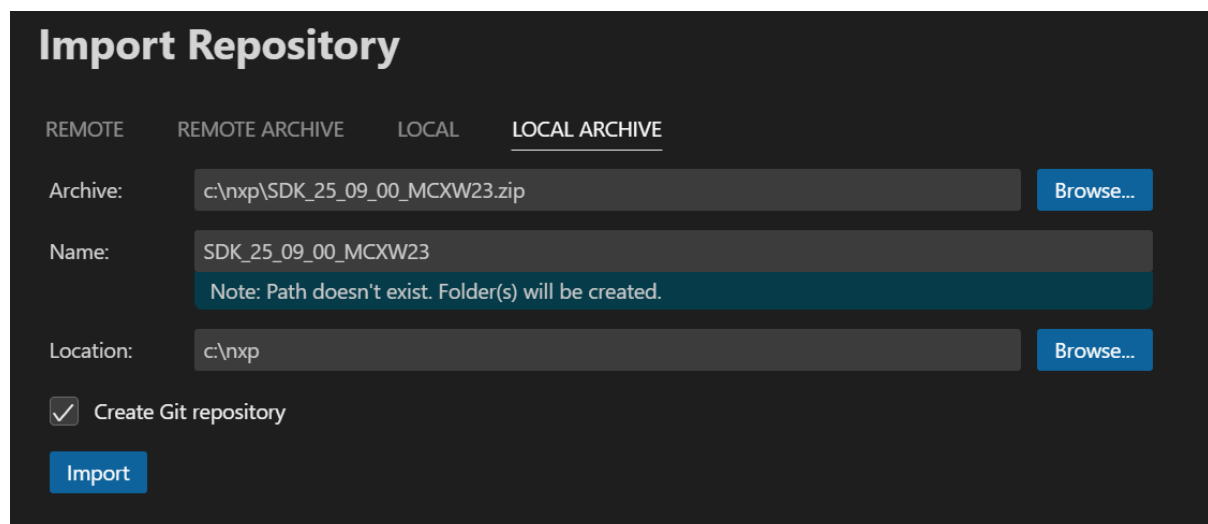


PANEL

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



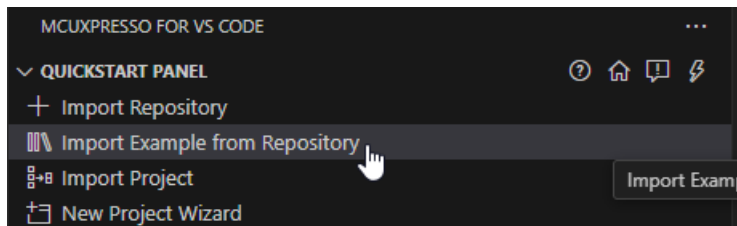
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



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**

Import Example from Repository

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

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

Board: FRDM-MCXC444 | v

 FRDM-MCXC444

Template: demo_apps/hello_world | v

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

App type: Freestanding application | v

Name: frdmxc444_hello_world

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

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

☐ Open readme file after project is imported

Import

Application Types **Repository Applications:**

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

Freestanding Applications:

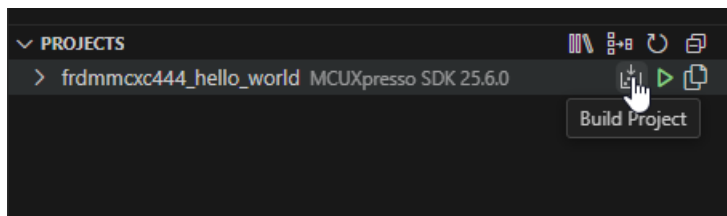
- Imported to user-defined location
- Independent of SDK location

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

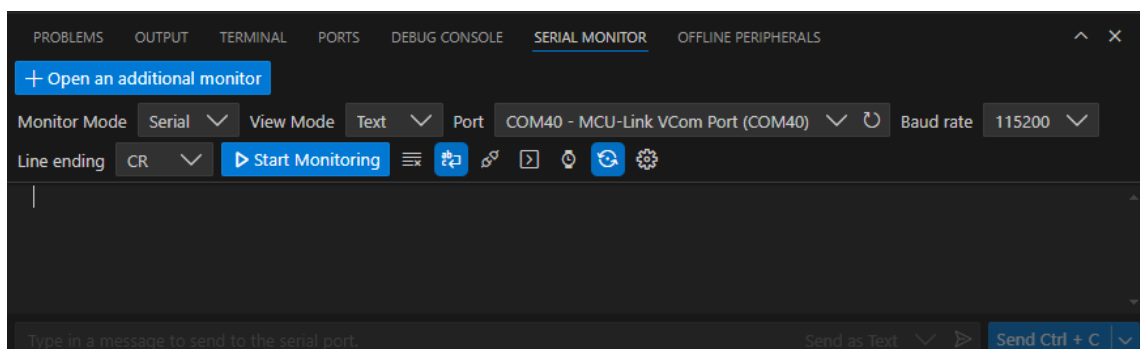


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

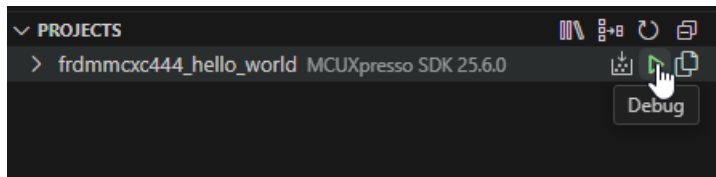
1. Open **Serial Monitor** from VS Code's integrated terminal



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

Debug Session

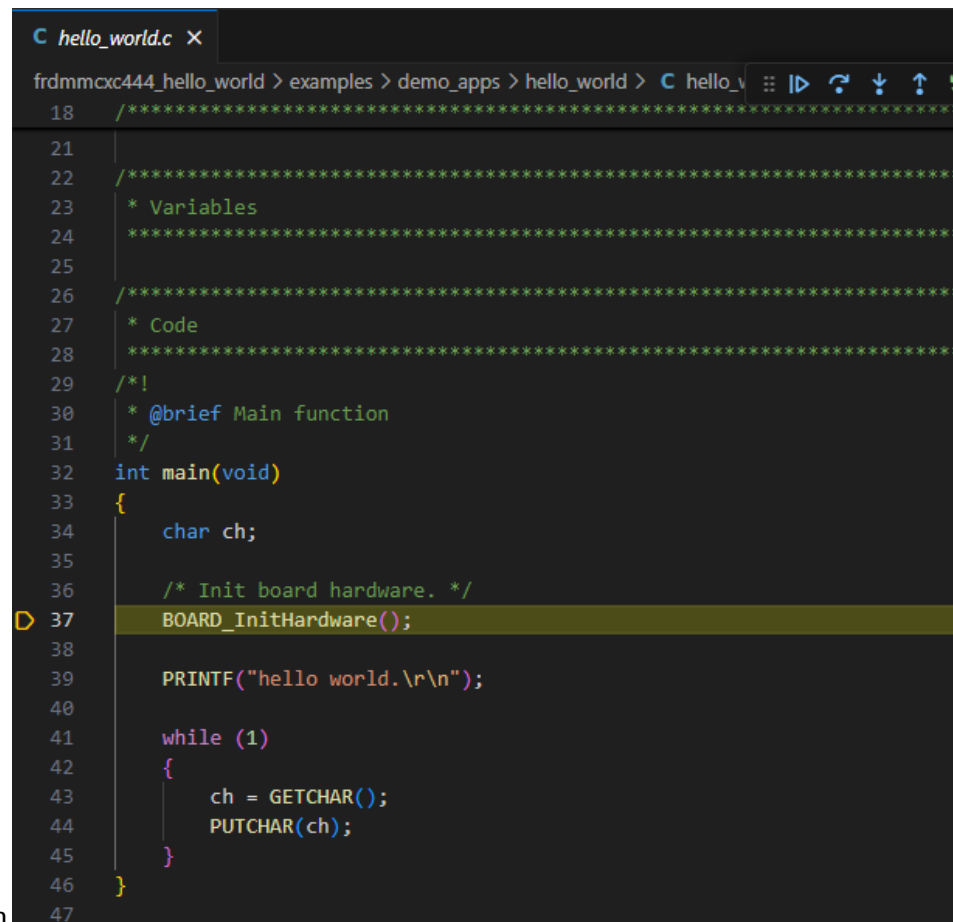
1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



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

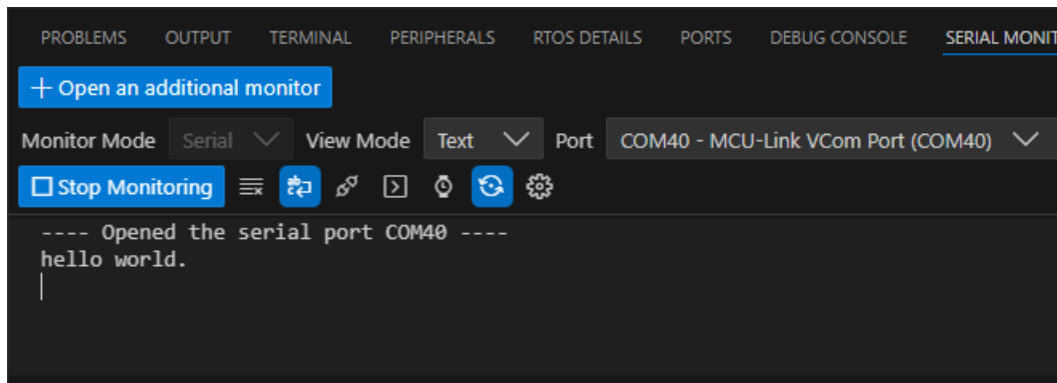
Debug Controls Use the debug controls to manage execution:

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



- **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 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
```

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:

Di- rec- tory	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 extension command `update_board` 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 frdm-mcxw23 --list-repo

# Export repository information to YAML file for reference
west update_board --set board frdm-mcxw23 --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 frdm-mcxw23 -o frdm-mcxw23-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 9.60.4
- Keil MDK, version is 5.42
- MCUXpresso for VS Code v25.09
- 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.

De- velop- ment boards	MCU devices			
FRDM- KE02Z40	MKE02Z16VFM4, MKE02Z32VLC4, MKE02Z64VFM4, MKE02Z64VQH4	MKE02Z16VLC4, MKE02Z32VLD4, MKE02Z64VLC4,	MKE02Z16VLD4, MKE02Z32VLH4, MKE02Z64VLD4,	MKE02Z32VFM4, MKE02Z32VQH4, MKE02Z64VLH4,

MCUXpresso SDK release package

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

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

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

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

Middleware

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

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

Release contents

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

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

Known issues

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

Cannot add SDK components into FreeRTOS projects

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

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version
-

ACMP

[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 ACMP_GetInstance() exceeding the array bounds.
 - Fixed the violation of MISRA C-2012 rules:
 - * Rule 3.1 8.3 10.3 17.7.

[2.0.1]

- Bug Fixes
 - Fixed the missing right pair definition for extern C.

[2.0.0]

- Initial version.
-

ADC

[2.1.0]

- Improvements
 - Added the ADC_GetDefaultFIFOConfig() API to get default setting for FIFO configuration.
- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.3.

[2.0.2]

- Bug Fixes
 - Fixed the violations of MISRA C-2012 rules:
 - * Rule 10.1 10.3 10.4 15.5 17.7.

[2.0.1]

- Bug Fixes
 - Fixed the missing right pair definition for extern C.

[2.0.0]

- Initial version.
-

CLOCK**[2.2.3]**

- Bug Fixes
 - Updated maximum value of 32K OSC from 32768 to 39063.

[2.2.2]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 18.1.

[2.2.1]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.1, rule 10.4, rule 10.8 and so on.

[2.2.0]

- New feature:
 - Moved SDK_DelayAtLeastUs function from clock driver to common driver.

[2.1.0]

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

[2.0.2]

- some minor fixes.

[2.0.0]

- initial version.
-

COMMON

[2.6.3]

- 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

[2.0.5]

- Bug fix:
 - Fix CERT-C issue with boolean-to-unsigned integer conversion.

[2.0.4]

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

[2.0.3]

- Bug fix:
 - Fix MISRA issues.

[2.0.2]

- Bug fix:
 - Fix MISRA issues.

[2.0.1]

- Bug fix:
 - DATA and DATALL macro definition moved from header file to source file.

[2.0.0]

- Initial version.
-

FLASH

[2.1.2]

- Improvements — The improved FLASH_EepromWrite function can write more data at once time.

[2.1.1]

- Bug Fixes — MISRA C-2012 issue fixed: rule 14.4

[2.1.0]

- New Features
 - add feature macro before the declaration of the EEPROM_check_range.

[2.0.0]

- Initial version.
-

FTM

[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

[2.1.1]

- Improvements:
 - Enhanced FGPIO_PinInit to enable clock internally.

[2.1.0]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 8.6.
 - Updated parameter from base into port in port_init() API.

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 3.1, 10.1, 10.3, 10.6, 10.7.

[2.0.0]

- Initial version.
-

I2C**[2.0.10]**

- Bug Fixes
 - Fixed coverity issues.

[2.0.9]

- Bug Fixes
 - Fixed the MISRA-2012 violations.
 - * Fixed rule 8.4, 10.1, 10.4, 13.5, 20.8.

[2.0.8]

- Bug Fixes
 - Fixed the bug that DFEN bit of I2C Status register 2 could not be set in I2C_MasterInit.
 - MISRA C-2012 issue fixed: rule 14.2, 15.7, and 16.4.
 - Eliminated IAR Pa082 warnings from I2C_MasterTransferDMA and I2C_MasterTransferCallbackDMA by assigning volatile variables to local variables and using local variables instead.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 11.9, 14.4, 15.7, 17.7.
- Improvements
 - Improved timeout mechanism when waiting certain state in transfer API.
 - Updated the I2C_WAIT_TIMEOUT macro to unified name I2C_RETRY_TIMES.
 - Moved the master manually acknowledge byte operation into static function I2C_MasterAckByte.
 - Fixed control/status clean flow issue inside I2C_MasterReadBlocking to avoid potential issue that pending status is cleaned before it's proceeded.

[2.0.7]

- Bug Fixes
 - Fixed the issue for MISRA-2012 check.
 - * Fixed rule 11.9 ,15.7 ,14.4 ,10.4 ,10.8 ,10.3, 10.1, 10.6, 13.5, 11.3, 13.2, 17.7, 5.7, 8.3, 8.5, 11.1, 16.1.

- Fixed Coverity issue of unchecked return value in I2C_RTOS_Transfer.
- Fixed variable redefine issue by moving i2cBases from fsl_i2c.h to fsl_i2c.c.
- Improvements
 - Added I2C_MASTER_FACK_CONTROL macro to enable FACK control for master transfer receive flow with IP supporting double buffer, then master could hold the SCL by manually setting TX AK/NAK during data transfer.

[2.0.6]

- Bug Fixes
 - Fixed the issue that I2C Master transfer APIs(blocking/non-blocking) did not support the situation of master transfer with subaddress and transfer data size being zero, which means no data followed by the subaddress.

[2.0.5]

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

[2.0.4]

- Bug Fixes
 - Added a proper handle for transfer config flag kI2C_TransferNoStartFlag to support transmit with kI2C_TransferNoStartFlag flag. Support write only or write+read with no start flag; does not support read only with no start flag.

[2.0.3]

- Bug Fixes
 - Removed enableHighDrive member in the master/slave configuration structure because the operation to HDRS bit is useless, the user need to use DSE bit in port register to configure the high drive capability.
 - Added register reset operation in I2C_MasterInit and I2C_SlaveInit APIs. Fixed issue where I2C could not switch between master and slave mode.
 - Improved slave IRQ handler to handle the corner case that stop flag and address match flag come synchronously.

[2.0.2]

- Bug Fixes
 - Fixed issue in master receive and slave transmit mode with no stop flag. The master could not succeed to start next transfer because the master could not send out re-start signal.
 - Fixed the out-of-order issue of data transfer due to memory barrier.
 - Added hold time configuration for slave. By leaving the SCL divider and MULT reset values when configured to slave mode, the setup and hold time of the slave is then reduced outside of spec for lower baudrates. This can cause intermittent arbitration loss on the master side.

- New Features
 - Added address nak event for master.
 - Added general call event for slave.

[2.0.1]

- New Features
 - Added double buffer enable configuration for SoCs which have the DFEN bit in S2 register.
 - Added flexible transmit/receive buffer size support in I2C_SlaveHandleIRQ.
 - Added start flag clear, address match, and release bus operation in I2C_SlaveWrite/ReadBlocking API.
- Bug Fixes
 - Changed the kI2C_SlaveRepeatedStartEvent to kI2C_SlaveStartEvent.

[2.0.0]

- Initial version.
-

IRQ

[2.0.2]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 3.1, 8.4, 10.3 and 10.6.

[2.0.1]

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

[2.0.0]

- Initial version.
-

KBI

[2.0.3]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rules 10.8.

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 17.7.

[2.0.0]

- Initial version.
-

MCM

[2.2.0]

- Improvements
 - Support platforms with less features.

[2.1.0]

- Others
 - Remove byteID from `mcm_lmem_fault_attribute_t` for document update.

[2.0.0]

- Initial version.
-

PIT

[2.2.0]

- Bug Fixes
 - According to ERR050763, `PIT_LDVAL_STAT` register is not reliable in dynamic load mode, so remove the status check in `PIT_SetRtiTimerPeriod` which added since 2.1.1.
 - Removed not used bit `PIT_RTI_TCTRL_CHN_MASK`.
- Improvements
 - Added more guide about get RTI load status in `PIT_SetRtiTimerPeriod`'s API comment.
 - Change `PIT_RTI_Deinit` to inline API.
 - Ensure PIT peripheral clock enabled in `PIT_RTI_Init`.
- New Features
 - Added `PIT_ClearRtiSyncStatus` API to clear the `RTI_LDVAL_STAT` register.

[2.1.1]

- Bug Fixes
 - Enable PIT when using RTI to ensure RTI can work properly in debug mode.
- Improvements
 - Added status check in PIT_SetRtiTimerPeriod to ensure the load value is synchronized into the RTI clock domain.
 - Added note for PIT_RTI_Init to remind users wait RTI sync.

[2.1.0]

- New Features
 - Support RTI (Real Time Interrupt) timer.

[2.0.5]

- Improvements
 - Support workaround for ERR007914. This workaround guarantee the write to MCR register is not ignored.

[2.0.4]

- Bug Fixes
 - Fixed PIT_SetTimerPeriod implementation, the load value trigger should be PIT clock cycles minus 1.

[2.0.3]

- Bug Fixes
 - Clear all status bits for all channels to make sure the status of all TCTRL registers is clean.

[2.0.2]

- Bug Fixes
 - Fixed MISRA-2012 issues.
 - * Rule 10.1.

[2.0.1]

- Bug Fixes
 - Cleared timer enable bit for all channels in function PIT_Init() to make sure all channels stay in disable status before setting other configurations.
 - Fixed MISRA-2012 rules.
 - * Rule 14.4, rule 10.4.

[2.0.0]

- Initial version.
-

PORT

[2.0.2]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 10.1, rule 10.3, rule 10.4, rule 10.7, rule 14.4.

[2.0.1]

- Change pin index enum `port_pin_index_t` to `uint8_t` in `PORT_SetPinPullUpEnable()`;

[2.0.0]

- initial version;
-

RTC

[2.0.6]

- Bug Fixes
 - Fix `RTC_GetDatetime` function validating datetime issue.

[2.0.5]

- Bug Fixes
 - Fixed CERT INT30-C, INT31-C violations.

[2.0.4]

- Improvements
 - Changed the behavior of calling alarm callback when alarm seconds reach counter seconds, instead of previous behavior when counter seconds reach alarm seconds and counter seconds increments.

[2.0.3]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.1, 10.3, 10.4 and 14.4.

[2.0.2]

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

[2.0.1]

- Bug Fixes
 - Fixed the issue of Pa082 warning.

[2.0.0]

- Initial version.
-

SPI**[2.1.4]**

- Bug Fixes
 - Fixed coverity issues.

[2.1.3]

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

[2.1.2]

- Improvements
 - Changed SPI_DUMMYDATA to 0x00.

[2.1.1]

- Bug Fixes
 - Fixed MISRA 10.3 violation.

[2.1.0]

- Improvements
 - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
 - Fixed the bug that, when working as a slave, instance that does not have FIFO may miss some rx data.
 - Fixed master RX data overflow issue by synchronizing transmit and receive process.
 - Fixed issue that slave should not share the same non-blocking initialization API and IRQ handler with master to prevent dead lock issue.
 - Fixed issue that callback should be invoked after all data is sent out to bus.
 - Added code in SPI_SlaveTransferNonBlocking to empty rx buffer before initializing transfer.

[2.0.5]

- Bug Fixes
 - Eliminated Pa082 warnings from SPI_WriteNonBlocking and SPI_GetStatusFlags.
 - Fixed MISRA issues.
 - * Fixed issues 10.1, 10.3, 10.4, 10.7, 10.8, 11.9, 14.4, 17.7.

[2.0.4]

- New Features
 - Supported 3-wire mode for SPI driver. Added new API SPI_SetPinMode() to control the transfer direction of the single wire. For master instance, MOSI is selected as I/O pin. For slave instance, MISO is selected as I/O pin.
 - Added dummy data setup API to allow users to configure the dummy data to be transferred.

[2.0.3]

- Bug Fixes
 - Fixed the potential interrupt race condition at high baudrate when calling API SPI_MasterTransferNonBlocking.

[2.0.2]

- New Features
 - Allowed users to set the transfer size for SPI_TransferNoBlocking non-integer times of watermark.
 - Allowed users to define the dummy data. Users only need to define the macro SPI_DUMMYDATA in applications.

[2.0.1]

- Bug Fixes
 - Fixed SPI_Enable function parameter error.
 - Set the s_dummy variable as static variable in fsl_spi_dma.c.
- Improvements
 - Optimized the code size while not using transactional API.
 - Improved performance in polling method.
 - Added #ifndef/#endif to allow users to change the default tx value at compile time.

[2.0.0]

- Initial version.
-

TPM

[2.4.1]

- Improvements
 - Add Coverage Justification for uncovered code.

[2.4.0]

- New Feature
 - Added while loop timeout for MOD CnV CnSC and SC register write sequence.
 - Change the return type from void to status_t for following API:
 - * TPM_DisableChannel
 - * TPM_EnableChannel
 - * TPM_SetupOutputCompare
 - * TPM_SetTimerPeriod
 - * TPM_StopTimer

[2.3.6]

- Bug Fixes
 - Fixed CERT INT30-C INT31-C issue for TPM_SetupDualEdgeCapture.

[2.3.5]

- New Feature
 - Added IRQ handler entry for TPM2.

[2.3.4]

- New Feature
 - Added common IRQ handler entry TPM_DriverIRQHandler.

[2.3.3]

- Improvements
 - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.3.2]

- Bug Fixes
 - Fixed ERR008085 TPM writing the TPMx_MOD or TPMx_CnV registers more than once may fail when the timer is disabled.

[2.3.1]

- Bug Fixes
 - Fixed compilation error when macro `FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL` is 1.

[2.3.0]

- Improvements
 - Create callback feature for TPM match and timer overflow interrupts.

[2.2.4]

- Improvements
 - Add feature macros(`FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_EN`, `FSL_FEATURE_TPM_HAS_GLOBAL_TIME_BASE_SYNC`).

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Bug Fixes
 - Fixed CCM issue by splitting function from `TPM_SetupPwm()` function to reduce function complexity.
 - Fixed violations of MISRA C-2012 rule 17.7.

[2.2.0]

- Improvements
 - Added `TPM_SetChannelPolarity` to support select channel input/output polarity.
 - Added `TPM_EnableChannelExtTrigger` to support enable external trigger input to be used by channel.
 - Added `TPM_CalculateCounterClkDiv` to help calculates the counter clock prescaler.
 - Added `TPM_GetChannelValue` to support get TPM channel value.
 - Added new TPM configuration.
 - * `syncGlobalTimeBase`
 - * `extTriggerPolarity`
 - * `chnlPolarity`
 - Added new PWM signal configuration.

- * secPauseLevel

- Bug Fixes
 - Fixed TPM_SetupPwm can't configure 0% combined PWM issues.

[2.1.1]

- Improvements
 - Add feature macro for PWM pause level select feature.

[2.1.0]

- Improvements
 - Added TPM_EnableChannel and TPM_DisableChannel APIs.
 - Added new PWM signal configuration.
 - * pauseLevel - Support select output level when counter first enabled or paused.
 - * enableComplementary - Support enable/disable generate complementary PWM signal.
 - * deadTimeValue - Support deadtime insertion for each pair of channels in combined PWM mode.
- Bug Fixes
 - Fixed issues about channel MSnB:MSnA and ELSnB:ELSnA bit fields and CnV register change request acknowledgement. Writes to these bits are ignored when the interval between successive writes is less than the TPM clock period.

[2.0.8]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.4, 10.7 and 14.4.

[2.0.7]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.4 and 17.7.

[2.0.6]

- Bug Fixes
 - Fixed Out-of-bounds issue.

[2.0.5]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 10.6, 10.7

[2.0.4]

- Bug Fixes
 - Fixed ERR050050 in functions TPM_SetupPwm/TPM_UpdatePwmDutycycle. When TPM was configured in EPWM mode as PS = 0, the compare event was missed on the first reload/overflow after writing 1 to the CnV register.

[2.0.3]

- Bug Fixes
 - MISRA-2012 issue fixed.
 - * Fixed rules: rule-12.1, rule-17.7, rule-16.3, rule-14.4, rule-1.3, rule-10.4, rule-10.3, rule-10.7, rule-10.1, rule-10.6, and rule-18.1.

[2.0.2]

- Bug Fixes
 - Fixed issues in functions TPM_SetupPwm/TPM_UpdateChnEdgeLevelSelect/TPM_SetupInputCapture/TPM_SetupOutputCompare/TPM_SetupDualEdgeCapture, wait acknowledgement when the channel is disabled.

[2.0.1]

- Bug Fixes
 - Fixed TPM_UpdateChnIEdgeLevelSelect ACK wait issue.
 - Fixed the issue that TPM_SetupdualEdgeCapture could not set FILTER register.
 - Fixed TPM_UpdateChnEdgeLevelSelect ACK wait issue.

[2.0.0]

- Initial version.
-

UART

[2.5.1]

- Improvements
 - Use separate data for TX and RX in uart_transfer_t.
- Bug Fixes
 - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling UART_TransferReceiveNonBlocking, the received data count returned by UART_TransferGetReceiveCount is wrong.

[2.5.0]

- New Features
 - Added APIs UART_GetRxFifoCount/UART_GetTxFifoCount to get rx/tx FIFO data count.
 - Added APIs UART_SetRxFifoWatermark/UART_SetTxFifoWatermark to set rx/tx FIFO water mark.
- Bug Fixes
 - Fixed bug of race condition during UART transfer using transactional APIs, by disabling and re-enabling the global interrupt before and after critical operations on interrupt enable registers.
 - Fixed DMA/eDMA transfer blocking issue by enabling tx idle interrupt after DMA/eDMA transmission finishes.

[2.4.0]

- New Features
 - Added APIs to configure 9-bit data mode, set slave address and send address.

[2.3.0]

- Bug Fixes
 - Fixed the bug that, when framing/parity/noise/overflow flag or idle line detect flag is set, receive FIFO should be flushed to avoid FIFO pointer being in unknown state, since FIFO has no valid data.
- Improvements
 - Modified UART_TransferHandleIRQ so that txState will be set to idle only when all data has been sent out to bus.
 - Modified UART_TransferGetSendCount so that this API returns the real byte count that UART has sent out rather than the software buffer status.
 - Added timeout mechanism when waiting for certain states in transfer driver.

[2.2.0]

- New Features
 - Added UART hardware FIFO enable/disable API.
- Improvements
 - Added check for kUART_TransmissionCompleteFlag in UART_TransferHandleIRQ, UART_SentEDMACallback and UART_TransferSendDMACallback to ensure all the data would be sent out to bus.
- Bug Fixes
 - Eliminated IAR Pa082 warnings from UART_TransferGetRxRingBufferLength, UART_GetEnabledInterrupts, UART_GetStatusFlags and UART_TransferHandleIRQ.
 - Added code in UART_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.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 14.4, 11.6, 17.7.

[2.1.6]

- Bug Fixes
 - Fixed the issue of register's being in repeatedly reading status while performing the IRQ routine.

[2.1.5]

- Improvements
 - Added hardware flow control function support.
 - Added idle-line-detecting feature in UART_TransferNonBlocking function. If an idle line is detected, a callback will be triggered with status `kStatus_UART_IdleLineDetected` returned. This feature may be useful when the number of received bytes is less than the expected receive data size. Before triggering the callback, data in the FIFO is read out (if it has FIFO), and no interrupt will be disabled except for the case that the receive data size reaches 0.
 - Enabled the RX FIFO watermark function. With the idle-line-detecting feature enabled, you can set the watermark value to whatever you want (should not be bigger than the RX FIFO size). Data is then received and a callback will be triggered when data receive ends.

[2.1.4]

- Improvements
 - Changed parameter type in `UART_RTOS_Init()` struct `rtos_uart_config` → `uart_rtos_config_t`.
- Bug Fixes
 - Disabled UART receive interrupt instead of global interrupt when reading data from ring buffer. With ring buffer used, receive nonblocking will disable global interrupt to protect the ring buffer. This has a negative effect on other IPs using interrupt.

[2.1.3]

- New Features
 - Added RX framing error and parity error status check when using interrupt transfer.

[2.1.2]

- Bug Fixes
 - Fixed baud rate fine adjust bug to make the computed baud rate more accurate.

[2.1.1]

- Bug Fixes
 - Removed needless check of event flags and assert in `UART_RTOS_Receive`.
 - Always waited for RX event flag in `UART_RTOS_Receive`.

[2.1.0]

- Improvements
 - Added transactional API.

[2.0.0]

- Initial version.
-

WDOG8**[2.0.1]**

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7 and 11.9.
 - Fixed the issue of the inseparable process interrupted by other interrupt source.
 - * WDOG8_Refresh

[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.

[MKE02Z4](#)

1.7 Middleware Documentation

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

1.7.1 FreeMASTER

[freemaster](#)

Chapter 2

MKE02Z4

2.1 ACMP: Analog Comparator Driver

`void ACMP_Init(ACMP_Type *base, const acmp_config_t *config)`

Initialize the ACMP.

The default configuration can be got by calling `ACMP_GetDefaultConfig()`.

Parameters

- `base` – ACMP peripheral base address.
- `config` – Pointer to ACMP configuration structure.

`void ACMP_Deinit(ACMP_Type *base)`

De-Initialize the ACMP.

Parameters

- `base` – ACMP peripheral basic address.

`void ACMP_GetDefaultConfig(acmp_config_t *config)`

Gets the default configuration for ACMP.

This function initializes the user configuration structure to default value. The default value are: Example:

