MCUXpresso SDK Documentation
Release 25.09.00

Table of contents

This documentation contains information specific to the twrkm34z50mv3 board.

Chapter 1

TWR-KM34Z50MV3

1.1 Overview
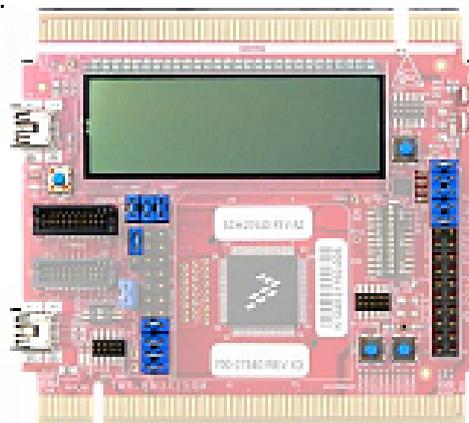
MCU device and part on board is shown below:

- Device: MKM34ZA5
- PartNumber: MKM34Z128ACLL5

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with MCUXpresso SDK Package

Overview

The NXP MCUXpresso software and tools offer comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on general purpose, crossover, and Bluetooth-enabled MCUs from NXP. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to full demo applications. The MCUXpresso SDK contains optional RTOS integrations such as FreeRTOS and Azure RTOS, and various other middleware to support rapid development.

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

For more details about MCUXpresso SDK, see MCUXpresso Software Development Kit (SDK).
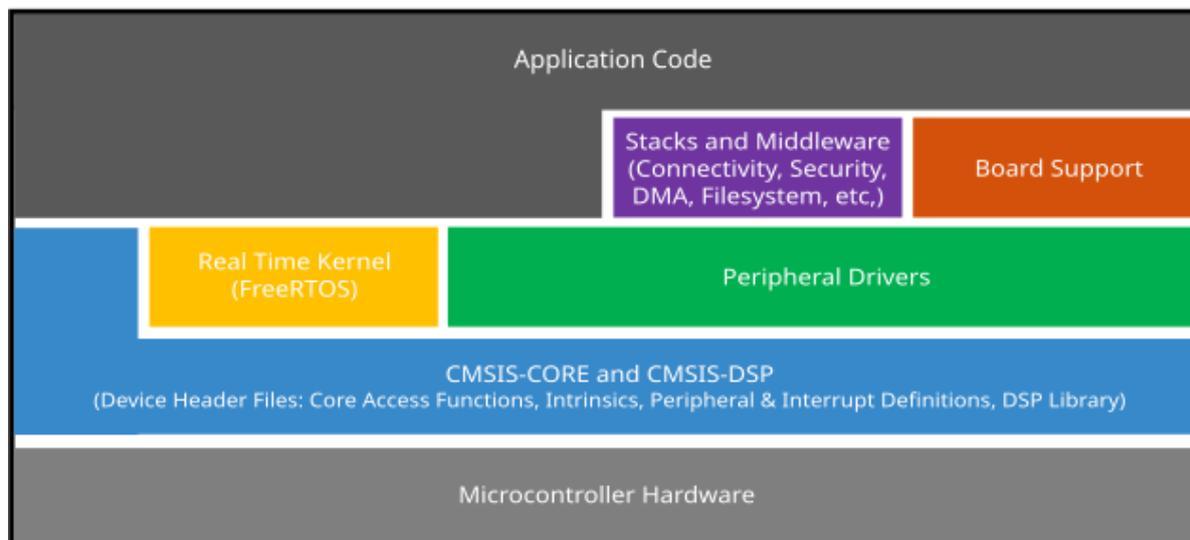
MCUXpresso SDK board support package folders

MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCX-presso 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:
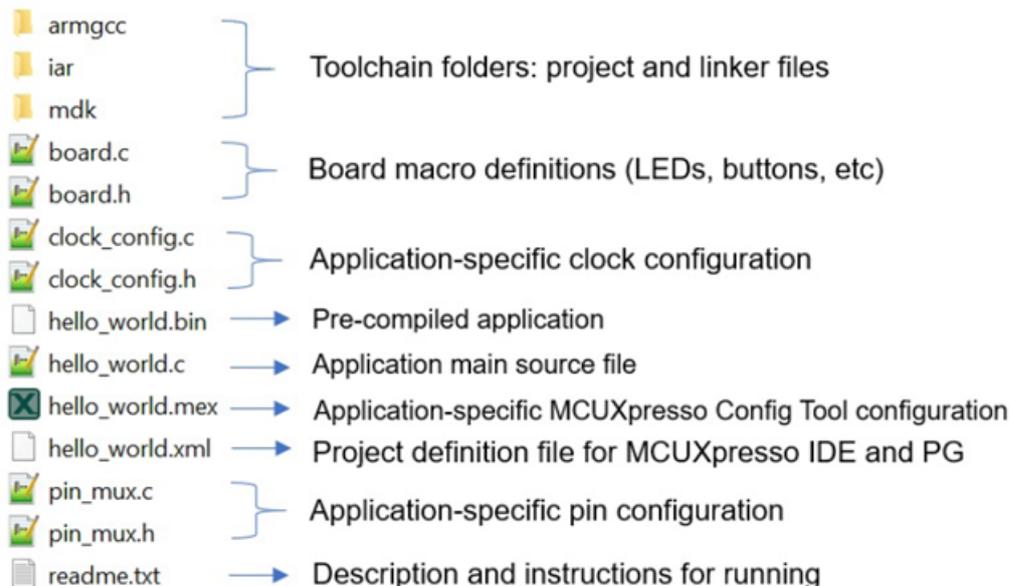
All files in the application folder are specific to that example, so it is easy to copy and paste an existing example to start developing a custom application based on a project provided in the MCUXpresso SDK.

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.
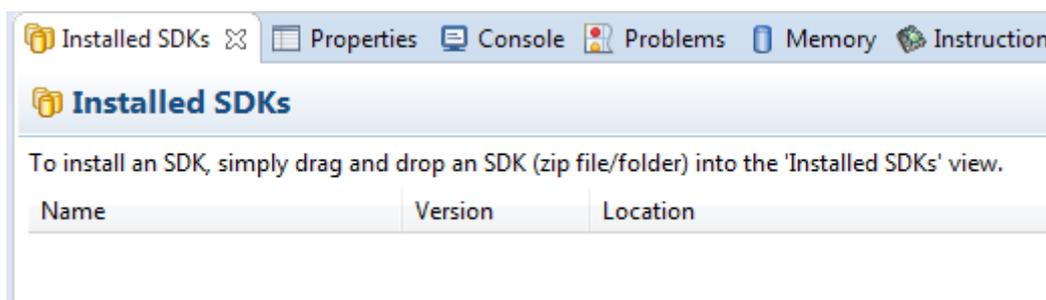
2. On the **Quickstart Panel**, click **Import SDK example(s)…**.
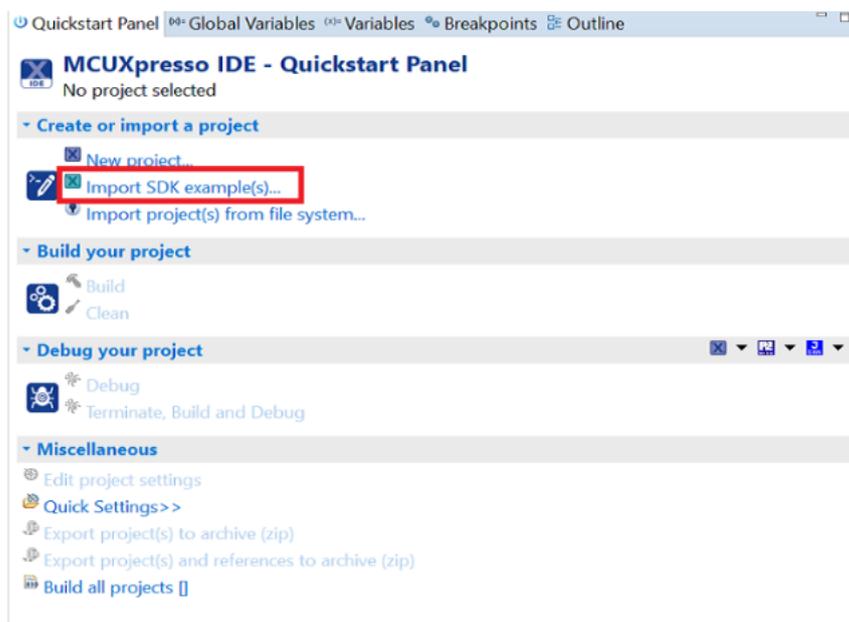
3. Expand the demo_apps folder and select hello_world.

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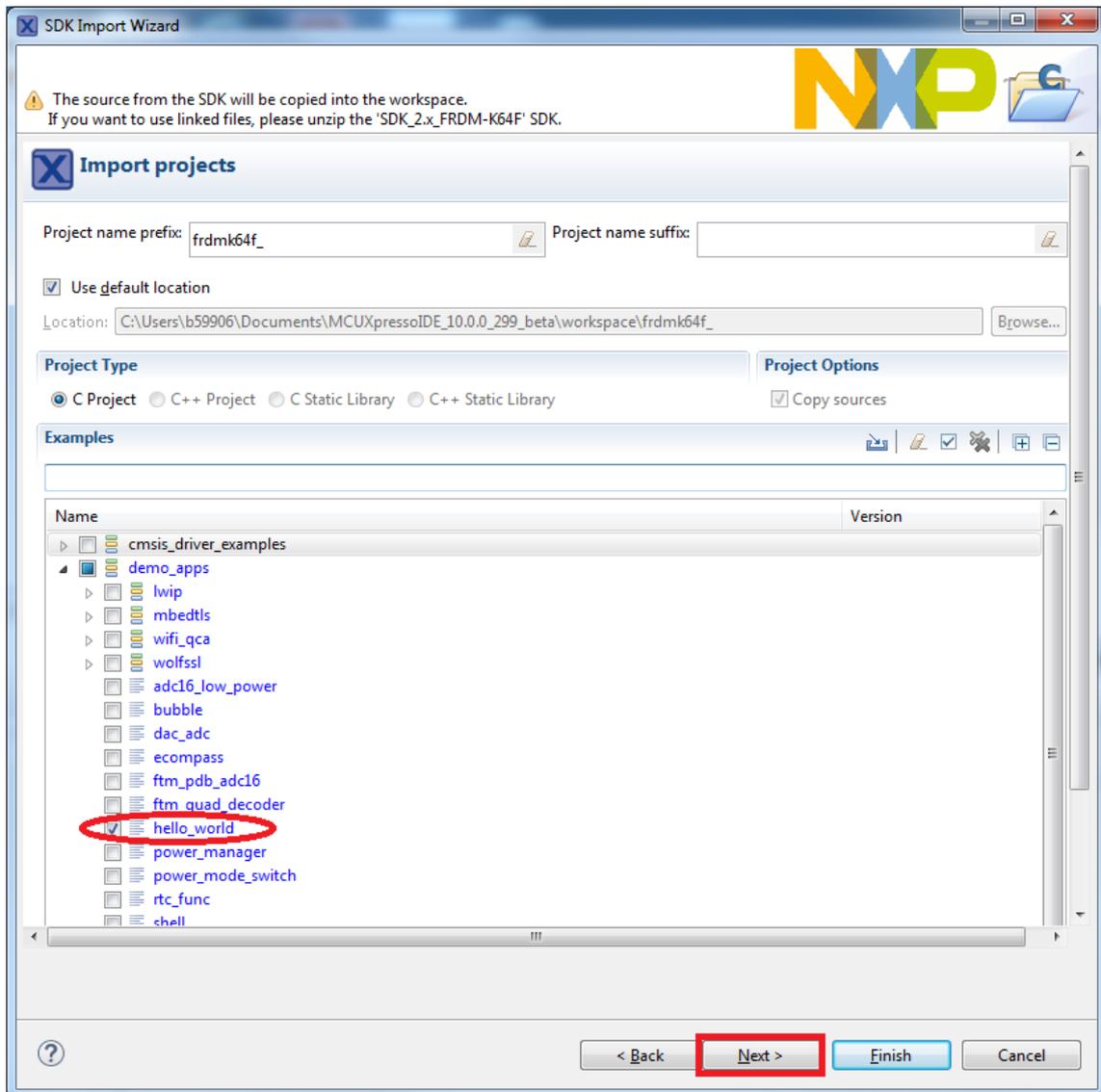
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminalfor demo applications such as adc_basic, adc_burst, adc_dma, and adc_interrupt. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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
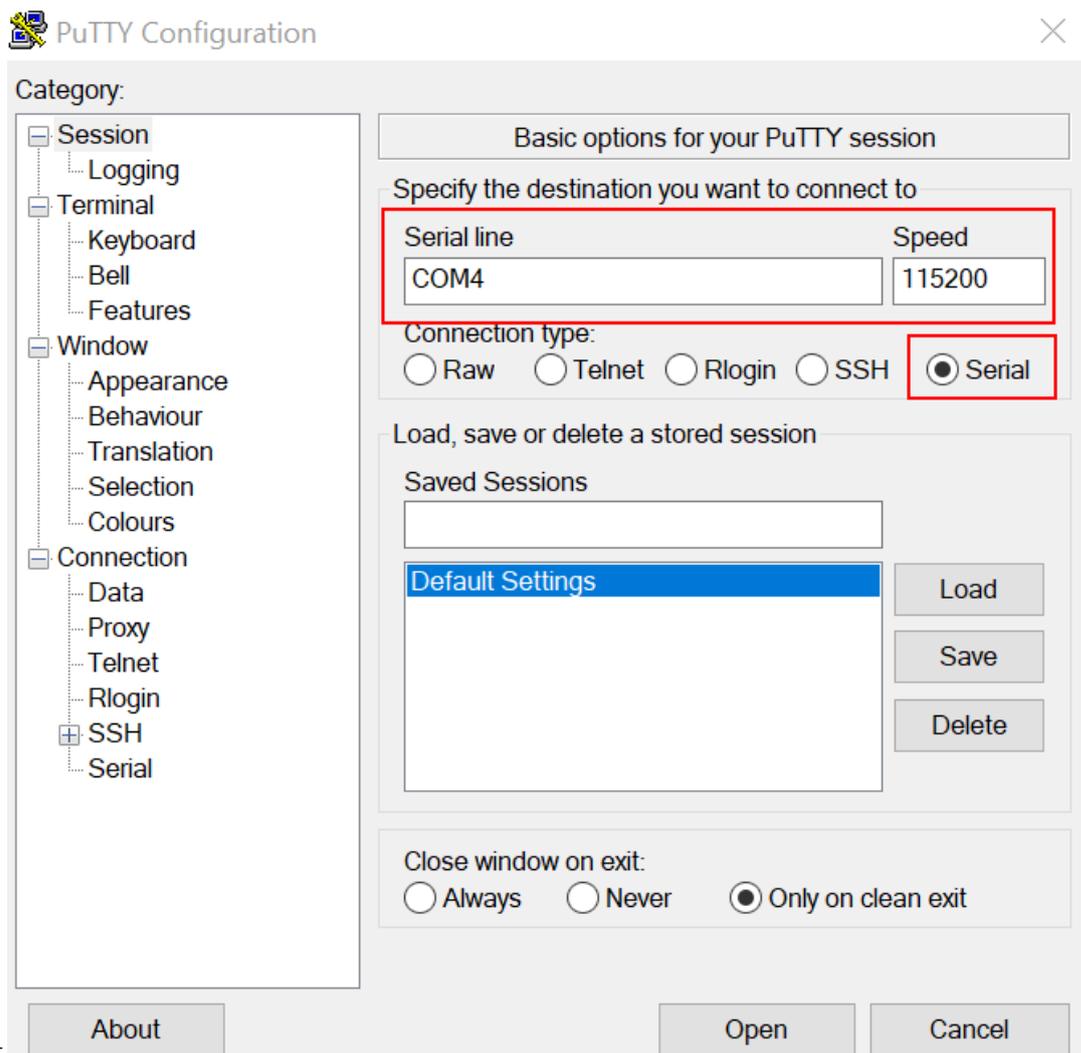
4. 1 stop bit

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.)
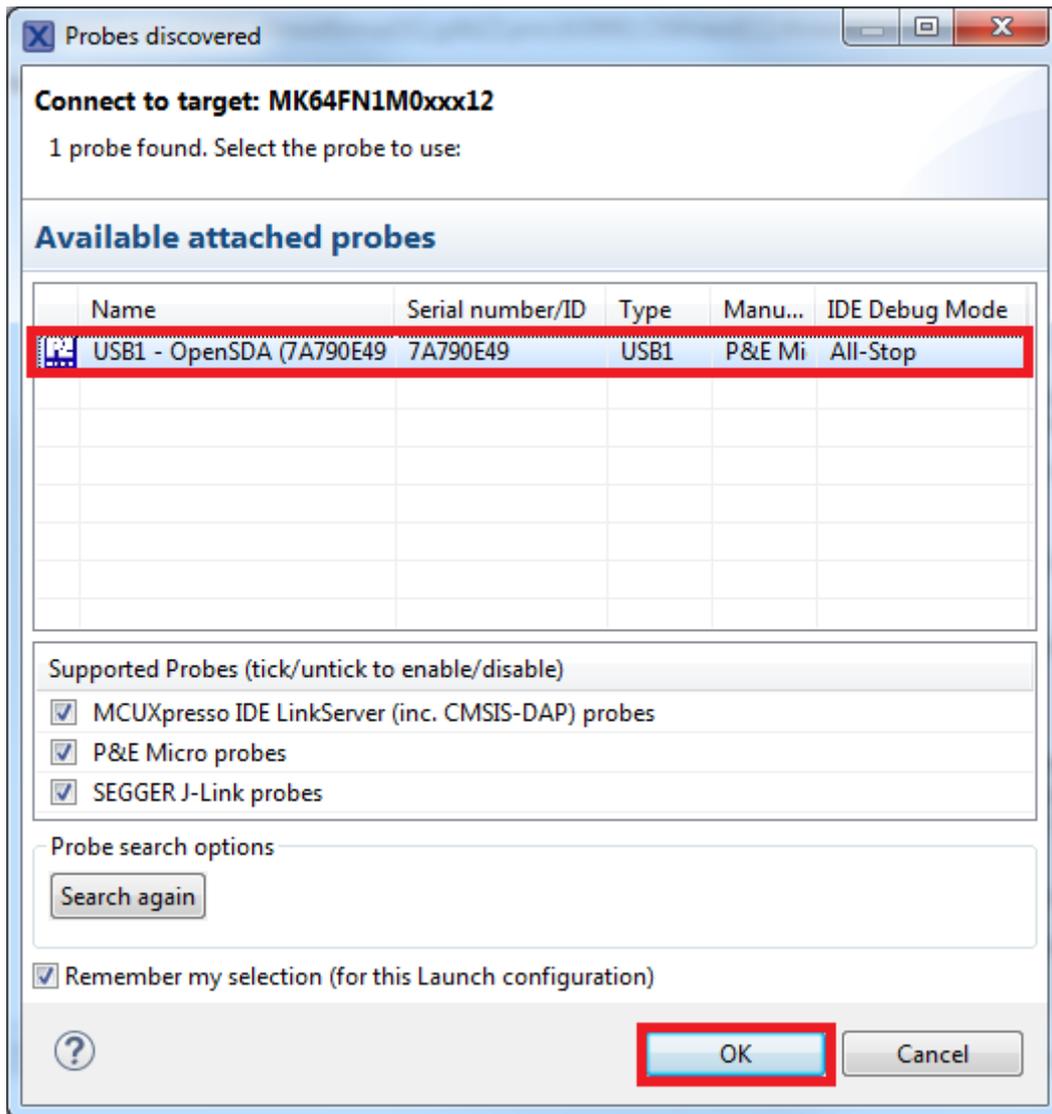
6. The application is downloaded to the target and automatically runs to $main()$.