```
config->enablePinOut = false;  
config->hysteresisMode = kACMP_HysterisisLevel1;
```

Parameters

- `config` – Pointer to ACMP configuration structure.

`static inline void ACMP_Enable(ACMP_Type *base, bool enable)`

Enable/Disable the ACMP module.

Parameters

- `base` – ACMP peripheral base address.
- `enable` – Switcher to enable/disable ACMP module.

`void ACMP_EnableInterrupt(ACMP_Type *base, acmp_interrupt_mode_t mode)`

Enable the ACMP interrupt and determines the sensitivity modes of the interrupt trigger.

Parameters

- `base` – ACMP peripheral base address.

- `mode` – Select one interrupt mode to generate interrupt.

static inline void ACMP_DisableInterrupt(ACMP_Type *base)

Disable the ACMP interrupt.

Parameters

- `base` – ACMP peripheral base address.

void ACMP_SetChannelConfig(ACMP_Type *base, *acmp_input_channel_selection_t* PositiveInput, *acmp_input_channel_selection_t* negativeInout)

Configure the ACMP positive and negative input channel.

Parameters

- `base` – ACMP peripheral base address.
- `PositiveInput` – ACMP Positive Input Select. Refer to “*acmp_input_channel_selection_t*”.
- `negativeInout` – ACMP Negative Input Select. Refer to “*acmp_input_channel_selection_t*”.

void ACMP_SetDACConfig(ACMP_Type *base, const *acmp_dac_config_t* *config)

void ACMP_EnableInputPin(ACMP_Type *base, uint32_t mask, bool enable)

Enable/Disable ACMP input pin. The API controls if the corresponding ACMP external pin can be driven by an analog input.

Parameters

- `base` – ACMP peripheral base address.
- `mask` – The mask of the pin associated with channel ADx. Valid range is AD0:0x1U ~ AD3:0x4U. For example: If enable AD0, AD1 and AD2 pins, mask should be set to 0x7U(0x1 | 0x2 | 0x4).
- `enable` – Switcher to enable/disable ACMP module.

static inline uint8_t ACMP_GetStatusFlags(ACMP_Type *base)

Get ACMP status flags.

Parameters

- `base` – ACMP peripheral base address.

Returns

Flags' mask if indicated flags are asserted. See “*_acmp_status_flags*”.

static inline void ACMP_ClearInterruptFlags(ACMP_Type *base)

Clear interrupts status flag.

Parameters

- `base` – ACMP peripheral base address.

FSL_ACMP_DRIVER_VERSION

ACMP driver version 2.0.2.

enum *_acmp_hysterisis_mode*

Analog Comparator Hysterisis Selection.

Values:

enumerator *kACMP_HysterisisLevel1*

ACMP hysterisis is 20mv. >

enumerator kACMP_HysterisisLevel2

ACMP hysteresis is 30mv. >

enum _acmp_reference_voltage_source

DAC Voltage Reference source.

Values:

enumerator kACMP_VrefSourceVin1

The DAC selects Bandgap as the reference.

enumerator kACMP_VrefSourceVin2

The DAC selects VDDA as the reference.

enum _acmp_interrupt_mode

The sensitivity modes of the interrupt trigger.

Values:

enumerator kACMP_OutputFallingInterruptMode

ACMP interrupt on output falling edge. >

enumerator kACMP_OutputRisingInterruptMode

ACMP interrupt on output rising edge. >

enumerator kACMP_OutputBothEdgeInterruptMode

ACMP interrupt on output falling or rising edge. >

enum _acmp_input_channel_selection

The ACMP input channel selection.

Values:

enumerator kACMP_ExternalReference0

External reference 0 is selected to as input channel. >

enumerator kACMP_ExternalReference1

External reference 1 is selected to as input channel. >

enumerator kACMP_ExternalReference2

External reference 2 is selected to as input channel. >

enumerator kACMP_InternalDACOutput

Internal DAC putput is selected to as input channel. >

enum _acmp_status_flags

The ACMP status flags.

Values:

enumerator kACMP_InterruptFlag

ACMP interrupt on output valid edge. >

enumerator kACMP_OutputFlag

The current value of the analog comparator output. >

typedef enum _acmp_hysterisis_mode acmp_hysterisis_mode_t

Analog Comparator Hysterisis Selection.

typedef enum _acmp_reference_voltage_source acmp_reference_voltage_source_t

DAC Voltage Reference source.

typedef enum _acmp_interrupt_mode acmp_interrupt_mode_t

The sensitivity modes of the interrupt trigger.

```
typedef enum _acmp_input_channel_selection acmp_input_channel_selection_t
```

The ACMP input channel selection.

```
typedef struct _acmp_config acmp_config_t
```

Configuration for ACMP.

```
typedef struct _acmp_dac_config acmp_dac_config_t
```

Configuration for Internal DAC.

```
struct _acmp_config
```

#include <fsl_acmp.h> Configuration for ACMP.

Public Members

```
bool enablePinOut
```

The comparator output is available on the associated pin.

```
acmp_hysteresis_mode_t hysteresisMode
```

Hysteresis mode.

```
struct _acmp_dac_config
```

#include <fsl_acmp.h> Configuration for Internal DAC.

Public Members

```
uint8_t DACValue
```

Value for DAC Output Voltage. Available range is 0-63.

```
acmp_reference_voltage_source_t referenceVoltageSource
```

Supply voltage reference source.

2.2 ADC: 12-bit Analog to Digital Converter Driver

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

Initializes the ADC module.

Parameters

- *base* – ADC peripheral base address.
- *config* – Pointer to configuration structure. See “*adc_config_t*”.

```
void ADC_Deinit(ADC_Type *base)
```

De-initialize the ADC module.

Parameters

- *base* – ADC peripheral base address.

```
void ADC_GetDefaultConfig(adc_config_t *config)
```

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

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

```

config->referenceVoltageSource = kADC_ReferenceVoltageSourceAlt0;
config->enableLowPower = false;
config->enableLongSampleTime = false;
config->clockDivider = kADC_ClockDivider1;
config->ResolutionMode = kADC_Resolution8BitMode;
config->clockSource = kADC_ClockSourceAlt0;

```

Parameters

- config – Pointer to the configuration structure.

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

Enable the hardware trigger mode.

Parameters

- base – ADC peripheral base address.
- enable – Switcher of the hardware trigger feature. “true” means enabled, “false” means not enabled.

void ADC_SetHardwareCompare(ADC_Type *base, const *adc_hardware_compare_config_t* *config)

Configure the hardware compare mode.

The compare function can be configured to check for an upper or lower limit. After the input is sampled and converted, the result is added to the complement of the compare value (ADC_CV).

Parameters

- base – ADC peripheral base address.
- config – Pointer to “*adc_hardware_compare_config_t*” structure.

void ADC_SetFifoConfig(ADC_Type *base, const *adc_fifo_config_t* *config)

Configure the Fifo mode.

The ADC module supports FIFO operation to minimize the interrupts to CPU in order to reduce CPU loading in ADC interrupt service routines. This module contains two FIFOs to buffer analog input channels and analog results respectively.

Parameters

- base – ADC peripheral base address.
- config – Pointer to “*adc_fifo_config_t*” structure.

void ADC_GetDefaultFIFOConfig(*adc_fifo_config_t* *config)

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

Parameters

- config – Pointer to the FIFO configuration structure, please refer to *adc_fifo_config_t* for details.

void ADC_SetChannelConfig(ADC_Type *base, const *adc_channel_config_t* *config)

Configures the conversion channel.

This operation triggers the conversion when in software trigger mode. When in hardware trigger mode, this API configures the channel while the external trigger source helps to trigger the conversion.

Parameters

- base – ADC peripheral base address.
- config – Pointer to “*adc_channel_config_t*” structure.

bool ADC_GetChannelStatusFlags(ADC_Type *base)

Get the status flags of channel.

Parameters

- base – ADC peripheral base address.

Returns

“True” means conversion has completed and “false” means conversion has not completed.

uint32_t ADC_GetStatusFlags(ADC_Type *base)

Get the ADC status flags.

Parameters

- base – ADC peripheral base address.

Returns

Flags’ mask if indicated flags are asserted. See “_adc_status_flags”.

static inline void ADC_EnableAnalogInput(ADC_Type *base, uint32_t mask, bool enable)

Disables the I/O port control of the pins used as analog inputs.

When a pin control register bit is set, the following conditions are forced for the associated MCU pin: -The output buffer is forced to its high impedance state. -The input buffer is disabled. A read of the I/O port returns a zero for any pin with its input buffer disabled. -The pullup is disabled.

Parameters

- base – ADC peripheral base address.
- mask – The mask of the pin associated with channel ADx. Valid range is AD0:0x1U ~ AD15:0x8000U. For example: If enable AD0, AD1 and AD2 pins, mask should be set to 0x7U.
- enable – The “true” means enabled, “false” means not enabled.

static inline uint32_t ADC_GetChannelConversionValue(ADC_Type *base)

Gets the conversion value.

Parameters

- base – ADC peripheral base address.

Returns

Conversion value.

static inline void ADC_SetHardwareTriggerMaskMode(ADC_Type *base,
adc_hardware_trigger_mask_mode_t
mode)

enum _adc_reference_voltage_source

Reference voltage source.

Values:

enumerator kADC_ReferenceVoltageSourceAlt0

Default voltage reference pin pair (VREFH/VREFL). >

enumerator kADC_ReferenceVoltageSourceAlt1

Analog supply pin pair (VDDA/VSSA). >

enum _adc_clock_divider

Clock divider for the converter.

Values:

enumerator kADC_ClockDivider1

Divide ration = 1, and clock rate = Input clock. >

enumerator kADC_ClockDivider2

Divide ration = 2, and clock rate = Input clock / 2. >

enumerator kADC_ClockDivider4

Divide ration = 3, and clock rate = Input clock / 4. >

enumerator kADC_ClockDivider8

Divide ration = 4, and clock rate = Input clock / 8. >

enum _adc_resolution_mode

ADC converter resolution mode.

Values:

enumerator kADC_Resolution8BitMode

8-bit conversion (N = 8). >

enumerator kADC_Resolution10BitMode

10-bit conversion (N = 10) >

enumerator kADC_Resolution12BitMode

12-bit conversion (N = 12) >

enum _adc_clock_source

ADC input Clock source.

Values:

enumerator kADC_ClockSourceAlt0

Bus clock. >

enumerator kADC_ClockSourceAlt1

Bus clock divided by 2. >

enumerator kADC_ClockSourceAlt2

Alternate clock (ALTCLK). >

enumerator kADC_ClockSourceAlt3

Asynchronous clock (ADACK). >

enum _adc_compare_mode

Compare function mode.

Values:

enumerator kADC_CompareDisableMode

Compare function disabled. >

enumerator kADC_CompareLessMode

Compare triggers when input is less than compare level. >

enumerator kADC_CompareGreaterOrEqualMode

Compare triggers when input is greater than or equal to compare level. >

enum _adc_status_flags

ADC status flags mask.

Values:

enumerator kADC_ActiveFlag

Indicates that a conversion is in progress. >

enumerator kADC_FifoEmptyFlag

Indicates that ADC result FIFO have no valid new data. >

enumerator kADC_FifoFullFlag

Indicates that ADC result FIFO is full. >

enum _adc_hardware_trigger_mask_mode

Hardware trigger mask mode.

Values:

enumerator kADC_HWTriggerMaskDisableMode

Hardware trigger mask disable and hardware trigger can trigger ADC conversion. >

enumerator kADC_HWTriggerMaskAutoMode

Hardware trigger mask automatically when data fifo is not empty. >

enumerator kADC_HWTriggerMaskEnableMode

Hardware trigger mask enable and hardware trigger cannot trigger ADC conversion.
>

typedef enum _adc_reference_voltage_source adc_reference_voltage_source_t

Reference voltage source.

typedef enum _adc_clock_divider adc_clock_divider_t

Clock divider for the converter.

typedef enum _adc_resolution_mode adc_resolution_mode_t

ADC converter resolution mode.

typedef enum _adc_clock_source adc_clock_source_t

ADC input Clock source.

typedef enum _adc_compare_mode adc_compare_mode_t

Compare function mode.

typedef enum _adc_hardware_trigger_mask_mode adc_hardware_trigger_mask_mode_t

Hardware trigger mask mode.

typedef struct _adc_config adc_config_t

ADC converter configuration.

typedef struct _adc_hardware_compare_config adc_hardware_compare_config_t

ADC hardware comparison configuration.

typedef struct _adc_fifo_config adc_fifo_config_t

ADC FIFO configuration.

typedef struct _adc_channel_config adc_channel_config_t

ADC channel conversion configuration.

FSL_ADC_DRIVER_VERSION

ADC driver version.

Version 2.1.0.

struct _adc_config

#include <fsl_adc.h> ADC converter configuration.

Public Members

adc_reference_voltage_source_t referenceVoltageSource

Selects the voltage reference source used for conversions. >

bool enableLowPower

Enable low power mode. The power is reduced at the expense of maximum clock speed. >

bool enableLongSampleTime

Enable long sample time mode. >

adc_clock_divider_t clockDivider

Select the divider of input clock source. >

adc_resolution_mode_t ResolutionMode

Select the sample resolution mode. >

adc_clock_source_t clockSource

Select the input Clock source. >

struct *_adc_hardware_compare_config*

#include <fsl_adc.h> ADC hardware comparison configuration.

Public Members

uint32_t compareValue

Setting the compare value. The value are compared to the conversion result. >

adc_compare_mode_t compareMode

Setting the compare mode. Refer to “*adc_compare_mode_t*”. >

struct *_adc_fifo_config*

#include <fsl_adc.h> ADC FIFO configuration.

Public Members

bool enableHWTriggerMultConv

The field is valid when FIFO is enabled. Enable hardware trigger multiple conversion. One hardware trigger pulse triggers multiple conversions in fifo mode. >

bool enableFifoScanMode

The field is valid when FIFO is enabled. Enable the FIFO scan mode. If enable, ADC will repeat using the first FIFO channel as the conversion channel until the result FIFO is fulfilled. >

bool enableCompareAndMode

The field is valid when FIFO is enabled. If enable, ADC will AND all of compare triggers and set COCO after all of compare triggers occur. If disable, ADC will OR all of compare triggers and set COCO after at least one of compare trigger occurs. >

uint32_t FifoDepth

Setting the depth of FIFO. Depth of fifo is FifoDepth + 1. When FifoDepth = 0U, the FIFO is DISABLED. When FifoDepth is set to nonzero, the FIFO function is ENABLED and the depth is indicated by the FifoDepth field. >

struct *_adc_channel_config*

#include <fsl_adc.h> ADC channel conversion configuration.

Public Members

uint32_t channelNumber

Setting the conversion channel number. The available range is 0-31. See channel connection information for each chip in Reference Manual document.

bool enableContinuousConversion

enables continuous conversions. >

bool enableInterruptOnConversionCompleted

Generate an interrupt request once the conversion is completed.

2.3 Clock Driver

enum _clock_name

Clock name used to get clock frequency.

Values:

enumerator kCLOCK_CoreSysClk

Core/system clock

enumerator kCLOCK_PlatClk

Platform clock

enumerator kCLOCK_BusClk

Bus clock

enumerator kCLOCK_FlashClk

Flash clock

enumerator kCLOCK_Osc0ErClk

OSC0 external reference clock (OSC0ERCLK)

enumerator kCLOCK_ICSFixedFreqClk

ICS fixed frequency clock (ICSFFCLK)

enumerator kCLOCK_ICSInternalRefClk

ICS internal reference clock (ICSIRCLK)

enumerator kCLOCK_ICSFllClk

ICSFLLCLK

enumerator kCLOCK_ICSOutClk

ICS Output clock

enumerator kCLOCK_LpoClk

LPO clock

enum _clock_ip_name

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

Values:

enumerator kCLOCK_IpInvalid

enumerator kCLOCK_I2c0

enumerator kCLOCK_Uart0

enumerator kCLOCK_Uart1

enumerator kCLOCK__Uart2
enumerator kCLOCK__Acmp0
enumerator kCLOCK__Acmp1
enumerator kCLOCK__Spi0
enumerator kCLOCK__Spi1
enumerator kCLOCK__Irq0
enumerator kCLOCK__Kbi0
enumerator kCLOCK__Kbi1
enumerator kCLOCK__Adc0
enumerator kCLOCK__Crc0
enumerator kCLOCK__Ftm0
enumerator kCLOCK__Ftm1
enumerator kCLOCK__Ftm2
enumerator kCLOCK__Pit0
enumerator kCLOCK__Rtc0

enum __osc_work_mode

OSC work mode.

Values:

enumerator kOSC__ModeExt
OSC source from external clock.
enumerator kOSC__ModeOscLowPower
Oscillator low freq low power.
enumerator kOSC__ModeOscHighGain
Oscillator low freq high gain.

enum __osc_enable_mode

OSC enable mode.

Values:

enumerator kOSC__Enable
Enable.
enumerator kOSC__EnableInStop
Enable in stop mode.

enum __ics_fll_src

ICS FLL reference clock source select.

Values:

enumerator kICS__FllSrcExternal
External reference clock is selected
enumerator kICS__FllSrcInternal
The slow internal reference clock is selected

enum `_ics_clkout_src`

ICSOUT clock source.

Values:

enumerator `kICS_ClkOutSrcFll`

Output of the FLL is selected (reset default)

enumerator `kICS_ClkOutSrcInternal`

Internal reference clock is selected, FLL is bypassed

enumerator `kICS_ClkOutSrcExternal`

External reference clock is selected, FLL is bypassed

ICS status. .

Values:

enumerator `kStatus_ICS_ModeUnreachable`

Can't switch to target mode.

enumerator `kStatus_ICS_SourceUsed`

Can't change the clock source because it is in use.

enum `_ics_irelk_enable_mode`

ICS internal reference clock (ICSIRCLK) enable mode definition.

Values:

enumerator `kICS_IrclkDisable`

ICSIRCLK disable.

enumerator `kICS_IrclkEnable`

ICSIRCLK enable.

enumerator `kICS_IrclkEnableInStop`

ICSIRCLK enable in stop mode.

enum `_ics_mode`

ICS mode definitions.

Values:

enumerator `kICS_ModeFEI`

FEI - FLL Engaged Internal

enumerator `kICS_ModeFBI`

FBI - FLL Bypassed Internal

enumerator `kICS_ModeBILP`

BILP - Bypassed Low Power Internal

enumerator `kICS_ModeFEE`

FEE - FLL Engaged External

enumerator `kICS_ModeFBE`

FBE - FLL Bypassed External

enumerator `kICS_ModeBELP`

BELP - Bypassed Low Power External

enumerator `kICS_ModeError`

Unknown mode

```
typedef enum _clock_name clock_name_t
```

Clock name used to get clock frequency.

```
typedef enum _clock_ip_name clock_ip_name_t
```

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

```
typedef struct _sim_clock_config sim_clock_config_t
```

SIM configuration structure for clock setting.

```
typedef struct _osc_config osc_config_t
```

OSC Initialization Configuration Structure.

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

- a. freq: The external frequency.
- b. workMode: The OSC module mode.
- c. enableMode: The OSC enable mode.

```
typedef enum _ics_fll_src ics_fll_src_t
```

ICS FLL reference clock source select.

```
typedef enum _ics_clkout_src ics_clkout_src_t
```

ICSOUT clock source.

```
typedef enum _ics_mode ics_mode_t
```

ICS mode definitions.

```
typedef struct _ics_config ics_config_t
```

ICS configuration structure.

When porting to a new board, set the following members according to the board setting:

- a. icsMode: ICS mode
- b. irClkEnableMode: ICSIRCLK enable mode
- c. rDiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by rDiv is in the 31.25 kHz to 39.0625 kHz range.
- d. bDiv, this divider determine the ISCOUNT clock

```
volatile uint32_t g_xtal0Freq
```

External XTAL0 (OSC0) clock frequency.

The XTAL0/EXTAL0 (OSC0) clock frequency in Hz. When the clock is set up, use the function CLOCK_SetXtal0Freq to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
CLOCK_InitOsc0(...);
CLOCK_SetXtal0Freq(8000000)
```

This is important for the multicore platforms where only one core needs to set up the OSC0 using the CLOCK_InitOsc0. All other cores need to call the CLOCK_SetXtal0Freq to get a valid clock frequency.

```
static inline void CLOCK_EnableClock(clock_ip_name_t name)
```

Enable the clock for specific IP.

Parameters

- name – Which clock to enable, see clock_ip_name_t.

static inline void CLOCK_DisableClock(*clock_ip_name_t* name)

Disable the clock for specific IP.

Parameters

- name – Which clock to disable, see *clock_ip_name_t*.

static inline void CLOCK_SetBusClkDiv(uint32_t busDiv)

clock divider

Set the SIM_BUSDIV. Carefully configure the SIM_BUSDIV to avoid bus/flash clock frequency higher than 24MHZ.

Parameters

- busDiv – bus clock output divider value.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in *clock_name_t*. The ICS must be properly configured before using this function.

Parameters

- clockName – Clock names defined in *clock_name_t*

Returns

Clock frequency value in Hertz

uint32_t CLOCK_GetCoreSysClkFreq(void)

Get the core clock or system clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetBusClkFreq(void)

Get the bus clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetFlashClkFreq(void)

Get the flash clock frequency.

Returns

Clock frequency in Hz.

uint32_t CLOCK_GetOsc0ErClkFreq(void)

Get the OSC0 external reference clock frequency (OSC0ERCLK).

Returns

Clock frequency in Hz.

void CLOCK_SetSimConfig(*sim_clock_config_t* const *config)

Set the clock configure in SIM module.

This function sets system layer clock settings in SIM module.

Parameters

- config – Pointer to the configure structure.

`static inline void CLOCK_SetSimSafeDivs(void)`

Set the system clock dividers in SIM to safe value.

The system level clocks (core clock, bus clock, and flash clock) must be in allowed ranges. During ICS clock mode switch, the ICS output clock changes then the system level clocks may be out of range. This function could be used before ICS mode change, to make sure system level clocks are in allowed range.

`FSL_CLOCK_DRIVER_VERSION`

CLOCK driver version 2.2.3.

`SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY`

`UART_CLOCKS`

Clock ip name array for UART.

`ADC_CLOCKS`

Clock ip name array for ADC16.

`IRQ_CLOCKS`

Clock ip name array for IRQ.

`KBI_CLOCKS`

Clock ip name array for KBI.

`SPI_CLOCKS`

Clock ip name array for SPI.

`I2C_CLOCKS`

Clock ip name array for I2C.

`FTM_CLOCKS`

Clock ip name array for FTM.

`ACMP_CLOCKS`

Clock ip name array for CMP.

`CRC_CLOCKS`

Clock ip name array for CRC.

`PIT_CLOCKS`

Clock ip name array for PIT.

`RTC_CLOCKS`

Clock ip name array for RTC.

`LPO_CLK_FREQ`

LPO clock frequency.

`CLK_GATE_REG_OFFSET_SHIFT`

`CLK_GATE_REG_OFFSET_MASK`

`CLK_GATE_BIT_SHIFT_SHIFT`

`CLK_GATE_BIT_SHIFT_MASK`

`CLK_GATE_DEFINE(reg_offset, bit_shift)`

`CLK_GATE_ABSTRACT_REG_OFFSET(x)`

`CLK_GATE_ABSTRACT_BITS_SHIFT(x)`

uint32_t CLOCK_GetICSOutClkFreq(void)

Gets the ICS output clock (ICSOUTCLK) frequency.

This function gets the ICS output clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSOUTCLK.

uint32_t CLOCK_GetFllFreq(void)

Gets the ICS FLL clock (ICSFLLCLK) frequency.

This function gets the ICS FLL clock frequency in Hz based on the current ICS register value. The FLL is enabled in FEI/FBI/FEE/FBE mode and disabled in low power state in other modes.

Returns

The frequency of ICSFLLCLK.

uint32_t CLOCK_GetInternalRefClkFreq(void)

Gets the ICS internal reference clock (ICSIRCLK) frequency.

This function gets the ICS internal reference clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSIRCLK.

uint32_t CLOCK_GetICSFixedFreqClkFreq(void)

Gets the ICS fixed frequency clock (ICSFFCLK) frequency.

This function gets the ICS fixed frequency clock frequency in Hz based on the current ICS register value.

Returns

The frequency of ICSFFCLK.

static inline void CLOCK_SetLowPowerEnable(bool enable)

Enables or disables the ICS low power.

Enabling the ICS low power disables the PLL and FLL in bypass modes. In other words, in FBE and PBE modes, enabling low power sets the ICS to BELP mode. In FBI and PBI modes, enabling low power sets the ICS to BILP mode. When disabling the ICS low power, the PLL or FLL are enabled based on ICS settings.

Parameters

- enable – True to enable ICS low power, false to disable ICS low power.

static inline void CLOCK_SetInternalRefClkConfig(uint8_t enableMode)

Configures the Internal Reference clock (ICSIRCLK).

This function sets the ICSIRCLK base on parameters. This function also sets whether the ICSIRCLK is enabled in stop mode.

Parameters

- enableMode – ICSIRCLK enable mode, OR'ed value of _ICS_ircclk_enable_mode.

Return values

- kStatus_IC_SourceUsed – Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
- kStatus_Success – ICSIRCLK configuration finished successfully.

static inline void CLOCK_SetFllExtRefDiv(uint8_t rdiv)

Set the FLL external reference clock divider value.

Sets the FLL external reference clock divider value, the register ICS_C1[RDIV]. Resulting frequency must be in the range 31.25KHZ to 39.0625KHZ.

Parameters

- rdiv – The FLL external reference clock divider value, ICS_C1[RDIV].

static inline void CLOCK_SetOsc0MonitorMode(bool enable)

Sets the OSC0 clock monitor mode.

This function sets the OSC0 clock monitor mode. See ics_monitor_mode_t for details.

Parameters

- enable – True to enable clock monitor, false to disable clock monitor.

void CLOCK_InitOsc0(osc_config_t const *config)

Initializes the OSC0.

This function initializes the OSC0 according to the board configuration.

Parameters

- config – Pointer to the OSC0 configuration structure.

void CLOCK_DeinitOsc0(void)

Deinitializes the OSC0.

This function deinitializes the OSC0.

static inline void CLOCK_SetXtal0Freq(uint32_t freq)

Sets the XTAL0 frequency based on board settings.

Parameters

- freq – The XTAL0/EXTAL0 input clock frequency in Hz.

static inline void CLOCK_SetOsc0Enable(uint8_t enable)

Sets the OSC enable.

Parameters

- enable – osc enable mode.

ics_mode_t CLOCK_GetMode(void)

Gets the current ICS mode.

This function checks the ICS registers and determines the current ICS mode.

Returns

Current ICS mode or error code; See ics_mode_t.

status_t CLOCK_SetFeiMode(uint8_t bDiv)

Sets the ICS to FEI mode.

This function sets the ICS to FEI mode. If setting to FEI mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetFeeMode(uint8_t bDiv, uint8_t rDiv)

Sets the ICS to FEE mode.

This function sets the ICS to FEE mode. If setting to FEE mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider
- rDiv – FLL reference clock divider setting, RDIV.

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetFbiMode(uint8_t bDiv)

Sets the ICS to FBI mode.

This function sets the ICS to FBI mode. If setting to FBI mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetFbeMode(uint8_t bDiv, uint8_t rDiv)

Sets the ICS to FBE mode.

This function sets the ICS to FBE mode. If setting to FBE mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider
- rDiv – FLL reference clock divider setting, RDIV.

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetBilpMode(uint8_t bDiv)

Sets the ICS to BILP mode.

This function sets the ICS to BILP mode. If setting to BILP mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetBelpMode(uint8_t bDiv)

Sets the ICS to BELP mode.

This function sets the ICS to BELP mode. If setting to BELP mode fails from the current mode, this function returns an error.

Parameters

- bDiv – bus clock divider

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToFeiMode(uint8_t bDiv)

Sets the ICS to FEI mode during system boot up.

This function sets the ICS to FEI mode from the reset mode. It can also be used to set up ICS during system boot up.

Parameters

- bDiv – bus clock divider.

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToFeeMode(uint8_t bDiv, uint8_t rDiv)

Sets the ICS to FEE mode during system bootup.

This function sets ICS to FEE mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

- bDiv – bus clock divider.
- rDiv – FLL reference clock divider setting, RDIV.

Return values

- kStatus_ICS_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToBilpMode(uint8_t bDiv)

Sets the ICS to BILP mode during system boot up.

This function sets the ICS to BILP mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

- bDiv – bus clock divider.

Return values

- kStatus_ICS_SourceUsed – Could not change ICSIRCLK setting.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToBelpMode(uint8_t bDiv)

Sets the ICS to BELP mode during system boot up.

This function sets the ICS to BELP mode from the reset mode. It can also be used to set up the ICS during system boot up.

Parameters

- bDiv – bus clock divider.

Return values

- kStatus_Ics_ModeUnreachable – Could not switch to the target mode.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetIcsConfig(*ics_config_t* const *config)

Sets the ICS to a target mode.

This function sets ICS to a target mode defined by the configuration structure. If switching to the target mode fails, this function chooses the correct path.

Note: If the external clock is used in the target mode, ensure that it is enabled. For example, if the OSC0 is used, set up OSC0 correctly before calling this function.

Parameters

- config – Pointer to the target ICS mode configuration structure.

Returns

Return kStatus_Success if switched successfully; Otherwise, it returns an error code `_ICS_status`.

uint32_t busDiv

SIM_BUSDIV.

uint8_t busClkPrescaler

A option prescaler for bus clock

uint32_t freq

External clock frequency.

uint8_t workMode

OSC work mode setting.

uint8_t enableMode

Configuration for OSCERCLK.

ics_mode_t icsMode

ICS mode.

uint8_t irClkEnableMode

ICSIRCLK enable mode.

uint8_t rDiv

Divider for external reference clock, ICS_C1[RDIV].

uint8_t bDiv

Divider for ICS output clock ICS_C2[BDIV].

ICS_CONFIG_CHECK_PARAM

Configures whether to check a parameter in a function.

Some ICS settings must be changed with conditions, for example:

- a. ICSIRCLK settings, such as the source, divider, and the trim value should not change when ICSIRCLK is used as a system clock source.
- b. ICS_C7[OSCSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BELP/PBE modes.

- c. The users should only switch between the supported clock modes.

ICS functions check the parameter and ICS status before setting, if not allowed to change, the functions return error. The parameter checking increases code size, if code size is a critical requirement, change ICS_CONFIG_CHECK_PARAM to 0 to disable parameter checking.

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

struct _sim_clock_config

#include <fsl_clock.h> SIM configuration structure for clock setting.

struct _osc_config

#include <fsl_clock.h> OSC Initialization Configuration Structure.

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

- a. freq: The external frequency.
- b. workMode: The OSC module mode.
- c. enableMode: The OSC enable mode.

struct _ics_config

#include <fsl_clock.h> ICS configuration structure.

When porting to a new board, set the following members according to the board setting:

- a. icsMode: ICS mode
- b. irClkEnableMode: ICSIRCLK enable mode
- c. rDiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by rDiv is in the 31.25 kHz to 39.0625 kHz range.
- d. bDiv, this divider determine the ISCOUT clock

2.4 CRC: Cyclic Redundancy Check Driver

FSL_CRC_DRIVER_VERSION

CRC driver version. Version 2.0.5.

Current version: 2.0.5

Change log:

- Version 2.0.5
 - Fix CERT-C issue with boolean-to-unsigned integer conversion.
- Version 2.0.4
 - Release peripheral from reset if necessary in init function.
- Version 2.0.3

- Fix MISRA issues
- Version 2.0.2
 - Fix MISRA issues
- Version 2.0.1
 - move DATA and DATALL macro definition from header file to source file

enum `_crc_bits`

CRC bit width.

Values:

enumerator `kCrcBits16`

Generate 16-bit CRC code

enumerator `kCrcBits32`

Generate 32-bit CRC code

enum `_crc_result`

CRC result type.

Values:

enumerator `kCrcFinalChecksum`

CRC data register read value is the final checksum. Reflect out and final xor protocol features are applied.

enumerator `kCrcIntermediateChecksum`

CRC data register read value is intermediate checksum (raw value). Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for `CRC_Init()` to continue adding data to this checksum.

typedef enum `_crc_bits` `crc_bits_t`

CRC bit width.

typedef enum `_crc_result` `crc_result_t`

CRC result type.

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 function enables the clock gate in the SIM module for the CRC peripheral. It also configures the CRC module and starts a 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 function disables the clock gate in the SIM module for the CRC peripheral.

Parameters

- `base` – CRC peripheral address.

`void CRC_GetDefaultConfig(crc_config_t *config)`

Loads default values to the CRC protocol configuration structure.

Loads default values to the CRC protocol configuration structure. The default values are as follows.

```
config->polynomial = 0x1021;
config->seed = 0xFFFF;
config->reflectIn = false;
config->reflectOut = false;
config->complementChecksum = false;
config->crcBits = kCrcBits16;
config->crcResult = kCrcFinalChecksum;
```

Parameters

- `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 the CRC data register. The configured type of transpose is applied.

Parameters

- `base` – CRC peripheral address.
- `data` – Input data stream, MSByte in `data[0]`.
- `dataSize` – Size in bytes of the input data buffer.

`uint32_t CRC_Get32bitResult(CRC_Type *base)`

Reads the 32-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- `base` – CRC peripheral address.

Returns

An intermediate or the final 32-bit checksum, after configured transpose and complement operations.

`uint16_t CRC_Get16bitResult(CRC_Type *base)`

Reads a 16-bit checksum from the CRC module.

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

- `base` – CRC peripheral address.

Returns

An intermediate or the final 16-bit checksum, after configured transpose and complement operations.

`CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT`

Default configuration structure filled by `CRC_GetDefaultConfig()`. Use CRC16-CCIT-FALSE as default.

`struct _crc_config`

`#include <fsl_crc.h>` CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

Public Members

`uint32_t` polynomial
CRC Polynomial, MSBit first. Example polynomial: $0x1021 = 1_0000_0010_0001 = x^{12} + x^5 + 1$

`uint32_t` seed
Starting checksum value

`bool` reflectIn
Reflect bits on input.

`bool` reflectOut
Reflect bits on output.

`bool` complementChecksum
True if the result shall be complement of the actual checksum.

`crc_bits_t` crcBits
Selects 16- or 32- bit CRC protocol.

`crc_result_t` crcResult
Selects final or intermediate checksum return from `CRC_Get16bitResult()` or `CRC_Get32bitResult()`

2.5 FGPIO Driver

`void` `FGPIO_PortInit(gpio_port_num_t port)`

Initializes the FGPIO peripheral.

This function ungates the FGPIO clock.

Parameters

- port – FGPIO PORT number, see “`gpio_port_num_t`”. For each group FGPIO (FGPIOA, FGPIOB, etc) control registers, they handles four PORT number controls. FGPIOA serial registers – PTA 0 ~ 7, PTB 0 ~ 7 ... PTD 0 ~ 7. FGPIOB serial registers – PTE 0 ~ 7, PTF 0 ~ 7 ... PTH 0 ~ 7. ...

`void` `FGPIO_PinInit(gpio_port_num_t port, uint8_t pin, const gpio_pin_config_t *config)`

Initializes a FGPIO pin used by the board.

To initialize the FGPIO driver, define a pin configuration, as either input or output, in the user file. Then, call the `FGPIO_PinInit()` function.

This is an example to define an input pin or an 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

- port – FGPIOPORT number, see “gpio_port_num_t”. For each group FGPIOP (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- pin – FGPIOP port pin number
- config – FGPIOP pin configuration pointer

void FGPIOP_PinWrite(*gpio_port_num_t* port, uint8_t pin, uint8_t output)

Sets the output level of the multiple FGPIOP pins to the logic 1 or 0.

Parameters

- port – FGPIOPORT number, see “gpio_port_num_t”. For each group FGPIOP (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- pin – FGPIOP pin number
- output – FGPIOPin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

void FGPIOP_PortSet(*gpio_port_num_t* port, uint8_t mask)

Sets the output level of the multiple FGPIOP pins to the logic 1.

Parameters

- port – FGPIOPORT number, see “gpio_port_num_t”. For each group FGPIOP (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- mask – FGPIOP pin number macro

void FGPIOP_PortClear(*gpio_port_num_t* port, uint8_t mask)

Sets the output level of the multiple FGPIOP pins to the logic 0.

Parameters

- port – FGPIOPORT number, see “gpio_port_num_t”. For each group FGPIOP (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- mask – FGPIOP pin number macro

void FGPIOP_PortToggle(*gpio_port_num_t* port, uint8_t mask)

Reverses the current output logic of the multiple FGPIOP pins.

Parameters

- port – FGPIOPORT number, see “gpio_port_num_t”. For each group FGPIOP (FGPIOA, FGPIOB,etc) control registers, they handles four PORT number controls. FGPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. FGPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- mask – FGPIOP pin number macro

uint32_t FGPIOP_PinRead(*gpio_port_num_t* port, uint8_t pin)

Reads the current input value of the FGPIOP port.

Parameters

- port – FGPIO PORT number, see “gpio_port_num_t”. For each group FGPIO (FGPIOA, FGPIOB, etc) control registers, they handle four PORT number controls. FGPIOA serial registers – PTA 0 ~ 7, PTB 0 ~ 7 ... PTD 0 ~ 7. FGPIOB serial registers – PTE 0 ~ 7, PTF 0 ~ 7 ... PTH 0 ~ 7. ...
- pin – FGPIO pin number

Return values

FGPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

2.6 FTMRx Flash Driver

enum _flash_driver_version_constants

Flash driver version for ROM.

Values:

enumerator kFLASH_DriverVersionName

Flash driver version name.

enumerator kFLASH_DriverVersionMajor

Major flash driver version.

enumerator kFLASH_DriverVersionMinor

Minor flash driver version.

enumerator kFLASH_DriverVersionBugfix

Bugfix for flash driver version.

MAKE_VERSION(major, minor, bugfix)

Constructs the version number for drivers.

FSL_FLASH_DRIVER_VERSION

Flash driver version for SDK.

Version 2.1.2.

Flash driver status codes.

Values:

enumerator kStatus_FLASH_Success

API is executed successfully

enumerator kStatus_FLASH_InvalidArgument

Invalid argument

enumerator kStatus_FLASH_SizeError

Error size

enumerator kStatus_FLASH_AlignmentError

Parameter is not aligned with the specified baseline

enumerator kStatus_FLASH_AddressError

Address is out of range

enumerator kStatus_FLASH_AccessError

Invalid instruction codes and out-of bound addresses

enumerator kStatus_FLASH_ProtectionViolation

The program/erase operation is requested to execute on protected areas

enumerator kStatus_FLASH_CommandFailure

Run-time error during command execution.

enumerator kStatus_FLASH_UnknownProperty

Unknown property.

enumerator kStatus_FLASH_EraseKeyError

API erase key is invalid.

enumerator kStatus_FLASH_RegionExecuteOnly

The current region is execute-only.

enumerator kStatus_FLASH_ExecuteInRamFunctionNotReady

Execute-in-RAM function is not available.

enumerator kStatus_FLASH_PartitionStatusUpdateFailure

Failed to update partition status.

enumerator kStatus_FLASH_SetFlexramAsEepromError

Failed to set FlexRAM as EEPROM.

enumerator kStatus_FLASH_RecoverFlexramAsRamError

Failed to recover FlexRAM as RAM.

enumerator kStatus_FLASH_SetFlexramAsRamError

Failed to set FlexRAM as RAM.

enumerator kStatus_FLASH_RecoverFlexramAsEepromError

Failed to recover FlexRAM as EEPROM.

enumerator kStatus_FLASH_CommandNotSupported

Flash API is not supported.

enumerator kStatus_FLASH_SwapSystemNotInUninitialized

Swap system is not in an uninitialized state.

enumerator kStatus_FLASH_SwapIndicatorAddressError

The swap indicator address is invalid.

enumerator kStatus_FLASH_ReadOnlyProperty

The flash property is read-only.

enumerator kStatus_FLASH_InvalidPropertyValue

The flash property value is out of range.

enumerator kStatus_FLASH_InvalidSpeculationOption

The option of flash prefetch speculation is invalid.

enumerator kStatus_FLASH_ClockDivider

Flash clock prescaler is wrong

enumerator kStatus_FLASH_EepromDoubleBitFault

A double bit fault was detected in the stored parity.

enumerator kStatus_FLASH_EepromSingleBitFault

A single bit fault was detected in the stored parity.

kStatusGroupGeneric

Flash driver status group.

kStatusGroupFlashDriver

MAKE_STATUS(group, code)

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

enum _flash_driver_api_keys

Enumeration for Flash driver API keys.

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

Values:

enumerator kFLASH_ApiEraseKey

Key value used to validate all flash erase APIs.

FOUR_CHAR_CODE(a, b, c, d)

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

status_t FLASH_Init(flash_config_t *config)

Initializes the global flash properties structure members.

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

Parameters

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

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_ClockDivider – Flash clock prescaler is wrong.
- kStatus_FLASH_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

status_t FLASH_SetCallback(flash_config_t *config, flash_callback_t callback)

Sets the desired flash callback function.

Parameters

- config – Pointer to the storage for the driver runtime state.
- callback – A callback function to be stored in the driver.

Return values

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

status_t FLASH_PrepareExecuteInRamFunctions(flash_config_t *config)

Prepares flash execute-in-RAM functions.

Parameters

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

Return values

- kStatus_FLASH_Success – API was executed successfully.

- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.

`status_t FLASH_EraseAll(flash_config_t *config, uint32_t key)`

Erases entire flash.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_EraseKeyError` – API erase key is invalid.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during command execution.
- `kStatus_FLASH_EepromSingleBitFault` – EEPROM single bit fault error code.
- `kStatus_FLASH_EepromDoubleBitFault` – EEPROM double bit fault error code.

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

Erases the flash sectors encompassed by parameters passed into function.

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AddressError` – The address is out of range.
- `kStatus_FLASH_EraseKeyError` – The API erase key is invalid.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.

- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

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

Erases the eeprom sectors encompassed by parameters passed into function.

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AddressError` – The address is out of range.
- `kStatus_FLASH_EraseKeyError` – The API erase key is invalid.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_EraseAllUnsecure(flash_config_t *config, uint32_t key)`

Erases the entire flash, including protected sectors.

Parameters

- `config` – Pointer to the storage for the driver runtime state.
- `key` – A value used to validate all flash erase APIs.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_EraseKeyError` – API erase key is invalid.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.

- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during command execution.
- `kStatus_FLASH_EepromSingleBitFault` – EEPROM single bit fault error code.
- `kStatus_FLASH_EepromDoubleBitFault` – EEPROM double bit fault error code.

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

Programs flash with data at locations passed in through parameters.

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_ProgramOnce(flash_config_t *config, uint32_t index, uint32_t *src, uint32_t lengthInBytes)`

Programs Program Once Field through parameters.

This function programs the Program Once Field with the desired data for a given flash area as determined by the index and length.

Parameters

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

- `index` – The index indicating which area of the Program Once Field to be programmed.
- `src` – A pointer to the source buffer of data that is to be programmed into the Program Once Field.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

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

Programs the EEPROM with data at locations passed in through parameters.

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_EepromSingleBitFault` – EEPROM single bit fault error code.
- `kStatus_FLASH_EepromDoubleBitFault` – EEPROM double bit fault error code.

`status_t FLASH_ReadOnce(flash_config_t *config, uint32_t index, uint32_t *dst, uint32_t lengthInBytes)`

Reads the Program Once Field through parameters.

This function reads the read once feild with given index and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `index` – The index indicating the area of program once field to be read.
- `dst` – A pointer to the destination buffer of data that is used to store data to be read.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

status_t FLASH_GetSecurityState(*flash_config_t* *config, *flash_security_state_t* *state)

Returns the security state via the pointer passed into the function.

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

- `config` – A pointer to storage for the driver runtime state.
- `state` – A pointer to the value returned for the current security status code:

Return values

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

status_t FLASH_SecurityBypass(*flash_config_t* *config, const uint8_t *backdoorKey)

Allows users to bypass security with a backdoor key.

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `backdoorKey` – A pointer to the user buffer containing the backdoor key.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.

- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_VerifyEraseAll(flash_config_t *config, flash_margin_value_t margin)`

Verifies erasure of the entire flash at a specified margin level.

This function checks whether the flash is erased to the specified read margin level.

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_EepromSingleBitFault` – EEPROM single bit fault error code.
- `kStatus_FLASH_EepromDoubleBitFault` – EEPROM double bit fault error code.

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

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

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.

- `kStatus_FLASH_ExecuteInRamFunctionNotReady` – Execute-in-RAM function is not available.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

`status_t FLASH_IsProtected(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_protection_state_t *protection_state)`

Returns the protection state of the desired flash area via the pointer passed into the function.

This function retrieves the current flash protect status for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be checked. Must be word-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be checked. Must be word-aligned.
- `protection_state` – A pointer to the value returned for the current protection status code for the desired flash area.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – The address is out of range.

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

Returns the desired flash property.

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_UnknownProperty` – An unknown property tag.

`status_t FLASH_SetProperty(flash_config_t *config, flash_property_tag_t whichProperty, uint32_t value)`

Sets the desired flash property.

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_UnknownProperty` – An unknown property tag.
- `kStatus_FLASH_InvalidPropertyValue` – An invalid property value.
- `kStatus_FLASH_ReadOnlyProperty` – An read-only property tag.

`status_t` FLASH_PflashSetProtection(*flash_config_t* *config, *pflash_protection_status_t* *protectStatus)

Sets the PFlash Protection to the intended protection status.

Parameters

- `config` – A pointer to storage for the driver runtime state.
- `protectStatus` – The expected protect status to set to the PFlash protection register.

Return values

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

`status_t` FLASH_PflashGetProtection(*flash_config_t* *config, *pflash_protection_status_t* *protectStatus)

Gets the PFlash protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – Protect status returned by the PFlash IP.

Return values

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

`status_t` FLASH_EepromSetProtection(*flash_config_t* *config, `uint8_t` protectStatus)

Sets the EEPROM protection to the intended protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – The expected protect status to set to the EEPROM protection register.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.

- `kStatus_FLASH_CommandFailure` – Run-time error during command execution.

`status_t FLASH_EepromGetProtection(flash_config_t *config, uint8_t *protectStatus)`

Gets the EEPROM protection status.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `protectStatus` – EEPROM Protect status returned by the EEPROM IP.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.

`status_t FLASH_PflashSetPrefetchSpeculation(flash_prefetch_speculation_status_t *speculationStatus)`

Sets the PFlash prefetch speculation to the intended speculation status.