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

The $hello\_world$ application is now running and a banner is displayed on the terminal. If 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**.
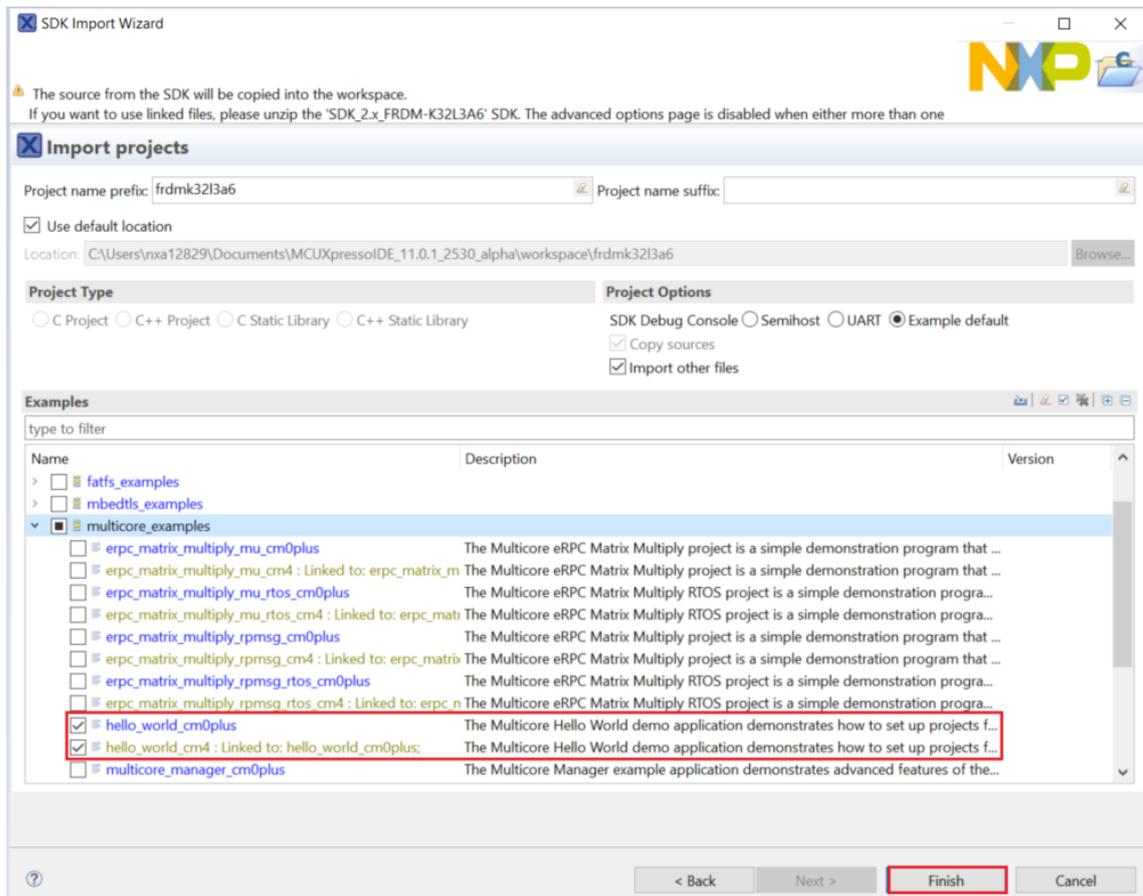
3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the lpcxpresso54114_multicore_examples_hello_world_cm4 project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in the figure. For this example, select **Debug**.
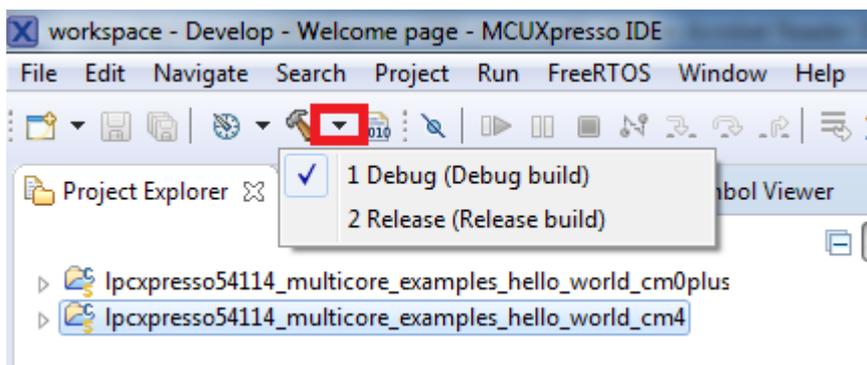
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

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.
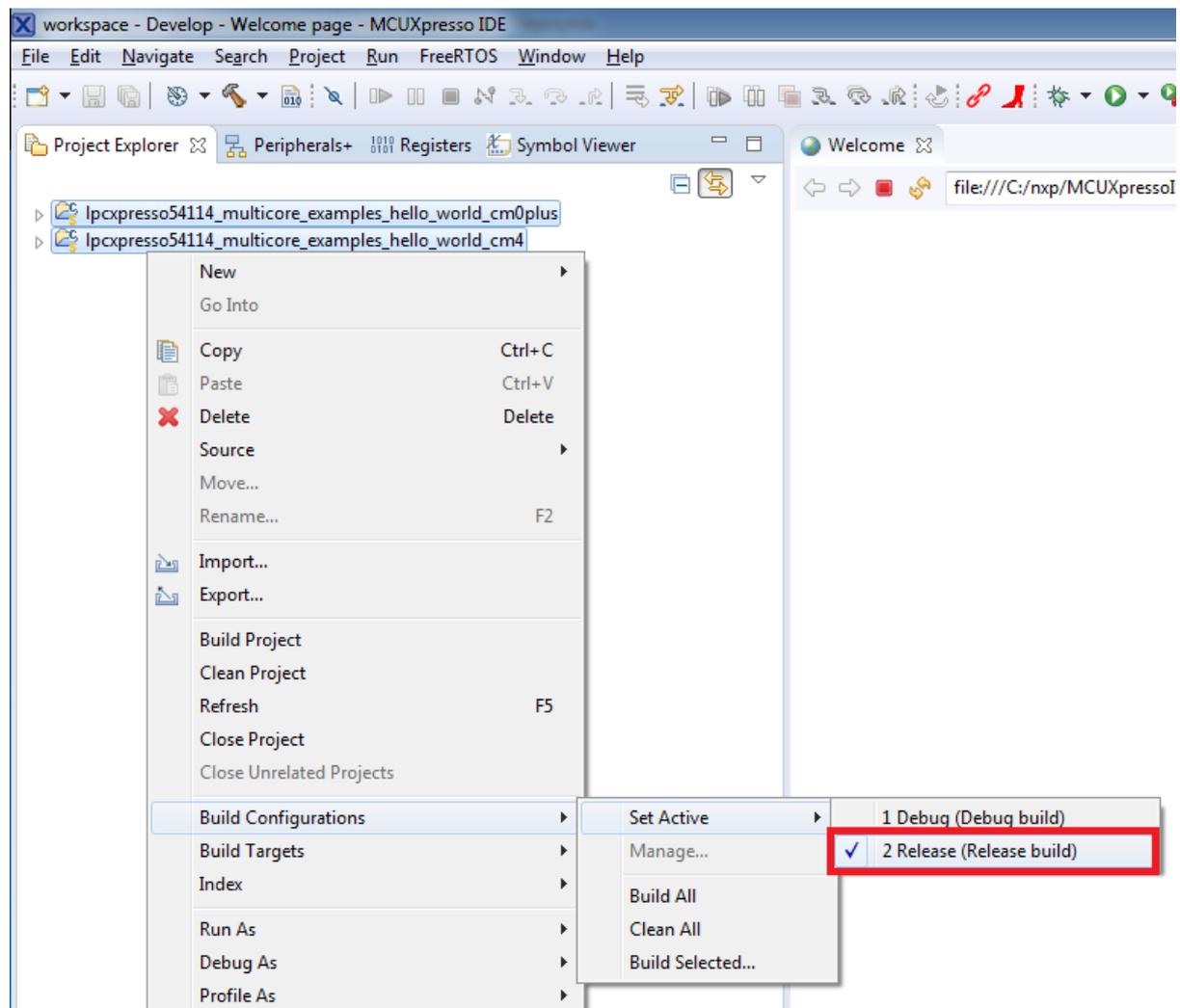
Run a multicore example application   The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run 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.
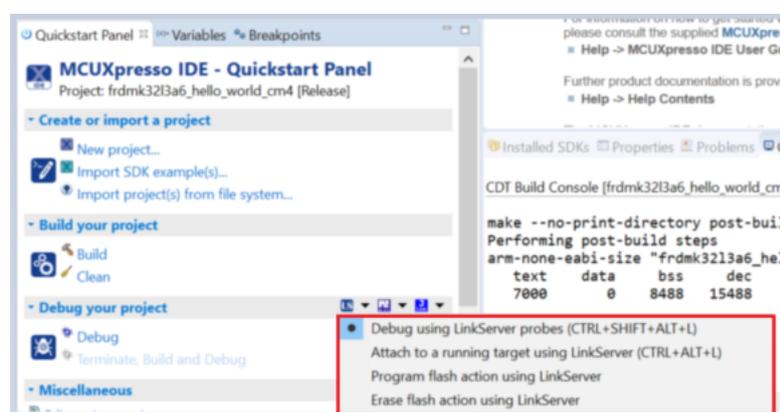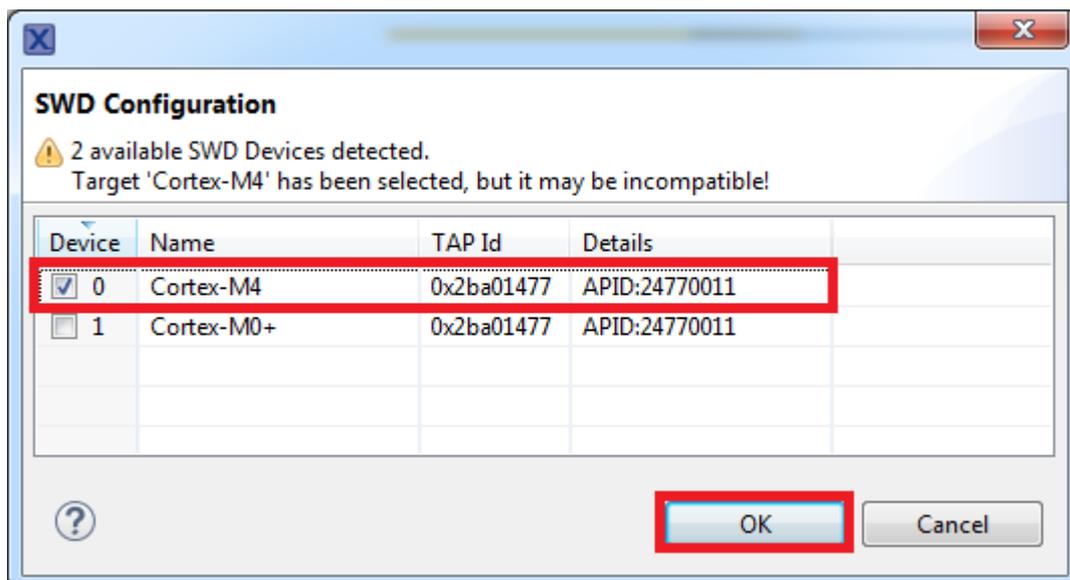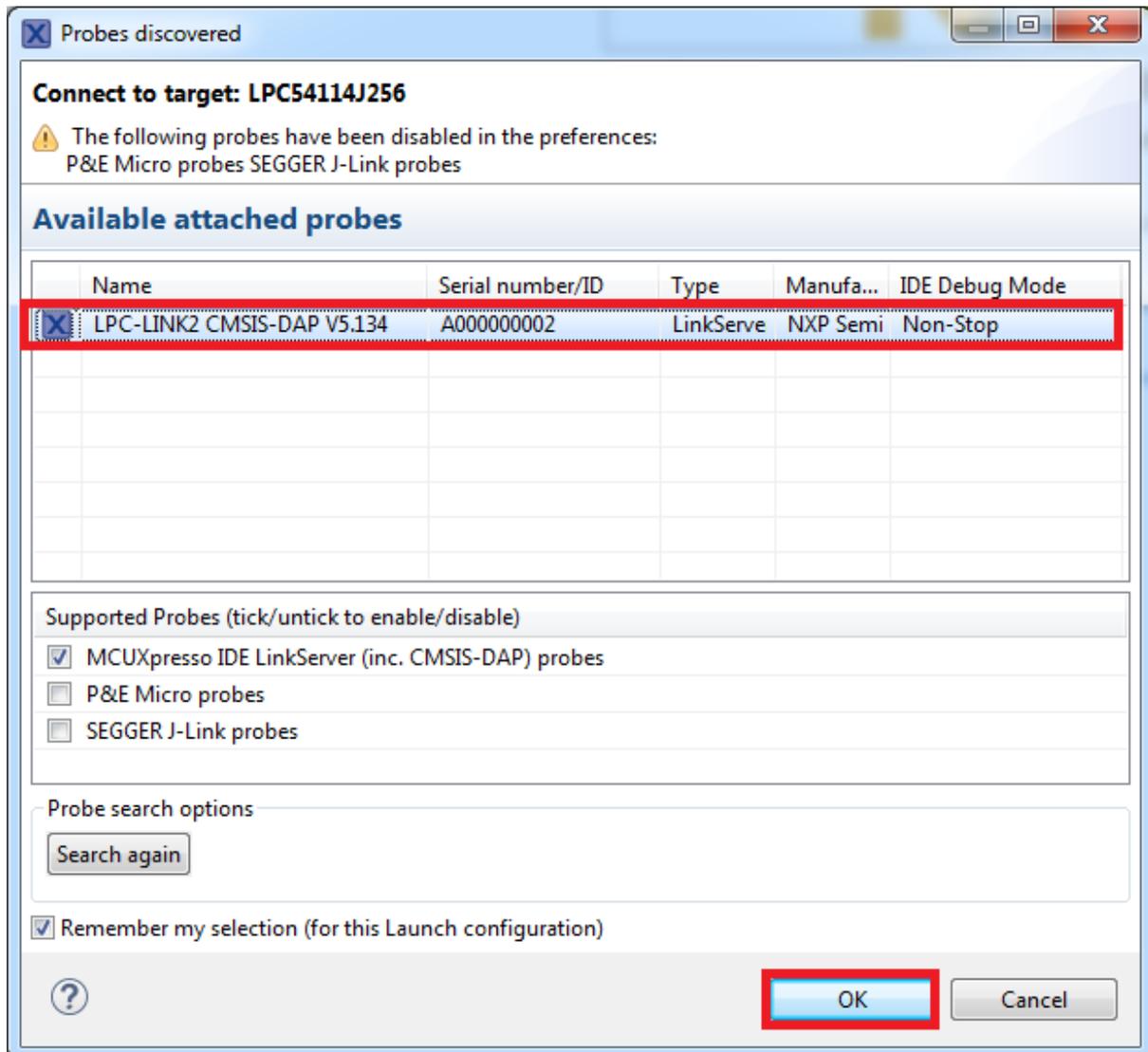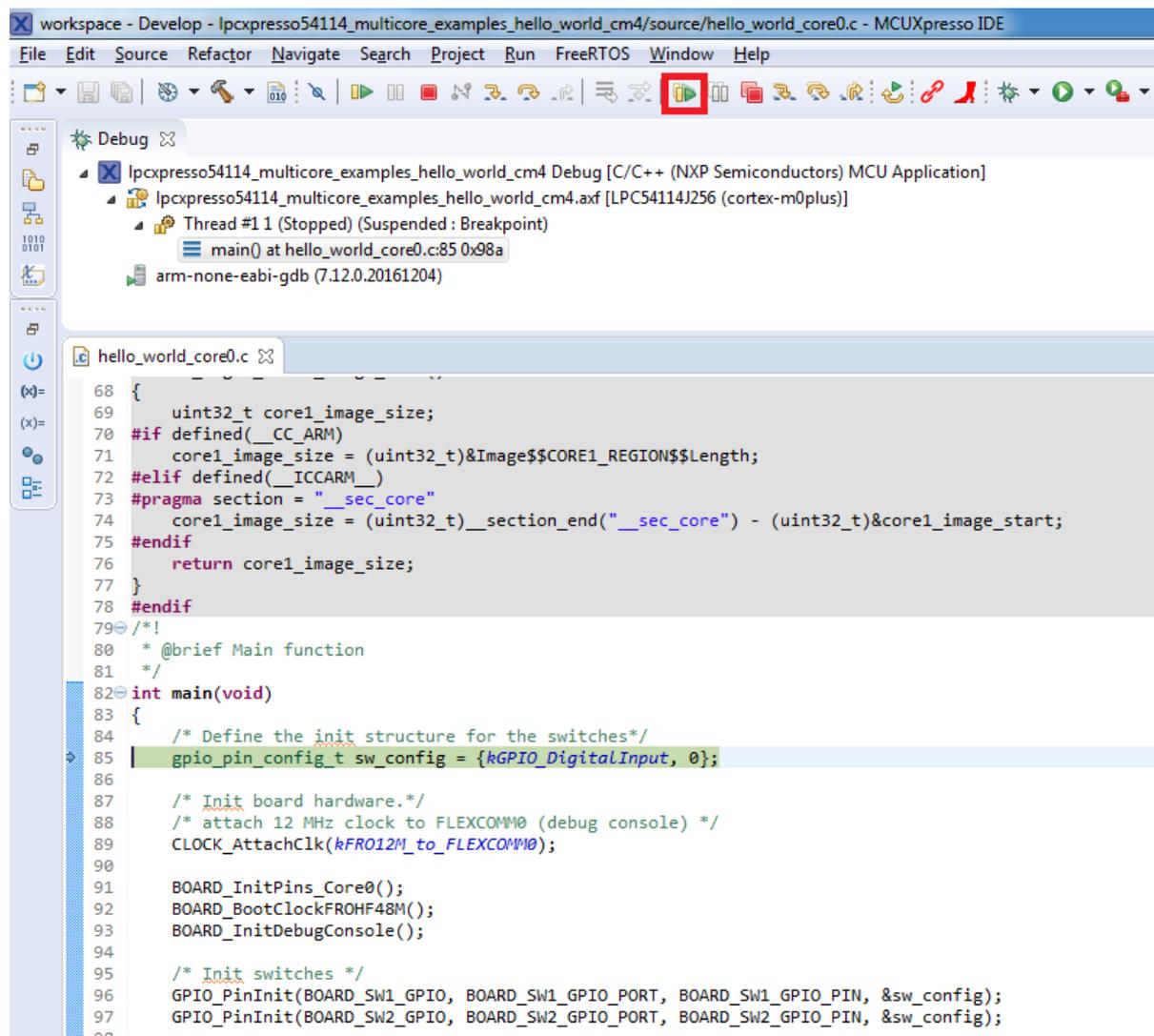
After clicking the "Resume All Debug sessions" button, the hello_world multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.
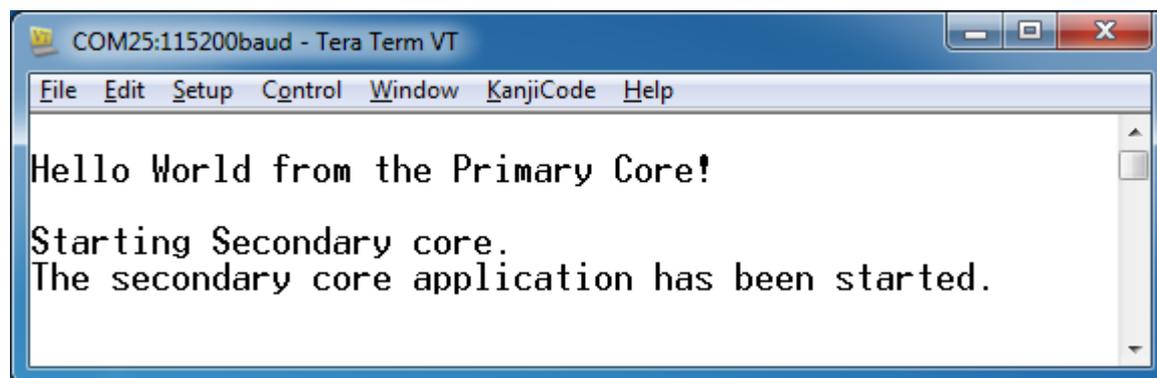
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps 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.

```
68  {
69      uint32_t core1_image_size;
70  #if defined(__CC_ARM)
71      core1_image_size = (uint32_t)&Image$$CORE1_REGION$$Length;
72  #elif defined(__ICCARM__)
73  #pragma section = "__sec_core"
74      core1_image_size = (uint32_t)__section_end("__sec_core") - (uint32_t)&core1_image_start;
75  #endif
76      return core1_image_size;
77  }
78  #endif
79  /*!
80   * @brief Main function
81   */
82  int main(void)
83  {
84      /* Define the init structure for the switches*/
85      gpio_pin_config_t sw_config = {kGPIO_DigitalInput, 0};
86
87      /* Init board hardware.*/
88      /* attach 12 MHz clock to FLEXCOMM0 (debug console) */
89      CLOCK_AttachClk(kFRO12M_to_FLEXCOMM0);
90
91      BOARD_InitPins_Core0();
92      BOARD_BootClockFROHF48M();
93      BOARD_InitDebugConsole();
94
95      /* Init switches */
96      GPIO_PinInit(BOARD_SW1_GPIO, BOARD_SW1_GPIO_PORT, BOARD_SW1_GPIO_PIN, &sw_config);
97      GPIO_PinInit(BOARD_SW2_GPIO, BOARD_SW2_GPIO_PORT, BOARD_SW2_GPIO_PIN, &sw_config);
98
```