Parameters

- `speculationStatus` – The expected protect status to set to the PFlash protection register. Each bit is

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidSpeculationOption` – An invalid speculation option argument is provided.

`status_t FLASH_PflashGetPrefetchSpeculation(flash_prefetch_speculation_status_t *speculationStatus)`

Gets the PFlash prefetch speculation status.

Parameters

- `speculationStatus` – Speculation status returned by the PFlash IP.

Return values

`kStatus_FLASH_Success` – API was executed successfully.

`FLASH_SSD_CONFIG_ENABLE_EEPROM_SUPPORT`

Indicates whether to support EEPROM in the Flash driver.

Enables the EEPROM support.

`FLASH_SSD_IS_EEPROM_ENABLED`

Indicates whether the EEPROM is enabled in the Flash driver.

`FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT`

Indicates whether to support Secondary flash in the Flash driver.

Enables the secondary flash support by default.

`FLASH_SSD_IS_SECONDARY_FLASH_ENABLED`

Indicates whether the secondary flash is supported in the Flash driver.

`FLASH_DRIVER_IS_FLASH_RESIDENT`

Flash driver location.

Used for the flash resident application.

FLASH_DRIVER_IS_EXPORTED

Flash Driver Export option.

Used for the MCUXpresso SDK application.

FLASH_ENABLE_STALLING_FLASH_CONTROLLER

Enable flash stalling controller.

enum _flash_user_margin_value

Enumeration for supported flash user margin levels.

Values:

enumerator kFLASH_ReadMarginValueNormal

Use the 'normal' read level for 1s.

enumerator kFLASH_UserMarginValue1

Apply the 'User' margin to the normal read-1 level.

enumerator kFLASH_UserMarginValue0

Apply the 'User' margin to the normal read-0 level.

enum _flash_factory_margin_value

Enumeration for supported factory margin levels.

Values:

enumerator kFLASH_FactoryMarginValue1

Apply the 'Factory' margin to the normal read-1 level.

enumerator kFLASH_FactoryMarginValue0

Apply the 'Factory' margin to the normal read-0 level.

enum _flash_margin_value

Enumeration for supported flash margin levels.

Values:

enumerator kFLASH_MarginValueNormal

Use the 'normal' read level for 1s.

enumerator kFLASH_MarginValueUser

Apply the 'User' margin to the normal read-1 level.

enumerator kFLASH_MarginValueFactory

Apply the 'Factory' margin to the normal read-1 level.

enumerator kFLASH_MarginValueInvalid

Not real margin level, Used to determine the range of valid margin level.

enum _flash_security_state

Enumeration for the three possible flash security states.

Values:

enumerator kFLASH_SecurityStateNotSecure

Flash is not secure.

enumerator kFLASH_SecurityStateBackdoorEnabled

Flash backdoor is enabled.

enumerator kFLASH_SecurityStateBackdoorDisabled

Flash backdoor is disabled.

enum _flash_protection_state

Enumeration for the three possible flash protection levels.

Values:

enumerator kFLASH_ProtectionStateUnprotected

Flash region is not protected.

enumerator kFLASH_ProtectionStateProtected

Flash region is protected.

enumerator kFLASH_ProtectionStateMixed

Flash is mixed with protected and unprotected region.

enum _flash_property_tag

Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflashSectorSize

Pflash sector size property.

enumerator kFLASH_PropertyPflashTotalSize

Pflash total size property.

enumerator kFLASH_PropertyPflashBlockSize

Pflash block size property.

enumerator kFLASH_PropertyPflashBlockCount

Pflash block count property.

enumerator kFLASH_PropertyPflashBlockBaseAddr

Pflash block base address property.

enumerator kFLASH_PropertyPflashFacSupport

Pflash fac support property.

enumerator kFLASH_PropertyEepromTotalSize

EEPROM total size property.

enumerator kFLASH_PropertyFlashMemoryIndex

Flash memory index property.

enumerator kFLASH_PropertyFlashCacheControllerIndex

Flash cache controller index property.

enumerator kFLASH_PropertyEepromBlockBaseAddr

EEPROM block base address property.

enumerator kFLASH_PropertyEepromSectorSize

EEPROM sector size property.

enumerator kFLASH_PropertyEepromBlockSize

EEPROM block size property.

enumerator kFLASH_PropertyEepromBlockCount

EEPROM block count property.

enumerator kFLASH_PropertyFlashClockFrequency

Flash peripheral clock property.

Constants for execute-in-RAM flash function. `_flash_execute_in_ram_function_constants`.

Values:

enumerator kFLASH_ExecuteInRamFunctionMaxSizeInWords
The maximum size of execute-in-RAM function.

enumerator kFLASH_ExecuteInRamFunctionTotalNum
Total number of execute-in-RAM functions.

enum _flash_memory_index
Enumeration for the flash memory index.

Values:

enumerator kFLASH_MemoryIndexPrimaryFlash
Current flash memory is primary flash.

enumerator kFLASH_MemoryIndexSecondaryFlash
Current flash memory is secondary flash.

enum _flash_cache_controller_index
Enumeration for the flash cache controller index.

Values:

enumerator kFLASH_CacheControllerIndexForCore0
Current flash cache controller is for core 0.

enumerator kFLASH_CacheControllerIndexForCore1
Current flash cache controller is for core 1.

enum _flash_prefetch_speculation_option
Enumeration for the two possible options of flash prefetch speculation.

Values:

enumerator kFLASH_prefetchSpeculationOptionEnable

enumerator kFLASH_prefetchSpeculationOptionDisable

enum _flash_cache_clear_process
Flash cache clear process code.

Values:

enumerator kFLASH_CacheClearProcessPre
Pre flash cache clear process.

enumerator kFLASH_CacheClearProcessPost
Post flash cache clear process.

typedef enum _flash_user_margin_value flash_user_margin_value_t
Enumeration for supported flash user margin levels.

typedef enum _flash_factory_margin_value flash_factory_margin_value_t
Enumeration for supported factory margin levels.

typedef enum _flash_margin_value flash_margin_value_t
Enumeration for supported flash margin levels.

typedef enum _flash_security_state flash_security_state_t
Enumeration for the three possible flash security states.

typedef enum _flash_protection_state flash_protection_state_t
Enumeration for the three possible flash protection levels.

typedef enum _flash_property_tag flash_property_tag_t
Enumeration for various flash properties.

typedef struct *_pflash_protection_status* pflash_protection_status_t

PFlash protection status - full.

typedef enum *_flash_memory_index* flash_memory_index_t

Enumeration for the flash memory index.

typedef enum *_flash_cache_controller_index* flash_cache_controller_index_t

Enumeration for the flash cache controller index.

typedef void (*flash_callback_t)(void)

A callback type used for the Pflash block.

typedef enum *_flash_prefetch_speculation_option* flash_prefetch_speculation_option_t

Enumeration for the two possible options of flash prefetch speculation.

typedef struct *_flash_prefetch_speculation_status* flash_prefetch_speculation_status_t

Flash prefetch speculation status.

typedef enum *_flash_cache_clear_process* flash_cache_clear_process_t

Flash cache clear process code.

typedef struct *_flash_protection_config* flash_protection_config_t

Active flash protection information for the current operation.

typedef struct *_flash_operation_config* flash_operation_config_t

Active flash information for the current operation.

typedef struct *_flash_execute_in_ram_function_config* flash_execute_in_ram_function_config_t

Flash execute-in-RAM function information.

typedef struct *_flash_config* flash_config_t

Flash driver state information.

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

struct *_pflash_protection_status*

#include <fsl_flash.h> PFlash protection status - full.

Public Members

uint8_t fprotvalue

FPROT[7:0] .

struct *_flash_prefetch_speculation_status*

#include <fsl_flash.h> Flash prefetch speculation status.

Public Members

flash_prefetch_speculation_option_t instructionOption

Instruction speculation.

flash_prefetch_speculation_option_t dataOption

Data speculation.

struct *_flash_protection_config*

#include <fsl_flash.h> Active flash protection information for the current operation.

Public Members

uint32_t lowRegionStart
Start address of flash protection low region.

uint32_t lowRegionEnd
End address of flash protection low region.

uint32_t highRegionStart
Start address of flash protection high region.

uint32_t highRegionEnd
End address of flash protection high region.

struct _flash_operation_config
#include <fsl_flash.h> Active flash information for the current operation.

Public Members

uint32_t convertedAddress
A converted address for the current flash type.

uint32_t activeSectorSize
A sector size of the current flash type.

uint32_t activeBlockSize
A block size of the current flash type.

uint32_t blockWriteUnitSize
The write unit size.

uint32_t sectorCmdAddressAligment
An erase sector command address alignment.

uint32_t sectionCmdAddressAligment
A program/verify section command address alignment.

uint32_t programCmdAddressAligment
A program flash command address alignment.

union function_run_command_t
#include <fsl_flash.h> Flash execute-in-RAM command.

Public Members

uint32_t commadAddr

void (*callFlashCommand)(volatile uint8_t *FTMRx_fstat)

union function_common_bit_operation_t
#include <fsl_flash.h>

Public Members

uint32_t bitOperationAddr

void (*callCommonBitOperationCommand)(volatile uint32_t *base, uint32_t bitMask, uint32_t bitShift, uint32_t bitValue)

struct _flash_execute_in_ram_function_config
#include <fsl_flash.h> Flash execute-in-RAM function information.

Public Members

uint32_t activeFunctionCount

Number of available execute-in-RAM functions.

function_run_command_t runCmdFuncAddr

Execute-in-RAM function: flash_run_command.

struct _flash_config

#include <fsl_flash.h> Flash driver state information.

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

Public Members

uint32_t PFlashBlockBase

A base address of the first PFlash block

uint32_t PFlashTotalSize

The size of the combined PFlash block.

uint8_t PFlashBlockCount

A number of PFlash blocks.

uint8_t FlashMemoryIndex

0 - primary flash; 1 - secondary flash

uint8_t FlashCacheControllerIndex

0 - Controller for core 0; 1 - Controller for core 1

uint8_t Reserved0

Reserved field 0

uint32_t PFlashSectorSize

The size in bytes of a sector of PFlash.

flash_callback_t PFlashCallback

The callback function for the flash API.

uint32_t *flashExecuteInRamFunctionInfo

An information structure of the flash execute-in-RAM function.

uint32_t EEPromTotalSize

For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM For the non-FlexNVM device, this field is unused

uint32_t EEPromBlockBase

This is the base address of the Eeprom For the non-Eeprom device, this field is unused

uint8_t EEPromBlockCount

A number of EEPROM blocks. For the non-Eeprom device, this field is unused

uint8_t EEPromSectorSize

The size in bytes of a sector of EEPROM. For the non-Eeprom device, this field is unused

uint8_t Reserved1[2]

Reserved field 1

uint32_t PFlashClockFreq

The flash peripheral clock frequency

uint32_t PFlashMarginLevel
The margin level

2.7 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:

- 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.
 - 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 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 2.7.4.

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_Deadtme_Prescale_4
Divide by 4

enumerator kFTM_Deadtme_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

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 reload 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

2.8 GPIO: General-Purpose Input/Output Driver

`FSL_GPIO_DRIVER_VERSION`

GPIO driver version.

`enum _gpio_port_num`

PORT definition.

Values:

enumerator `kGPIO_PORTA`

enumerator `kGPIO_PORTB`

enumerator `kGPIO_PORTC`

enumerator `kGPIO_PORTD`

enumerator `kGPIO_PORTE`

enumerator kGPIO_PORTF

enumerator kGPIO_PORTG

enumerator kGPIO_PORTH

enum _gpio_pin_direction

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_port_num gpio_port_num_t

PORT definition.

typedef enum _gpio_pin_direction gpio_pin_direction_t

GPIO direction definition.

typedef struct _gpio_pin_config gpio_pin_config_t

The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

struct _gpio_pin_config

#include <fsl_gpio.h> The GPIO pin configuration structure.

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig().

Public Members

gpio_pin_direction_t pinDirection

GPIO direction, input or output

uint8_t outputLogic

Set a default output logic, which has no use in input

2.9 GPIO Driver

void GPIO_PinInit(*gpio_port_num_t* port, uint8_t pin, const *gpio_pin_config_t* *config)

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or an output pin configuration.

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
```

(continues on next page)

(continued from previous page)

```

}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}

```

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- pin – GPIO port pin number
- config – GPIO pin configuration pointer

void GPIO_PinWrite(*gpio_port_num_t* port, uint8_t pin, uint8_t output)

Sets the output level of the multiple GPIO pins to the logic 1 or 0.

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- pin – GPIO pin number
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

void GPIO_PortSet(*gpio_port_num_t* port, uint8_t mask)

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- mask – GPIO pin number macro

void GPIO_PortClear(*gpio_port_num_t* port, uint8_t mask)

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB,etc) control registers, they handles four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~7 ... PTH 0 ~ 7. ...
- mask – GPIO pin number macro

void GPIO_PortToggle(*gpio_port_num_t* port, uint8_t mask)

Reverses the current output logic of the multiple GPIO pins.

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB, etc) control registers, they handle four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~ 7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~ 7 ... PTH 0 ~ 7. ...
- mask – GPIO pin number macro

uint32_t GPIO_PinRead(*gpio_port_num_t* port, uint8_t pin)

Reads the current input value of the GPIO port.

Parameters

- port – GPIO PORT number, see “gpio_port_num_t”. For each group GPIO (GPIOA, GPIOB, etc) control registers, they handle four PORT number controls. GPIOA serial registers –— PTA 0 ~ 7, PTB 0 ~ 7 ... PTD 0 ~ 7. GPIOB serial registers –— PTE 0 ~ 7, PTF 0 ~ 7 ... PTH 0 ~ 7. ...
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

2.10 I2C: Inter-Integrated Circuit Driver

2.11 I2C Driver

void I2C_MasterInit(I2C_Type *base, const *i2c_master_config_t* *masterConfig, uint32_t srcClock_Hz)

Initializes the I2C peripheral. Call this API to ungate the I2C clock and configure the I2C with master configuration.

Note: This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can be custom filled or it can be set with default values by using the I2C_MasterGetDefaultConfig(). After calling this API, the master is ready to transfer. This is an example.

```
i2c_master_config_t config = {
    .enableMaster = true,
    .enableStopHold = false,
    .highDrive = false,
    .baudRate_Bps = 100000,
    .glitchFilterWidth = 0
};
I2C_MasterInit(I2C0, &config, 12000000U);
```

Parameters

- base – I2C base pointer
- masterConfig – A pointer to the master configuration structure
- srcClock_Hz – I2C peripheral clock frequency in Hz

void I2C_SlaveInit(I2C_Type *base, const *i2c_slave_config_t* *slaveConfig, uint32_t srcClock_Hz)
Initializes the I2C peripheral. Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note: This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can partly be set with default values by I2C_SlaveGetDefaultConfig() or it can be custom filled by the user. This is an example.

```
i2c_slave_config_t config = {  
    .enableSlave = true,  
    .enableGeneralCall = false,  
    .addressingMode = kI2C_Address7bit,  
    .slaveAddress = 0x1DU,  
    .enableWakeUp = false,  
    .enablehighDrive = false,  
    .enableBaudRateCtl = false,  
    .sclStopHoldTime_ns = 4000  
};  
I2C_SlaveInit(I2C0, &config, 12000000U);
```

Parameters

- base – I2C base pointer
- slaveConfig – A pointer to the slave configuration structure
- srcClock_Hz – I2C peripheral clock frequency in Hz

void I2C_MasterDeinit(I2C_Type *base)
De-initializes the I2C master peripheral. Call this API to gate the I2C clock. The I2C master module can't work unless the I2C_MasterInit is called.

Parameters

- base – I2C base pointer

void I2C_SlaveDeinit(I2C_Type *base)
De-initializes the I2C slave peripheral. Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C_SlaveInit is called to enable the clock.

Parameters

- base – I2C base pointer

uint32_t I2C_GetInstance(I2C_Type *base)
Get instance number for I2C module.

Parameters

- base – I2C peripheral base address.

void I2C_MasterGetDefaultConfig(*i2c_master_config_t* *masterConfig)
Sets the I2C master configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in the I2C_MasterConfigure(). Use the initialized structure unchanged in the I2C_MasterConfigure() or modify the structure before calling the I2C_MasterConfigure(). This is an example.

```
i2c_master_config_t config;  
I2C_MasterGetDefaultConfig(&config);
```

Parameters

- masterConfig – A pointer to the master configuration structure.

void I2C_SlaveGetDefaultConfig(*i2c_slave_config_t* *slaveConfig)

Sets the I2C slave configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in the I2C_SlaveConfigure(). Modify fields of the structure before calling the I2C_SlaveConfigure(). This is an example.

```
i2c_slave_config_t config;
I2C_SlaveGetDefaultConfig(&config);
```

Parameters

- slaveConfig – A pointer to the slave configuration structure.

static inline void I2C_Enable(I2C_Type *base, bool enable)

Enables or disables the I2C peripheral operation.

Parameters

- base – I2C base pointer
- enable – Pass true to enable and false to disable the module.

uint32_t I2C_MasterGetStatusFlags(I2C_Type *base)

Gets the I2C status flags.

Parameters

- base – I2C base pointer

Returns

status flag, use status flag to AND `_i2c_flags` to get the related status.

static inline uint32_t I2C_SlaveGetStatusFlags(I2C_Type *base)

Gets the I2C status flags.

Parameters

- base – I2C base pointer

Returns

status flag, use status flag to AND `_i2c_flags` to get the related status.

static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)

Clears the I2C status flag state.

The following status register flags can be cleared `kI2C_ArbitrationLostFlag` and `kI2C_IntPendingFlag`.

Parameters

- base – I2C base pointer
- statusMask – The status flag mask, defined in type `i2c_status_flag_t`. The parameter can be any combination of the following values:
 - `kI2C_StartDetectFlag` (if available)
 - `kI2C_StopDetectFlag` (if available)
 - `kI2C_ArbitrationLostFlag`
 - `kI2C_IntPendingFlag`

```
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 kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag

Parameters

- base – I2C base pointer
- statusMask – The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values:
 - kI2C_StartDetectFlag (if available)
 - kI2C_StopDetectFlag (if available)
 - kI2C_ArbitrationLostFlag
 - kI2C_IntPendingFlagFlag

```
void I2C_EnableInterrupts(I2C_Type *base, uint32_t mask)
```

Enables I2C interrupt requests.

Parameters

- base – I2C base pointer
- mask – interrupt source The parameter can be combination of the following source if defined:
 - kI2C_GlobalInterruptEnable
 - kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable
 - kI2C_SdaTimeoutInterruptEnable

```
void I2C_DisableInterrupts(I2C_Type *base, uint32_t mask)
```

Disables I2C interrupt requests.

Parameters

- base – I2C base pointer
- mask – interrupt source The parameter can be combination of the following source if defined:
 - kI2C_GlobalInterruptEnable
 - kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable
 - kI2C_SdaTimeoutInterruptEnable

```
static inline void I2C_EnableDMA(I2C_Type *base, bool enable)
```

Enables/disables the I2C DMA interrupt.

Parameters

- base – I2C base pointer
- enable – true to enable, false to disable

```
static inline uint32_t I2C_GetDataRegAddr(I2C_Type *base)
```

Gets the I2C tx/rx data register address. This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

- base – I2C base pointer

Returns

data register address

void I2C_MasterSetBaudRate(I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
 Sets the I2C master transfer baud rate.

Parameters

- *base* – I2C base pointer
- *baudRate_Bps* – the baud rate value in bps
- *srcClock_Hz* – Source clock

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.

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 uint8_t *txBuff, size_t txSize, uint32_t flags)

Performs a polling send transaction on the I2C bus.

Parameters

- *base* – The I2C peripheral base pointer.
- *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 decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.

status_t I2C_MasterReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transaction on the I2C bus.

Note: The I2C_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

Parameters

- base – I2C peripheral base pointer.
- rxBuff – The pointer to the data to store the received data.
- rxSize – The length in bytes of the data to be received.
- flags – Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_Timeout – Send stop signal failed, timeout.

status_t I2C_SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)

Performs a polling send transaction on the I2C bus.

Parameters

- base – The I2C peripheral base pointer.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.

status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)

Performs a polling receive transaction on the I2C bus.

Parameters

- base – I2C peripheral base pointer.
- rxBuff – The pointer to the data to store the received data.
- rxSize – The length in bytes of the data to be received.

Return values

- kStatus_Success – Successfully complete data receive.

- `kStatus_I2C_Timeout` – Wait status flag timeout.

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)

Initializes the I2C handle which is used in transactional functions.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to *i2c_master_handle_t* structure to store the transfer state.
- `callback` – pointer to user callback function.
- `userData` – user parameter passed to the callback function.

status_t I2C_MasterTransferNonBlocking(I2C_Type *base, *i2c_master_handle_t* *handle,
i2c_master_transfer_t *xfer)

Performs a master interrupt non-blocking transfer on the I2C bus.

Note: Calling the API returns immediately after transfer initiates. The user needs to call `I2C_MasterGetTransferCount` to poll the transfer status to check whether the transfer is finished. If the return status is not `kStatus_I2C_Busy`, the transfer is finished.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to *i2c_master_handle_t* structure which stores the transfer state.
- `xfer` – pointer to *i2c_master_transfer_t* structure.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_I2C_Busy` – Previous transmission still not finished.
- `kStatus_I2C_Timeout` – Transfer error, wait signal timeout.

status_t I2C_MasterTransferGetCount(I2C_Type *base, *i2c_master_handle_t* *handle, size_t *count)

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- base – I2C base pointer.
- handle – pointer to *i2c_master_handle_t* structure which stores the transfer state.
- count – Number of bytes transferred so far by the non-blocking transaction.

Return values

- *kStatus_InvalidArgument* – count is Invalid.
- *kStatus_Success* – Successfully return the count.

status_t I2C_MasterTransferAbort(I2C_Type *base, *i2c_master_handle_t* *handle)

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – I2C base pointer.
- handle – pointer to *i2c_master_handle_t* structure which stores the transfer state

Return values

- *kStatus_I2C_Timeout* – Timeout during polling flag.
- *kStatus_Success* – Successfully abort the transfer.

void I2C_MasterTransferHandleIRQ(I2C_Type *base, void *i2cHandle)

Master interrupt handler.

Parameters

- base – I2C base pointer.
- i2cHandle – pointer to *i2c_master_handle_t* structure.

void I2C_SlaveTransferCreateHandle(I2C_Type *base, *i2c_slave_handle_t* *handle, *i2c_slave_transfer_callback_t* callback, void *userData)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – I2C base pointer.
- handle – pointer to *i2c_slave_handle_t* structure to store the transfer state.
- callback – pointer to user callback function.
- userData – user parameter passed to the callback function.

status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, *i2c_slave_handle_t* *handle, uint32_t eventMask)

Starts accepting slave transfers.

Call this API after calling the *I2C_SlaveInit()* and *I2C_SlaveTransferCreateHandle()* to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes

events to the callback that was passed into the call to `I2C_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 `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kLPI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, 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.

`void I2C_SlaveTransferAbort(I2C_Type *base, i2c_slave_handle_t *handle)`

Aborts the slave transfer.

Note: This API can be called at any time to stop slave for handling the bus events.

Parameters

- *base* – I2C base pointer.
- *handle* – pointer to `i2c_slave_handle_t` structure which stores the transfer state.

`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)`

Slave interrupt handler.

Parameters

- *base* – I2C base pointer.
- *i2cHandle* – pointer to `i2c_slave_handle_t` structure which stores the transfer state

FSL_I2C_DRIVER_VERSION

I2C driver version.

I2C status return codes.

Values:

enumerator kStatus_I2C_Busy

I2C is busy with current transfer.

enumerator kStatus_I2C_Idle

Bus is Idle.

enumerator kStatus_I2C_Nak

NAK received during transfer.

enumerator kStatus_I2C_ArbitrationLost

Arbitration lost during transfer.

enumerator kStatus_I2C_Timeout

Timeout polling status flags.

enumerator kStatus_I2C_Addr_Nak

NAK received during the address probe.

enum _i2c_flags

I2C peripheral flags.

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

Values:

enumerator kI2C_ReceiveNakFlag

I2C receive NAK flag.

enumerator kI2C_IntPendingFlag

I2C interrupt pending flag. This flag can be cleared.

enumerator kI2C_TransferDirectionFlag

I2C transfer direction flag.

enumerator kI2C_RangeAddressMatchFlag

I2C range address match flag.

enumerator kI2C_ArbitrationLostFlag

I2C arbitration lost flag. This flag can be cleared.

enumerator kI2C_BusBusyFlag

I2C bus busy flag.

enumerator kI2C_AddressMatchFlag

I2C address match flag.

enumerator kI2C_TransferCompleteFlag

I2C transfer complete flag.

enumerator kI2C_StopDetectFlag

I2C stop detect flag. This flag can be cleared.

enumerator kI2C_StartDetectFlag

I2C start detect flag. This flag can be cleared.

enum `_i2c_interrupt_enable`

I2C feature interrupt source.

Values:

enumerator `kI2C_GlobalInterruptEnable`

I2C global interrupt.

enumerator `kI2C_StopDetectInterruptEnable`

I2C stop detect interrupt.

enumerator `kI2C_StartStopDetectInterruptEnable`

I2C start&stop detect interrupt.

enum `_i2c_direction`

The direction of master and slave transfers.

Values:

enumerator `kI2C_Write`

Master transmits to the slave.

enumerator `kI2C_Read`

Master receives from the slave.

enum `_i2c_slave_address_mode`

Addressing mode.

Values:

enumerator `kI2C_Address7bit`

7-bit addressing mode.

enumerator `kI2C_RangeMatch`

Range address match addressing mode.

enum `_i2c_master_transfer_flags`

I2C transfer control flag.

Values:

enumerator `kI2C_TransferDefaultFlag`

A transfer starts with a start signal, stops with a stop signal.

enumerator `kI2C_TransferNoStartFlag`

A transfer starts without a start signal, only support write only or write+read with no start flag, do not support read only with no start flag.

enumerator `kI2C_TransferRepeatedStartFlag`

A transfer starts with a repeated start signal.

enumerator `kI2C_TransferNoStopFlag`

A transfer ends without a stop signal.

enum `_i2c_slave_transfer_event`

Set of events sent to the callback for nonblocking 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()` 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`

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

enumerator `kI2C_SlaveReceiveEvent`

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

enumerator `kI2C_SlaveTransmitAckEvent`

A callback needs to either transmit an ACK or NACK.

enumerator `kI2C_SlaveStartEvent`

A start/repeated start was detected.

enumerator `kI2C_SlaveCompletionEvent`

A stop was detected or finished transfer, completing the transfer.

enumerator `kI2C_SlaveGeneralCallEvent`

Received the general call address after a start or repeated start.

enumerator `kI2C_SlaveAllEvents`

A bit mask of all available events.

Common sets of flags used by the driver.

Values:

enumerator `kClearFlags`

All flags which are cleared by the driver upon starting a transfer.

enumerator `kIrqFlags`

typedef enum `_i2c_direction` `i2c_direction_t`

The direction of master and slave transfers.

typedef enum `_i2c_slave_address_mode` `i2c_slave_address_mode_t`

Addressing mode.

typedef enum `_i2c_slave_transfer_event` `i2c_slave_transfer_event_t`

Set of events sent to the callback for nonblocking 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()` 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_master_config` `i2c_master_config_t`

I2C master user configuration.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`

I2C slave user configuration.

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 status, void *userData)
```

I2C master transfer callback typedef.

```
typedef struct _i2c_slave_handle i2c_slave_handle_t
```

I2C slave handle typedef.

```
typedef struct _i2c_master_transfer i2c_master_transfer_t
```

I2C master transfer structure.

```
typedef struct _i2c_slave_transfer i2c_slave_transfer_t
```

I2C slave transfer structure.

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

I2C slave transfer callback typedef.

```
I2C_RETRY_TIMES
```

Retry times for waiting flag.

```
I2C_MASTER_FACK_CONTROL
```

Master Fast ack control, control if master needs to manually write ack, this is used to low the speed of transfer for SoCs with feature FSL_FEATURE_I2C_HAS_DOUBLE_BUFFERING.

```
I2C_HAS_STOP_DETECT
```

```
struct _i2c_master_config
```

#include <fsl_i2c.h> I2C master user configuration.

Public Members

```
bool enableMaster
```

Enables the I2C peripheral at initialization time.

```
bool enableStopHold
```

Controls the stop hold enable.

```
bool enableDoubleBuffering
```

Controls double buffer enable; notice that enabling the double buffer disables the clock stretch.

```
uint32_t baudRate_Bps
```

Baud rate configuration of I2C peripheral.

```
uint8_t glitchFilterWidth
```

Controls the width of the glitch.

```
struct _i2c_slave_config
```

#include <fsl_i2c.h> I2C slave user configuration.

Public Members

```
bool enableSlave
```

Enables the I2C peripheral at initialization time.

```
bool enableGeneralCall
```

Enables the general call addressing mode.

```
bool enableWakeUp
```

Enables/disables waking up MCU from low-power mode.

bool enableDoubleBuffering

Controls a double buffer enable; notice that enabling the double buffer disables the clock stretch.

bool enableBaudRateCtl

Enables/disables independent slave baud rate on SCL in very fast I2C modes.

uint16_t slaveAddress

A slave address configuration.

uint16_t upperAddress

A maximum boundary slave address used in a range matching mode.

i2c_slave_address_mode_t addressingMode

An addressing mode configuration of *i2c_slave_address_mode_config_t*.

uint32_t sclStopHoldTime_ns

the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.

struct *_i2c_master_transfer*

#include <fsl_i2c.h> I2C master transfer structure.

Public Members

uint32_t flags

A transfer flag which controls the transfer.

uint8_t slaveAddress

7-bit slave address.

i2c_direction_t direction

A transfer direction, read or write.

uint32_t subaddress

A sub address. Transferred MSB first.

uint8_t subaddressSize

A size of the command buffer.

uint8_t *volatile data

A transfer buffer.

volatile size_t dataSize

A transfer size.

struct *_i2c_master_handle*

#include <fsl_i2c.h> I2C master handle structure.

Public Members

i2c_master_transfer_t transfer

I2C master transfer copy.

size_t transferSize

Total bytes to be transferred.

uint8_t state

A transfer state maintained during transfer.

i2c_master_transfer_callback_t completionCallback

A callback function called when the transfer is finished.

void *userData

A callback parameter passed to the callback function.

struct __i2c_slave_transfer

#include <fsl_i2c.h> I2C slave transfer structure.

Public Members

i2c_slave_transfer_event_t event

A reason that the callback is invoked.

uint8_t *volatile data

A transfer buffer.

volatile size_t dataSize

A transfer size.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI2C_SlaveCompletionEvent.

size_t transferredCount

A number of bytes actually transferred since the start or since the last repeated start.

struct __i2c_slave_handle

#include <fsl_i2c.h> I2C slave handle structure.

Public Members

volatile bool isBusy

Indicates whether a transfer is busy.

i2c_slave_transfer_t transfer

I2C slave transfer copy.

uint32_t eventMask

A mask of enabled events.

i2c_slave_transfer_callback_t callback

A callback function called at the transfer event.

void *userData

A callback parameter passed to the callback.

2.12 Irq

uint32_t IRQ_GetInstance(IRQ_Type *base)

Get irq instance.

Parameters

- base – IRQ peripheral base pointer

Return values

Irq – instance number.

`void IRQ_Init(IRQ_Type *base, const irq_config_t *config)`

Initializes the IRQ pin used by the board.

To initialize the IRQ pin, define a `irq` configuration, specify whether enable pull-up, the edge and detect mode. Then, call the `IRQ_Init()` function.

This is an example to initialize `irq` configuration.

```
irq_config_t config =
{
    true,
    kIRQ_FallingEdgeorLowlevel,
    kIRQ_DetectOnEdgesOnly
}
```

Parameters

- `base` – IRQ peripheral base pointer
- `config` – IRQ configuration pointer

`void IRQ_Deinit(IRQ_Type *base)`

Deinitialize IRQ peripheral.

This function disables the IRQ clock.

Parameters

- `base` – IRQ peripheral base pointer.

Return values

None. –

`static inline void IRQ_Enable(IRQ_Type *base, bool enable)`

Enable/disable IRQ pin.

Parameters

- `base` – IRQ peripheral base pointer.
- `enable` – true to enable IRQ pin, else disable IRQ pin.

Return values

None. –

`static inline void IRQ_EnableInterrupt(IRQ_Type *base, bool enable)`

Enable/disable IRQ pin interrupt.

Parameters

- `base` – IRQ peripheral base pointer.
- `enable` – true to enable IRQ assert interrupt request, else disable.

Return values

None. –

`static inline void IRQ_ClearIRQFlag(IRQ_Type *base)`

Clear IRQF flag.

This function clears the IRQF flag.

Parameters

- `base` – IRQ peripheral base pointer.

Return values

None. –

```
static inline uint32_t IRQ_GetIRQFlag(IRQ_Type *base)
```

Get IRQF flag.

This function returns the IRQF flag.

Parameters

- base – IRQ peripheral base pointer.

Return values

status – = 0 IRQF flag deasserted. = 1 IRQF flag asserted.

```
FSL_IRQ_DRIVER_VERSION
```

Version 2.0.2.

```
enum _irq_edge
```

Interrupt Request (IRQ) Edge Select.

Values:

```
enumerator kIRQ_FallingEdgeorLowlevel
```

IRQ is falling-edge or falling-edge/low-level sensitive

```
enumerator kIRQ_RisingEdgeorHighlevel
```

IRQ is rising-edge or rising-edge/high-level sensitive

```
enum _irq_mode
```

Interrupt Request (IRQ) Detection Mode.

Values:

```
enumerator kIRQ_DetectOnEdgesOnly
```

IRQ event is detected only on falling/rising edges

```
enumerator kIRQ_DetectOnEdgesAndEdges
```

IRQ event is detected on falling/rising edges and low/high levels

```
typedef enum _irq_edge irq_edge_t
```

Interrupt Request (IRQ) Edge Select.

```
typedef enum _irq_mode irq_mode_t
```

Interrupt Request (IRQ) Detection Mode.

```
typedef struct _irq_config irq_config_t
```

The IRQ pin configuration structure.

```
struct _irq_config
```

#include <fsl_irq.h> The IRQ pin configuration structure.

Public Members

```
bool enablePullDevice
```

Enable/disable the internal pullup device when the IRQ pin is enabled

```
irq_edge_t edgeSelect
```

Select the polarity of edges or levels on the IRQ pin that cause IRQF to be set

```
irq_mode_t detectMode
```

select either edge-only detection or edge-and-level detection

2.13 IRQ: external interrupt (IRQ) module

2.14 KBI: Keyboard interrupt Driver

`void KBI_Init(KBI_Type *base, kbi_config_t *configure)`

KBI initialize. This function ungates the KBI clock and initializes KBI. This function must be called before calling any other KBI driver functions.

Parameters

- `base` – KBI peripheral base address.
- `configure` – The KBI configuration structure pointer.

`void KBI_Deinit(KBI_Type *base)`

Deinitializes the KBI module and gates the clock. This function gates the KBI clock. As a result, the KBI module doesn't work after calling this function.

Parameters

- `base` – KBI peripheral base address.

`static inline void KBI_EnableInterrupts(KBI_Type *base)`

Enables the interrupt.

Parameters

- `base` – KBI peripheral base address.

`static inline void KBI_DisableInterrupts(KBI_Type *base)`

Disables the interrupt.

Parameters

- `base` – KBI peripheral base address.

`static inline bool KBI_IsInterruptRequestDetected(KBI_Type *base)`

Gets the KBI interrupt event status.

Parameters

- `base` – KBI peripheral base address.

Returns

The status of the KBI interrupt request is detected.

`static inline void KBI_ClearInterruptFlag(KBI_Type *base)`

Clears KBI status flag.

Parameters

- `base` – KBI peripheral base address.

`static inline uint32_t KBI_GetSourcePinStatus(KBI_Type *base)`

Gets the KBI Source pin status.

Parameters

- `base` – KBI peripheral base address.

Returns

The status indicates the active pin defined as keyboard interrupt which is pushed.

`FSL_KBI_DRIVER_VERSION`

KBI driver version.

enum *_kbi_detect_mode*

KBI detection mode.

Values:

enumerator *kKBI_EdgesDetect*

The keyboard detects edges only.

enumerator *kKBI_EdgesLevelDetect*

The keyboard detects both edges and levels.

typedef uint32_t *kbi_reg_t*

typedef enum *_kbi_detect_mode* *kbi_detect_mode_t*

KBI detection mode.

typedef struct *_kbi_config* *kbi_config_t*

KBI configuration.

struct *_kbi_config*

#include <fsl_kbi.h> KBI configuration.

Public Members

uint32_t *pinsEnabled*

The eight kbi pins, set 1 to enable the corresponding KBI interrupt pins.

uint32_t *pinsEdge*

The edge selection for each kbi pin: 1 — rising edge, 0 — falling edge.

kbi_detect_mode_t *mode*

The kbi detection mode.

2.15 Common Driver

FSL_COMMON_DRIVER_VERSION

common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE

No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART

Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART

Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI

Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(addr, clearBits, setBits)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(addr, expected, newValue)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true**, else return **false**.

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_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 SDSPIC 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_CODEEC
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_SDIO SLV
Group number for SDIO SLV 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_QUEUE DSPI
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().

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.

2.16 MCM: Miscellaneous Control Module

FSL_MCM_DRIVER_VERSION

MCM driver version.

Enum `_mcm_interrupt_flag`. Interrupt status flag mask. .

Values:

enumerator `kMCM_CacheWriteBuffer`

Cache Write Buffer Error Enable.

enumerator `kMCM_ParityError`

Cache Parity Error Enable.

enumerator `kMCM_FPUInvalidOperation`

FPU Invalid Operation Interrupt Enable.

enumerator `kMCM_FPUDivideByZero`

FPU Divide-by-zero Interrupt Enable.

enumerator `kMCM_FPUOverflow`

FPU Overflow Interrupt Enable.

enumerator `kMCM_FPUUnderflow`

FPU Underflow Interrupt Enable.

enumerator `kMCM_FPUInexact`

FPU Inexact Interrupt Enable.

enumerator `kMCM_FPUInputDenormalInterrupt`

FPU Input Denormal Interrupt Enable.

typedef union `_mcm_buffer_fault_attribute` `mcm_buffer_fault_attribute_t`

The union of buffer fault attribute.

typedef union `_mcm_lmem_fault_attribute` `mcm_lmem_fault_attribute_t`

The union of LMEM fault attribute.

static inline void `MCM_EnableCrossbarRoundRobin(MCM_Type *base, bool enable)`

Enables/Disables crossbar round robin.

Parameters

- `base` – MCM peripheral base address.
- `enable` – Used to enable/disable crossbar round robin.
 - **true** Enable crossbar round robin.
 - **false** disable crossbar round robin.

static inline void MCM_EnableInterruptStatus(MCM_Type *base, uint32_t mask)

Enables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(`_mcm_interrupt_flag`).

static inline void MCM_DisableInterruptStatus(MCM_Type *base, uint32_t mask)

Disables the interrupt.

Parameters

- base – MCM peripheral base address.
- mask – Interrupt status flags mask(`_mcm_interrupt_flag`).

static inline uint16_t MCM_GetInterruptStatus(MCM_Type *base)

Gets the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline void MCM_ClearCacheWriteBufferErrorStatus(MCM_Type *base)

Clears the Interrupt status .

Parameters

- base – MCM peripheral base address.

static inline uint32_t MCM_GetBufferFaultAddress(MCM_Type *base)

Gets buffer fault address.

Parameters

- base – MCM peripheral base address.

static inline void MCM_GetBufferFaultAttribute(MCM_Type *base, *mcm_buffer_fault_attribute_t* *bufferfault)

Gets buffer fault attributes.

Parameters

- base – MCM peripheral base address.
- bufferfault – Structure to store the result.

static inline uint32_t MCM_GetBufferFaultData(MCM_Type *base)

Gets buffer fault data.

Parameters

- base – MCM peripheral base address.

static inline void MCM_LimitCodeCachePeripheralWriteBuffering(MCM_Type *base, bool enable)

Limit code cache peripheral write buffering.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable limit code cache peripheral write buffering.
 - **true** Enable limit code cache peripheral write buffering.
 - **false** disable limit code cache peripheral write buffering.

static inline void MCM_BypassFixedCodeCacheMap(MCM_Type *base, bool enable)
Bypass fixed code cache map.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable bypass fixed code cache map.
 - **true** Enable bypass fixed code cache map.
 - **false** disable bypass fixed code cache map.

static inline void MCM_EnableCodeBusCache(MCM_Type *base, bool enable)
Enables/Disables code bus cache.

Parameters

- base – MCM peripheral base address.
- enable – Used to disable/enable code bus cache.
 - **true** Enable code bus cache.
 - **false** disable code bus cache.

static inline void MCM_ForceCodeCacheToNoAllocation(MCM_Type *base, bool enable)
Force code cache to no allocation.

Parameters

- base – MCM peripheral base address.
- enable – Used to force code cache to allocation or no allocation.
 - **true** Force code cache to no allocation.
 - **false** Force code cache to allocation.

static inline void MCM_EnableCodeCacheWriteBuffer(MCM_Type *base, bool enable)
Enables/Disables code cache write buffer.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable code cache write buffer.
 - **true** Enable code cache write buffer.
 - **false** Disable code cache write buffer.

static inline void MCM_ClearCodeBusCache(MCM_Type *base)
Clear code bus cache.

Parameters

- base – MCM peripheral base address.

static inline void MCM_EnablePcParityFaultReport(MCM_Type *base, bool enable)
Enables/Disables PC Parity Fault Report.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity Fault Report.
 - **true** Enable PC Parity Fault Report.
 - **false** disable PC Parity Fault Report.

```
static inline void MCM_EnablePcParity(MCM_Type *base, bool enable)
```

Enables/Disables PC Parity.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable PC Parity.
 - **true** Enable PC Parity.
 - **false** disable PC Parity.

```
static inline void MCM_LockConfigState(MCM_Type *base)
```

Lock the configuration state.

Parameters

- base – MCM peripheral base address.

```
static inline void MCM_EnableCacheParityReporting(MCM_Type *base, bool enable)
```

Enables/Disables cache parity reporting.

Parameters

- base – MCM peripheral base address.
- enable – Used to enable/disable cache parity reporting.
 - **true** Enable cache parity reporting.
 - **false** disable cache parity reporting.

```
static inline uint32_t MCM_GetLmemFaultAddress(MCM_Type *base)
```

Gets LMEM fault address.

Parameters

- base – MCM peripheral base address.

```
static inline void MCM_GetLmemFaultAttribute(MCM_Type *base, mcm_lmem_fault_attribute_t  
                                             *lmemFault)
```

Get LMEM fault attributes.

Parameters

- base – MCM peripheral base address.
- lmemFault – Structure to store the result.

```
static inline uint64_t MCM_GetLmemFaultData(MCM_Type *base)
```

Gets LMEM fault data.

Parameters

- base – MCM peripheral base address.

MCM_LMFATR_TYPE_MASK

MCM_LMFATR_MODE_MASK

MCM_LMFATR_BUFF_MASK

MCM_LMFATR_CACH_MASK

MCM_ISCR_STAT_MASK

FSL_COMPONENT_ID

```
union __mcm_buffer_fault_attribute
```

#include <fsl_mcm.h> The union of buffer fault attribute.

Public Members

uint32_t attribute

Indicates the faulting attributes, when a properly-enabled cache write buffer error interrupt event is detected.

struct *_mcm_buffer_fault_attribute._mcm_buffer_fault_attrib* attribute_memory

struct *_mcm_buffer_fault_attrib*

#include <fsl_mcm.h>

Public Members

uint32_t busErrorDataAccessType

Indicates the type of cache write buffer access.

uint32_t busErrorPrivilegeLevel

Indicates the privilege level of the cache write buffer access.

uint32_t busErrorSize

Indicates the size of the cache write buffer access.

uint32_t busErrorAccess

Indicates the type of system bus access.

uint32_t busErrorMasterID

Indicates the crossbar switch bus master number of the captured cache write buffer bus error.

uint32_t busErrorOverrun

Indicates if another cache write buffer bus error is detected.

union *_mcm_lmem_fault_attribute*

#include <fsl_mcm.h> The union of LMEM fault attribute.

Public Members

uint32_t attribute

Indicates the attributes of the LMEM fault detected.

struct *_mcm_lmem_fault_attribute._mcm_lmem_fault_attrib* attribute_memory

struct *_mcm_lmem_fault_attrib*

#include <fsl_mcm.h>

Public Members

uint32_t parityFaultProtectionSignal

Indicates the features of parity fault protection signal.

uint32_t parityFaultMasterSize

Indicates the parity fault master size.

uint32_t parityFaultWrite

Indicates the parity fault is caused by read or write.

uint32_t backdoorAccess

Indicates the LMEM access fault is initiated by core access or backdoor access.

`uint32_t parityFaultSyndrome`
Indicates the parity fault syndrome.

`uint32_t overrun`
Indicates the number of faultss.

2.17 PIT: Periodic Interrupt Timer

`void PIT_Init(PIT_Type *base, const pit_config_t *config)`

Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.

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

Parameters

- `base` – PIT peripheral base address
- `config` – Pointer to the user's PIT config structure

`void PIT_Deinit(PIT_Type *base)`

Gates the PIT clock and disables the PIT module.

Parameters

- `base` – PIT peripheral base address

`static inline void PIT_GetDefaultConfig(pit_config_t *config)`

Fills in the PIT configuration structure with the default settings.

The default values are as follows.