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.
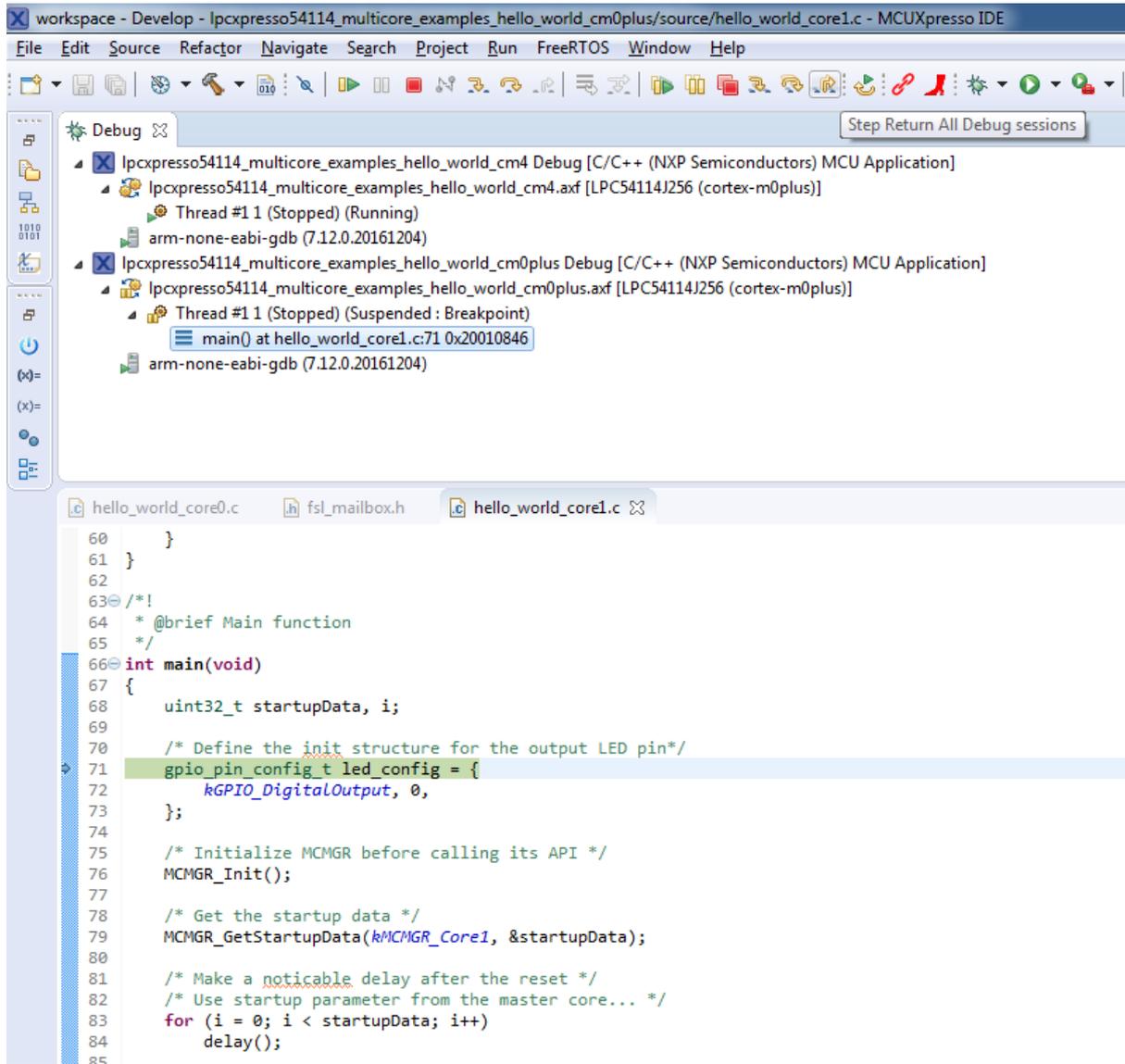
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the "Suspend" / "Resume" control button, or just using the "Suspend All Debug sessions" and the "Resume All Debug sessions" buttons.
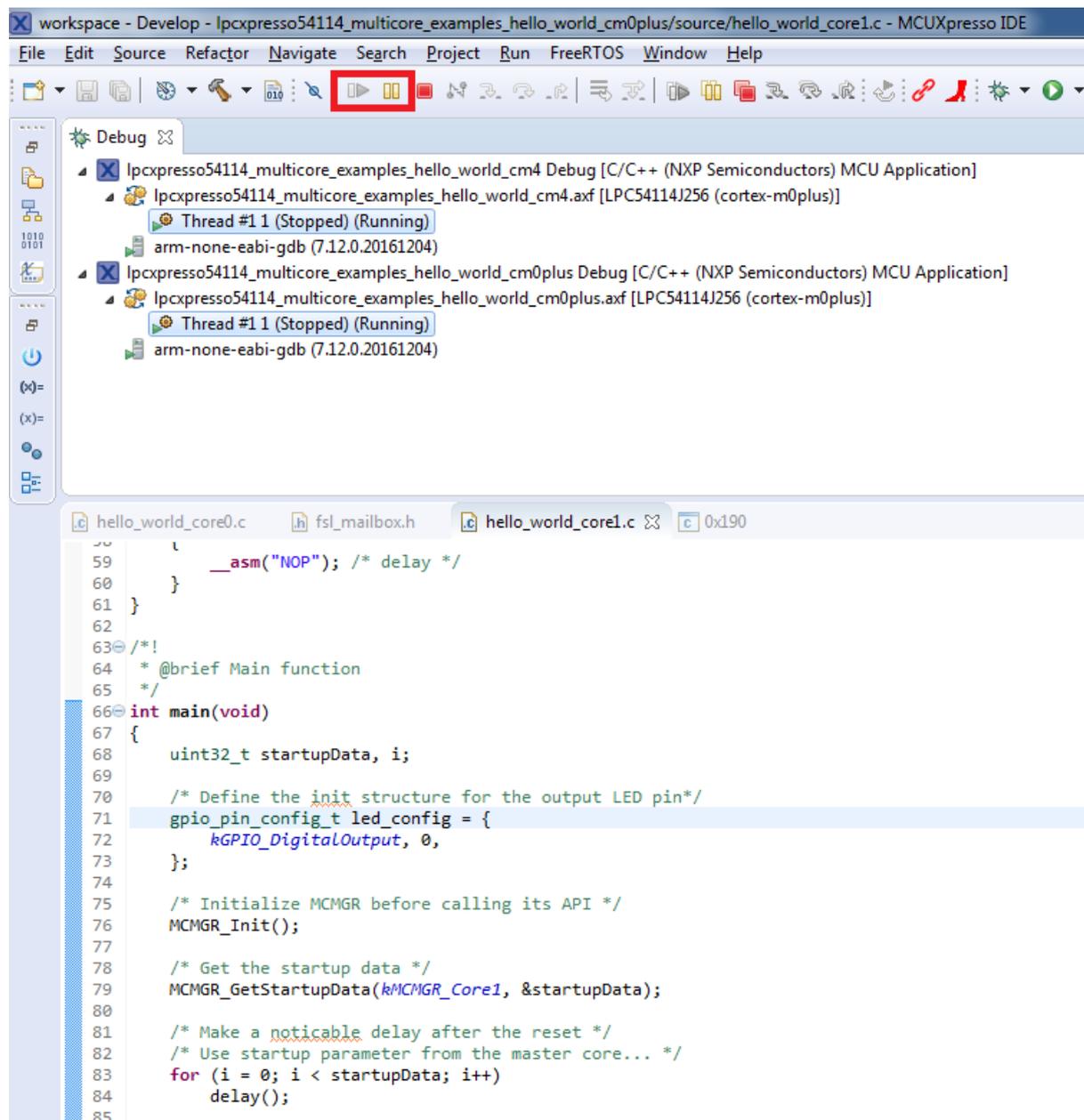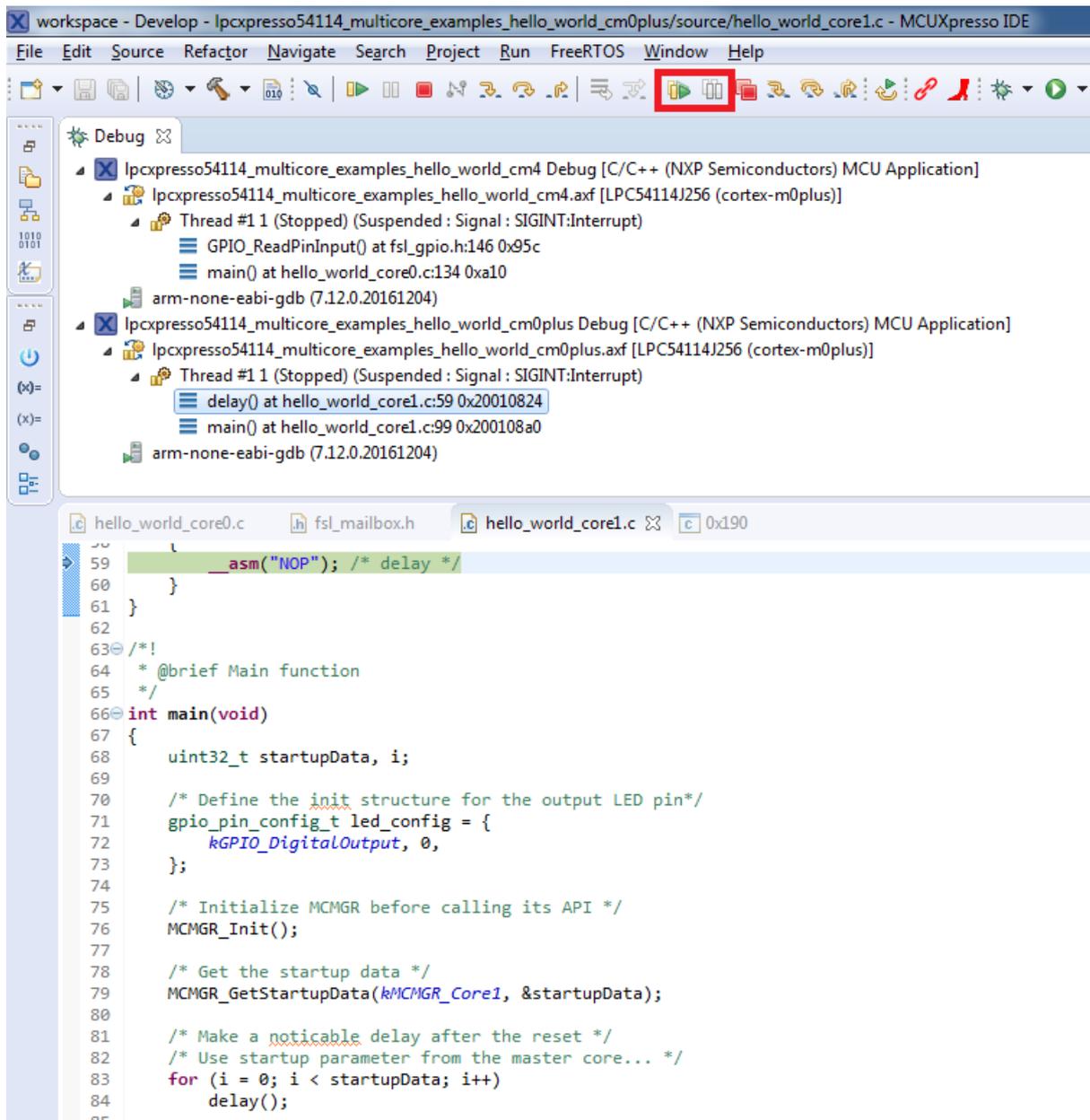
Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the hello_world example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though 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**.
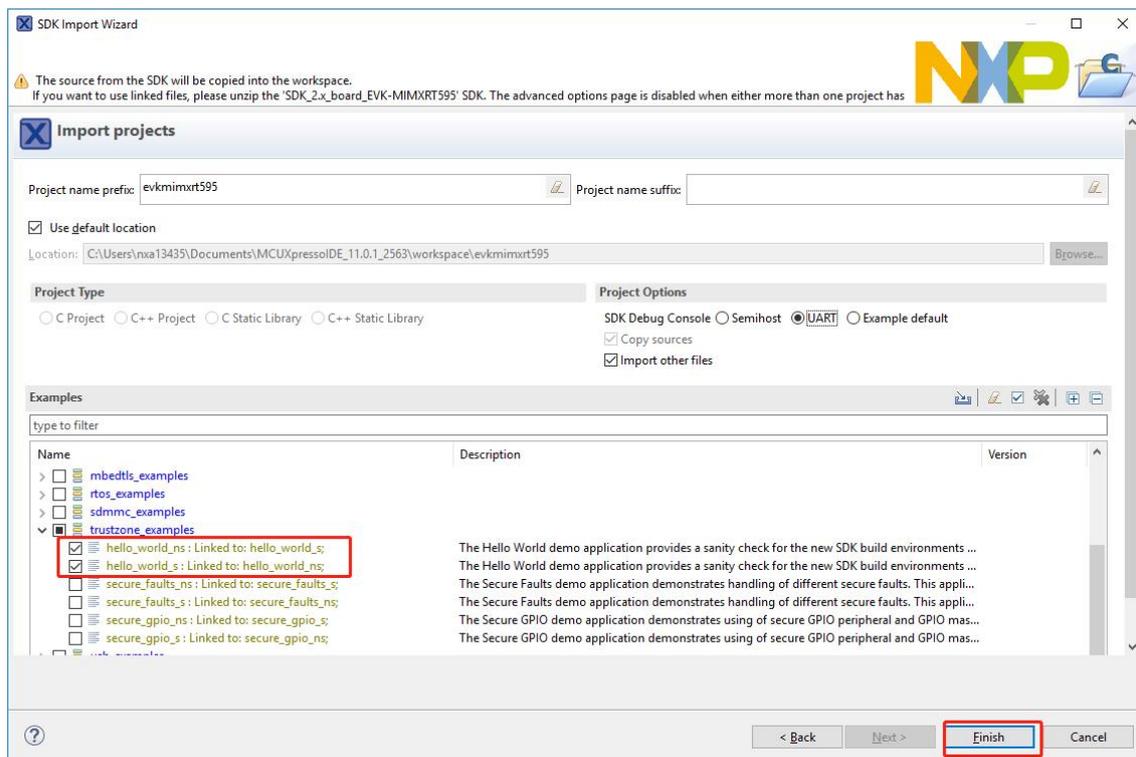
3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the evkmimxrt595_hello_world_s project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in following figure. For this example, select the **Debug** target.
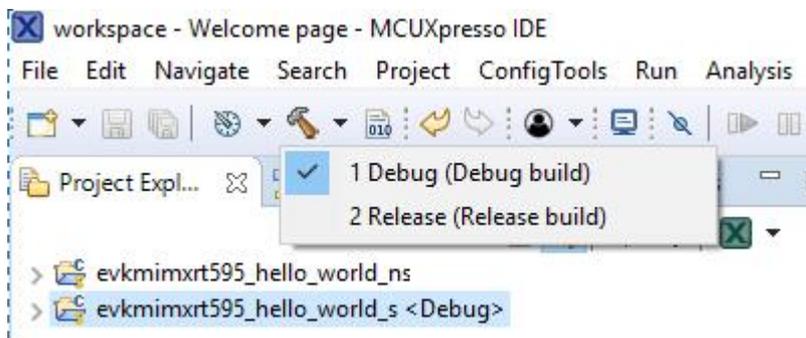

The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when 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.
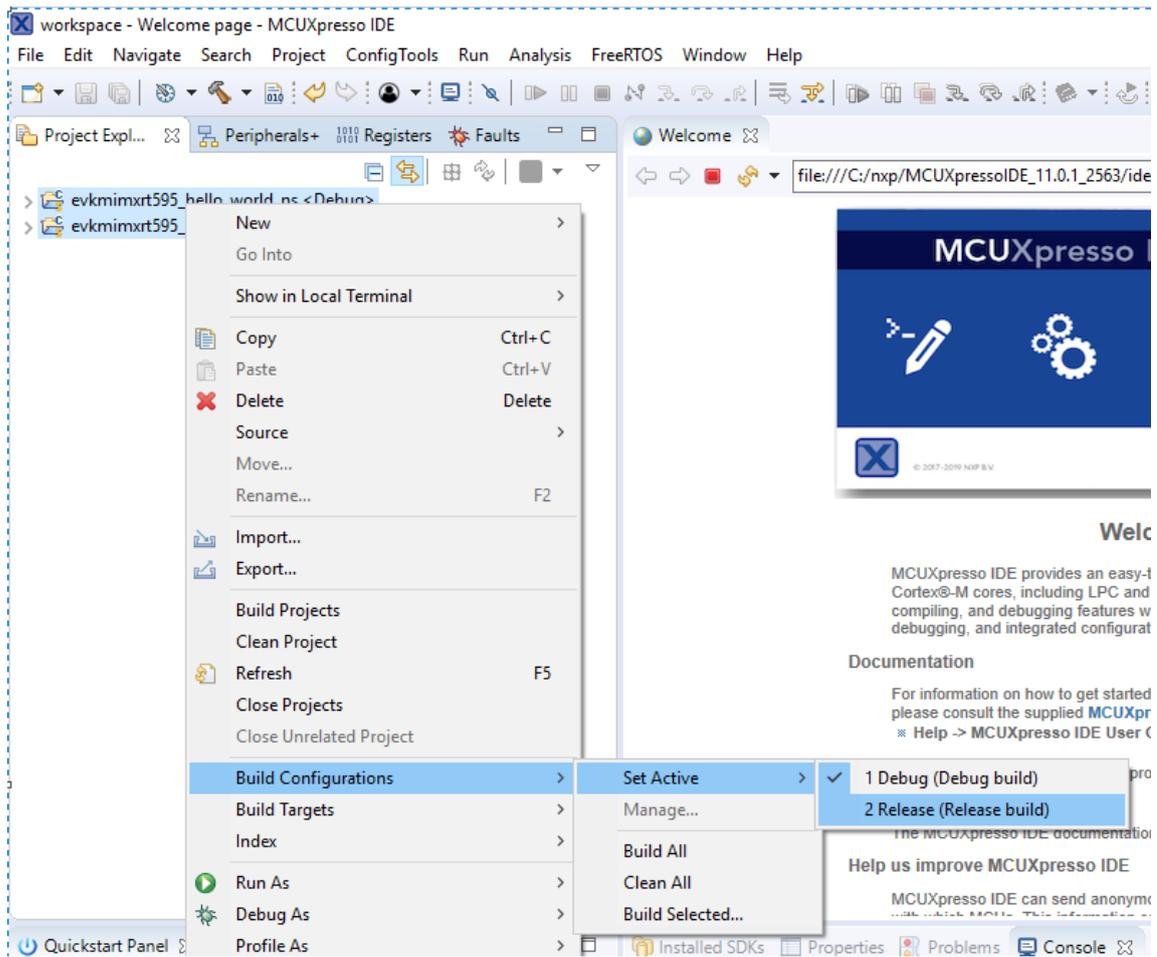
Run a TrustZone example application To download and run the application, perform all steps as described in **Run an example application**. These steps are common for single core, and TrustZone applications, ensuring <board_name>_hello_world_s is selected for debugging.

In the Quickstart Panel, click **Debug** to launch the second debug session.
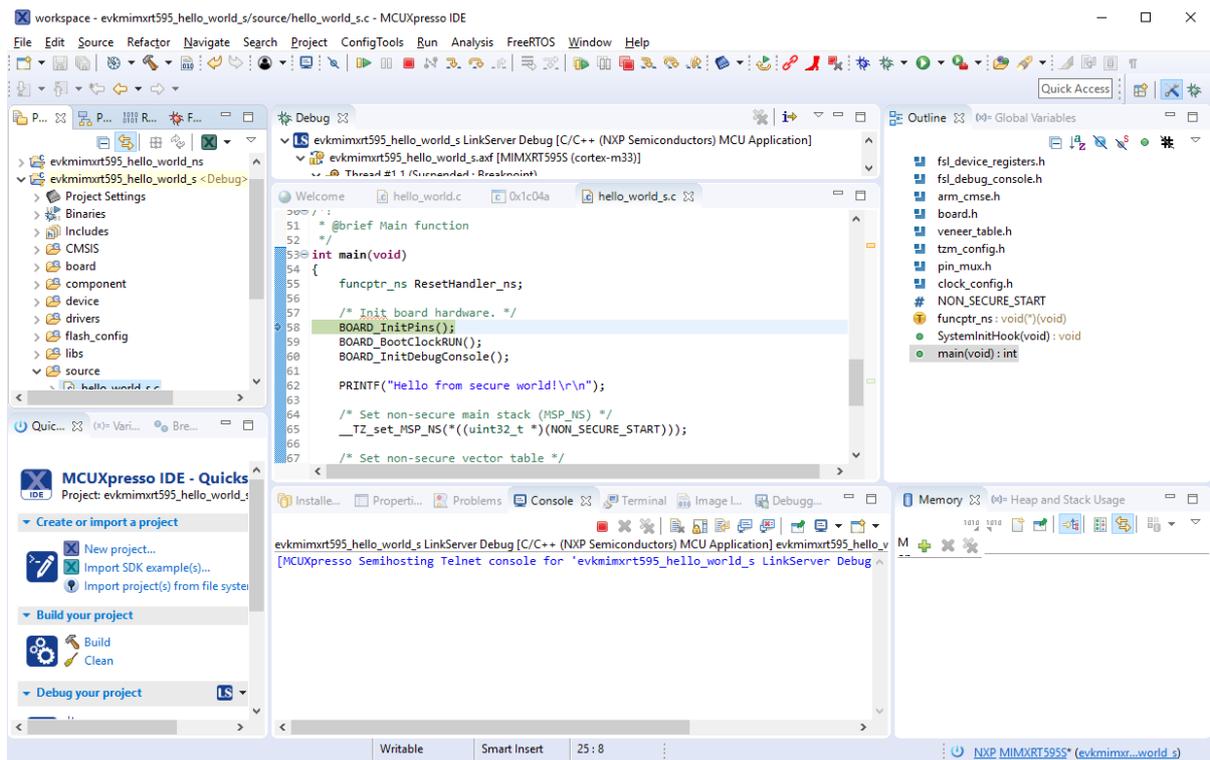
Now, the TrustZone sessions should be opened. Click **Resume**. The hello_world TrustZone application then starts running, and the secure application starts the non-secure application during runtime.

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**.
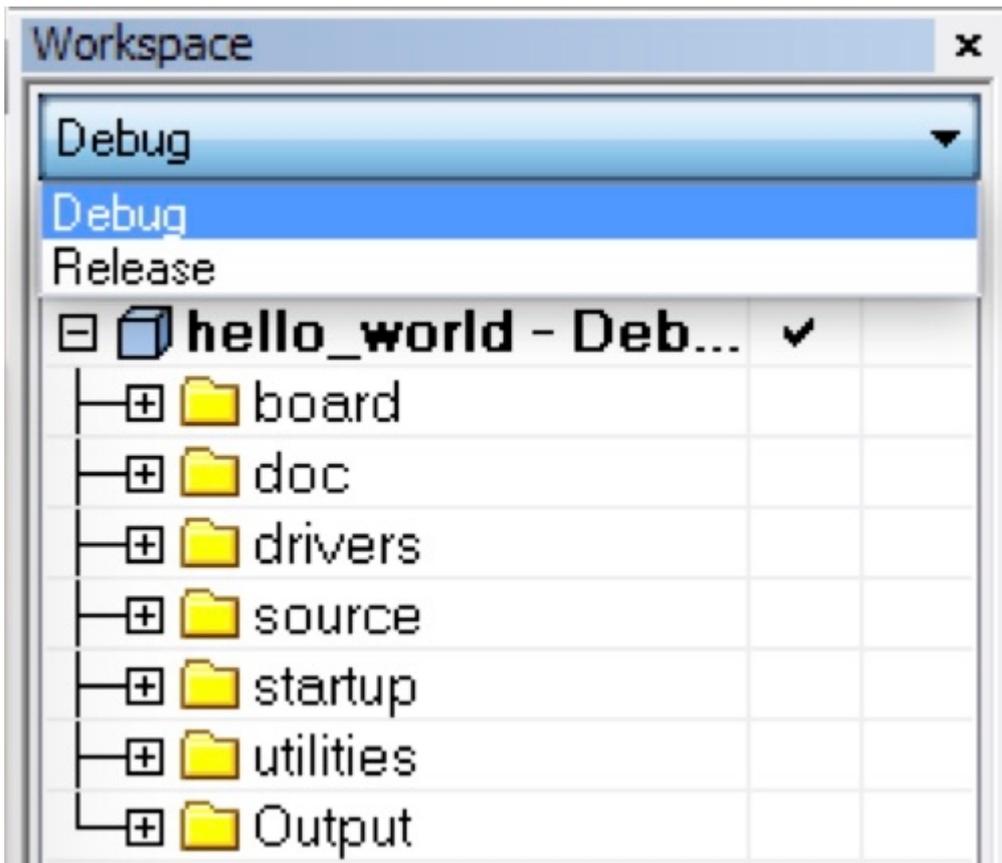
3. To build the demo application, click **Make**, highlighted in red in following figure.
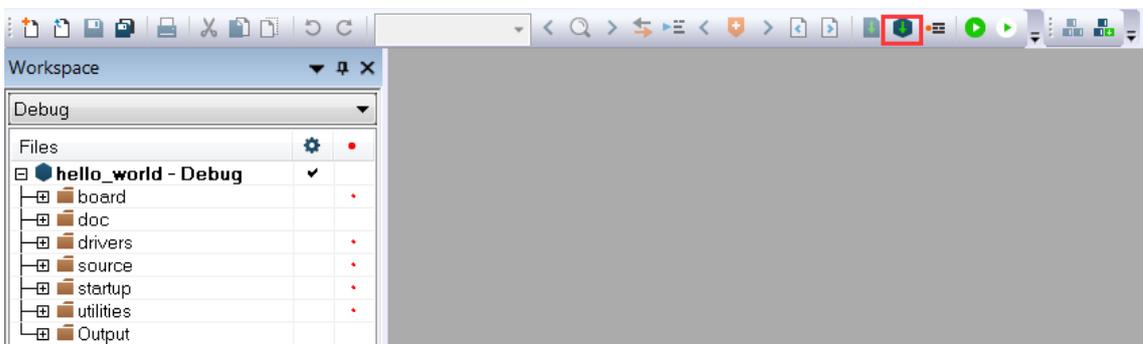
4. The build completes without errors.