```
config->enableRunInDebug = false;
```

Parameters

- `config` – Pointer to the configuration structure.

`static inline void PIT_SetTimerChainMode(PIT_Type *base, pit_chnl_t channel, bool enable)`

Enables or disables chaining a timer with the previous timer.

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer `n-1` has counted down to 0, counter `n` decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number which is chained with the previous timer
- `enable` – Enable or disable chain. `true`: Current timer is chained with the previous timer. `false`: Timer doesn't chain with other timers.

`static inline void PIT_EnableInterrupts(PIT_Type *base, pit_chnl_t channel, uint32_t mask)`

Enables the selected PIT interrupts.

Parameters

- `base` – PIT peripheral base address

- `channel` – Timer channel number
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline void PIT_DisableInterrupts(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)
Disables the selected PIT interrupts.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number
- `mask` – The interrupts to disable. This is a logical OR of members of the enumeration `pit_interrupt_enable_t`

static inline uint32_t PIT_GetEnabledInterrupts(PIT_Type *base, *pit_chnl_t* channel)
Gets the enabled PIT interrupts.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number

Returns

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

static inline uint32_t PIT_GetStatusFlags(PIT_Type *base, *pit_chnl_t* channel)
Gets the PIT status flags.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number

Returns

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

static inline void PIT_ClearStatusFlags(PIT_Type *base, *pit_chnl_t* channel, uint32_t mask)
Clears the PIT status flags.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `pit_status_flags_t`

static inline void PIT_SetTimerPeriod(PIT_Type *base, *pit_chnl_t* channel, uint32_t count)
Sets the timer period in units of count.

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note: Users can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

- `base` – PIT peripheral base address

- `channel` – Timer channel number
- `count` – Timer period in units of ticks

static inline uint32_t PIT_GetCurrentTimerCount(PIT_Type *base, *pit_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: Users can call the utility macros provided in `fsl_common.h` to convert ticks to usec or msec.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number

Returns

Current timer counting value in ticks

static inline void PIT_StartTimer(PIT_Type *base, *pit_chnl_t* channel)

Starts the timer counting.

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number.

static inline void PIT_StopTimer(PIT_Type *base, *pit_chnl_t* channel)

Stops the timer counting.

This function stops every timer counting. Timers reload their periods respectively after the next time they call the `PIT_DRV_StartTimer`.

Parameters

- `base` – PIT peripheral base address
- `channel` – Timer channel number.

FSL_PIT_DRIVER_VERSION

PIT Driver Version 2.2.0.

enum `_pit_chnl`

List of PIT channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator `kPIT_Chnl_0`

PIT channel number 0

enumerator `kPIT_Chnl_1`

PIT channel number 1

enumerator `kPIT_Chnl_2`

PIT channel number 2

enumerator kPIT_Chnl_3
PIT channel number 3

enum __pit_interrupt_enable
List of PIT interrupts.

Values:

enumerator kPIT_TimerInterruptEnable
Timer interrupt enable

enum __pit_status_flags
List of PIT status flags.

Values:

enumerator kPIT_TimerFlag
Timer flag

typedef enum __pit_chnl pit_chnl_t
List of PIT channels.

Note: Actual number of available channels is SoC dependent

typedef enum __pit_interrupt_enable pit_interrupt_enable_t
List of PIT interrupts.

typedef enum __pit_status_flags pit_status_flags_t
List of PIT status flags.

typedef struct __pit_config pit_config_t
PIT configuration structure.

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

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

uint64_t PIT_GetLifetimeTimerCount(PIT_Type *base)
Reads the current lifetime counter value.

The lifetime timer is a 64-bit timer which chains timer 0 and timer 1 together. Timer 0 and 1 are chained by calling the PIT_SetTimerChainMode before using this timer. The period of lifetime timer is equal to the “period of timer 0 * period of timer 1”. For the 64-bit value, the higher 32-bit has the value of timer 1, and the lower 32-bit has the value of timer 0.

Parameters

- base – PIT peripheral base address

Returns

Current lifetime timer value

struct __pit_config
#include <fsl_pit.h> PIT configuration structure.

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

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

Public Members

bool enableRunInDebug

true: Timers run in debug mode; false: Timers stop in debug mode

2.18 PORT Driver

enum __port_module_t

Module or peripheral for port pin selection.

Values:

enumerator kPORT_NMI

NMI port pin select.

enumerator kPORT_RESET

RESET pin select.

enumerator kPORT_SWDE

Single wire debug port pin.

enumerator kPORT_RTC

RTCO port pin select.

enumerator kPORT_I2C0

I2C0 Port pin select.

enumerator kPORT_SPI0

SPI0 port pin select.

enumerator kPORT_UART0

UART0 port pin select.

enumerator kPORT_FTM0CH0

FTM0_CH0 port pin select.

enumerator kPORT_FTM0CH1

FTM0_CH1 port pin select.

enumerator kPORT_FTM1CH0

FTM1_CH0 port pin select.

enumerator kPORT_FTM1CH1

FTM1_CH1 port pin select.

enumerator kPORT_FTM2CH0

FTM2_CH0 port pin select.

enumerator kPORT_FTM2CH1

FTM2_CH1 port pin select.

enumerator kPORT_FTM2CH2

FTM2_CH2 port pin select.

enumerator kPORT_FTM2CH3

FTM2_CH3 port pin select.

enum __port_type_t

Port type.

Values:

enumerator kPORT_PTA
PORT PTA.

enumerator kPORT_PTB
PORT PTB.

enumerator kPORT_PTC
PORT PTC.

enumerator kPORT_PTD
PORT PTD.

enumerator kPORT_PTE
PORT PTE.

enumerator kPORT_PTF
PORT PTF.

enumerator kPORT_PTG
PORT PTG.

enumerator kPORT_PTH
PORT PTH.

enum __port_pin_index_t

Pin number, Notice this index enum has been deprecated and it will be removed in the next release.

Values:

enumerator kPORT_PinIdx0
PORT PIN index 0.

enumerator kPORT_PinIdx1
PORT PIN index 1.

enumerator kPORT_PinIdx2
PORT PIN index 2.

enumerator kPORT_PinIdx3
PORT PIN index 3.

enumerator kPORT_PinIdx4
PORT PIN index 4.

enumerator kPORT_PinIdx5
PORT PIN index 5.

enumerator kPORT_PinIdx6
PORT PIN index 6.

enumerator kPORT_PinIdx7
PORT PIN index 7.

enum __port_pin_select_t

Pin selection.

Values:

enumerator kPORT_NMI_OTHERS
PTB4/FTM2_CH4 etc function as PTB4/FTM2_CH4 etc

enumerator kPORT_NMI_NMIE
PTB4/FTM2_CH4 etc function as NMI.

enumerator kPORT_RST_OTHERS

PTA5/IRQ etc function as PTA5/IRQ etc.

enumerator kPORT_RST_RSTPE

PTA5/IRQ etc function as REST.

enumerator kPORT_SWDE_OTHERS

PTA4/ACMP0 etc function as PTA4/ACMP0 etc.

enumerator kPORT_SWDE_SWDE

PTA4/ACMP0 etc function as SWD.

enumerator kPORT_RTCO_PTC4

RTCO is mapped to PTC4.

enumerator kPORT_RTCO_PTC5

RTCO is mapped to PTC5.

enumerator kPORT_I2C0_SCLPTA3_SDAPTA2

I2C0_SCL and I2C0_SDA are mapped on PTA3 and PTA2, respectively.

enumerator kPORT_I2C0_SCLPTB7_SDAPTB6

I2C0_SCL and I2C0_SDA are mapped on PTB7 and PTB6, respectively.

enumerator kPORT_SPI0_SCKPTB2_MOSIPTB3_MISOPTB4_PCSPTB5

SPI0_SCK/MOSI/MISO/PCS0 are mapped on PTB2/PTB3/PTB4/PTB5.

enumerator kPORT_SPI0_SCKPTE0_MOSIPTE1_MISOPTE2_PCSPTE3

SPI0_SCK/MOSI/MISO/PCS0 are mapped on PTE0/PTE1/PTE2/PTE3.

enumerator kPORT_UART0_RXPTB0_TXPTB1

UART0_RX and UART0_TX are mapped on PTB0 and PTB1.

enumerator kPORT_UART0_RXPTA2_TXPTA3

UART0_RX and UART0_TX are mapped on PTA2 and PTA3.

enumerator kPORT_FTM0_CH0_PTA0

FTM0_CH0 channels are mapped on PTA0.

enumerator kPORT_FTM0_CH0_PTB2

FTM0_CH0 channels are mapped on PTB2.

enumerator kPORT_FTM0_CH1_PTA1

FTM0_CH1 channels are mapped on PTA1.

enumerator kPORT_FTM0_CH1_PTB3

FTM0_CH1 channels are mapped on PTB3.

enumerator kPORT_FTM1_CH0_PTC4

FTM1_CH0 channels are mapped on PTC4.

enumerator kPORT_FTM1_CH0_PTH2

FTM1_CH0 channels are mapped on PTH2.

enumerator kPORT_FTM1_CH1_PTC5

FTM1_CH1 channels are mapped on PTC5.

enumerator kPORT_FTM1_CH1_PTE7

FTM1_CH1 channels are mapped on PTE7.

enumerator kPORT_FTM2_CH0_PTC0

FTM2_CH0 channels are mapped on PTC0.

enumerator kPORT_FTM2_CH0_PTH0

FTM2_CH0 channels are mapped on PTH0.

enumerator kPORT_FTM2_CH1_PTC1

FTM2_CH1 channels are mapped on PTC1.

enumerator kPORT_FTM2_CH1_PTH1

FTM2_CH1 channels are mapped on PTH1.

enumerator kPORT_FTM2_CH2_PTC2

FTM2_CH2 channels are mapped on PTC2.

enumerator kPORT_FTM2_CH2_PTD0

FTM2_CH2 channels are mapped on PTD0.

enumerator kPORT_FTM2_CH3_PTC3

FTM2_CH3 channels are mapped on PTC3.

enumerator kPORT_FTM2_CH3_PTD1

FTM2_CH3 channels are mapped on PTD1.

enum _port_filter_pin_t

The PORT pins for input glitch filter configure.

Values:

enumerator kPORT_FilterPTA

Filter for input from PTA.

enumerator kPORT_FilterPTB

Filter for input from PTB.

enumerator kPORT_FilterPTC

Filter for input from PTC.

enumerator kPORT_FilterPTD

Filter for input from PTD.

enumerator kPORT_FilterPTE

Filter for input from PTE.

enumerator kPORT_FilterPTF

Filter for input from PTF.

enumerator kPORT_FilterPTG

Filter for input from PTG.

enumerator kPORT_FilterPTH

Filter for input from PTH.

enumerator kPORT_FilterRST

Filter for input from RESET/IRQ.

enumerator kPORT_FilterKBI0

Filter for input from KBI0.

enumerator kPORT_FilterKBI1

Filter for input from KBI1.

enumerator kPORT_FilterNMI

Filter for input from NMI.

enum `_port_filter_select_t`

The Filter selection for input pins.

Values:

enumerator `kPORT_BUSCLK_OR_NOFILTER`

Filter section BUSCLK for PTA~PTH, No filter for REST/KBI0/KBI1/NMI.

enumerator `kPORT_FILTERDIV1`

Filter Division Set 1.

enumerator `kPORT_FILTERDIV2`

Filter Division Set 2.

enumerator `kPORT_FILTERDIV3`

Filter Division Set 3.

enum `_port_highdrive_pin_t`

Port pin for high driver enable/disable control.

Values:

enumerator `kPORT_HighDrive_PTB4`

PTB4.

enumerator `kPORT_HighDrive_PTB5`

PTB5.

enumerator `kPORT_HighDrive_PTD0`

PTD0.

enumerator `kPORT_HighDrive_PTD1`

PTD1.

enumerator `kPORT_HighDrive_PTE0`

PTE0.

enumerator `kPORT_HighDrive_PTE1`

PTE1.

enumerator `kPORT_HighDrive_PTH0`

PTH0.

enumerator `kPORT_HighDrive_PTH1`

PTH1.

typedef enum `_port_module_t` `port_module_t`

Module or peripheral for port pin selection.

typedef enum `_port_type_t` `port_type_t`

Port type.

typedef enum `_port_pin_index_t` `port_pin_index_t`

Pin number, Notice this index enum has been deprecated and it will be removed in the next release.

typedef enum `_port_pin_select_t` `port_pin_select_t`

Pin selection.

typedef enum `_port_filter_pin_t` `port_filter_pin_t`

The PORT pins for input glitch filter configure.

typedef enum `_port_filter_select_t` `port_filter_select_t`

The Filter selection for input pins.

```
typedef enum _port_highdrive_pin_t port_highdrive_pin_t
```

Port pin for high driver enable/disable control.

```
FSL_PORT_DRIVER_VERSION
```

Version 2.0.2.

```
FSL_PORT_FILTER_SELECT_BITMASK
```

The IOFLT Filter selection bit mask .

```
void PORT_SetPinSelect(port_module_t module, port_pin_select_t pin)
```

Selects pin for modules.

This API is used to select the port pin for the module with multiple port pin selection. For example the FTM Channel 0 can be mapped to ether PTA0 or PTB2. Select FTM channel 0 map to PTA0 port pin as:

```
PORT_SetPinSelect(kPORT_FTM0CH0, kPORT_FTM0_CH0_PTA0);
```

If you want to select a specified ALT for a given port pin, please add two more steps after calling PORT_SetPinSelect:

- Enable module or the port control in the module for the ALT you want to select. For I2C ALT feature:all port enable is controlled by the module enable, so set IICEN in I2CX_C1 to enable the port pins for I2C feature. For KBI ALT feature:each port pin is controlled independently by each bit in KBIx_PE. set related bit in this register to enable the KBI feature in the port pin.
- Make sure there is no module enabled with higher priority than the ALT module feature you want to select.

Note: This API doesn't support to select specified ALT for a given port pin. The ALT feature is automatically selected by hardware according to the ALT priority: Low —> high: Alt1, Alt2, ... when peripheral modules has been enabled.

Parameters

- module – Modules for pin selection. For NMI/RST module are write-once attribute after reset.
- pin – Port pin selection for modules.

```
static inline void PORT_SetFilterSelect(PORT_Type *base, port_filter_pin_t port,  
                                       port_filter_select_t filter)
```

Selects the glitch filter for input pins.

Parameters

- base – PORT peripheral base pointer.
- port – PORT pin, see “port_filter_pin_t”.
- filter – Filter select, see “port_filter_select_t”.

```
static inline void PORT_SetFilterDIV1WidthThreshold(PORT_Type *base, uint8_t threshold)
```

Sets the width threshold for glitch filter division set 1. ‘.

Parameters

- base – PORT peripheral base pointer.
- threshold – PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - LPOCLK 1 - LPOCLK/2 2 - LPOCLK/4 3 - LPOCLK/8 4 - LPOCLK/16 5 - LPOCLK/32 6 - LPOCLK/64 7 - LPOCLK/128

static inline void PORT_SetFilterDIV2WidthThreshold(PORT_Type *base, uint8_t threshold)
Sets the width threshold for glitch filter division set 2. ‘.

Parameters

- base – PORT peripheral base pointer.
- threshold – PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - BUSCLK/32 1 - BUSCLK/64 2 - BUSCLK/128 3 - BUSCLK/256 4 - BUSCLK/512 5 - BUSCLK/1024 6 - BUSCLK/2048 7 - BUSCLK/4096

static inline void PORT_SetFilterDIV3WidthThreshold(PORT_Type *base, uint8_t threshold)
Sets the width threshold for glitch filter division set 3. ‘.

Parameters

- base – PORT peripheral base pointer.
- threshold – PORT glitch filter width threshold, take refer to reference manual for detail information. 0 - BUSCLK/2 1 - BUSCLK/4 2 - BUSCLK/8 3 - BUSCLK/16

void PORT_SetPinPullUpEnable(PORT_Type *base, port_type_t port, uint8_t num, bool enable)
Enables or disables the port pull up.

Parameters

- base – PORT peripheral base pointer.
- port – PORT type, such as PTA/PTB/PTC etc, see “port_type_t”.
- num – PORT Pin number, such as 0, 1, 2.... There are seven pins not exists in this device: PTG: PTG4, PTG5, PTG6, PTG7. PTH: PTH3, PTH4, PTH5. so, when set PTG, and PTH, please don’t set the pins mentioned above. Please take refer to the reference manual.
- enable – Enable or disable the pull up feature switch.

static inline void PORT_SetHighDriveEnable(PORT_Type *base, port_highdrive_pin_t pin, bool enable)

Set High drive for port pins.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin support high drive.
- enable – Enable or disable the high driver feature switch.

2.19 RTC: Real Time Clock

void RTC_Init(RTC_Type *base, const rtc_config_t *config)
Ungates the RTC clock and configures the peripheral for basic operation.

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

Parameters

- base – RTC peripheral base address
- config – Pointer to the user’s RTC configuration structure.

void RTC_Deinit(RTC_Type *base)

Stops the timer and gate the RTC clock.

Parameters

- base – RTC peripheral base address

void RTC_GetDefaultConfig(*rtc_config_t* *config)

Fills in the RTC config struct with the default settings.

The default values are as follows.

```
config->clockSource = kRTC_BusClock;
config->prescaler = kRTC_ClockDivide_16_2048;
config->time_us = 1000000U;
```

Parameters

- config – Pointer to the user's RTC configuration structure.

status_t RTC_SetDatetime(*rtc_datetime_t* *datetime)

Sets the RTC date and time according to the given time structure.

Parameters

- datetime – Pointer to the structure where the date and time details are stored.

Returns

kStatus_Success: Success in setting the time and starting the RTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

void RTC_GetDatetime(*rtc_datetime_t* *datetime)

Gets the RTC time and stores it in the given time structure.

Parameters

- datetime – Pointer to the structure where the date and time details are stored.

void RTC_SetAlarm(uint32_t second)

Sets the RTC alarm time.

Parameters

- second – Second value. User input the number of second. After seconds user input, alarm occurs.

void RTC_GetAlarm(*rtc_datetime_t* *datetime)

Returns the RTC alarm time.

Parameters

- datetime – Pointer to the structure where the alarm date and time details are stored.

void RTC_SetAlarmCallback(*rtc_alarm_callback_t* callback)

Set the RTC alarm callback.

Parameters

- callback – The callback function.

static inline void RTC_SelectSourceClock(RTC_Type *base, *rtc_clock_source_t* clock,
rtc_clock_prescaler_t divide)

Select Real-Time Clock Source and Clock Prescaler.

Parameters

- base – RTC peripheral base address
- clock – Select RTC clock source
- divide – Select RTC clock prescaler value

uint32_t RTC_GetDivideValue(RTC_Type *base)

Get the RTC Divide value.

Note: This API should be called after selecting clock source and clock prescaler.

Parameters

- base – RTC peripheral base address

Returns

The Divider value. The Divider value depends on clock source and clock prescaler

static inline void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected RTC interrupts.

Parameters

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

static inline void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected RTC interrupts.

Parameters

- base – PIT peripheral base address
- mask – The interrupts to disable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

static inline uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)

Gets the enabled RTC interrupts.

Parameters

- base – RTC peripheral base address

Returns

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

static inline uint32_t RTC_GetInterruptFlags(RTC_Type *base)

Gets the RTC interrupt flags.

Parameters

- base – RTC peripheral base address

Returns

The interrupt flags. This is the logical OR of members of the enumeration rtc_interrupt_flags_t

static inline void RTC_ClearInterruptFlags(RTC_Type *base, uint32_t mask)

Clears the RTC interrupt flags.

Parameters

- base – RTC peripheral base address

- **mask** – The interrupt flags to clear. This is a logical OR of members of the enumeration `rtc_interrupt_flags_t`

static inline void RTC_EnableOutput(RTC_Type *base, uint32_t mask)

Enable the RTC output. If RTC output is enabled, the RTCO pinout will be toggled when RTC counter overflows.

Parameters

- **base** – RTC peripheral base address
- **mask** – The Output to enable. This is a logical OR of members of the enumeration `rtc_output_enable_t`

static inline void RTC_DisableOutput(RTC_Type *base, uint32_t mask)

Disable the RTC output.

Parameters

- **base** – RTC peripheral base address
- **mask** – The Output to disable. This is a logical OR of members of the enumeration `rtc_output_enable_t`

static inline void RTC_SetModuloValue(RTC_Type *base, uint32_t value)

Set the RTC module value.

Parameters

- **base** – RTC peripheral base address
- **value** – The Module Value. The RTC Modulo register allows the compare value to be set to any value from 0x0000 to 0xFFFF

static inline uint16_t RTC_GetCountValue(RTC_Type *base)

Get the RTC Count value.

Parameters

- **base** – RTC peripheral base address

Returns

The Count Value. The Count Value is allowed from 0x0000 to 0xFFFF

FSL_RTC_DRIVER_VERSION

Version 2.0.6

enum _rtc_clock_source

List of RTC clock source.

Values:

enumerator kRTC_ExternalClock

External clock source

enumerator kRTC_LPOCLK

Real-time clock source is 1 kHz (LPOCLK)

enumerator kRTC_ICSIKCLK

Internal reference clock (ICSIKCLK)

enumerator kRTC_BusClock

Bus clock

enum _rtc_clock_prescaler

List of RTC clock prescaler.

Values:

enumerator kRTC_ClockDivide_off

Off

enumerator kRTC_ClockDivide_1_128

If RTCLKS = x0, it is 1; if RTCLKS = x1, it is 128

enumerator kRTC_ClockDivide_2_256

If RTCLKS = x0, it is 2; if RTCLKS = x1, it is 256

enumerator kRTC_ClockDivide_4_512

If RTCLKS = x0, it is 4; if RTCLKS = x1, it is 512

enumerator kRTC_ClockDivide_8_1024

If RTCLKS = x0, it is 8; if RTCLKS = x1, it is 1024

enumerator kRTC_ClockDivide_16_2048

If RTCLKS = x0, it is 16; if RTCLKS = x1, it is 2048

enumerator kRTC_ClockDivide_32_100

If RTCLKS = x0, it is 32; if RTCLKS = x1, it is 100

enumerator kRTC_ClockDivide_64_1000

If RTCLKS = x0, it is 64; if RTCLKS = x1, it is 1000

enum _rtc_interrupt_enable

List of RTC interrupts.

Values:

enumerator kRTC_InterruptEnable

Interrupt enable

enum _RTC_interrupt_flags

List of RTC Interrupt flags.

Values:

enumerator kRTC_InterruptFlag

Interrupt flag

enum _RTC_output_enable

List of RTC Output.

Values:

enumerator kRTC_OutputEnable

Output enable

typedef struct _rtc_datetime rtc_datetime_t

Structure is used to hold the date and time.

typedef enum _rtc_clock_source rtc_clock_source_t

List of RTC clock source.

typedef enum _rtc_clock_prescaler rtc_clock_prescaler_t

List of RTC clock prescaler.

typedef enum _rtc_interrupt_enable rtc_interrupt_enable_t

List of RTC interrupts.

typedef enum _RTC_interrupt_flags rtc_interrupt_flags_t

List of RTC Interrupt flags.

```
typedef enum _RTC_output_enable rtc_output_enable_t
```

List of RTC Output.

```
typedef struct _rtc_config rtc_config_t
```

RTC config structure.

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

```
typedef void (*rtc_alarm_callback_t)(void)
```

RTC alarm callback function.

```
struct _rtc_datetime
```

#include <fsl_rtc.h> Structure is used to hold the date and time.

Public Members

```
uint16_t year
```

Range from 1970 to 2099.

```
uint8_t month
```

Range from 1 to 12.

```
uint8_t day
```

Range from 1 to 31 (depending on month).

```
uint8_t hour
```

Range from 0 to 23.

```
uint8_t minute
```

Range from 0 to 59.

```
uint8_t second
```

Range from 0 to 59.

```
struct _rtc_config
```

#include <fsl_rtc.h> RTC config structure.

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

2.20 SPI: Serial Peripheral Interface Driver

2.21 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

void 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 = 400000,
    ...
};
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

void 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 = kSPIClockPolarity_ActiveHigh;
    .phase = kSPIClockPhase_FirstEdge;
    .direction = kSPIMsbFirst;
    ...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

- base – SPI base pointer
- config – pointer to master configuration structure

void SPI_Deinit(SPI_Type *base)

De-initializes the SPI.

Calling this API resets the SPI module, gates the SPI clock. 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)

Enables or disables the SPI.

Parameters

- base – SPI base pointer
- enable – pass true to enable module, false to disable module

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_flags` could get the related status.

static inline void SPI_ClearInterrupt(SPI_Type *base, uint8_t mask)

Clear the interrupt if enable INCTLR.

Parameters

- base – SPI base pointer
- mask – Interrupt need to be cleared The parameter could be any combination of the following values:
 - kSPI_RxFullAndModfInterruptEnable
 - kSPI_TxEmptyInterruptEnable
 - kSPI_MatchInterruptEnable
 - kSPI_RxFifoNearFullInterruptEnable
 - kSPI_TxFifoNearEmptyInterruptEnable

void SPI_EnableInterrupts(SPI_Type *base, uint32_t mask)

Enables the interrupt for the SPI.

Parameters

- base – SPI base pointer
- mask – SPI interrupt source. The parameter can be any combination of the following values:
 - kSPI_RxFullAndModfInterruptEnable
 - kSPI_TxEmptyInterruptEnable
 - kSPI_MatchInterruptEnable
 - kSPI_RxFifoNearFullInterruptEnable
 - kSPI_TxFifoNearEmptyInterruptEnable

void SPI_DisableInterrupts(SPI_Type *base, uint32_t mask)

Disables the interrupt for the SPI.

Parameters

- base – SPI base pointer
- mask – SPI interrupt source. The parameter can be any combination of the following values:

- kSPI_RxFullAndModfInterruptEnable
- kSPI_TxEmptyInterruptEnable
- kSPI_MatchInterruptEnable
- kSPI_RxFifoNearFullInterruptEnable
- kSPI_TxFifoNearEmptyInterruptEnable

static inline void SPI_EnableDMA(SPI_Type *base, uint8_t mask, bool enable)

Enables the DMA source for SPI.

Parameters

- base – SPI base pointer
- mask – SPI DMA source.
- enable – True means enable DMA, false means disable DMA

static inline uint32_t SPI_GetDataRegisterAddress(SPI_Type *base)

Gets the SPI tx/rx data register address.

This API is used to provide a transfer address for the SPI DMA transfer configuration.

Parameters

- base – SPI base pointer

Returns

data register address

uint32_t SPI_GetInstance(SPI_Type *base)

Get the instance for SPI module.

Parameters

- base – SPI base address

static inline void SPI_SetPinMode(SPI_Type *base, spi_pin_mode_t pinMode)

Sets the pin mode for transfer.

Parameters

- base – SPI base pointer
- pinMode – pin mode for transfer AND `_spi_pin_mode` could get the related configuration.

void 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_SetMatchData(SPI_Type *base, uint32_t matchData)

Sets the match data for SPI.

The match data is a hardware comparison value. When the value received in the SPI receive data buffer equals the hardware comparison value, the SPI Match Flag in the S register (S[SPMF]) sets. This can also generate an interrupt if the enable bit sets.

Parameters

- base – SPI base pointer

- `matchData` – Match data.

`void SPI_EnableFIFO(SPI_Type *base, bool enable)`

Enables or disables the FIFO if there is a FIFO.

Parameters

- `base` – SPI base pointer
- `enable` – True means enable FIFO, false means disable FIFO.

`status_t SPI_WriteBlocking(SPI_Type *base, uint8_t *buffer, size_t size)`

Sends a buffer of data bytes using a blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- `base` – SPI base pointer
- `buffer` – The data bytes to send
- `size` – The number of data bytes to send

Returns

`kStatus_SPI_Timeout` The transfer timed out and was aborted.

`void SPI_WriteData(SPI_Type *base, uint16_t data)`

Writes a data into the SPI data register.

Parameters

- `base` – SPI base pointer
- `data` – needs to be write.

`uint16_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_SetDummyData(SPI_Type *base, uint8_t dummyData)`

Set up the dummy data.

Parameters

- `base` – SPI peripheral address.
- `dummyData` – Data to be transferred when tx buffer is NULL.

`void 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_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.

status_t SPI_MasterTransferNonBlocking(*SPI_Type* *base, *spi_master_handle_t* *handle, *spi_transfer_t* *xfer)

Performs a non-blocking SPI interrupt transfer.

Note: The API immediately returns after transfer initialization is finished. Call *SPI_GetStatusIRQ()* to get the transfer status.

Note: If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

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 bytes of the SPI interrupt transferred.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.
- count – Transferred bytes of SPI master.

Return values

- *kStatus_SPI_Success* – Succeed get the transfer count.
- *kStatus_NoTransferInProgress* – There is not a non-blocking transaction currently in progress.


```
void SPI_MasterTransferAbort(SPI_Type *base, spi_master_handle_t *handle)
```

Aborts an SPI transfer using interrupt.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.

```
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.

```
void 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. Call SPI_GetStatusIRQ() to get the transfer status.

Note: If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

- base – SPI peripheral base address.
- handle – pointer to spi_slave_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 bytes of the SPI interrupt transferred.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.
- count – Transferred bytes of SPI slave.

Return values

- kStatus_SPI_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

```
static inline void SPI_SlaveTransferAbort(SPI_Type *base, spi_slave_handle_t *handle)
```

Aborts an SPI slave transfer using interrupt.

Parameters

- base – SPI peripheral base address.
- handle – Pointer to SPI transfer handle, this should be a static variable.

```
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.

Return status for the SPI driver.

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_Timeout
    SPI timeout polling status flags.
```

```
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 SPCK occurs at the middle of the first cycle of a data transfer.

enumerator `kSPI_ClockPhaseSecondEdge`

First edge on SPCK occurs at the start of the first cycle of a data transfer.

enum `_spi_shift_direction`

SPI data shifter direction options.

Values:

enumerator `kSPI_MsbFirst`

Data transfers start with most significant bit.

enumerator `kSPI_LsbFirst`

Data transfers start with least significant bit.

enum `_spi_ss_output_mode`

SPI slave select output mode options.

Values:

enumerator `kSPI_SlaveSelectAsGpio`

Slave select pin configured as GPIO.

enumerator `kSPI_SlaveSelectFaultInput`

Slave select pin configured for fault detection.

enumerator `kSPI_SlaveSelectAutomaticOutput`

Slave select pin configured for automatic SPI output.

enum `_spi_pin_mode`

SPI pin mode options.

Values:

enumerator `kSPI_PinModeNormal`

Pins operate in normal, single-direction mode.

enumerator `kSPI_PinModeInput`

Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input.

enumerator `kSPI_PinModeOutput`

Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output.

enum `_spi_data_bitcount_mode`

SPI data length mode options.

Values:

enumerator `kSPI_8BitMode`

8-bit data transmission mode

enumerator `kSPI_16BitMode`

16-bit data transmission mode

enum `_spi_interrupt_enable`

SPI interrupt sources.

Values:

enumerator kSPI_RxFullAndModfInterruptEnable
Receive buffer full (SPRF) and mode fault (MODF) interrupt

enumerator kSPI_TxEmptyInterruptEnable
Transmit buffer empty interrupt

enumerator kSPI_MatchInterruptEnable
Match interrupt

enumerator kSPI_RxFifoNearFullInterruptEnable
Receive FIFO nearly full interrupt

enumerator kSPI_TxFifoNearEmptyInterruptEnable
Transmit FIFO nearly empty interrupt

enum _spi_flags
SPI status flags.

Values:

enumerator kSPI_RxBufferFullFlag
Read buffer full flag

enumerator kSPI_MatchFlag
Match flag

enumerator kSPI_TxBufferEmptyFlag
Transmit buffer empty flag

enumerator kSPI_ModeFaultFlag
Mode fault flag

enumerator kSPI_RxFifoNearFullFlag
Rx FIFO near full

enumerator kSPI_TxFifoNearEmptyFlag
Tx FIFO near empty

enumerator kSPI_TxFifoFullFlag
Tx FIFO full

enumerator kSPI_RxFifoEmptyFlag
Rx FIFO empty

enumerator kSPI_TxFifoError
Tx FIFO error

enumerator kSPI_RxFifoError
Rx FIFO error

enumerator kSPI_TxOverflow
Tx FIFO Overflow

enumerator kSPI_RxOverflow
Rx FIFO Overflow

enum _spi_w1c_interrupt
SPI FIFO write-1-to-clear interrupt flags.

Values:

enumerator kSPI_RxFifoFullClearInterrupt
Receive FIFO full interrupt

enumerator kSPI_TxFifoEmptyClearInterrupt
Transmit FIFO empty interrupt

enumerator kSPI_RxNearFullClearInterrupt
Receive FIFO nearly full interrupt

enumerator kSPI_TxNearEmptyClearInterrupt
Transmit FIFO nearly empty interrupt

enum _spi_txfifo_watermark
SPI TX FIFO watermark settings.

Values:

enumerator kSPI_TxFifoOneFourthEmpty
SPI tx watermark at 1/4 FIFO size

enumerator kSPI_TxFifoOneHalfEmpty
SPI tx watermark at 1/2 FIFO size

enum _spi_rxfifo_watermark
SPI RX FIFO watermark settings.

Values:

enumerator kSPI_RxFifoThreeFourthsFull
SPI rx watermark at 3/4 FIFO size

enumerator kSPI_RxFifoOneHalfFull
SPI rx watermark at 1/2 FIFO size

enum _spi_dma_enable_t
SPI DMA source.

Values:

enumerator kSPI_TxDmaEnable
Tx DMA request source

enumerator kSPI_RxDmaEnable
Rx DMA request source

enumerator kSPI_DmaAllEnable
All DMA request source

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_shift_direction spi_shift_direction_t
SPI data shifter direction options.

typedef enum _spi_ss_output_mode spi_ss_output_mode_t
SPI slave select output mode options.

typedef enum _spi_pin_mode spi_pin_mode_t
SPI pin mode options.

typedef enum _spi_data_bitcount_mode spi_data_bitcount_mode_t
SPI data length mode options.

```

typedef enum _spi_w1c_interrupt spi_w1c_interrupt_t
    SPI FIFO write-1-to-clear interrupt flags.

typedef enum _spi_txfifo_watermark spi_txfifo_watermark_t
    SPI TX FIFO watermark settings.

typedef enum _spi_rxfifo_watermark spi_rxfifo_watermark_t
    SPI RX FIFO watermark settings.

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

typedef spi_master_handle_t spi_slave_handle_t
    Slave handle is the same with master handle

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 master callback for finished transmit.

volatile uint8_t g_spiDummyData[]
    Global variable for dummy data value setting.

SPI_DUMMYDATA
    SPI dummy transfer data, the data is sent while txBuff is NULL.

SPI_RETRY_TIMES
    Retry times for waiting flag.

struct _spi_master_config
    #include <fsl_spi.h> SPI master user configure structure.

```

Public Members

```

bool enableMaster
    Enable SPI at initialization time

bool enableStopInWaitMode
    SPI stop in wait mode

spi_clock_polarity_t polarity
    Clock polarity

spi_clock_phase_t phase
    Clock phase

spi_shift_direction_t direction
    MSB or LSB

```

spi_data_bitcount_mode_t dataMode

8bit or 16bit mode

spi_txfifo_watermark_t txWatermark

Tx watermark settings

spi_rxfifo_watermark_t rxWatermark

Rx watermark settings

spi_ss_output_mode_t outputMode

SS pin setting

spi_pin_mode_t pinMode

SPI pin mode select

uint32_t baudRate_Bps

Baud Rate for SPI in Hz

struct *_spi_slave_config*

#include <fsl_spi.h> SPI slave user configure structure.

Public Members

bool enableSlave

Enable SPI at initialization time

bool enableStopInWaitMode

SPI stop in wait mode

spi_clock_polarity_t polarity

Clock polarity

spi_clock_phase_t phase

Clock phase

spi_shift_direction_t direction

MSB or LSB

spi_data_bitcount_mode_t dataMode

8bit or 16bit mode

spi_txfifo_watermark_t txWatermark

Tx watermark settings

spi_rxfifo_watermark_t rxWatermark

Rx watermark settings

spi_pin_mode_t pinMode

SPI pin mode select

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 flags
    SPI control flag, useless to SPI.
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
    Send data remaining in bytes
volatile size_t rxRemainingBytes
    Receive data remaining in bytes
volatile uint32_t state
    SPI internal state
size_t transferSize
    Bytes to be transferred
uint8_t bytePerFrame
    SPI mode, 2bytes or 1byte in a frame
uint8_t watermark
    Watermark value for SPI transfer
spi_master_callback_t callback
    SPI callback
void *userData
    Callback parameter
```

2.22 TPM: Timer PWM Module

```
uint32_t TPM_GetInstance(TPM_Type *base)
    Gets the instance from the base address.
```

Parameters

- base – TPM peripheral base address

Returns

The TPM instance

```
void TPM_Init(TPM_Type *base, const tpm_config_t *config)
    Ungates the TPM clock and configures the peripheral for basic operation.
```

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

Parameters

- base – TPM peripheral base address
- config – Pointer to user's TPM config structure.

void TPM_Deinit(TPM_Type *base)

Stops the counter and gates the TPM clock.

Parameters

- base – TPM peripheral base address

void TPM_GetDefaultConfig(*tpm_config_t* *config)

Fill in the TPM config struct with the default settings.

The default values are:

```
config->prescale = kTPM_Prescale_Divide_1;
config->useGlobalTimeBase = false;
config->syncGlobalTimeBase = false;
config->dozeEnable = false;
config->dbgMode = false;
config->enableReloadOnTrigger = false;
config->enableStopOnOverflow = false;
config->enableStartOnTrigger = false;
#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
config->enablePauseOnTrigger = false;
#endif
config->triggerSelect = kTPM_Trigger_Select_0;
#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
config->triggerSource = kTPM_TriggerSource_External;
config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
#endif
#if defined(FSL_FEATURE_TPM_HAS_POL) && FSL_FEATURE_TPM_HAS_POL
config->chnlPolarity = 0U;
#endif
```

Parameters

- config – Pointer to user's TPM config structure.

tpm_clock_prescale_t TPM_CalculateCounterClkDiv(TPM_Type *base, uint32_t
counterPeriod_Hz, uint32_t srcClock_Hz)

Calculates the counter clock prescaler.

This function calculates the values for SC[PS].

return Calculated clock prescaler value.

Parameters

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

status_t TPM_SetupPwm(TPM_Type *base, const *tpm_chnl_pwm_signal_param_t* *chnlParams,
uint8_t numOfChnls, *tpm_pwm_mode_t* mode, uint32_t pwmFreq_Hz,
uint32_t srcClock_Hz)

Configures the PWM signal parameters.

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

- `base` – TPM 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 `tpm_pwm_mode_t`
- `pwmFreq_Hz` – PWM signal frequency in Hz
- `srcClock_Hz` – TPM counter clock in Hz

Returns

`kStatus_Success` PWM setup successful
`kStatus_Error` PWM setup failed
`kStatus_Timeout` PWM setup timeout when write register `CnV` or `MOD`

`status_t` TPM_UpdatePwmDutycycle(TPM_Type *base, *tpm_chnl_t* chnlNumber, *tpm_pwm_mode_t* currentPwmMode, uint8_t dutyCyclePercent)

Update the duty cycle of an active PWM signal.

Parameters

- `base` – TPM peripheral base address
- `chnlNumber` – The channel number. In combined mode, this represents the channel pair number
- `currentPwmMode` – The current PWM mode set during PWM setup
- `dutyCyclePercent` – New PWM pulse width, 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 setup was successful, `kStatus_Error` on failure

`void` TPM_UpdateChnlEdgeLevelSelect(TPM_Type *base, *tpm_chnl_t* chnlNumber, uint8_t level)

Update the edge level selection for a channel.

Note: When the TPM has PWM pause level select feature (FSL_FEATURE_TPM_HAS_PAUSE_LEVEL_SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use `TPM_DisableChannel` API to close the PWM output.

Parameters

- `base` – TPM 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 appropriate SoC reference manual for details about this field.

`static inline uint8_t` TPM_GetChannelControlBits(TPM_Type *base, *tpm_chnl_t* chnlNumber)

Get the channel control bits value (mode, edge and level bit fields).

This function disable the channel by clear all mode and level control bits.

Parameters

- `base` – TPM peripheral base address

- `chnlNumber` – The channel number

Returns

The control bits value. This is the logical OR of members of the enumeration `tpm_chnl_control_bit_mask_t`.

```
static inline status_t TPM_DisableChannel(TPM_Type *base, tpm_chnl_t chnlNumber)
```

Disable the channel.

This function disable the channel by clear all mode and level control bits.

Parameters

- `base` – TPM peripheral base address
- `chnlNumber` – The channel number

Returns

`kStatus_Success` PWM setup successful `kStatus_Timeout` PWM setup timeout when write register `CnSC`

```
static inline status_t TPM_EnableChannel(TPM_Type *base, tpm_chnl_t chnlNumber, uint8_t control)
```

Enable the channel according to mode and level configs.

This function enable the channel output according to input mode/level config parameters.

Parameters

- `base` – TPM peripheral base address
- `chnlNumber` – The channel number
- `control` – The control bits value. This is the logical OR of members of the enumeration `tpm_chnl_control_bit_mask_t`.

Returns

`kStatus_Success` PWM setup successful `kStatus_Timeout` PWM setup timeout when write register `CnSC`

```
void TPM_SetupInputCapture(TPM_Type *base, tpm_chnl_t chnlNumber, tpm_input_capture_edge_t captureMode)
```

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 TPM counter is captured into the `CnV` register. The user has to read the `CnV` register separately to get this value.

Parameters

- `base` – TPM peripheral base address
- `chnlNumber` – The channel number
- `captureMode` – Specifies which edge to capture

```
status_t TPM_SetupOutputCompare(TPM_Type *base, tpm_chnl_t chnlNumber, tpm_output_compare_mode_t compareMode, uint32_t compareValue)
```

Configures the TPM to generate timed pulses.

When the TPM 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` – TPM 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.

Returns

`kStatus_Success` PWM setup successful
`kStatus_Timeout` PWM setup timeout when write register CnV

`void TPM_EnableInterrupts(TPM_Type *base, uint32_t mask)`

Enables the selected TPM interrupts.

Parameters

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

`void TPM_DisableInterrupts(TPM_Type *base, uint32_t mask)`

Disables the selected TPM interrupts.

Parameters

- `base` – TPM peripheral base address
- `mask` – The interrupts to disable. This is a logical OR of members of the enumeration `tpm_interrupt_enable_t`

`uint32_t TPM_GetEnabledInterrupts(TPM_Type *base)`

Gets the enabled TPM interrupts.

Parameters

- `base` – TPM peripheral base address

Returns

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

`void TPM_RegisterCallBack(TPM_Type *base, tpm_callback_t callback)`

Register callback.

If channel or overflow interrupt is enabled by the user, then a callback can be registered which will be invoked when the interrupt is triggered.

Parameters

- `base` – TPM peripheral base address
- `callback` – Callback function

`void TPM_DriverIRQHandler(uint32_t instance)`

TPM driver IRQ handler common entry.

This function provides the common IRQ request entry for TPM.

Parameters

- `instance` – TPM instance.

`static inline uint32_t TPM_GetChannelValue(TPM_Type *base, tpm_chnl_t chnlNumber)`

Gets the TPM channel value.

Note: The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

- base – TPM peripheral base address
- chnlNumber – The channel number

Returns

The channle CnV regisyer value.

```
static inline uint32_t TPM_GetStatusFlags(TPM_Type *base)
```

Gets the TPM status flags.

Parameters

- base – TPM peripheral base address

Returns

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

```
static inline void TPM_ClearStatusFlags(TPM_Type *base, uint32_t mask)
```

Clears the TPM status flags.

Parameters

- base – TPM peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `tpm_status_flags_t`

```
static inline status_t TPM_SetTimerPeriod(TPM_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:

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

Parameters

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

Returns

`kStatus_Success` PWM setup successful
`kStatus_Timeout` PWM setup timeout when write register CnSC

```
static inline uint32_t TPM_GetCurrentTimerCount(TPM_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 – TPM peripheral base address

Returns

The current counter value in ticks

```
static inline void TPM_StartTimer(TPM_Type *base, tpm_clock_source_t clockSource)
```

Starts the TPM counter.

Parameters

- base – TPM peripheral base address
- clockSource – TPM clock source; once clock source is set the counter will start running

```
static inline status_t TPM_StopTimer(TPM_Type *base)
```

Stops the TPM counter.

Parameters

- base – TPM peripheral base address

Returns

kStatus_Success PWM setup successful
kStatus_Timeout PWM setup timeout when write register CnSC

```
FSL_TPM_DRIVER_VERSION
```

TPM driver version 2.4.0.

```
enum _tpm_chnl
```

List of TPM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kTPM_Chnl_0
TPM channel number 0

enumerator kTPM_Chnl_1
TPM channel number 1

enumerator kTPM_Chnl_2
TPM channel number 2

enumerator kTPM_Chnl_3
TPM channel number 3

enumerator kTPM_Chnl_4
TPM channel number 4

enumerator kTPM_Chnl_5
TPM channel number 5

enumerator kTPM_Chnl_6
TPM channel number 6

enumerator kTPM_Chnl_7
TPM channel number 7

```
enum _tpm_pwm_mode
```

TPM PWM operation modes.

Values:

enumerator kTPM_EdgeAlignedPwm

Edge aligned PWM

enumerator kTPM_CenterAlignedPwm

Center aligned PWM

enum _tpm_pwm_level_select

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

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Values:

enumerator kTPM_NoPwmSignal

No PWM output on pin

enumerator kTPM_LowTrue

Low true pulses

enumerator kTPM_HighTrue

High true pulses

enum _tpm_chnl_control_bit_mask

List of TPM channel modes and level control bit mask.

Values:

enumerator kTPM_ChnlELSnAMask

Channel ELSA bit mask.

enumerator kTPM_ChnlELSnBMask

Channel ELSE bit mask.

enumerator kTPM_ChnlMSAMask

Channel MSA bit mask.

enumerator kTPM_ChnlMSBMask

Channel MSB bit mask.

enum _tpm_output_compare_mode

TPM output compare modes.

Values:

enumerator kTPM_NoOutputSignal

No channel output when counter reaches CnV

enumerator kTPM_ToggleOnMatch

Toggle output

enumerator kTPM_ClearOnMatch

Clear output

enumerator kTPM_SetOnMatch

Set output

enumerator kTPM_HighPulseOutput

Pulse output high

enumerator kTPM_LowPulseOutput
Pulse output low

enum _tpm_input_capture_edge
TPM input capture edge.

Values:

enumerator kTPM_RisingEdge
Capture on rising edge only

enumerator kTPM_FallingEdge
Capture on falling edge only

enumerator kTPM_RiseAndFallEdge
Capture on rising or falling edge

enum _tpm_clock_source
TPM clock source selection.

Values:

enumerator kTPM_SystemClock
System clock

enumerator kTPM_FixedClock
Fixed frequency clock

enumerator kTPM_ExternalClock
External TPM_EXTCLK pin clock

enum _tpm_clock_prescale
TPM prescale value selection for the clock source.

Values:

enumerator kTPM_Prescale_Divide_1
Divide by 1

enumerator kTPM_Prescale_Divide_2
Divide by 2

enumerator kTPM_Prescale_Divide_4
Divide by 4

enumerator kTPM_Prescale_Divide_8
Divide by 8

enumerator kTPM_Prescale_Divide_16
Divide by 16

enumerator kTPM_Prescale_Divide_32
Divide by 32

enumerator kTPM_Prescale_Divide_64
Divide by 64

enumerator kTPM_Prescale_Divide_128
Divide by 128

enum _tpm_interrupt_enable
List of TPM interrupts.

Values:

enumerator kTPM_Chnl0InterruptEnable
Channel 0 interrupt.

enumerator kTPM_Chnl1InterruptEnable
Channel 1 interrupt.

enumerator kTPM_Chnl2InterruptEnable
Channel 2 interrupt.

enumerator kTPM_Chnl3InterruptEnable
Channel 3 interrupt.

enumerator kTPM_Chnl4InterruptEnable
Channel 4 interrupt.

enumerator kTPM_Chnl5InterruptEnable
Channel 5 interrupt.

enumerator kTPM_Chnl6InterruptEnable
Channel 6 interrupt.

enumerator kTPM_Chnl7InterruptEnable
Channel 7 interrupt.

enumerator kTPM_TimeOverflowInterruptEnable
Time overflow interrupt.

enum _tpm_status_flags

List of TPM flags.

Values:

enumerator kTPM_Chnl0Flag
Channel 0 flag

enumerator kTPM_Chnl1Flag
Channel 1 flag

enumerator kTPM_Chnl2Flag
Channel 2 flag

enumerator kTPM_Chnl3Flag
Channel 3 flag

enumerator kTPM_Chnl4Flag
Channel 4 flag

enumerator kTPM_Chnl5Flag
Channel 5 flag

enumerator kTPM_Chnl6Flag
Channel 6 flag

enumerator kTPM_Chnl7Flag
Channel 7 flag

enumerator kTPM_TimeOverflowFlag
Time overflow flag

typedef enum _tpm_chnl tpm_chnl_t

List of TPM channels.

Note: Actual number of available channels is SoC dependent

typedef enum *_tpm_pwm_mode* tpm_pwm_mode_t

TPM PWM operation modes.

typedef enum *_tpm_pwm_level_select* tpm_pwm_level_select_t

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

Note: When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

typedef enum *_tpm_chnl_control_bit_mask* tpm_chnl_control_bit_mask_t

List of TPM channel modes and level control bit mask.

typedef struct *_tpm_chnl_pwm_signal_param* tpm_chnl_pwm_signal_param_t

Options to configure a TPM channel's PWM signal.

typedef enum *_tpm_output_compare_mode* tpm_output_compare_mode_t

TPM output compare modes.

typedef enum *_tpm_input_capture_edge* tpm_input_capture_edge_t

TPM input capture edge.

typedef enum *_tpm_clock_source* tpm_clock_source_t

TPM clock source selection.

typedef enum *_tpm_clock_prescale* tpm_clock_prescale_t

TPM prescale value selection for the clock source.

typedef struct *_tpm_config* tpm_config_t

TPM config structure.

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

The config struct can be made const so it resides in flash

typedef enum *_tpm_interrupt_enable* tpm_interrupt_enable_t

List of TPM interrupts.

typedef enum *_tpm_status_flags* tpm_status_flags_t

List of TPM flags.

typedef void (*tpm_callback_t)(TPM_Type *base)

TPM callback function pointer.

Param base

TPM peripheral base address.

TPM_TIMEOUT

Max loops to wait for writing register.

When writing MOD CnV CnSC and SC register, driver will wait until register is updated. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

TPM_MAX_COUNTER_VALUE(x)

Help macro to get the max counter value.

struct *_tpm_chnl_pwm_signal_param*

#include <fsl_tpm.h> Options to configure a TPM channel's PWM signal.

Public Members

tpm_chnl_t chnlNumber

TPM channel to configure. In combined mode (available in some SoC's), this represents the channel pair number

tpm_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)

`struct _tpm_config`

`#include <fsl_tpm.h>` TPM config structure.

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

tpm_clock_prescale_t prescale

Select TPM clock prescale value

2.23 UART: Universal Asynchronous Receiver/Transmitter Driver

2.24 UART Driver

status_t UART_Init(UART_Type *base, const *uart_config_t* *config, uint32_t srcClock_Hz)

Initializes a UART instance with a user configuration structure and peripheral clock.

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the `UART_GetDefaultConfig()` function. The example below shows how to use this API to configure UART.

```
uart_config_t uartConfig;
uartConfig.baudRate_Bps = 115200U;
uartConfig.parityMode = kUART_ParityDisabled;
uartConfig.stopBitCount = kUART_OneStopBit;
uartConfig.txFifoWatermark = 0;
uartConfig.rxFifoWatermark = 1;
UART_Init(UART1, &uartConfig, 2000000U);
```

Parameters

- base – UART peripheral base address.
- config – Pointer to the user-defined configuration structure.
- srcClock_Hz – UART clock source frequency in HZ.

Return values

- `kStatus_UART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Status UART initialize succeed

`void UART_Deinit(UART_Type *base)`

Deinitializes a UART instance.

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

- `base` – UART peripheral base address.

`void UART_GetDefaultConfig(uart_config_t *config)`

Gets the default configuration structure.

This function initializes the UART configuration structure to a default value. The default values are as follows. `uartConfig->baudRate_Bps = 115200U`; `uartConfig->bitCountPerChar = kUART_8BitsPerChar`; `uartConfig->parityMode = kUART_ParityDisabled`; `uartConfig->stopBitCount = kUART_OneStopBit`; `uartConfig->txFifoWatermark = 0`; `uartConfig->rxFifoWatermark = 1`; `uartConfig->idleType = kUART_IdleTypeStartBit`; `uartConfig->enableTx = false`; `uartConfig->enableRx = false`;

Parameters

- `config` – Pointer to configuration structure.

`status_t UART_SetBaudRate(UART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)`

Sets the UART instance baud rate.

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the `UART_Init`.

```
UART_SetBaudRate(UART1, 115200U, 20000000U);
```

Parameters

- `base` – UART peripheral base address.
- `baudRate_Bps` – UART baudrate to be set.
- `srcClock_Hz` – UART clock source frequency in Hz.

Return values

- `kStatus_UART_BaudrateNotSupport` – Baudrate is not support in the current clock source.
- `kStatus_Success` – Set baudrate succeeded.

`void UART_Enable9bitMode(UART_Type *base, bool enable)`

Enable 9-bit data mode for UART.

This function set the 9-bit mode for UART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- `base` – UART peripheral base address.
- `enable` – true to enable, false to disable.

`static inline void UART_SetMatchAddress(UART_Type *base, uint8_t address1, uint8_t address2)`

Set the UART slave address.

This function configures the address for UART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable

bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer; otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any UART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- base – UART peripheral base address.
- address1 – UART slave address 1.
- address2 – UART slave address 2.

static inline void UART_EnableMatchAddress(UART_Type *base, bool match1, bool match2)

Enable the UART match address feature.

Parameters

- base – UART peripheral base address.
- match1 – true to enable match address1, false to disable.
- match2 – true to enable match address2, false to disable.

static inline void UART_Set9thTransmitBit(UART_Type *base)

Set UART 9th transmit bit.

Parameters

- base – UART peripheral base address.

static inline void UART_Clear9thTransmitBit(UART_Type *base)

Clear UART 9th transmit bit.

Parameters

- base – UART peripheral base address.

uint32_t UART_GetStatusFlags(UART_Type *base)

Gets UART status flags.

This function gets all UART status flags. The flags are returned as the logical OR value of the enumerators `_uart_flags`. To check a specific status, compare the return value with enumerators in `_uart_flags`. For example, to check whether the TX is empty, do the following.

```
if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
{
    ...
}
```

Parameters

- base – UART peripheral base address.

Returns

UART status flags which are Ored by the enumerators in the `_uart_flags`.

`status_t UART_ClearStatusFlags(UART_Type *base, uint32_t mask)`

Clears status flags with the provided mask.

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. `kUART_TxDataRegEmptyFlag`, `kUART_TransmissionCompleteFlag`, `kUART_RxDataRegFullFlag`, `kUART_RxActiveFlag`, `kUART_NoiseErrorInRxDataRegFlag`, `kUART_ParityErrorInRxDataRegFlag`, `kUART_TxFifoEmptyFlag`, `kUART_RxFifoEmptyFlag`

Note: that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

Parameters

- `base` – UART peripheral base address.
- `mask` – The status flags to be cleared; it is logical OR value of `_uart_flags`.

Return values

- `kStatus_UART_FlagCannotClearManually` – The flag can't be cleared by this function but it is cleared automatically by hardware.
- `kStatus_Success` – Status in the mask is cleared.

`void UART_EnableInterrupts(UART_Type *base, uint32_t mask)`

Enables UART interrupts according to the provided mask.

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_uart_interrupt_enable`. For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
UART_EnableInterrupts(UART1, kUART_TxDataRegEmptyInterruptEnable | kUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – UART peripheral base address.
- `mask` – The interrupts to enable. Logical OR of `_uart_interrupt_enable`.

`void UART_DisableInterrupts(UART_Type *base, uint32_t mask)`

Disables the UART interrupts according to the provided mask.

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_uart_interrupt_enable`. For example, to disable TX empty interrupt and RX full interrupt do the following.

```
UART_DisableInterrupts(UART1, kUART_TxDataRegEmptyInterruptEnable | kUART_
↳ RxDataRegFullInterruptEnable);
```

Parameters

- `base` – UART peripheral base address.
- `mask` – The interrupts to disable. Logical OR of `_uart_interrupt_enable`.

`uint32_t UART_GetEnabledInterrupts(UART_Type *base)`

Gets the enabled UART interrupts.

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_uart_interrupt_enable`. To check a specific interrupts enable status, compare the return value with enumerators in `_uart_interrupt_enable`. For example, to check whether TX empty interrupt is enabled, do the following.