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

---
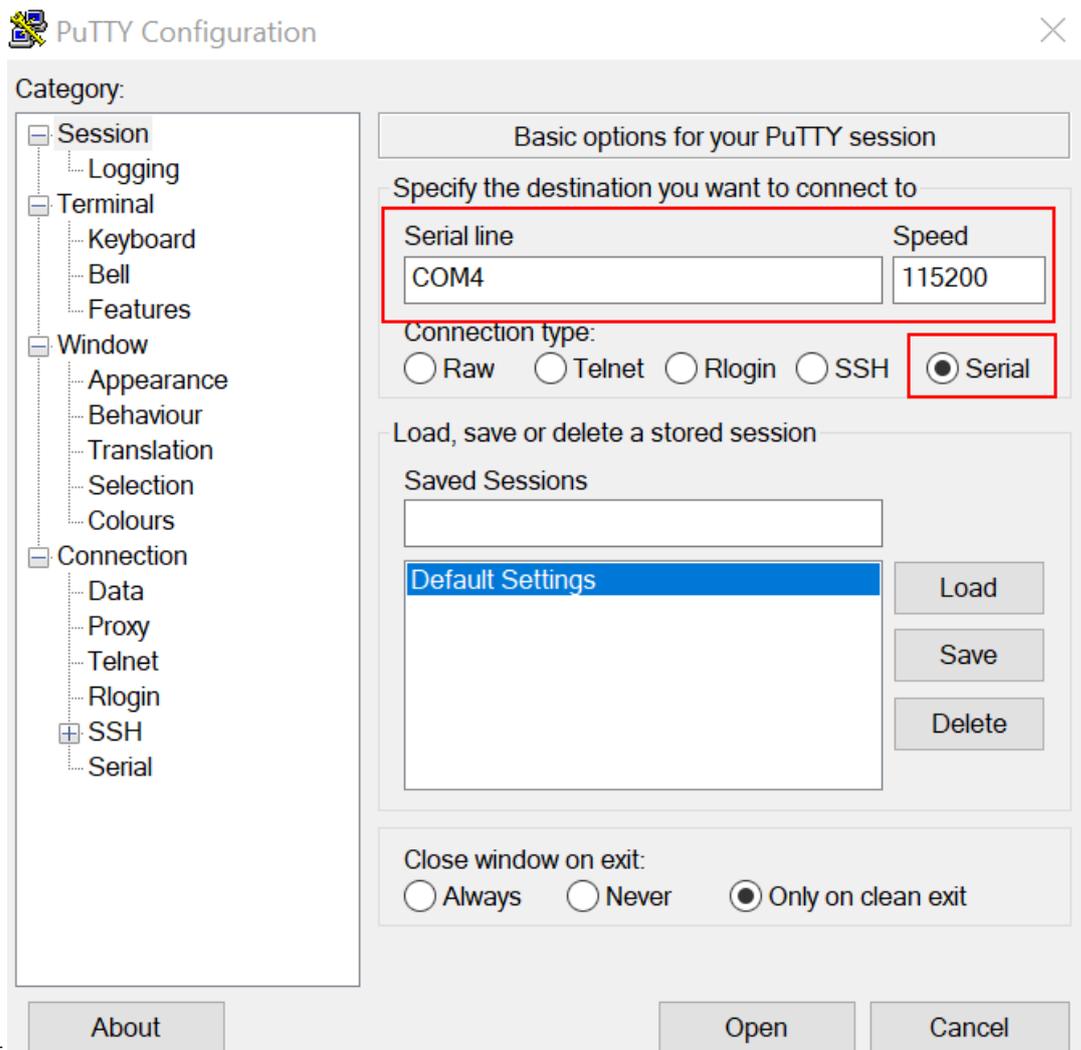
4. 1 stop bit

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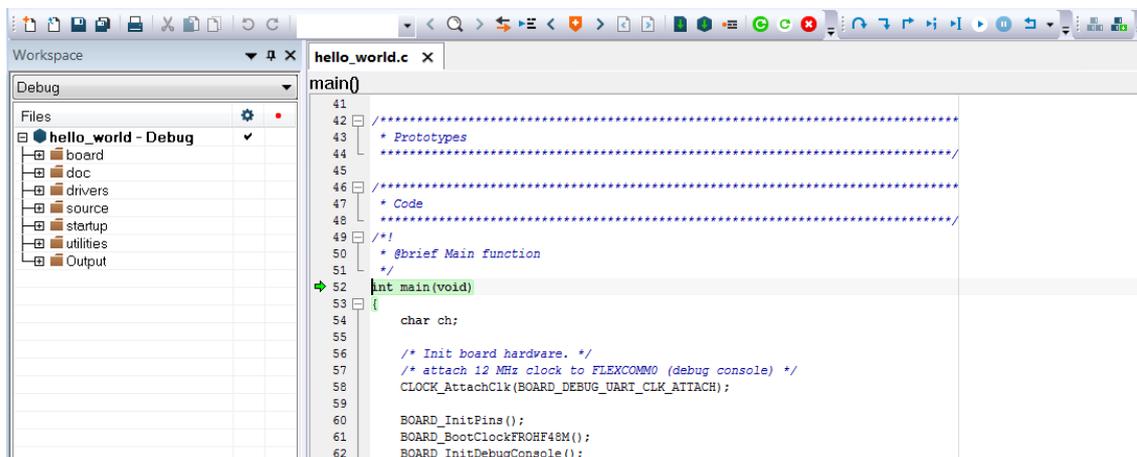
6. Run the code by clicking the **Go** button.

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.
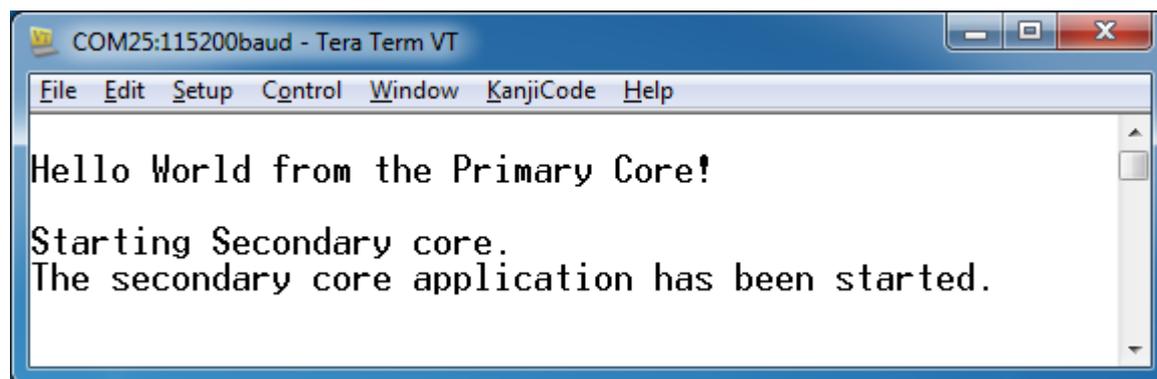
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and is running correctly. When both cores are running, use the "Stop all cores", and "Start all cores" control buttons to stop or run both cores simultaneously.
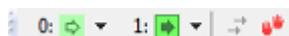
Build a TrustZone example application   This section describes the particular steps that must be done in order to build and run a TrustZone application. The demo applications workspace files are located in this folder:

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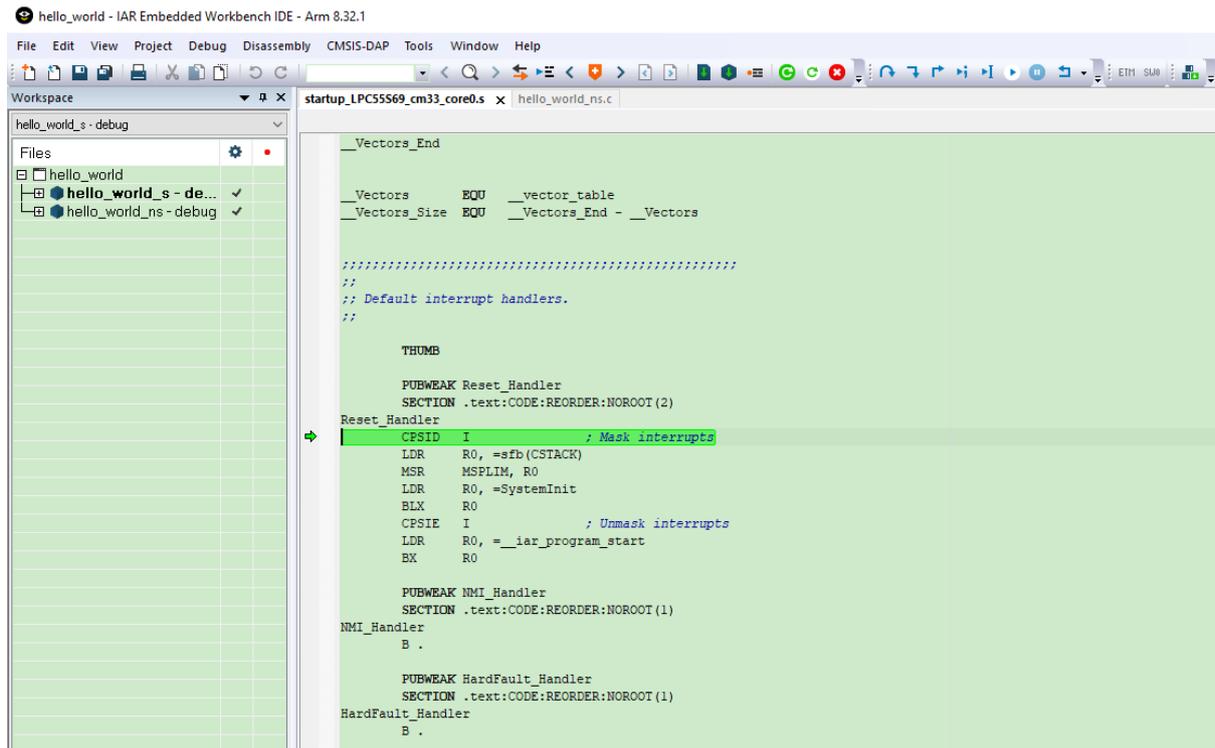
Run the code by clicking **Go** to start the application.

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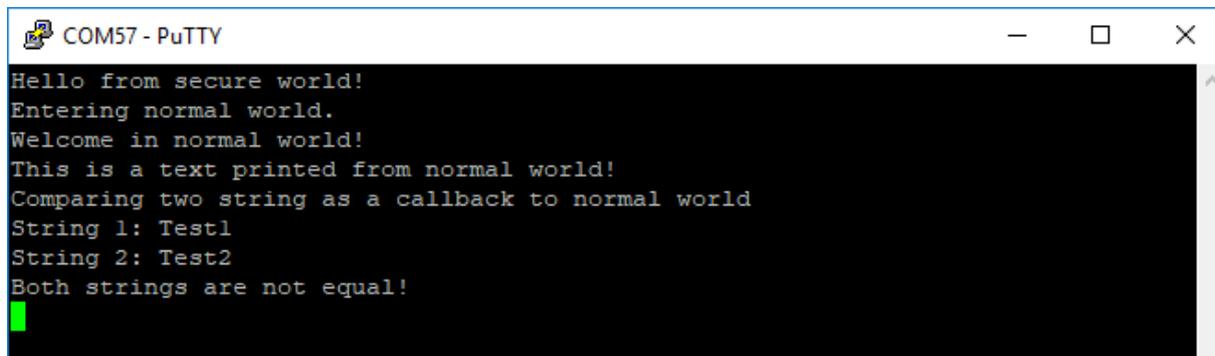
Note: If the application is running in RAM (debug/release build target), in **Options**>**Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the __ns download issue on i.MXRT500.
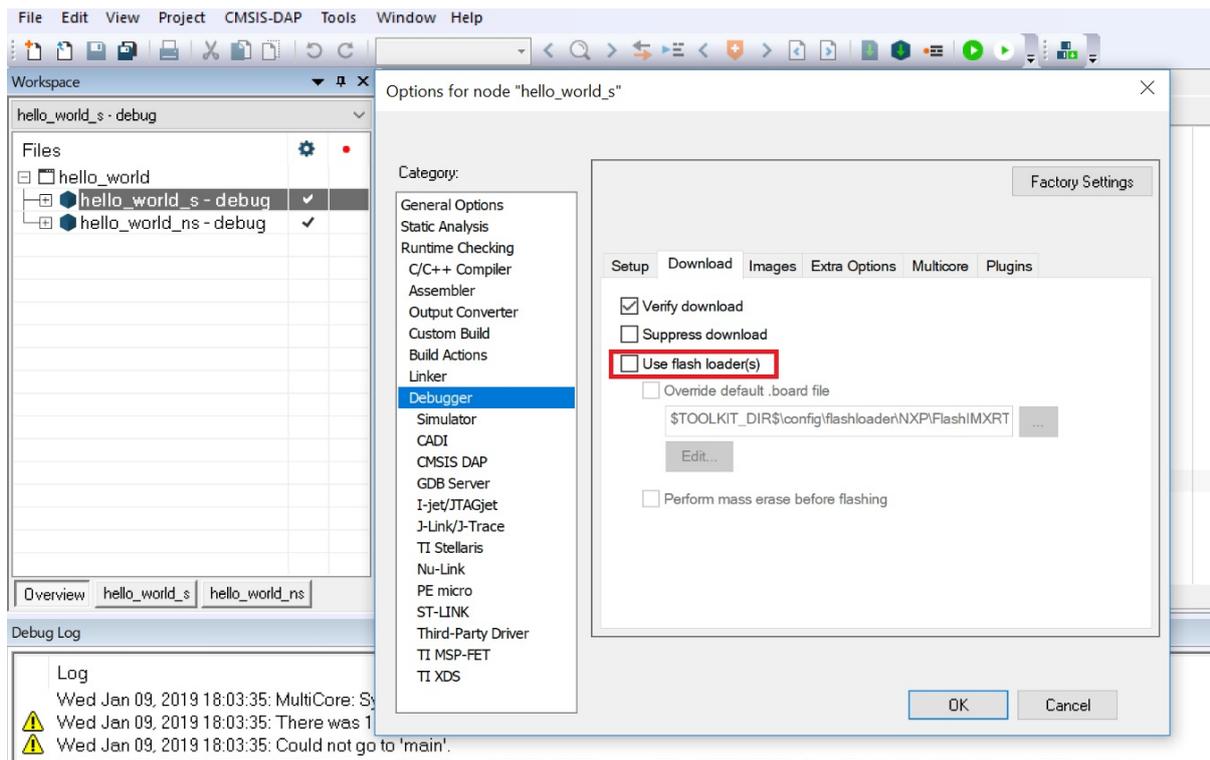
Run a demo using Keil MDK/µVision

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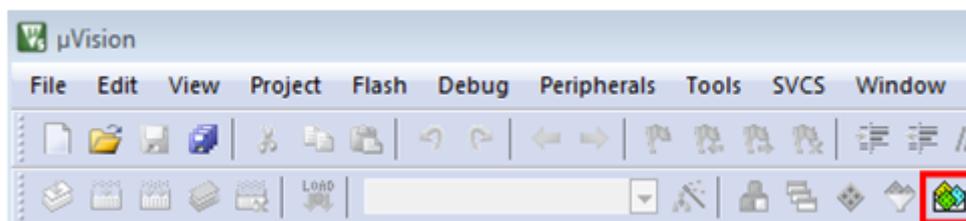
2. After the installation finishes, close the Pack Installer window and return to the µVision IDE.

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:

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

3. 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 using USB connector.

3. Open the terminal application on the PC, such as PuTTY or TeraTerm and connect to the debug serial port number (to determine the COM port number, see *How to determine COM port*. Configure the terminal with these settings:

   1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)

   2. No parity

   3. 8 data bits
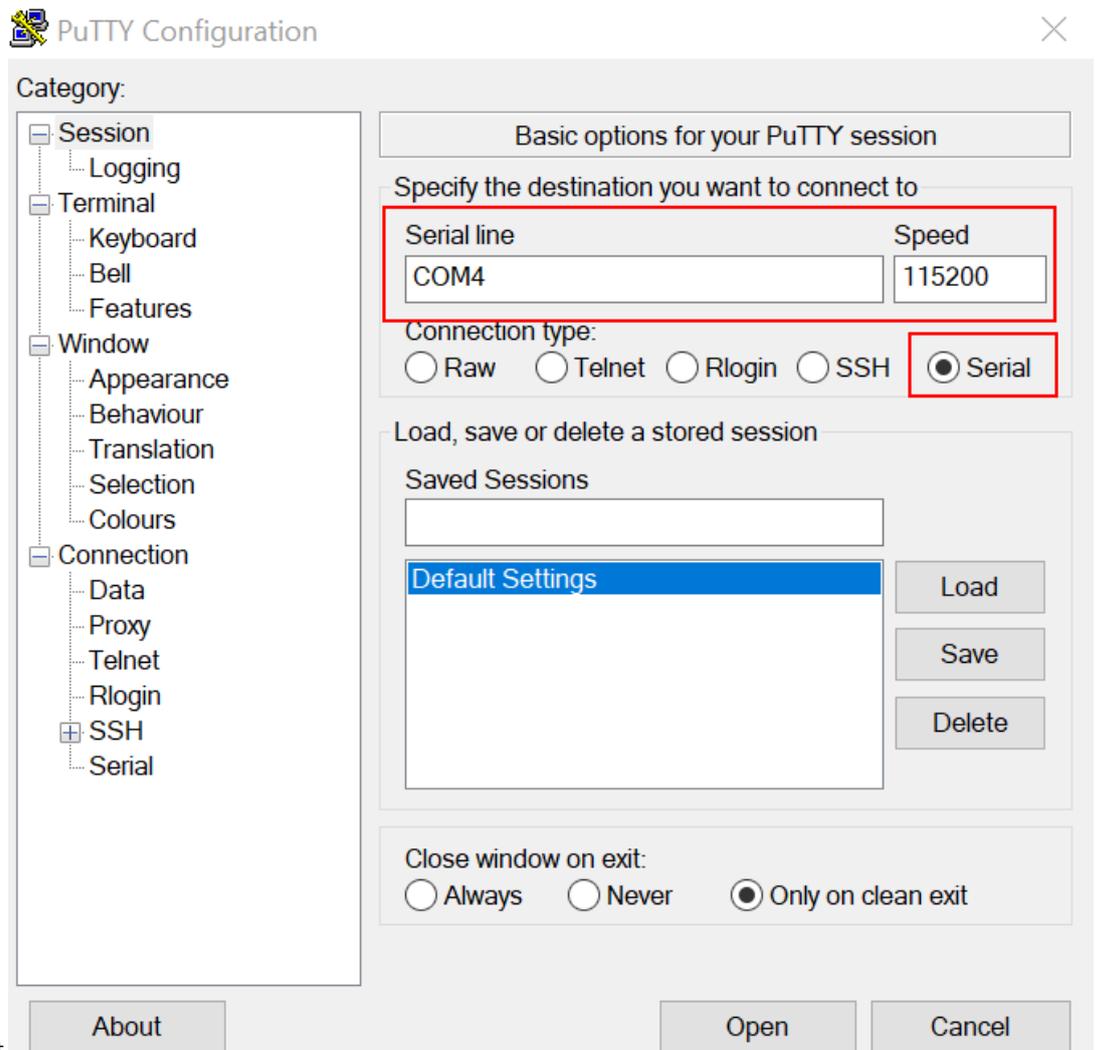
   4. 1 stop bit

4. In μVision, 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.
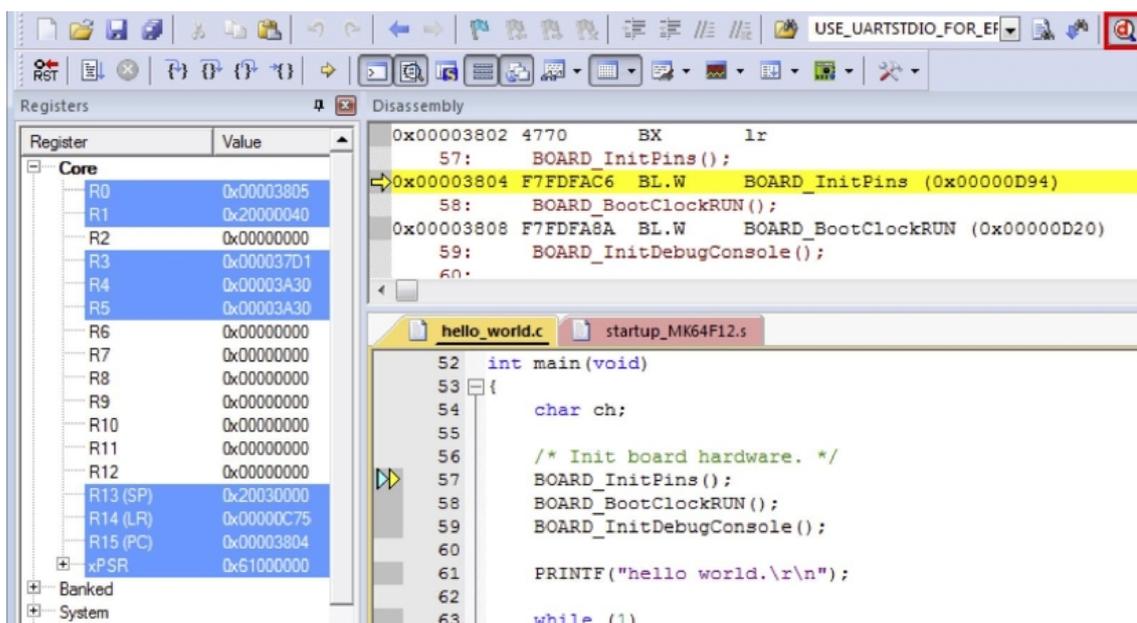
6. Run the code by clicking the **Run** button to start the application.
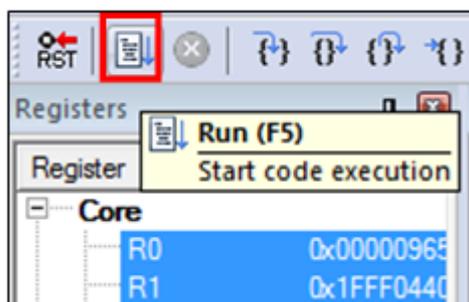
The hello_world application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.

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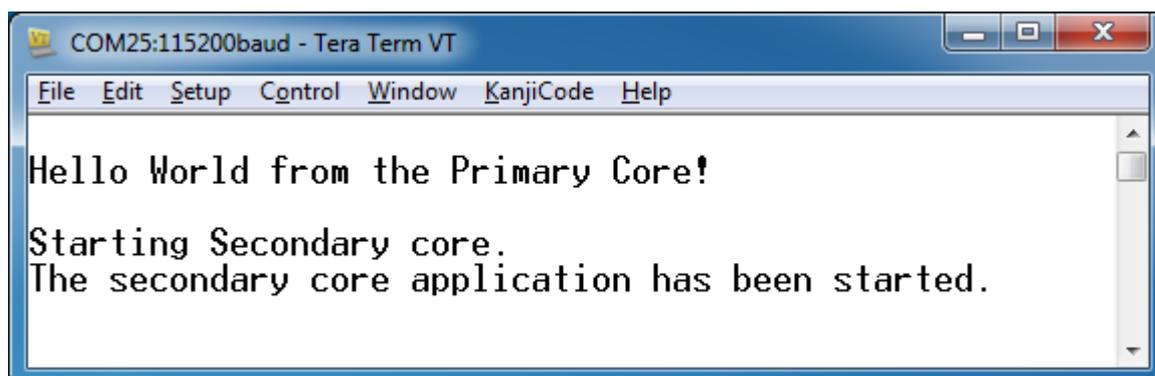
An LED controlled by the auxiliary core starts flashing indicating that the auxiliary core has been released from the reset and is running correctly.

Attach the running application of the auxiliary core by opening the auxiliary core project in the second µVision 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.
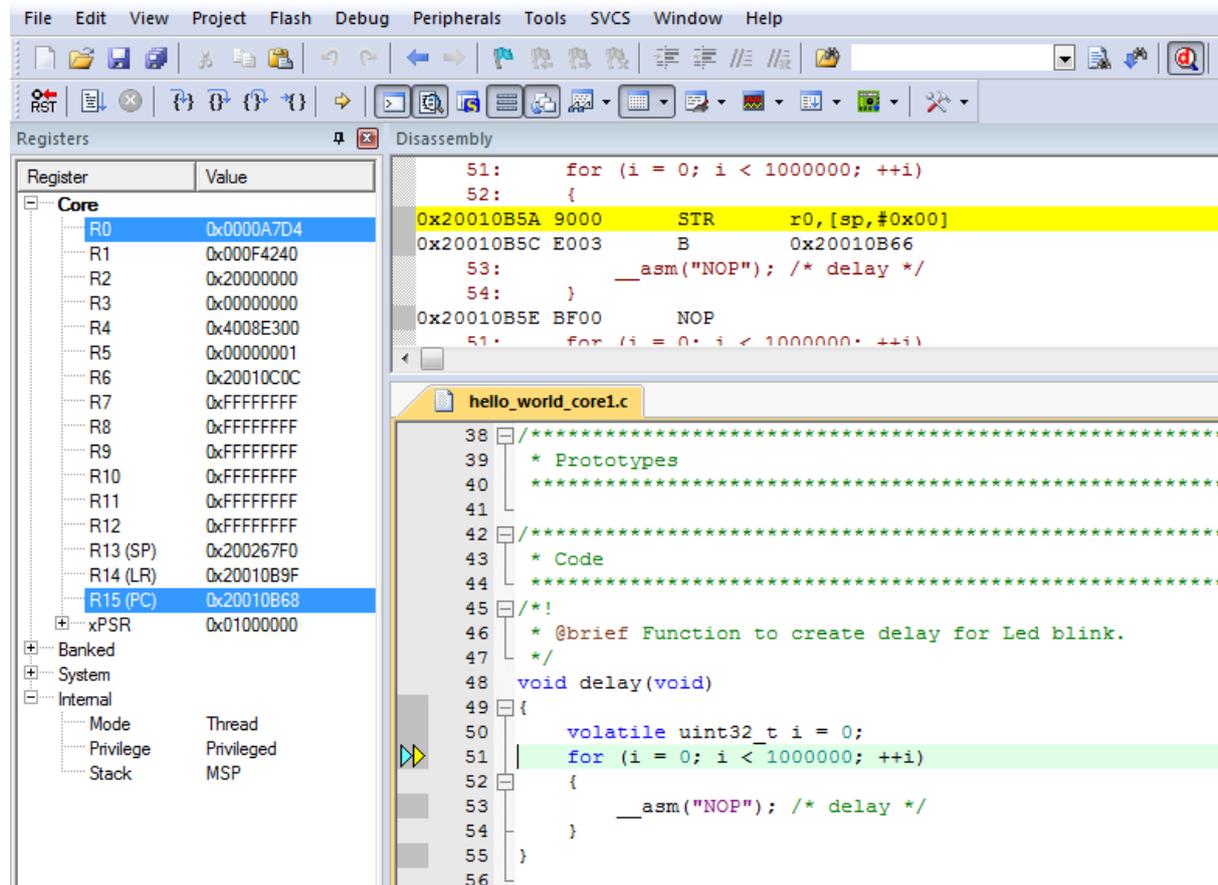
Arm describes multicore debugging using the NXP LPC54114 Cortex-M4/M0+ dual-core processor and Keil uVision IDE in Application Note 318 at www.keil.com/appnotes/docs/apnt_318.asp. The associated video can be found here.

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/µVision 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 µVision. After clicking **Download and Debug**, both the secure and non-secure images are loaded into the device flash memory, and the secure application is executed. It stops at the main() function.
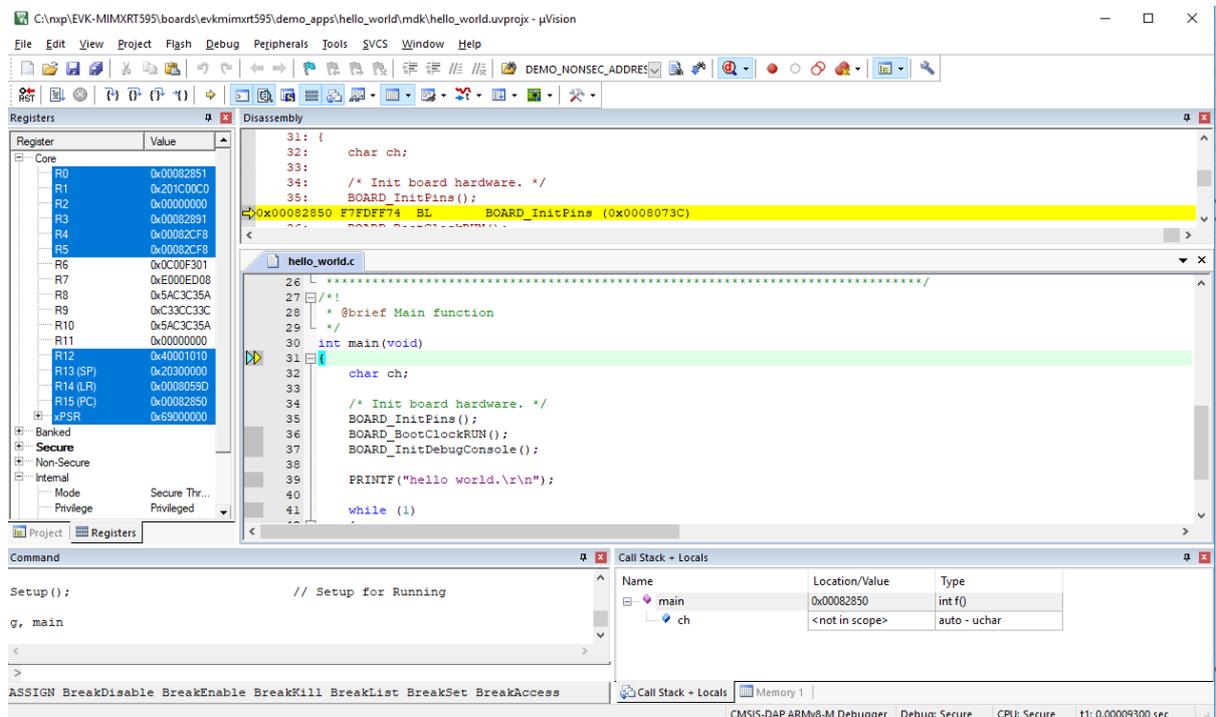

Run the code by clicking **Run** to start the application.

The hello_world application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.
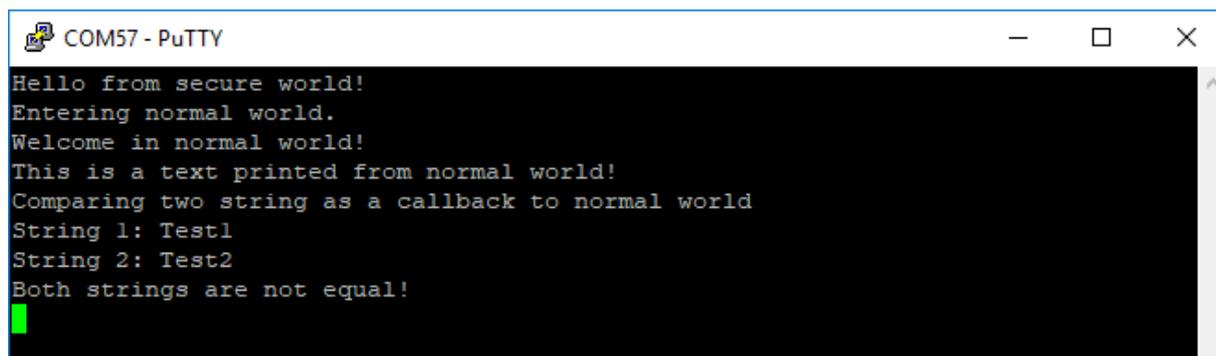
Run a demo using Arm GCC

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

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

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

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

1. Download the latest MinGW mingw-get-setup installer from MinGW.

2. Run the installer. The recommended installation path is C:\MinGW, however, you may install to any location.

   Note: The installation path cannot contain any spaces.

3. Ensure that the **mingw32-base** and **msys-base** are selected under **Basic Setup**.
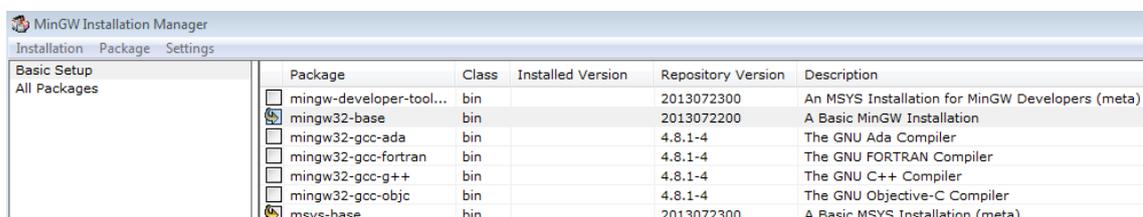
4. In the **Installation** menu, click **Apply Changes** and follow the remaining instructions to complete the installation.
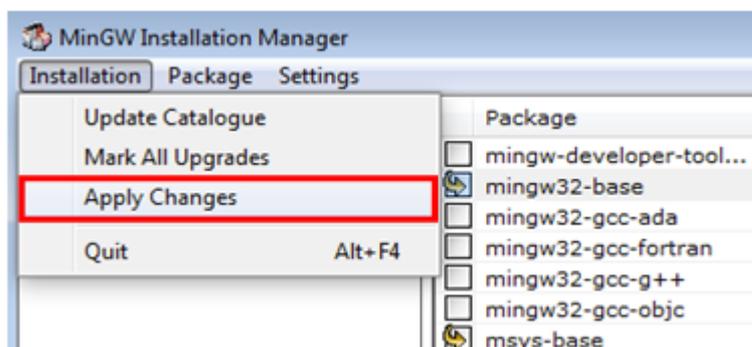
5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel**->**System and Security**->**System**->**Advanced System Settings** in the **Environment Variables...** section. The path is:

```
<mingw_install_dir>\bin
```

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

Note: If you have C:\MinGW\msys\x.x\bin in your PATH variable (as required by Kinetis SDK 1.0.0), remove it to ensure that the new GCC build system works correctly.
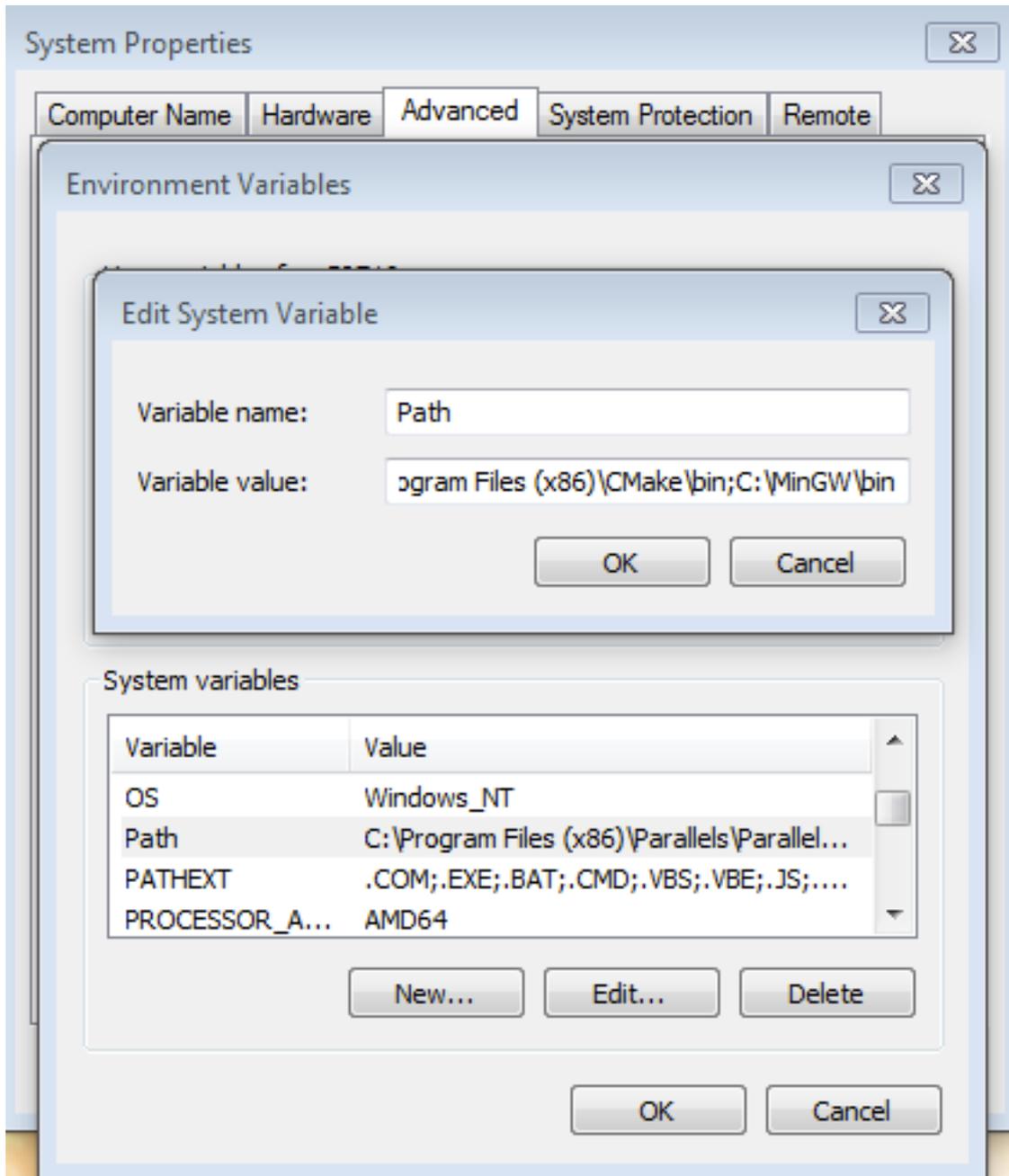
Add a new system environment variable for ARMGCC_DIR Create a new *system* environment variable and name it as ARMGCC_DIR. The value of this variable should point to the Arm GCC Embedded tool chain installation path. For this example, the path is:

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

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

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

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

C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>echo C:\PROGRA~2\GNUTOO~1\82018-~1
C:\PROGRA~2\GNUTOO~1\82018-~1
```
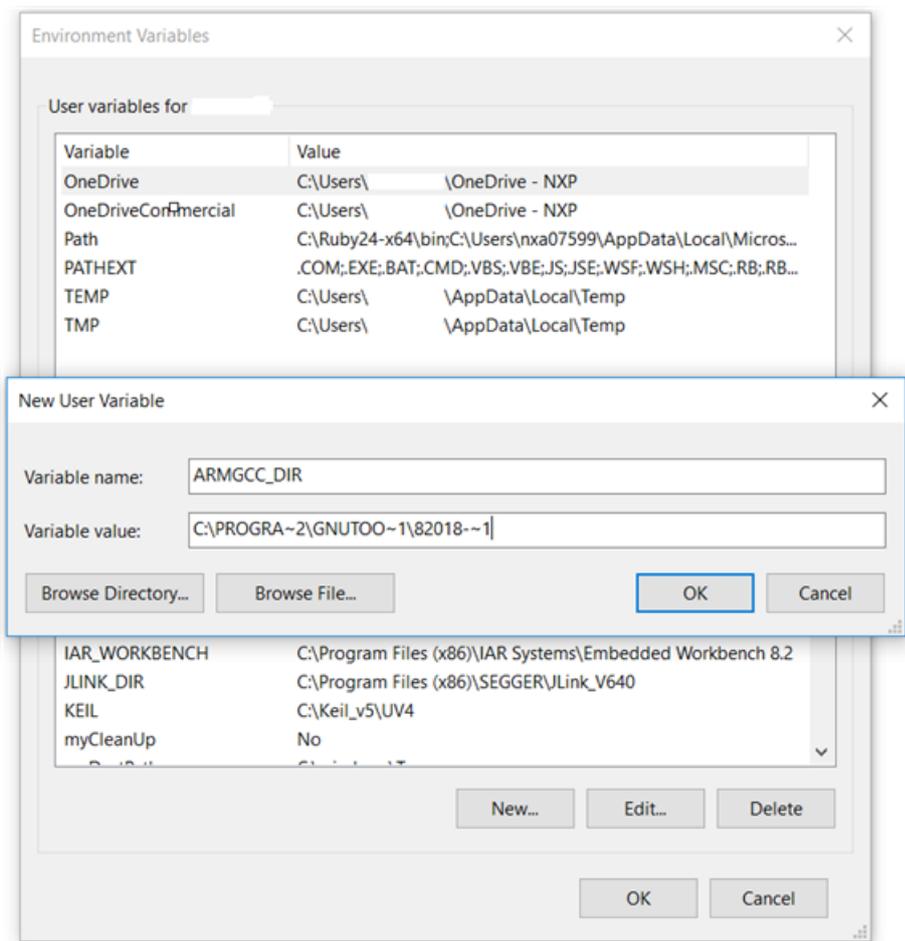
Install CMake

Windows OS

1. Download CMake 3.0.x from www.cmake.org/cmake/resources/software.html.

2. Install CMake, ensuring that the option **Add CMake to system PATH** is selected when installing. The user chooses to select whether it is installed into the PATH for all users or just the current user. In this example, it is installed for all users.
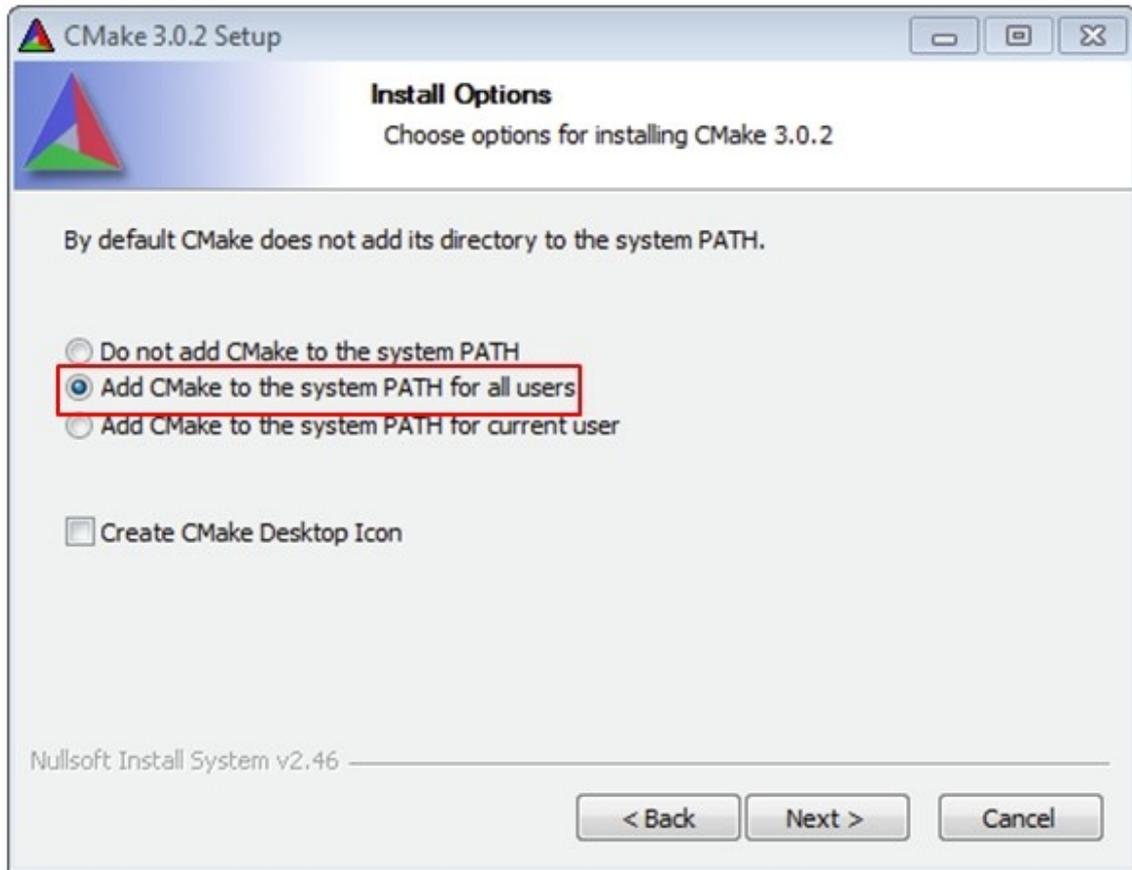
3. Follow the remaining instructions of the installer.

4. You may need to reboot your system for the PATH changes to take effect.

5. Make sure sh.exe is not in the Environment Variable PATH. This is a limitation of mingw32-make.

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

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

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

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

1. Open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system **Start** menu, go to **Programs** >**GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.
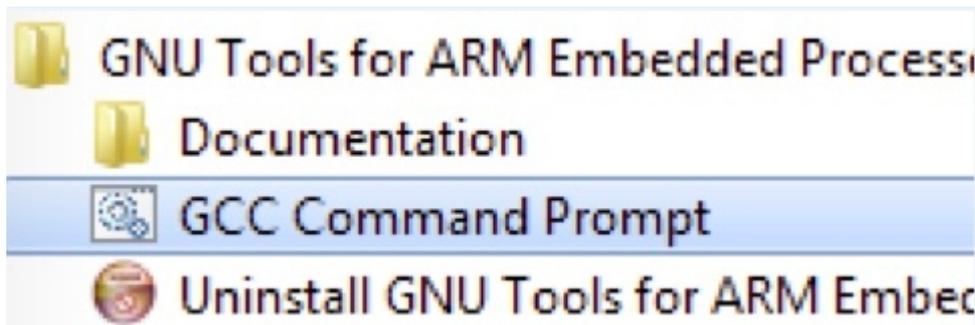
2. Change the directory to the example application project directory which has a path similar to the following:

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

For this example, the exact path is:

Note: To change directories, use the cd command.

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

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

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

1. Connect the development platform to your PC via USB cable between the on-board debugger USB connector and the PC USB connector. If using a standalone J-Link debug pod, connect it to the SWD/JTAG connector of the board.

2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see *How to determine COM port*). Configure the terminal with these settings:

   1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)

   2. No parity

   3. 8 data bits

   4. 1 stop bit
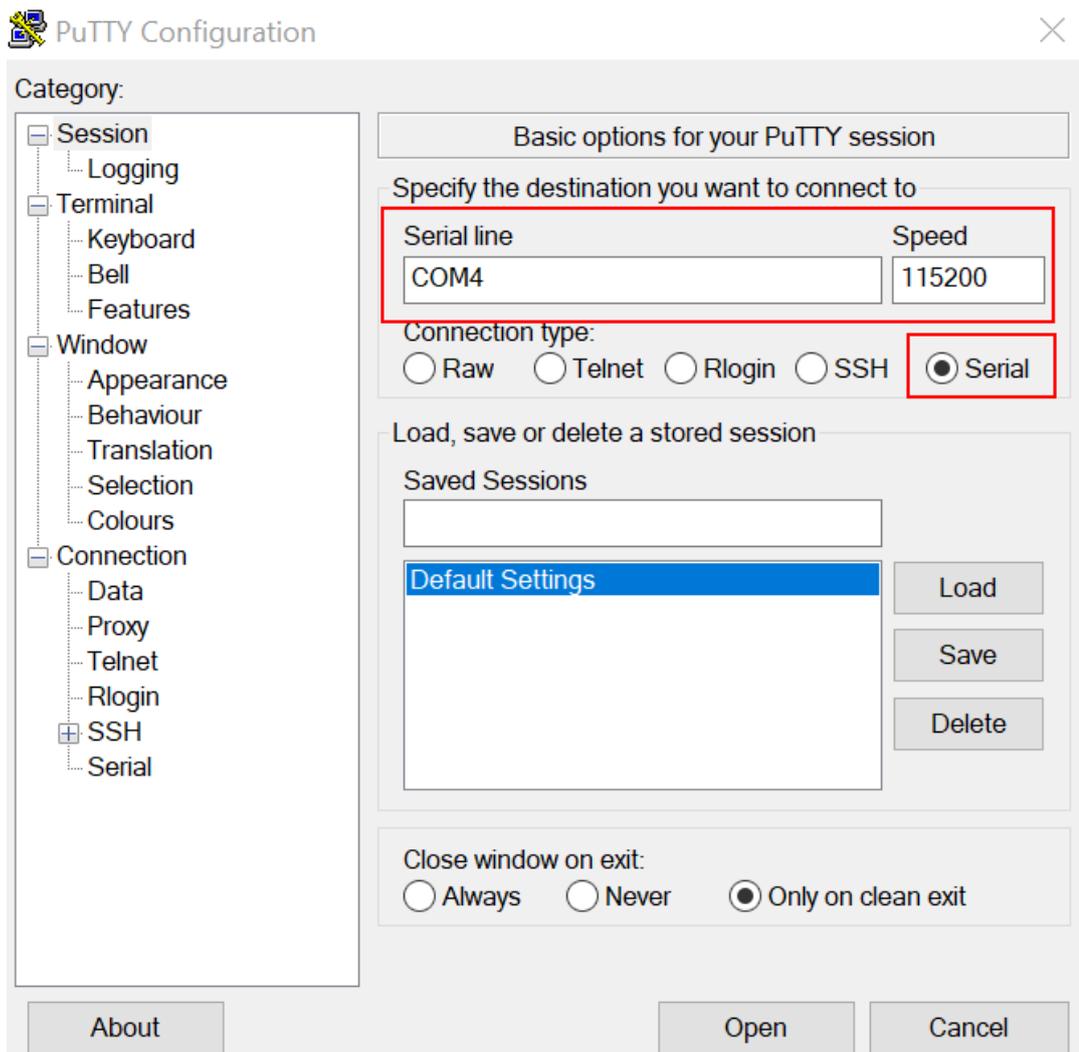
3. To launch the application, open the Windows **Start** menu and select **Programs** > **SEGGER** > **J-Link <version> J-Link GDB Server**.

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

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

4. Make sure to check the following options.

   1. **Target interface**: The debug connection on board uses internal SWD signaling. In case of a wrong setting J-Link is unable to communicate with device under test.

   2. **Script file**: If required, a J-Link init script file can be used for board initialization. The file with the ".jlinkscript" file extension is located in the <install_dir>/boards/ <board_name>/ directory.

   3. Under the **Server settings**, check the GDB port for connection with the gdb target remote command. For more information, see step 9.

   4. There is a command line version of J-Link GDB server "JLinkGDBServerCL.exe". Typical path is C:\Program Files\SEGGER\JLink\. To start the J-Link GDB server with the same settings as are selected in the UI, you can use these command line options.
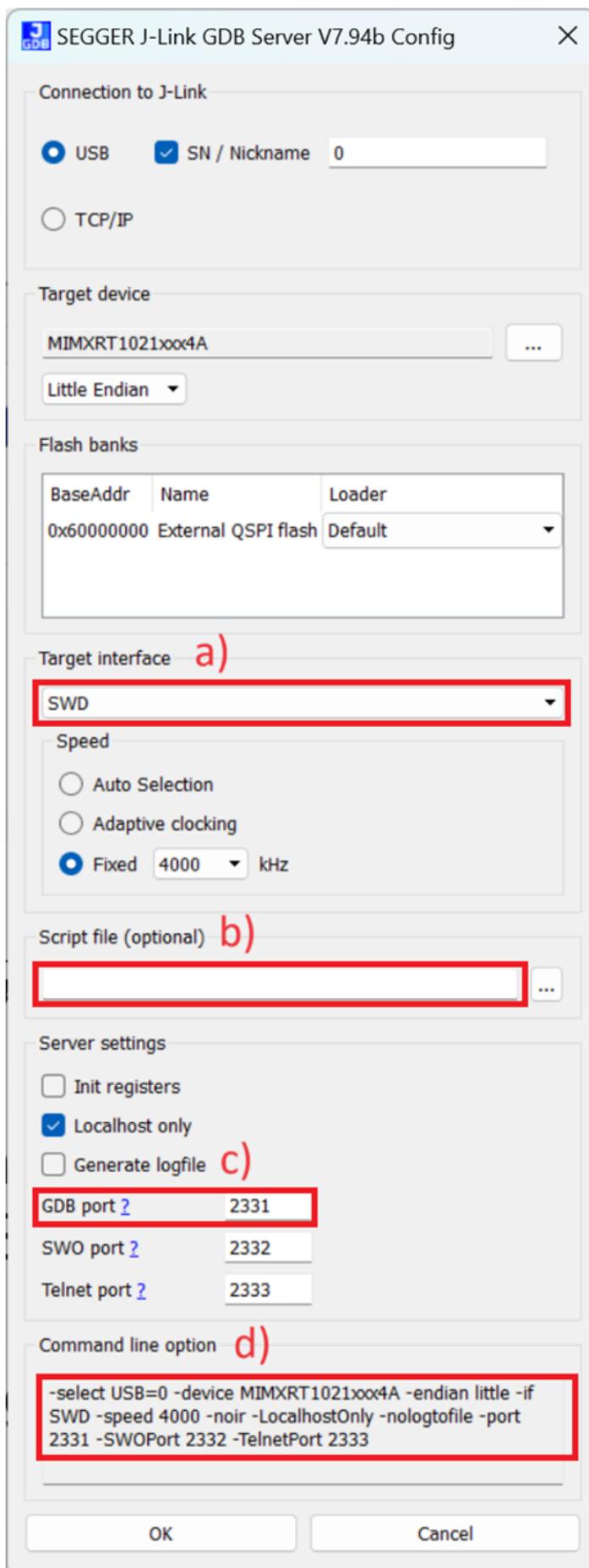
5. After it is connected, the screen should look like this figure:
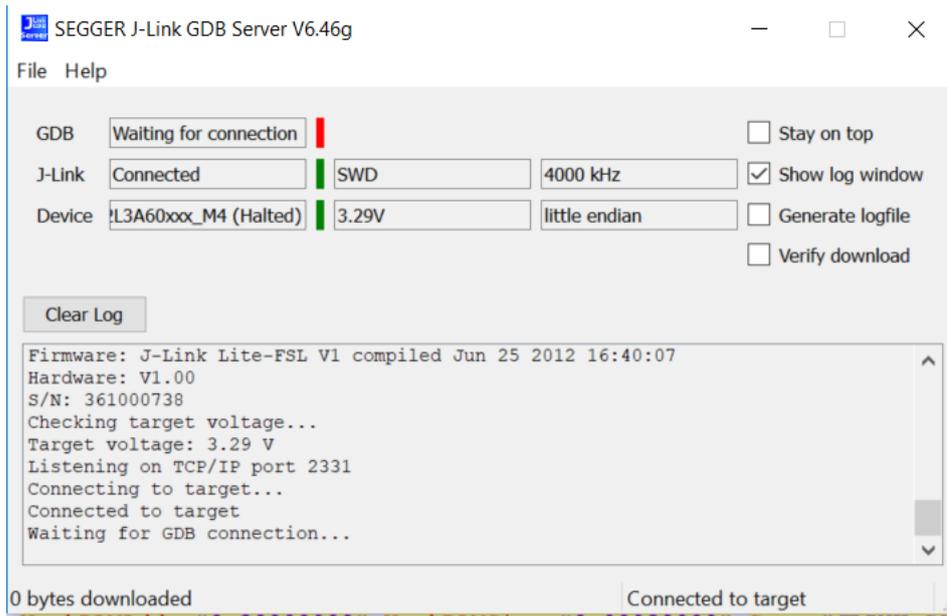
6. If not already running, open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system Start menu, go to **Programs** - **GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.
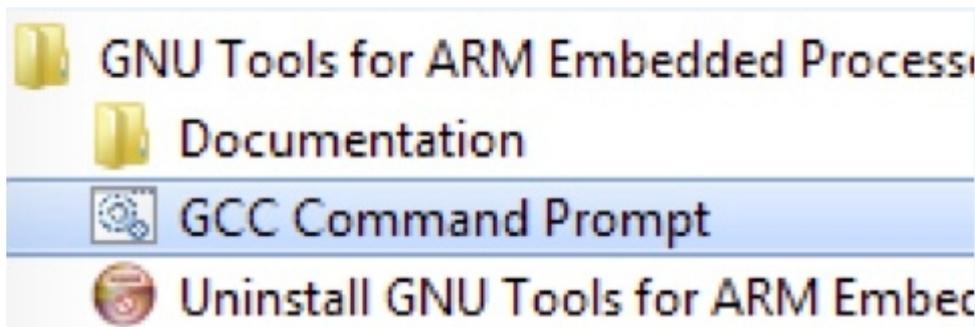
7. Change to the directory that contains the example application output. The output can be found in using one of these paths, depending on the build target selected:

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

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

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

9. Run these commands:

    1. `target remote localhost:2331`

    2. `monitor reset`

    3. `monitor halt`

    4. `load`

    5. `monitor reset`

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

    The `hello_world` application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.
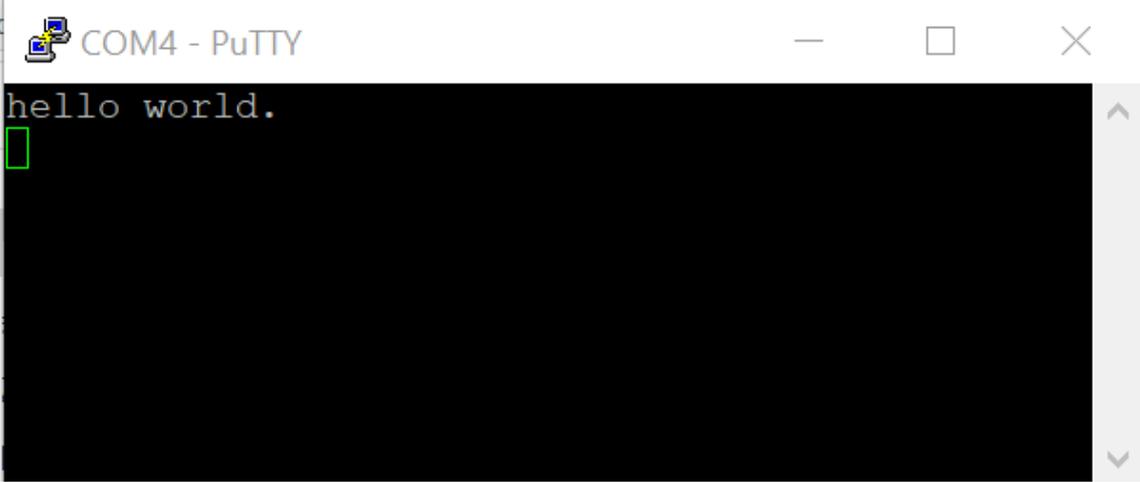
Build a multicore example application   This section describes the steps to build and run a dual-core application. The demo application build scripts are located in this folder:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then, run these commands:

1. arm-none-eabi-gdb.exe

2. target remote localhost:2331

3. monitor reset

4. monitor halt

5. monitor exec SetFlashDLNoRMWThreshold = 0x20000

6. load <install_dir>/boards/evkmimxrt595/trustzone_examples/hello_world/hello_world_ns/armgcc/debug/hello_world_ns.elf

7. load <install_dir>/boards/evkmimxrt595/trustzone_examples/hello_world/hello_world_s/armgcc/debug/hello_world_s.elf

8. monitor reset

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

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**:
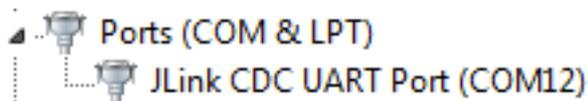
4. **P&E Micro OSJTAG**:

5. **MRB-KW01**:
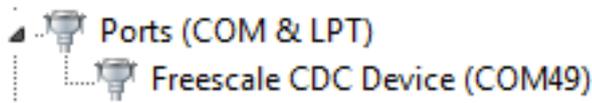
On-board Debugger

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 LPCScrypt. 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 LPCScrypt 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 LPCScrypt.

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in LPCScrypt user guide (LPCScrypt, select **LPCScrypt**, 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 LPCScrypt installation directory (<LPCScrypt install dir>).

   1. To program CMSIS-DAP debug firmware: <LPCScrypt install dir>/scripts/program_CMSIS

   2. To program J-Link debug firmware: <LPCScrypt 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**.
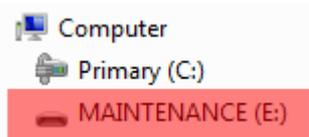
4. Drag and drop the new firmware image onto the MAINTENANCE drive.

   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-MIMX8MNDDR3L	N/A	N/A
EVK-MIMX8MP	N/A	N/A
EVK-MIMX8MQ	N/A	N/A
EVK-MIMX8ULP	N/A	N/A
EVK-MIMXRT1010	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1015	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1020	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1064	CMSIS-DAP	LPC-Link2
EVK-MIMXRT595	CMSIS-DAP	LPC-Link2
EVK-MIMXRT685	CMSIS-DAP	LPC-Link2
EVK9-MIMX8ULP	N/A	N/A
EVKB-IMXRT1050	CMSIS-DAP	LPC-Link2
FRDM-K22F	CMSIS-DAP	OpenSDA v2
FRDM-K32L2A4S	CMSIS-DAP	OpenSDA v2
FRDM-K32L2B	CMSIS-DAP	OpenSDA v2
FRDM-K32L3A6	CMSIS-DAP	OpenSDA v2
FRDM-KE02Z40M	P&E Micro	OpenSDA v1
FRDM-KE15Z	CMSIS-DAP	OpenSDA v2
FRDM-KE16Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z512	CMSIS-DAP	MCU-Link
FRDM-MCXA153	CMSIS-DAP	MCU-Link
FRDM-MCXA156	CMSIS-DAP	MCU-Link
FRDM-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
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

continues on next page

Table 1 – continued from previous page

Hardware platform	Default debugger firmware	On-board debugger probe
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
// 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
// cpp
extern "C" {
    void PIT_IRQHandler(void);
}
void PIT_IRQHandler(void)
{
    // Your code
}
```

1.3 Getting Started with MCUXpresso SDK GitHub

1.3.1 Getting Started with MCUXpresso SDK Repository

Installation

NOTE

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

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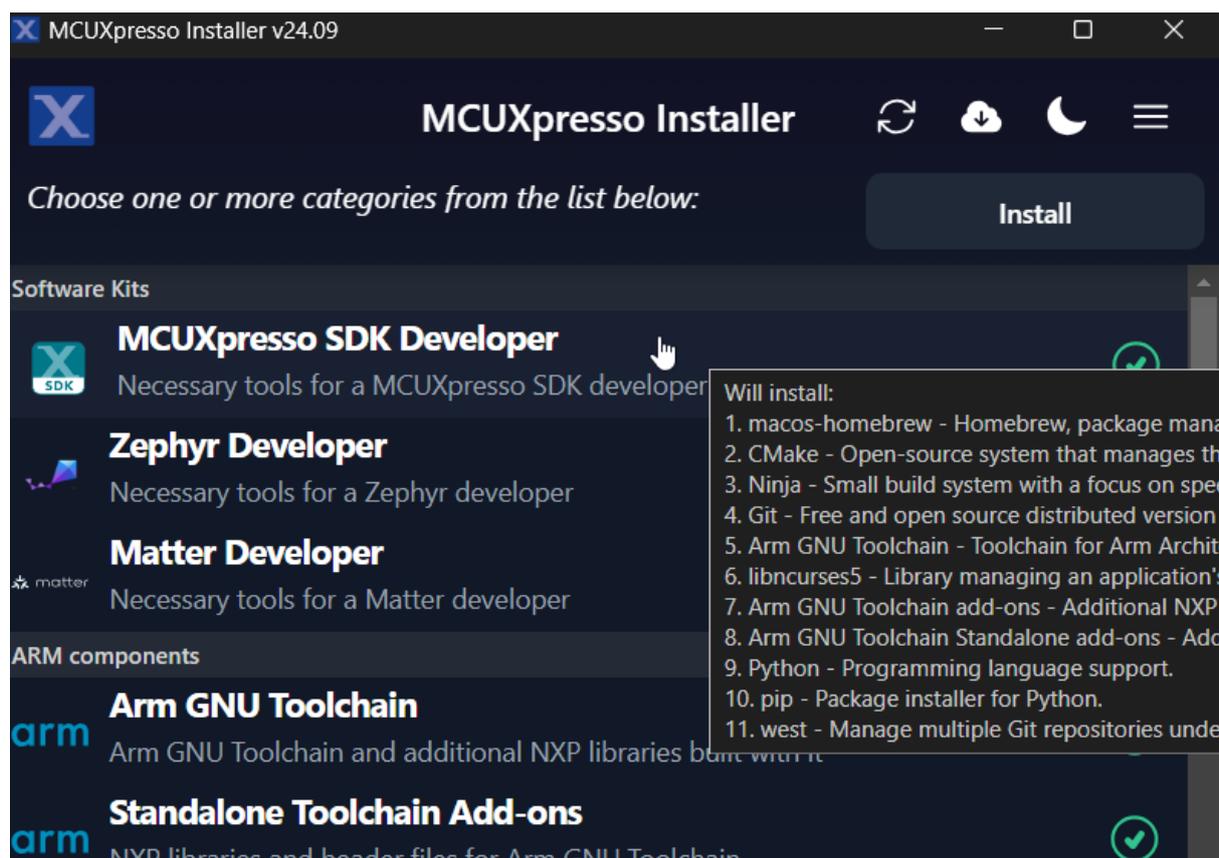
Alternative: Manual Installation

Basic tools

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

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

Then configure your username and email using the commands:

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

Python   Install python 3.10 or latest. Follow the Python Download guide.

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

West   Please use the west version equal or greater than 1.2.0

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

Build And Configuration System

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

For Windows, you can directly use the .msi installer like cmake-3.31.4-windows-x86_64.msi to install.

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

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

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

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

For Linux, you can use your system package manager or you can directly download the ninja binary to work.

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

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

Please make sure *python* environment is setup ready then you can use the Kconfig.

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

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

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

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

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

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

- The <toolchain>_DIR is the root installation folder, not the binary location folder. For IAR, it is directory containing following installation folders:
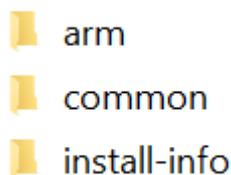
- MDK IDE using armclang toolchain only officially supports Windows. In Linux, please directly use armclang toolchain by setting ARMCLANG_DIR. In Windows, since most Keil users will install MDK IDE instead of standalone armclang toolchain, the MDK_DIR has higher priority than ARMCLANG_DIR.
- For Xtensa toolchain, please set the XTENSA_CORE environment variable. Here's an example list:

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

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

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

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

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

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

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

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

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

- Linux:

  1. Open the $HOME/.bashrc file using a text editor, such as vim.

  2. Go to the end of the file.

  3. Add the line which appends the tool installation path to the PATH variable and export PATH at the end of the file. For example, export PATH="/Directory1:$PATH".

  4. Save and exit.

  5. Execute the script with source .bashrc or reboot the system to make the changes live. To verify the changes, run echo $PATH.

- macOS:

  1. Open the $HOME/.bash_profile file using a text editor, such as nano.

  2. Go to the end of the file.

  3. Add the line which appends the tool installation path to the PATH variable and export PATH at the end of the file. For example, export PATH="/Directory1:$PATH".

  4. Save and exit.

  5. Execute the script with source .bash_profile or reboot the system to make the changes live. To verify the changes, run echo $PATH.