```
uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);  
  
if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)  
{  
    ...  
}
```

Parameters

- base – UART peripheral base address.

Returns

UART interrupt flags which are logical OR of the enumerators in `_uart_interrupt_enable`.

static inline uint32_t UART_GetDataRegisterAddress(UART_Type *base)

Gets the UART data register address.

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

- base – UART peripheral base address.

Returns

UART data register addresses which are used both by the transmitter and the receiver.

static inline void UART_EnableTxDMA(UART_Type *base, bool enable)

Enables or disables the UART transmitter DMA request.

This function enables or disables the transmit data register empty flag, S1[TDRE], to generate the DMA requests.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

static inline void UART_EnableRxDMA(UART_Type *base, bool enable)

Enables or disables the UART receiver DMA.

This function enables or disables the receiver data register full flag, S1[RDRF], to generate DMA requests.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

static inline void UART_EnableTx(UART_Type *base, bool enable)

Enables or disables the UART transmitter.

This function enables or disables the UART transmitter.

Parameters

- base – UART peripheral base address.
- enable – True to enable, false to disable.

static inline void UART_EnableRx(UART_Type *base, bool enable)

Enables or disables the UART receiver.

This function enables or disables the UART receiver.

Parameters

- `base` – UART peripheral base address.
- `enable` – True to enable, false to disable.

`static inline void UART_WriteByte(UART_Type *base, uint8_t data)`

Writes to the TX register.

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

- `base` – UART peripheral base address.
- `data` – The byte to write.

`static inline uint8_t UART_ReadByte(UART_Type *base)`

Reads the RX register directly.

This function reads data from the RX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

- `base` – UART peripheral base address.

Returns

The byte read from UART data register.

`static inline uint8_t UART_GetRxFifoCount(UART_Type *base)`

Gets the rx FIFO data count.

Parameters

- `base` – UART peripheral base address.

Returns

rx FIFO data count.

`static inline uint8_t UART_GetTxFifoCount(UART_Type *base)`

Gets the tx FIFO data count.

Parameters

- `base` – UART peripheral base address.

Returns

tx FIFO data count.

`void UART_SendAddress(UART_Type *base, uint8_t address)`

Transmit an address frame in 9-bit data mode.

Parameters

- `base` – UART peripheral base address.
- `address` – UART slave address.

`status_t UART_WriteBlocking(UART_Type *base, const uint8_t *data, size_t length)`

Writes to the TX register using a blocking method.

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Parameters

- `base` – UART peripheral base address.
- `data` – Start address of the data to write.
- `length` – Size of the data to write.

Return values

- `kStatus_UART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully wrote all data.

`status_t UART_ReadBlocking(UART_Type *base, uint8_t *data, size_t length)`

Read RX data register using a blocking method.

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data, and reads data from the TX register.

Parameters

- `base` – UART peripheral base address.
- `data` – Start address of the buffer to store the received data.
- `length` – Size of the buffer.

Return values

- `kStatus_UART_RxHardwareOverrun` – Receiver overrun occurred while receiving data.
- `kStatus_UART_NoiseError` – A noise error occurred while receiving data.
- `kStatus_UART_FramingError` – A framing error occurred while receiving data.
- `kStatus_UART_ParityError` – A parity error occurred while receiving data.
- `kStatus_UART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

`void UART_TransferCreateHandle(UART_Type *base, uart_handle_t *handle, uart_transfer_callback_t callback, void *userData)`

Initializes the UART handle.

This function initializes the UART handle which can be used for other UART transactional APIs. Usually, for a specified UART instance, call this API once to get the initialized handle.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `callback` – The callback function.
- `userData` – The parameter of the callback function.

`void UART_TransferStartRingBuffer(UART_Type *base, uart_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 UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the `UART_TransferReceiveNonBlocking()` API. If data is already 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, only 31 bytes are used for saving data.

Parameters

- `base` – UART peripheral base address.

- `handle` – UART 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 UART_TransferStopRingBuffer(UART_Type *base, uart_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` – UART peripheral base address.
- `handle` – UART handle pointer.

`size_t UART_TransferGetRxRingBufferLength(uart_handle_t *handle)`

Get the length of received data in RX ring buffer.

Parameters

- `handle` – UART handle pointer.

Returns

Length of received data in RX ring buffer.

`status_t UART_TransferSendNonBlocking(UART_Type *base, uart_handle_t *handle, uart_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 ISR, the UART driver calls the callback function and passes the `kStatus_UART_TxIdle` as status parameter.

Note: The `kStatus_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the `kUART_TransmissionCompleteFlag` to ensure that the TX is finished.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `xfer` – UART transfer structure. See `uart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_UART_TxBusy` – Previous transmission still not finished; data not all written to TX register yet.
- `kStatus_InvalidArgument` – Invalid argument.

`void UART_TransferAbortSend(UART_Type *base, uart_handle_t *handle)`

Aborts the interrupt-driven data transmit.

This function aborts the interrupt-driven data sending. The user can get the `remainBytes` to find out how many bytes are not sent out.

Parameters

- `base` – UART peripheral base address.

- `handle` – UART handle pointer.

`status_t` UART_TransferGetSendCount(UART_Type *base, `uart_handle_t` *handle, `uint32_t` *count)

Gets the number of bytes sent out to bus.

This function gets the number of bytes sent out to bus by using the interrupt method.

Parameters

- `base` – UART peripheral base address.
- `handle` – UART handle pointer.
- `count` – Send bytes count.

Return values

- `kStatus_NoTransferInProgress` – No send in progress.
- `kStatus_InvalidArgument` – The parameter is invalid.
- `kStatus_Success` – Get successfully through the parameter `count`;

`status_t` UART_TransferReceiveNonBlocking(UART_Type *base, `uart_handle_t` *handle, `uart_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 UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter `kStatus_UART_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 UART 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` – UART peripheral base address.
- `handle` – UART handle pointer.
- `xfer` – UART transfer structure, see `uart_transfer_t`.
- `receivedBytes` – Bytes received from the ring buffer directly.

Return values

- `kStatus_Success` – Successfully queue the transfer into transmit queue.
- `kStatus_UART_RxBusy` – Previous receive request is not finished.
- `kStatus_InvalidArgument` – Invalid argument.

`void` UART_TransferAbortReceive(UART_Type *base, `uart_handle_t` *handle)

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to know how many bytes are not received yet.

Parameters

- `base` – UART peripheral base address.

- handle – UART handle pointer.

status_t UART_TransferGetReceiveCount(UART_Type *base, *uart_handle_t* *handle, uint32_t *count)

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – UART peripheral base address.
- handle – UART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

status_t UART_EnableTxFIFO(UART_Type *base, bool enable)

Enables or disables the UART Tx FIFO.

This function enables or disables the UART Tx FIFO.

param base UART peripheral base address. param enable true to enable, false to disable. retval kStatus_Success Successfully turn on or turn off Tx FIFO. retval kStatus_Fail Fail to turn on or turn off Tx FIFO.

status_t UART_EnableRxFIFO(UART_Type *base, bool enable)

Enables or disables the UART Rx FIFO.

This function enables or disables the UART Rx FIFO.

param base UART peripheral base address. param enable true to enable, false to disable. retval kStatus_Success Successfully turn on or turn off Rx FIFO. retval kStatus_Fail Fail to turn on or turn off Rx FIFO.

static inline void UART_SetRxFifoWatermark(UART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – UART peripheral base address.
- water – Rx FIFO watermark.

static inline void UART_SetTxFifoWatermark(UART_Type *base, uint8_t water)

Sets the tx FIFO watermark.

Parameters

- base – UART peripheral base address.
- water – Tx FIFO watermark.

void UART_TransferHandleIRQ(UART_Type *base, void *irqHandle)

UART IRQ handle function.

This function handles the UART transmit and receive IRQ request.

Parameters

- base – UART peripheral base address.
- irqHandle – UART handle pointer.

`void UART_TransferHandleErrorIRQ(UART_Type *base, void *irqHandle)`

UART Error IRQ handle function.

This function handles the UART error IRQ request.

Parameters

- `base` – UART peripheral base address.
- `irqHandle` – UART handle pointer.

`FSL_UART_DRIVER_VERSION`

UART driver version.

Error codes for the UART driver.

Values:

enumerator `kStatus_UART_TxBusy`

Transmitter is busy.

enumerator `kStatus_UART_RxBusy`

Receiver is busy.

enumerator `kStatus_UART_TxIdle`

UART transmitter is idle.

enumerator `kStatus_UART_RxIdle`

UART receiver is idle.

enumerator `kStatus_UART_TxWatermarkTooLarge`

TX FIFO watermark too large

enumerator `kStatus_UART_RxWatermarkTooLarge`

RX FIFO watermark too large

enumerator `kStatus_UART_FlagCannotClearManually`

UART flag can't be manually cleared.

enumerator `kStatus_UART_Error`

Error happens on UART.

enumerator `kStatus_UART_RxRingBufferOverrun`

UART RX software ring buffer overrun.

enumerator `kStatus_UART_RxHardwareOverrun`

UART RX receiver overrun.

enumerator `kStatus_UART_NoiseError`

UART noise error.

enumerator `kStatus_UART_FramingError`

UART framing error.

enumerator `kStatus_UART_ParityError`

UART parity error.

enumerator `kStatus_UART_BaudrateNotSupport`

Baudrate is not support in current clock source

enumerator `kStatus_UART_IdleLineDetected`

UART IDLE line detected.

enumerator kStatus_UART_Timeout
UART times out.

enum _uart_parity_mode
UART parity mode.

Values:

enumerator kUART_ParityDisabled
Parity disabled

enumerator kUART_ParityEven
Parity enabled, type even, bit setting: PE | PT = 10

enumerator kUART_ParityOdd
Parity enabled, type odd, bit setting: PE | PT = 11

enum _uart_stop_bit_count
UART stop bit count.

Values:

enumerator kUART_OneStopBit
One stop bit

enumerator kUART_TwoStopBit
Two stop bits

enum _uart_idle_type_select
UART idle type select.

Values:

enumerator kUART_IdleTypeStartBit
Start counting after a valid start bit.

enumerator kUART_IdleTypeStopBit
Start counting after a stop bit.

enum _uart_interrupt_enable
UART interrupt configuration structure, default settings all disabled.
This structure contains the settings for all of the UART interrupt configurations.

Values:

enumerator kUART_LinBreakInterruptEnable
LIN break detect interrupt.

enumerator kUART_RxActiveEdgeInterruptEnable
RX active edge interrupt.

enumerator kUART_TxDataRegEmptyInterruptEnable
Transmit data register empty interrupt.

enumerator kUART_TransmissionCompleteInterruptEnable
Transmission complete interrupt.

enumerator kUART_RxDataRegFullInterruptEnable
Receiver data register full interrupt.

enumerator kUART_IdleLineInterruptEnable
Idle line interrupt.

enumerator kUART_RxOverrunInterruptEnable
Receiver overrun interrupt.

enumerator kUART_NoiseErrorInterruptEnable
Noise error flag interrupt.

enumerator kUART_FramingErrorInterruptEnable
Framing error flag interrupt.

enumerator kUART_ParityErrorInterruptEnable
Parity error flag interrupt.

enumerator kUART_RxFifoOverflowInterruptEnable
RX FIFO overflow interrupt.

enumerator kUART_TxFifoOverflowInterruptEnable
TX FIFO overflow interrupt.

enumerator kUART_RxFifoUnderflowInterruptEnable
RX FIFO underflow interrupt.

enumerator kUART_AllInterruptsEnable

UART status flags.

This provides constants for the UART status flags for use in the UART functions.

Values:

enumerator kUART_TxDataRegEmptyFlag
TX data register empty flag.

enumerator kUART_TransmissionCompleteFlag
Transmission complete flag.

enumerator kUART_RxDataRegFullFlag
RX data register full flag.

enumerator kUART_IdleLineFlag
Idle line detect flag.

enumerator kUART_RxOverrunFlag
RX overrun flag.

enumerator kUART_NoiseErrorFlag
RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets

enumerator kUART_FramingErrorFlag
Frame error flag, sets if logic 0 was detected where stop bit expected

enumerator kUART_ParityErrorFlag
If parity enabled, sets upon parity error detection

enumerator kUART_LinBreakFlag
LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled

enumerator kUART_RxActiveEdgeFlag
RX pin active edge interrupt flag, sets when active edge detected

enumerator kUART_RxActiveFlag
Receiver Active Flag (RAF), sets at beginning of valid start bit

enumerator `kUART_NoiseErrorInRxDataRegFlag`
Noisy bit, sets if noise detected.

enumerator `kUART_ParityErrorInRxDataRegFlag`
Parity bit, sets if parity error detected.

enumerator `kUART_TxFifoEmptyFlag`
TXEMPT bit, sets if TX buffer is empty

enumerator `kUART_RxFifoEmptyFlag`
RXEMPT bit, sets if RX buffer is empty

enumerator `kUART_TxFifoOverflowFlag`
TXOF bit, sets if TX buffer overflow occurred

enumerator `kUART_RxFifoOverflowFlag`
RXOF bit, sets if receive buffer overflow

enumerator `kUART_RxFifoUnderflowFlag`
RXUF bit, sets if receive buffer underflow

typedef enum `_uart_parity_mode` `uart_parity_mode_t`
UART parity mode.

typedef enum `_uart_stop_bit_count` `uart_stop_bit_count_t`
UART stop bit count.

typedef enum `_uart_idle_type_select` `uart_idle_type_select_t`
UART idle type select.

typedef struct `_uart_config` `uart_config_t`
UART configuration structure.

typedef struct `_uart_transfer` `uart_transfer_t`
UART transfer structure.

typedef struct `_uart_handle` `uart_handle_t`

typedef void (`*uart_transfer_callback_t`)(UART_Type *base, `uart_handle_t` *handle, `status_t` status, void *userData)
UART transfer callback function.

typedef void (`*uart_isr_t`)(UART_Type *base, void *handle)

void *s_uartHandle[]
Pointers to uart handles for each instance.

const IRQn_Type s_uartIRQ[]

`uart_isr_t` s_uartIsr
Pointer to uart IRQ handler for each instance.

uint32_t UART_GetInstance(UART_Type *base)
Get the UART instance from peripheral base address.

Parameters

- base – UART peripheral base address.

Returns
UART instance.

UART_RETRY_TIMES
Retry times for waiting flag.

struct __uart_config
 #include <fsl_uart.h> UART configuration structure.

Public Members

uint32_t baudRate_Bps
 UART baud rate

uart_parity_mode_t parityMode
 Parity mode, disabled (default), even, odd

uart_stop_bit_count_t stopBitCount
 Number of stop bits, 1 stop bit (default) or 2 stop bits

uint8_t txFifoWatermark
 TX FIFO watermark

uint8_t rxFifoWatermark
 RX FIFO watermark

bool enableRxRTS
 RX RTS enable

bool enableTxCTS
 TX CTS enable

uart_idle_type_select_t idleType
 IDLE type select.

bool enableTx
 Enable TX

bool enableRx
 Enable RX

struct __uart_transfer
 #include <fsl_uart.h> UART transfer structure.

Public Members

size_t dataSize
 The byte count to be transfer.

struct __uart_handle
 #include <fsl_uart.h> UART 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.

`uart_transfer_callback_t callback`
Callback function.

`void *userData`
UART callback function parameter.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state

`union __unnamed11__`

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.

2.25 WDOG8: 8-bit Watchdog Timer

`void WDOG8_GetDefaultConfig(wdog8_config_t *config)`

Initializes the WDOG8 configuration structure.

This function initializes the WDOG8 configuration structure to default values. The default values are:

```
wdog8Config->enableWdog8 = true;
wdog8Config->clockSource = kWDOG8_ClockSource1;
wdog8Config->prescaler = kWDOG8_ClockPrescalerDivide1;
wdog8Config->workMode.enableWait = true;
wdog8Config->workMode.enableStop = false;
wdog8Config->workMode.enableDebug = false;
wdog8Config->testMode = kWDOG8_TestModeDisabled;
```

(continues on next page)

(continued from previous page)

```
wdog8Config->enableUpdate = true;
wdog8Config->enableInterrupt = false;
wdog8Config->enableWindowMode = false;
wdog8Config->windowValue = 0U;
wdog8Config->timeoutValue = 0xFFFFU;
```

See also:`wdog8_config_t`**Parameters**

- `config` – Pointer to the WDOG8 configuration structure.

```
void WDOG8_Init(WDOG_Type *base, const wdog8_config_t *config)
```

Initializes the WDOG8 module.

This function initializes the WDOG8. To reconfigure the WDOG8 without forcing a reset first, `enableUpdate` must be set to true in the configuration.

Example:

```
wdog8_config_t config;
WDOG8_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG8_Init(wdog_base, &config);
```

Parameters

- `base` – WDOG8 peripheral base address.
- `config` – The configuration of the WDOG8.

```
void WDOG8_Deinit(WDOG_Type *base)
```

De-initializes the WDOG8 module.

This function shuts down the WDOG8. Ensure that the `WDOG_CS1.UPDATE` is 1, which means that the register update is enabled.

Parameters

- `base` – WDOG8 peripheral base address.

```
static inline void WDOG8_Enable(WDOG_Type *base)
```

Enables the WDOG8 module.

This function writes a value into the `WDOG_CS1` register to enable the WDOG8. The `WDOG_CS1` register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG8 peripheral base address.

```
static inline void WDOG8_Disable(WDOG_Type *base)
```

Disables the WDOG8 module.

This function writes a value into the `WDOG_CS1` register to disable the WDOG8. The `WDOG_CS1` register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG8 peripheral base address

```
static inline void WDOG8_EnableInterrupts(WDOG_Type *base, uint8_t mask)
```

Enables the WDOG8 interrupt.

This function writes a value into the WDOG_CS1 register to enable the WDOG8 interrupt. The WDOG_CS1 register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG8 peripheral base address.
- `mask` – The interrupts to enable. The parameter can be a combination of the following source if defined:
 - `kWDOG8_InterruptEnable`

```
static inline void WDOG8_DisableInterrupts(WDOG_Type *base, uint8_t mask)
```

Disables the WDOG8 interrupt.

This function writes a value into the WDOG_CS register to disable the WDOG8 interrupt. The WDOG_CS register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- `base` – WDOG8 peripheral base address.
- `mask` – The interrupts to disabled. The parameter can be a combination of the following source if defined:
 - `kWDOG8_InterruptEnable`

```
static inline uint8_t WDOG8_GetStatusFlags(WDOG_Type *base)
```

Gets the WDOG8 all status flags.

This function gets all status flags.

Example to get the running flag:

```
uint32_t status;
status = WDOG8_GetStatusFlags(wdog_base) & kWDOG8_RunningFlag;
```

See also:

`_wdog8_status_flags_t`

- `true`: related status flag has been set.
- `false`: related status flag is not set.

Parameters

- `base` – WDOG8 peripheral base address

Returns

State of the status flag: asserted (`true`) or not-asserted (`false`).

```
void WDOG8_ClearStatusFlags(WDOG_Type *base, uint8_t mask)
```

Clears the WDOG8 flag.

This function clears the WDOG8 status flag.

Example to clear an interrupt flag:

```
WDOG8_ClearStatusFlags(wdog_base, kWDOG8_InterruptFlag);
```

Parameters

- `base` – WDOG8 peripheral base address.

- mask – The status flags to clear. The parameter can be any combination of the following values:
 - kWDOG8_InterruptFlag

static inline void WDOG8_SetTimeoutValue(WDOG_Type *base, uint16_t timeoutCount)

Sets the WDOG8 timeout value.

This function writes a timeout value into the WDOG_TOVALH/L register. The WDOG_TOVALH/L register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG8 peripheral base address
- timeoutCount – WDOG8 timeout value, count of WDOG8 clock ticks.

static inline void WDOG8_SetWindowValue(WDOG_Type *base, uint16_t windowValue)

Sets the WDOG8 window value.

This function writes a window value into the WDOG_WINH/L register. The WDOG_WINH/L register is a write-once register. Ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

- base – WDOG8 peripheral base address.
- windowValue – WDOG8 window value.

static inline void WDOG8_Unlock(WDOG_Type *base)

Unlocks the WDOG8 register written.

This function unlocks the WDOG8 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG8 peripheral base address

static inline void WDOG8_Refresh(WDOG_Type *base)

Refreshes the WDOG8 timer.

This function feeds the WDOG8. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG8 peripheral base address

static inline uint16_t WDOG8_GetCounterValue(WDOG_Type *base)

Gets the WDOG8 counter value.

This function gets the WDOG8 counter value.

Parameters

- base – WDOG8 peripheral base address.

Returns

Current WDOG8 counter value.

Chapter 3

Middleware

3.1 Motor Control

3.1.1 FreeMASTER

Communication Driver User Guide

Introduction

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

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

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.
- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- ...and all of the above also using a **Zephyr** generic drivers.

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

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

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

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

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

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

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

- **General CPU Platform** (see source code in the `src/platforms` directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.
- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called `FMSTR_TRANSPORT` with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The `mcuxsdk` folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The “ampsdk” drivers target automotive-specific MCUs and their respective SDKs. The “dreg” implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

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

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

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

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

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

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

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

Related links:

- [The official FreeMASTER middleware repository.](#)
- [Online version of this document](#)

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

Example applications

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

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer’s physical or virtual COM port. The typical transmission speed is 115200 bps.

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

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

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

Description

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

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

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

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

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

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

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

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

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

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

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

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

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

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

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

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

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

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

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

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

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

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

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

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

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

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

- “External” single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- “True” single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

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

Zephyr-specific

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

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

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

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

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the .c files must be added to the project, compiled, and linked together with the application.
 - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
 - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
 - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
 - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
 - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
 - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.
- **src/drivers/[sdk]/serial** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- **src/drivers/[sdk]/can** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- **src/drivers/[sdk]/network** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
 - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
 - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

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

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

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

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

Interrupt modes

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

Value Type boolean (0 or 1)

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

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

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

Protocol transport

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

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

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

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

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

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

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

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

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

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

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

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

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

FMSTR_SERIAL_BASE

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

Value Type Optional address value (numeric or symbolic)

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

FMSTR_COMM_BUFFER_SIZE

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

Value Type 0 or a value in range 32...255

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

FMSTR_COMM_RQUEUE_SIZE

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

Value Type Value in range 0...255

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

FMSTR_SERIAL_SINGLEWIRE

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

Value Type Boolean 0 or 1.

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

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

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

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

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

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

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

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

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

FMSTR_CAN_BASE

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

Value Type Optional address value (numeric or symbolic)

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

FMSTR_CAN_CMDID

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

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

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

FMSTR_CAN_RSPID

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

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

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

FMSTR_FLEXCAN_TXMB

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

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

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

FMSTR_FLEXCAN_RXMB

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

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

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

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

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

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

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

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

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

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

Add another row below:

FMSTR_NET_PORT

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

Value Type TCP or UDP port number (short integer)

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

FMSTR_NET_BLOCKING_TIMEOUT

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

Value Type Timeout as number of milliseconds

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

FMSTR_NET_AUTODISCOVERY

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

Value Type Boolean 0 or 1.

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

Debugging options**FMSTR_DISABLE**

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

Value Type boolean (0 or 1)

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

FMSTR_DEBUG_TX

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

Value Type Boolean 0 or 1.

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

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

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

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_WRITEMEM

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

Value Type Boolean 0 or 1.

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

Oscilloscope options**FMSTR_USE_SCOPE**

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

Value Type Integer number.

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

FMSTR_MAX_SCOPE_VARS

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

Value Type Integer number larger than 2.

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

Recorder options**FMSTR_USE_RECORDER**

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

Value Type Integer number.

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

Default value is 0.

FMSTR_REC_BUFF_SIZE

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

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.

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

FMSTR_REC_TIMEBASE

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

Value Type Number (nanoseconds time).

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

Use one of the following macros:

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

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

FMSTR_REC_FLOAT_TRIG

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

Value Type Boolean 0 or 1.

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

Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

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

Value Type Boolean 0 or 1.

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

FMSTR_APPCMD_BUFF_SIZE

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

Value Type Numeric buffer size in range 1..255

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

FMSTR_MAX_APPCMD_CALLS

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

Value Type Number in range 0..255

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

TSA options**FMSTR_USE_TSA**

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_SAFETY

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_INROM

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_DYNAMIC

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

Value Type Boolean 0 or 1.

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

Pipes options

FMSTR_USE_PIPES

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

Value Type Boolean 0 or 1.

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

FMSTR_MAX_PIPES_COUNT

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

Value Type Number in range 1..63.

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

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

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

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

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

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

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

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

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

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

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_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_RQUEUE_SIZE* macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

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

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

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

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

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

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

Recorder API

FMSTR_RecorderCreate

Prototype

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

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

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

For more information, see [Configurable items](#).

FMSTR_Recorder

Prototype

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

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

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

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

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

FMSTR_RecorderTrigger

Prototype

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

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

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

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

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

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

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

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

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-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 *fmrstr:* 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 resultDataAddr, FMSTR_SIZE resultDataLen);
```

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

Arguments

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

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

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

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

FMSTR_RegisterAppCmdCall

Prototype

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

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

Arguments

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

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

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

The prototype of the callback function is

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

Where:

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

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

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

Pipes API

FMSTR_PipeOpen

Prototype

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

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

Arguments

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

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

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

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

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

FMSTR_PipeClose

Prototype

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

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

Arguments

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

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

FMSTR_PipeWrite

Prototype

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

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

Arguments

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

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

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the *nGranularity* value equal to the *nLength* value, all data are considered as one chunk which is either written successfully as a whole or not at all. The *nGranularity* value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

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

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

Arguments

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

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

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

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

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

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

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

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

<i>FM-STR_TSA_TII</i>	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables. By default, this is defined as <i>FM-STR_SIZE</i> .
<i>FM-STR_TSA_TS</i>	Data type used to hold a memory block size, as used in the TSA descriptors. By default, this is defined as <i>FM-STR_SIZE</i> .

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

<i>FM-STR_HPIPE</i>	Pipe handle that identifies the open-pipe object. Generally, this is a pointer to a void type.
<i>FM-STR_PIPE_PC</i>	Integer type required to hold at least 7 bits of data. Generally, this is an unsigned 8-bit or 16-bit type.
<i>FM-STR_PIPE_SI</i>	Integer type required to hold at least 16 bits of data. This is used to store the data buffer sizes.
<i>FM-STR_PPIPEF</i>	Pointer to the pipe handler function. See FM-STR_PipeOpen for more details.

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

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

Document references

Links

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

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

Documents

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

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

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

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