Get MCUXpresso SDK Repo

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

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

# Update the west projects
cd mcuxpresso-sdk
west update

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

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

1. Navigate to the core directory:

   ```
   cd mcuxsdk
   ```

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

   ```
   python -m venv .venv

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

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

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

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

3. Install the required Python packages:

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

Explore Contents

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

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

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

All the projects record in the Manifest repo are checked out to the folder mcuxsdk/, the layout of mcuxsdk folder is shown as below:

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

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

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

- demo_apps: Basic demo set to start using SDK, including hello_world and led_blinky.

- driver_examples: Simple applications that show how to use the peripheral drivers for a single use case. These applications typically only use a single peripheral but there are cases where multiple peripherals are used (for example, SPI transfer using DMA).

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

Run a demo using MCUXpresso for VS Code

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

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

Build an example application   This section assumes that the user has already obtained the SDK as outlined in *Get MCUXpresso SDK Repo*.

To build an example application:

1. Import the SDK into your workspace. Click **Import Repository** from the **QUICKSTART PANEL**.
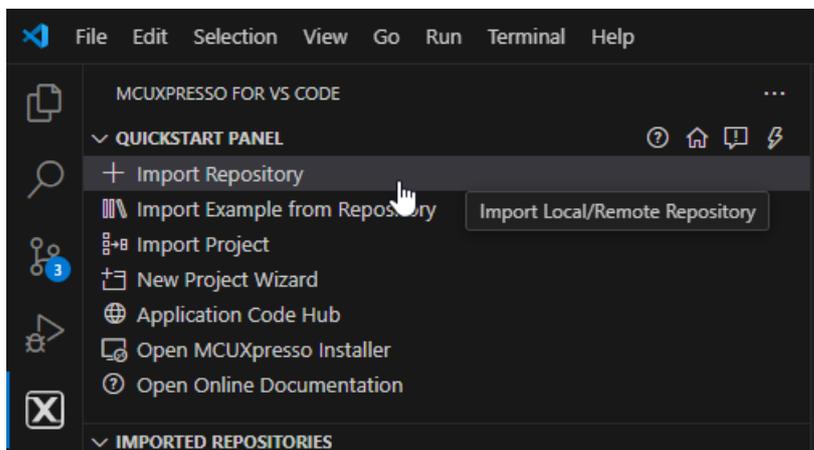
Note: You can import the SDK in several ways. Refer to MCUXpresso for VS Code Wiki for details.

Select **Local** if you've already obtained the SDK as seen in *Get MCUXpresso SDK Repo*. Select your location and click **Import**.
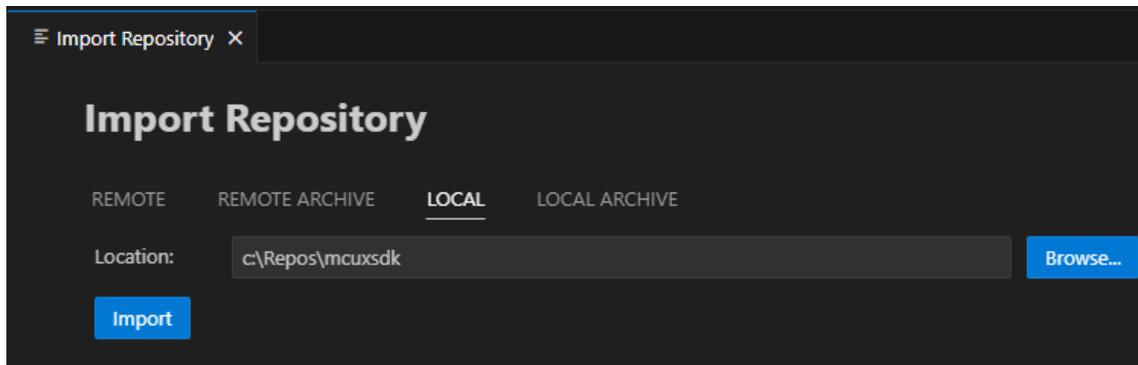
2. Click **Import Example from Repository** from the **QUICKSTART PANEL**.
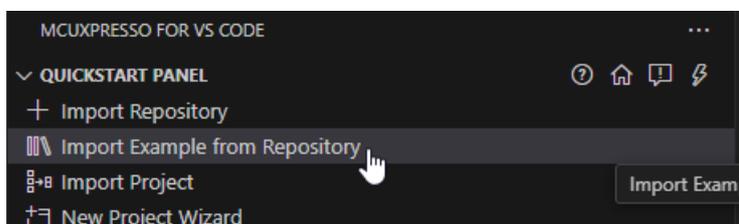
In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.
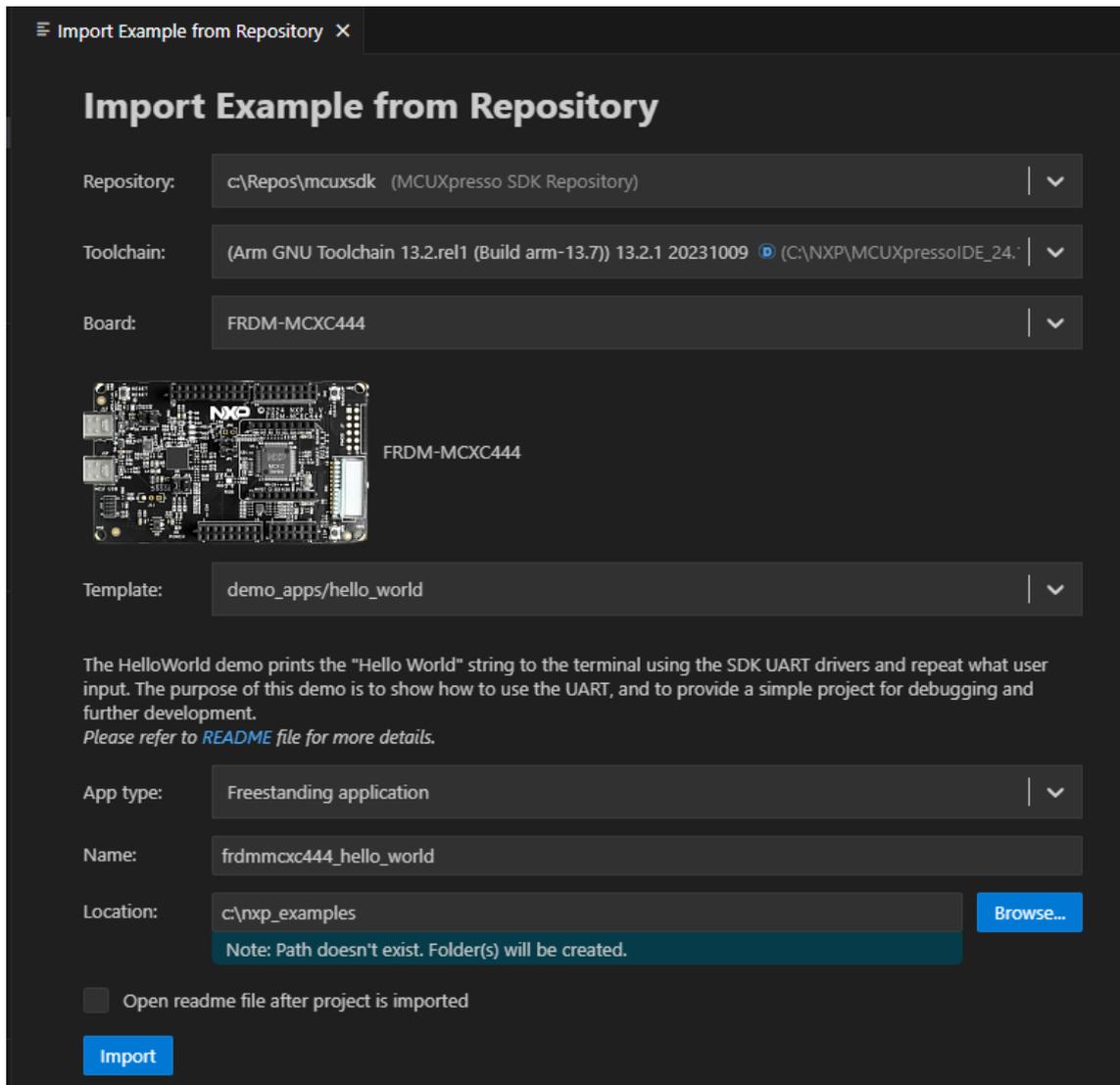
Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Freestanding applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Freestanding examples can be imported to a user-defined location. Select between these by designating your selection in the **App type** dropdown menu.

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

4. Navigate to the **PROJECTS** view. Find your project and click the **Build Project** icon.
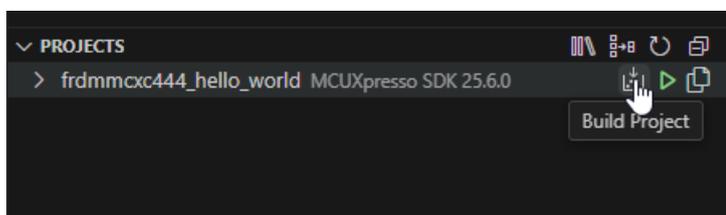
The integrated terminal will open at the bottom and will display the build output.

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

1. Open the **Serial Monitor** from the VS Code's integrated terminal. Select the VCom Port for your device and set the baud rate to 115200.
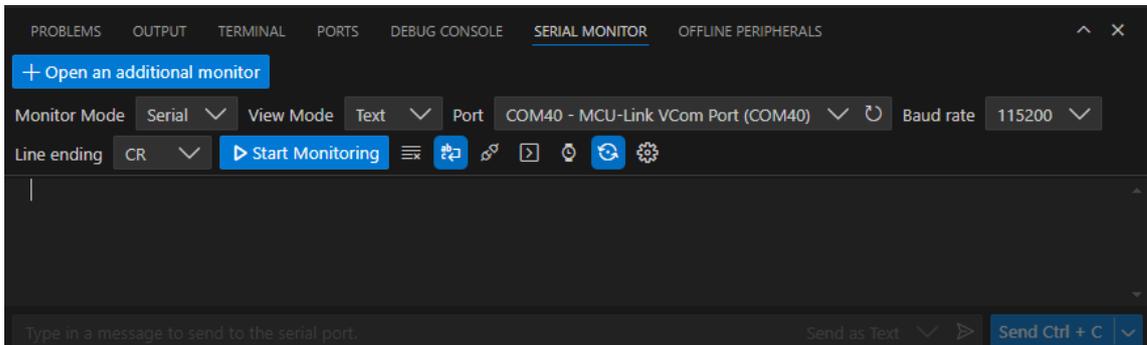
2. Navigate to the **PROJECTS** view and click the play button to initiate a debug session.
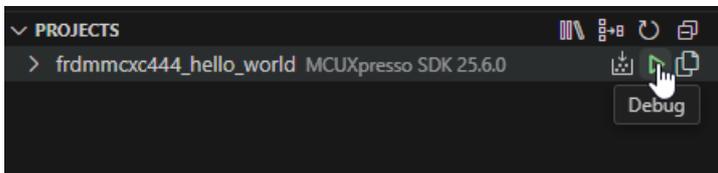
The debug session will begin. The debug controls are initially at the top.

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

Running a demo using ARMGCC CLI/IAR/MDK

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

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

INFO: [   1][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
↪evk9mimx8ulp -Dcore_id=cm33]
INFO: [   2][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
↪evkbimxrt1050]
INFO: [   3][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
```

```
→evkbmimxrt1060]
INFO: [    4][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
→evkbmimxrt1170 -Dcore_id=cm4]
INFO: [    5][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
→evkbmimxrt1170 -Dcore_id=cm7]
INFO: [    6][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
→evkcmimxrt1060]
INFO: [    7][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b␣
→evkmcimx7ulp]
...
```

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

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

- `--toolchain`: specify the toolchain for this build, default `armgcc`.

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

Here are some typical usages for generating a SDK example:

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

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

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

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

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

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

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config␣
→flexspi_nor_debug
```

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

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

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

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

For more details, please refer to System build.

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

1. Run cmake configuration

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

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

2. Run guiconfig target

```
west build -t guiconfig
```

Then you will get the Kconfig GUI launched, like
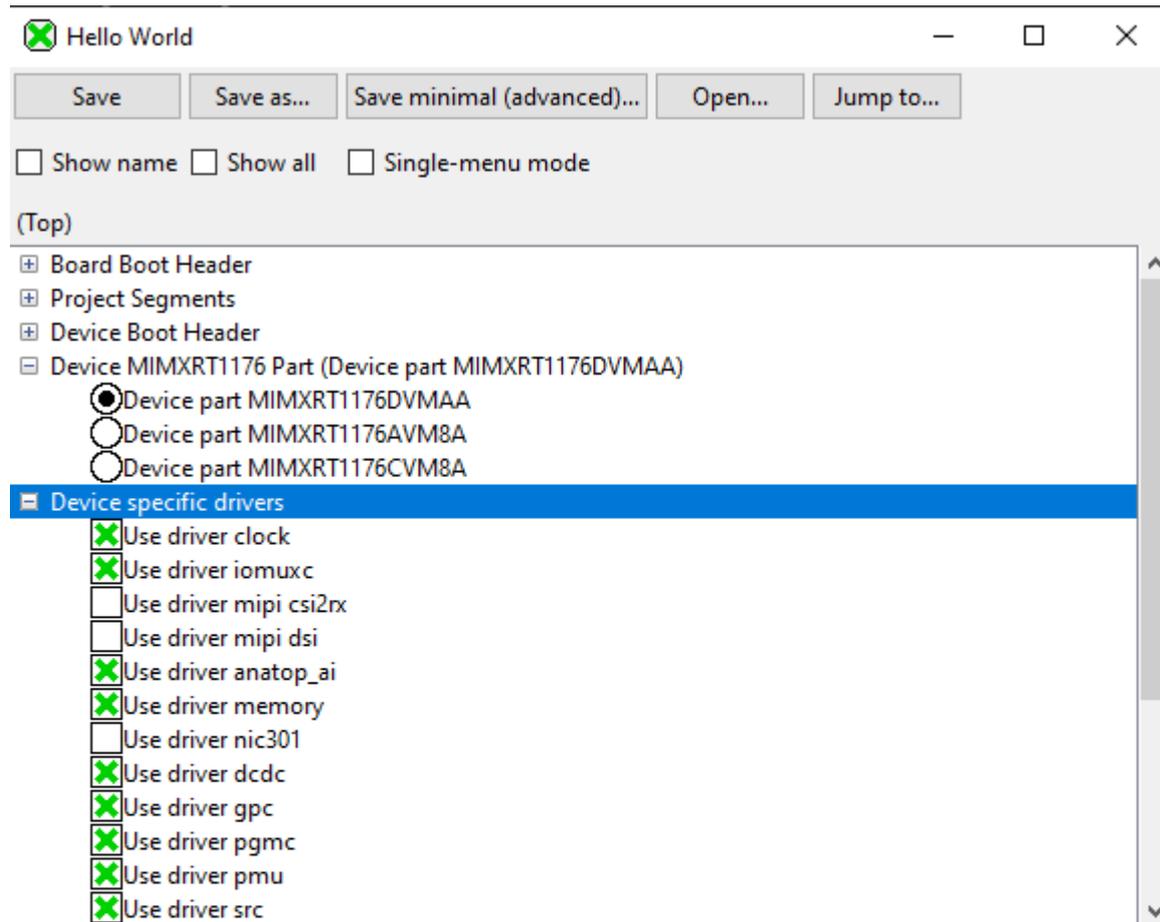
You can reconfigure the project by selecting/deselecting Kconfig options.

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

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

Flash the hello_world example:

```
west flash -r linkserver
```

Debug   Start a gdb interface by following command:

```
west debug -r linkserver
```

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

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

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

By default, the IDE project files are generated in mcuxsdk/build/<toolchain> folder, you can open the project file with the IDE tool to work:
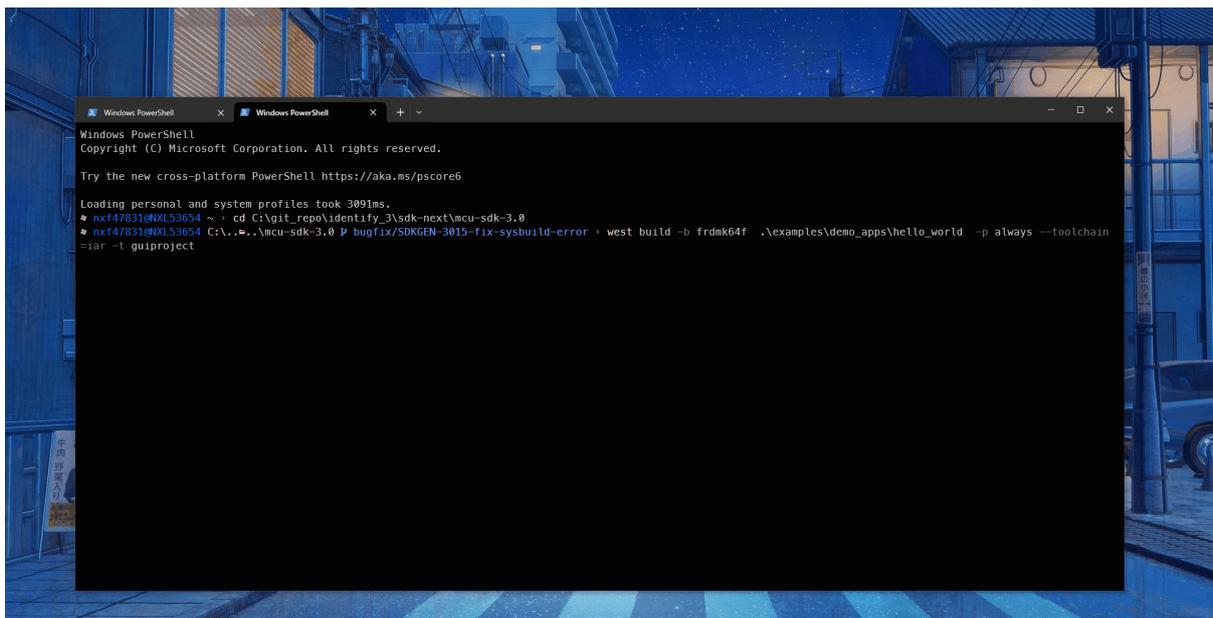
Note, please follow the *Installation* to setup the environment especially make sure that *ruby* has been installed.

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

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

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

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

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

MCUXpresso SDK

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

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

Development tools

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

- MCUXpresso IDE, Rev. 25.06.xx
- IAR Embedded Workbench for Arm, version is 9.60.4
- Keil MDK, version is 5.42
- 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.

Development boards	MCU devices
TWR-KM34Z50MV	MKM14Z128ACHH5, MKM14Z64ACHH5, MKM33Z128ACLH5, MKM33Z128ACLL5, MKM33Z64ACLH5, MKM33Z64ACLL5, **MKM34Z128ACLL5**

MCUXpresso SDK release package

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

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

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

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

RTOS

FreeRTOS   Real-time operating system for microcontrollers from Amazon

Middleware

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

TinyCBOR   Concise Binary Object Representation (CBOR) Library

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

llhttp   HTTP parser llhttp

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

ADC16

[2.3.0]

- Improvements

    – Added new API ADC16_EnableAsynchronousClockOutput() to enable/disable ADACK output.

    – In ADC16_GetDefaultConfig(), set enableAsynchronousClock to false.

[2.2.0]

- Improvements

    – Added hardware average mode in adc_config_t structure, then the hardware average mode can be set by invoking ADC16_Init() function.

[2.1.0]

- New Features:

    – Supported KM series' new ADC reference voltage source, bandgap from PMC.

[2.0.3]

- Bug Fixes

    – Fixed IAR warning Pa082: the order of volatile access should be defined.

[2.0.2]

- Improvements

    – Used conversion control feature macro instead of that in IO map.

[2.0.1]

- Bug Fixes

    – Fixed MISRA-2012 rules.

        * Rule 16.4, 10.1, 13.2, 14.4 and 17.7.

[2.0.0]

- Initial version

AFE

[2.0.3]

- Improvements
  - Fixed CERT-C issues

[2.0.2]

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

[2.0.1]

- Improvements
  - Changed type modifiers from const xx_Type * s_xxBases to xx_Type *const s_xxBases.
  - Added static modifier for s_xxx variables defined in drivers.

[2.0.0]

- Initial version.

CLOCK

[2.0.1]

- Bug Fixes
  - Fixed an issue that in CLOCK_SetFbeMode() C4 register not updated.

[2.0.0]

- Initial version.

CMP

[2.0.3]

- Improvements
  - Updated to clear CMP settings in DeInit function.

[2.0.2]

- Bug Fixes
    - Fixed the violations of MISRA 2012 rules:
        * Rule 10.3

[2.0.1]

- Bug Fixes
    - Fixed MISRA-2012 rules.
        * Rule 14.4, rule 10.3, rule 10.1, rule 10.4 and rule 17.7.

[2.0.0]

- Initial version.

COMMON

[2.6.0]

- Bug Fixes
    - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

- Improvements
    - Enable 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

[2.0.4]

- 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.

DMA

[2.1.2]

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

[2.1.1]

- Improvements
    - Corrected the dma channel feature macro from FSL_FEATURE_DMAMUX_MODULE_CHANNEL to FSL_FEATURE_DMA_MODULE_CHANNEL.

[2.1.0]

- Improvements
    - Added api DMA_PrepareTransferConfig to expose option address increment.
    - Added api DMA_EnableAutoStopRequest to support auto stop request feature.

[2.0.2]

- Bug Fixes
    - Fixed violations of MISRA C-2012 rule 10.4, 10.3, 14.4, 16.4, 11.6, 10.1.

[2.0.1]

- Bug Fixes
    - By adding parenthesis, fixed the build fail of DMA driver due to rule 12.5, MISRA C 2004.

[2.0.0]

- Initial version.

DMAMUX

[2.1.3]

- Improvements

    - Wrap DMAMUX_GetInstance into FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL to avoid build issues.

[2.1.2]

- Bug Fixes

    - Add macro FSL_DMAMUX_CHANNEL_NUM to calculat correct DMAMUX channel number when input EDAM channel number.

[2.1.1]

- Improvements

    - Add macro FSL_FEATURE_DMAMUX_CHANNEL_NEEDS_ENDIAN_CONVERT and DMAMUX_CHANNEL_ENDIAN_CONVERTn do channel endian convert.

[2.1.0]

- Improvements

    - Modify the type of parameter source from uint32_t to int32_t in the DMA-MUX_SetSource.

[2.0.5]

- Improvements

    - Added feature FSL_FEATURE_DMAMUX_CHCFG_REGISTER_WIDTH for the difference of CHCFG register width.

[2.0.4]

- Bug Fixes

    - Fixed violations of MISRA C-2012 rule 10.4.

[2.0.3]

- Bug Fixes

    - Fixed the issue for MISRA-2012 check.

        * Fixed rule 10.4 and rule 10.3.

[2.0.2]

- New Features

    - Added an always-on enable feature to a DMA channel for ULP1 DMAMUX support.

[2.0.1]

- Bug Fixes

    - Fixed the build warning issue by changing the type of parameter source from uint8_t to uint32_t when setting DMA request source in DMAMUX_SetSourceChange.

[2.0.0]

- Initial version.

EWM

[2.0.4]

- Bug Fixes

    - Fixed CERT INT31-C violations.

[2.0.3]

- Bug Fixes

    - Fixed violation of MISRA C-2012 rules: 10.1, 10.3.

[2.0.2]

- Bug Fixes

    - Fixed violation of MISRA C-2012 rules: 10.3, 10.4.

[2.0.1]

- Bug Fixes

    - Fixed the hard fault in EWM_Deinit.

[2.0.0]

- Initial version.

FLASH

[3.3.0]

- New Feature

    - Support for EEPROM Quick Write on devices with FTFC

[3.2.0]

- New Feature
    - Basic support for FTFC

[3.1.3]

- New Feature
    - Support 512KB flash for Kinetis E serials.

[3.1.2]

- Bug Fixes — Remove redundant comments.

[3.1.1]

- Bug Fixes — MISRA C-2012 issue fixed: rule 10.3

[3.1.0]

- New Feature
    - Support erase flash asynchronously.

[3.0.2]

- Bug Fixes — MISRA C-2012 issue fixed: rule 8.4, 17.7, 10.4, 16.1, 21.15, 11.3, 10.7 — building warning -Wnull-dereference on arm compiler v6

[3.0.1]

- New Features
    - Added support FlexNVM alias for (kw37/38/39).

[3.0.0]

- Improvements
    - Reorganized FTFx flash driver source file.
    - Extracted flash cache driver from FTFx driver.
    - Extracted flexnvm flash driver from FTFx driver.

[2.3.1]

- Bug Fixes
    - Unified Flash IFR design from K3.
    - New encoding rule for K3 flash size.

[2.3.0]

- New Features

    - Added support for device with LP flash (K3S/G).

    - Added flash prefetch speculation APIs.

- Improvements

    - Refined flash_cache_clear function.

    - Reorganized the member of flash_config_t struct.

[2.2.0]

- New Features

    - Supported FTFL device in FLASH_Swap API.

    - Supported various pflash start addresses.

    - Added support for KV58 in cache clear function.

    - Added support for device with secondary flash (KW40).

- Bug Fixes

    - Compiled execute-in-ram functions as PIC binary code for driver use.

    - Added missed flexram properties.

    - Fixed unaligned variable issue for execute-in-ram function code array.

[2.1.0]

- Improvements

    - Updated coding style to align with KSDK 2.0.

    - Different-alignment-size support for pflash and flexnvm.

    - Improved the implementation of execute-in-ram functions.

[2.0.0]

- Initial version

GPIO

[2.8.2]

- Bug Fixes

    - Fixed COVERITY issue that GPIO_GetInstance could return clock array overflow values due to GPIO base and clock being out of sync.

[2.8.1]

- Bug Fixes

    - Fixed CERT INT31-C issues.

[2.8.0]

- Improvements
  - Add API GPIO_PortInit/GPIO_PortDeinit to set GPIO clock enable and releasing GPIO reset.

[2.8.0]

- Improvements
  - Add API GPIO_PortInit/GPIO_PortDeinit to set GPIO clock enable and releasing GPIO reset.
  - Remove support for API GPIO_GetPinsDMARequestFlags with GPIO_ISFR_COUNT <= 1.

[2.7.3]

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

[2.7.2]

- New Features
  - Support devices without PORT module.

[2.7.1]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.4 issues in GPIO_GpioGetInterruptChannelFlags() function and GPIO_GpioClearInterruptChannelFlags() function.

[2.7.0]

- New Features
  - Added API to support Interrupt select (IRQS) bitfield.

[2.6.0]

- New Features
  - Added API to get GPIO version information.
  - Added API to control a pin for general purpose input.
  - Added some APIs to control pin in secure and previliege status.

[2.5.3]

- Bug Fixes
  - Correct the feature macro typo: FSL_FEATURE_GPIO_HAS_NO_INDEP_OUTPUT_CONTORL.

[2.5.2]

- Improvements

  - Improved GPIO_PortSet/GPIO_PortClear/GPIO_PortToggle functions to support devices without Set/Clear/Toggle registers.

[2.5.1]

- Bug Fixes

  - Fixed wrong macro definition.

  - Fixed MISRA C-2012 rule issues in the FGPIO_CheckAttributeBytes() function.

  - Defined the new macro to separate the scene when the width of registers is different.

  - Removed some redundant macros.

- New Features

  - Added some APIs to get/clear the interrupt status flag when the port doesn't control pins' interrupt.

[2.4.1]

- Improvements

  - Improved GPIO_CheckAttributeBytes() function to support 8 bits width GACR register.

[2.4.0]

- Improvements

  - API interface added:

    * New APIs were added to configure the GPIO interrupt clear settings.

[2.3.2]

- Bug Fixes

  - Fixed the issue for MISRA-2012 check.

    * Fixed rule 3.1, 10.1, 8.6, 10.6, and 10.3.

[2.3.1]

- Improvements

  - Removed deprecated APIs.

[2.3.0]

- New Features

  - Updated the driver code to adapt the case of interrupt configurations in GPIO module. New APIs were added to configure the GPIO interrupt settings if the module has this feature on it.

[2.2.1]

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

[2.1.1]

- Improvements
  - API interface changes:
    * Added an API for the check attribute bytes.

[2.1.0]

- Improvements
  - API interface changes:
    * Added "pins" or "pin" to some APIs' names.
    * Renamed "_PinConfigure" to "GPIO_PinInit".

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.

IRTC

[2.3.3]

- Bug Fixes

    - Fix CERT INT31-C issue.

[2.3.2]

- Bug Fixes

    - Fixed API IRTC_GetDatetime read YEARMON, DAYS, HOURMIN, SECONDS registers issue.

[2.3.1]

- Bug Fixes

    - Fixed MISRA C-2012 issue 10.4.

[2.3.0]

- New Feature
    - Supported platforms with multiple IRTC instances.

[2.2.4]

- Bug Fixes
    - Fixed MISRA C-2012 issue 10.1, 10.3, 10.4, 10.7, 12.2.

[2.2.3]

- Bug Fixes
    - Updated undefined macro names by available ones.

[2.2.2]

- Bug Fixes
    - Fixed MISRA C-2012 issue 10.3.

[2.2.1]

- Bug Fixes
    - Fixed MISRA issues.

[2.2.0]

- New Feature
    - Add new APIs for CLK_SEL and CLKO to select RTC clock and enable/disable output to peripherals.
    - Supported platforms without tamper feature.

[2.1.3]

- Bug Fixes
    - Fixed MISRA C-2012 issue 10.1 and 10.4.

[2.1.2]

- Bug Fixes
    - Fixed kIRTC_TamperFlag flag can't be cleared issue.

[2.1.1]

- Bug Fixes
    - MISRA C-2012 issue check.
        * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4.

[2.1.0]

- Bug Fixes

  – Fixed incorrect leap year check in IRTC_CheckDatetimeFormat.

- New Feature

  – Added new APIs for new feature FSL_FEATURE_RTC_HAS_SUBSYSTEM.

  – Added new APIs for TAMPER, TAMPER QUEUE status get and clear.

  – Added new API to enable/disable 32 kHz RTC OSC clock during RTC register write.

  – Updated IRTC_SetTamperParams to support new feature FSL_FEATURE_RTC_HAS_FILTER23_CFG

  – Updated irtc_config_t to exclude member wakeupSelect for new feature FSL_FEATURE_RTC_HAS_NO_CTRL2_WAKEUP_MODE.

[2.0.2]

- Bug Fixes

  – MISRA C-2012 issue check.

    * Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-10.6, rule-10.8, rule-11.9, rule-12.2, rule-15.5, rule-16.4, rule-17.7.

[2.0.1]

- Bug Fixes

  – Fixed the issue of hard code in IRTC_Init.

[2.0.0]

- Initial version.

LLWU

[2.0.5]

- Bug Fixes

  – Fixed violations of the MISRA C-2012 rules 10.3.

  – Fixed the issue that function LLWU_SetExternalWakeupPinMode() does not work on 32-bit width platforms.

[2.0.4]

- Bug Fixes

  – Fixed violations of the MISRA C-2012 rules 10.3, 10.4, 10.6, 10.7, 11.3.

  – Fixed issue that LLWU_ClearExternalWakeupPinFlag may clear other filter flags by mistake on platforms with 32-bit LLWU registers.

[2.0.3]

- Bug Fixes
    - Fixed MISRA-2012 rules.
        * Rule 16.4.

[2.0.2]

- Improvements
    - Corrected driver function LLWU_SetResetPinMode parameter name.
- Bug Fixes
    - Fixed MISRA-2012 rules.
        * Rule 14.4, 10.8, 10.4, 10.3.

[2.0.1]

- Other Changes
    - Updates for KL8x.

[2.0.0]

- Initial version.

LPTMR

[2.2.1]

- Bug Fixes
    - Fix CERT INT31-C issues.

[2.2.0]

- Improvements
    - Updated lptmr_prescaler_clock_select_t, only define the valid options.

[2.1.1]

- Improvements
    - Updated the characters from "PTMR" to "LPTMR" in "FSL_FEATURE_PTMR_HAS_NO_PRESCALER_CLOCK_SOURCE_1_SUPPORT" feature definition.

[2.1.0]

- Improvements
    - Implement for some special devices' not supporting for all clock sources.
- Bug Fixes
    - Fixed issue when accessing CMR register.

[2.0.2]

- Bug Fixes

  – Fixed MISRA-2012 issues.

    ∗ Rule 10.1.

[2.0.1]

- Improvements

  – Updated the LPTMR driver to support 32-bit CNR and CMR registers in some devices.

[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.

PMC

[2.0.3]

- Bug Fixes
  - Fixed the violation of MISRA C-2012 rule 11.3.

[2.0.2]

- Bug Fixes
  - Fixed the violations of MISRA 2012 rules:
    * Rule 10.3.

[2.0.1]

- Bug Fixes
  - Fixed MISRA issues.
    * Rule 10.8, Rule 10.3.

[2.0.0]

- Initial version.

PORT

[2.5.1]

- Bug Fixes
  - Fix CERT INT31-C issues.

[2.5.0]

- Bug Fixes
  - Correct the kPORT_MuxAsGpio for some platforms.

[2.4.1]

- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules: 10.1, 10.8 and 14.4.

[2.4.0]

- New Features
  - Updated port_pin_config_t to support input buffer and input invert.

[2.3.0]

- New Features

    – Added new APIs for Electrical Fast Transient(EFT) detect.

    – Added new API to configure port voltage range.

[2.2.0]

- New Features

    – Added new api PORT_EnablePinDoubleDriveStrength.

[2.1.1]

- Bug Fixes

    – Fixed the violations of MISRA C-2012 rules: 10.1, 10.4□11.3□11.8, 14.4.

[2.1.0]

- New Features

    – Updated the driver code to adapt the case of the interrupt configurations in GPIO module. Will move the pin configuration APIs to GPIO module.

[2.0.2]

- Other Changes

    – Added feature guard macros in the driver.

[2.0.1]

- Other Changes

    – Added "const" in function parameter.

    – Updated some enumeration variables' names.

QTMR

[2.0.1]

- Bug Fixes

    – MISRA C-2012 issue check.

        ∗ Fixed rules, containing: rule-10.1, rule-10.3, rule-10.4, rule-11.9, rule-14.4, rule-15.5, rule-17.7.

    – Changed FSL_COMPONENT_ID as platform.drivers.qtmr_2.

[2.0.0]

- Initial version.

RCM

[2.0.4]

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

[2.0.3]

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

[2.0.2]

- Bug Fixes
  - Fixed MISRA issue.
    * Rule 10.8, rule 10.1, rule 13.2, rule 3.1.

[2.0.1]

- Bug Fixes
  - Fixed kRCM_SourceSw bit shift issue.

[2.0.0]

- Initial version.

RNGA

[2.0.2]

- Bug fix:
  - Fix MISRA issue.

[2.0.1]

- Bug fix:
  - Fixed C++ build warning in RNGA driver.

[2.0.0]

- Initial version.

SIM

[2.2.0]

- Improvements
    - Added API to trigger TRGMUX.

[2.1.3]

- Improvements
    - Updated function SIM_GetUniqueId to support different register names.

[2.1.2]

- Bug Fixes
    - Fixed SIM_GetUniqueId bug that could not get UIDH.

[2.1.1]

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

[2.1.0]

- Improvements
    - Added new APIs: SIM_GetRfAddr() and SIM_EnableSystickClock().

[2.0.0]

- Initial version.

SLCD

[2.1.0]

- New Features
    - Added new enumerations, updated SLCD_Init and SLCD_GetDefaultConfig to support new low power IP on new SoCs.

[2.0.4]

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

[2.0.3]

- Bug Fixes
    - Fixed SLCD_Init bug that some bit-fileds are cleared by mistake.

[2.0.2]

- Bug Fixes

    - Fixed violations of the MISRA C-2012 rules 3.1, 10.1, 10.3, 10.3, 10.4 11.4, 17.7

[2.0.1]

- Bug Fixes

    - Changed the Blink mode start setting flow.

- Other Changes

    - Added static to SLCD global variables.

[2.0.0]

- Initial version.

SMC

[2.0.7]

- Bug Fixes

    - Fixed MISRA-2012 issue 10.3.

[2.0.6]

- Bug Fixes

    - Fixed issue for MISRA-2012 check.

        * Fixed rule 10.3, rule 11.3.

[2.0.5]

- Bug Fixes

    - Fixed issue for MISRA-2012 check.

        * Fixed rule 15.7, rule 14.4, rule 10.3, rule 10.1, rule 10.4.

[2.0.4]

- Bug Fixes

    - When entering stop modes, used RAM function for the flash synchronization issue. Application should make sure that, the RW data of fsl_smc.c is located in memory region which is not powered off in stop modes.

[2.0.3]

- Improvements

    - Added APIs SMC_PreEnterStopModes, SMC_PreEnterWaitModes, SMC_PostExitWaitModes, and SMC_PostExitStopModes.

[2.0.2]

- Bug Fixes
    - Added DSB before WFI while ISB after WFI.
- Other Changes
    - Updated SMC_SetPowerModeVlpw implementation.

[2.0.1]

- Other Changes
    - Updated for KL8x.

[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.

SPI DMA Driver

[2.1.1]

- Bug Fixes
    - Fixed the bug that TX data not sent to bus when transfer finish callback is called.

[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.

SYSMPU

[2.2.3]

- Bug Fixes
    - Fixed violation of MISRA C-2012 Rule 10.4, a part of issues is ignored before.

[2.2.2]

- Bug Fixes
    - Fixed violation of MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.7, 10.6, 10.8, 12.2.

[2.2.1]

- Bug Fixes
    - Fixed MISRA issue.

[2.2.0]

- Improvements
    - Renamed MPU to SYSMPU.
    - Changed macro definition for slave number and fixed the get error status calculation.

[2.1.1]

- Improvements

  - Added the feature file macro definition limitation for the MPU_SetRegionRwMasterAccessRights().

[2.1.0]

- Other Changes

  - API changes:

    * Changed the mpu_region_num_t and mpu_master_t to uint32_t.

    * Changed the mpu_low_masters_access_rights_t, mpu_high_masters_access_rights_t to mpu_rwxrights_master_access_control_t, mpu_rwrights_master_access_control_t.

    * Changed the MPU_SetRegionLowMasterAccessRights(), MPU_SetRegionHighMasterAccessRights() to MPU_SetRegionRwxMasterAccessRights(), MPU_SetRegionRwMasterAccessRights().

[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_SendEDMACallback 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.

UART_DMA

[2.5.0]

- Refer UART driver change log 2.1.0 to 2.5.0

VREF

[2.1.3]

- Improvements
  - Add timeout for APIs with dfmea issues.

[2.1.2]

- Bug Fixes
  - Fixed the violation of MISRA-2012 rule 10.3.
  - Fixed MISRA C-2012 rule 10.3, rule 10.4 violation.

[2.1.1]

- Bug Fixes
  - MISRA-2012 issue fixed.
    * Fixed rules containing: rule-10.4, rule-10.3, rule-10.1.

[2.1.0]

- Improvements
  - Added new functions to support L5K board: added VREF_SetTrim2V1Val() and VREF_GetTrim2V1Val() functions to supply 2V1 output mode.

[2.0.0]

- Initial version.

WDOG

[2.0.2]

- Improvements
  - WDG_Init() adds a 256 bus clock delay for WCT window finish.

[2.0.1]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7 11.9 and 17.7.

[2.0.0]

- Initial version.

XBAR

[2.1.0]

- Improvements

    - Improved to support XBAR which has less than 4 interrupt output.

- Bug Fixes

    - Fixed violations of MISRA C-2012 rule 12.2.

[2.0.5]

- Bug Fixes

    - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.7, 10.8, 12.2, 18.1, 20.7.

[2.0.4]

- Bug Fixes

    - Fixed IAR build warning Pa082.

[2.0.3]

- Improvements

    - Optimized XBAR_SetOutputSignalConfig.

[2.0.2]

- Bug Fixes

    - Corrected configuration for function XBAR_SetOutputSignalConfig.

[2.0.1]

- Bug Fixes

    - Fixed w1c bits for XBAR_SetOutputSignalConfig function.

[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.

MKM34ZA5

1.7 Middleware Documentation

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

1.7.1 FreeMASTER

freemaster

1.7.2 FreeRTOS

FreeRTOS

Chapter 2

MKM34ZA5

2.1 ADC16: 16-bit SAR Analog-to-Digital Converter Driver

void ADC16_Init(ADC_Type *base, const *adc16_config_t* *config)

> Initializes the ADC16 module.

> > **Parameters**

> > > • base – ADC16 peripheral base address.

> > > • config – Pointer to configuration structure. See "adc16_config_t".

void ADC16_Deinit(ADC_Type *base)

> De-initializes the ADC16 module.

> > **Parameters**

> > > • base – ADC16 peripheral base address.

void ADC16_GetDefaultConfig(*adc16_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     = kADC16_ReferenceVoltageSourceVref;
config->clockSource                = kADC16_ClockSourceAsynchronousClock;
config->enableAsynchronousClock    = false;
config->clockDivider               = kADC16_ClockDivider8;
config->resolution                 = kADC16_ResolutionSE12Bit;
config->longSampleMode             = kADC16_LongSampleDisabled;
config->enableHighSpeed            = false;
config->enableLowPower             = false;
config->enableContinuousConversion = false;
```

> > **Parameters**

> > > • config – Pointer to the configuration structure.

status_t ADC16_DoAutoCalibration(ADC_Type *base)

> Automates the hardware calibration.

> This auto calibration helps to adjust the plus/minus side gain automatically. Execute the calibration before using the converter. Note that the hardware trigger should be used during the calibration.

> > **Parameters**

- base – ADC16 peripheral base address.

Return values

- kStatus_Success – Calibration is done successfully.

- kStatus_Fail – Calibration has failed.

Returns

Execution status.

static inline void ADC16_SetOffsetValue(ADC_Type *base, int16_t value)

Sets the offset value for the conversion result.

This offset value takes effect on the conversion result. If the offset value is not zero, the reading result is subtracted by it. Note, the hardware calibration fills the offset value automatically.

Parameters

- base – ADC16 peripheral base address.

- value – Setting offset value.

static inline void ADC16_EnableDMA(ADC_Type *base, bool enable)

Enables generating the DMA trigger when the conversion is complete.

Parameters

- base – ADC16 peripheral base address.

- enable – Switcher of the DMA feature. "true" means enabled, "false" means not enabled.

static inline void ADC16_EnableHardwareTrigger(ADC_Type *base, bool enable)

Enables the hardware trigger mode.

Parameters

- base – ADC16 peripheral base address.

- enable – Switcher of the hardware trigger feature. "true" means enabled, "false" means not enabled.

void ADC16_SetChannelMuxMode(ADC_Type *base, *adc16_channel_mux_mode_t* mode)

Sets the channel mux mode.

Some sample pins share the same channel index. The channel mux mode decides which pin is used for an indicated channel.

Parameters

- base – ADC16 peripheral base address.

- mode – Setting channel mux mode. See "adc16_channel_mux_mode_t".

void ADC16_SetHardwareCompareConfig(ADC_Type *base, const
adc16_hardware_compare_config_t *config)

Configures the hardware compare mode.

The hardware compare mode provides a way to process the conversion result automatically by using hardware. Only the result in the compare range is available. To compare the range, see "adc16_hardware_compare_mode_t" or the appopriate reference manual for more information.

Parameters

- base – ADC16 peripheral base address.

- config – Pointer to the "adc16_hardware_compare_config_t" structure. Passing "NULL" disables the feature.

void ADC16_SetHardwareAverage(ADC_Type *base, *adc16_hardware_average_mode_t* mode)

Sets the hardware average mode.

The hardware average mode provides a way to process the conversion result automatically by using hardware. The multiple conversion results are accumulated and averaged internally making them easier to read.

Parameters

- base – ADC16 peripheral base address.
- mode – Setting the hardware average mode. See "adc16_hardware_average_mode_t".

void ADC16_SetPGAConfig(ADC_Type *base, const *adc16_pga_config_t* *config)

Configures the PGA for the converter's front end.

Parameters

- base – ADC16 peripheral base address.
- config – Pointer to the "adc16_pga_config_t" structure. Passing "NULL" disables the feature.

uint32_t ADC16_GetStatusFlags(ADC_Type *base)

Gets the status flags of the converter.

Parameters

- base – ADC16 peripheral base address.

Returns

Flags' mask if indicated flags are asserted. See "_adc16_status_flags".

void ADC16_ClearStatusFlags(ADC_Type *base, uint32_t mask)

Clears the status flags of the converter.

Parameters

- base – ADC16 peripheral base address.
- mask – Mask value for the cleared flags. See "_adc16_status_flags".

static inline void ADC16_EnableAsynchronousClockOutput(ADC_Type *base, bool enable)

Enable/disable ADC Asynchronous clock output to other modules.

Parameters

- base – ADC16 peripheral base address.
- enable – Used to enable/disable ADC ADACK output.
  - **true** Asynchronous clock and clock output is enabled regardless of the state of the ADC.
  - **false** Asynchronous clock output disabled, asynchronous clock is enabled only if it is selected as input clock and a conversion is active.

void ADC16_SetChannelConfig(ADC_Type *base, uint32_t channelGroup, const *adc16_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.

Note that the "Channel Group" has a detailed description. To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC has more than one group of status and control registers, one for each conversion. The channel group parameter indicates which group of registers are used, for example, channel group 0 is for Group A registers and channel group 1 is for Group B registers. The channel groups are used in a "ping-pong" approach to control the ADC operation. At any point, only one of the channel groups is actively controlling ADC conversions. The channel group 0 is used for both software and hardware trigger modes. Channel group 1 and greater indicates multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual for the number of SC1n registers (channel groups) specific to this device. Channel group 1 or greater are not used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

- base – ADC16 peripheral base address.

- channelGroup – Channel group index.

- config – Pointer to the "adc16_channel_config_t" structure for the conversion channel.

static inline uint32_t ADC16_GetChannelConversionValue(ADC_Type *base, uint32_t channelGroup)

Gets the conversion value.

Parameters

- base – ADC16 peripheral base address.

- channelGroup – Channel group index.

Returns

Conversion value.

uint32_t ADC16_GetChannelStatusFlags(ADC_Type *base, uint32_t channelGroup)

Gets the status flags of channel.

Parameters

- base – ADC16 peripheral base address.

- channelGroup – Channel group index.

Returns

Flags' mask if indicated flags are asserted. See "_adc16_channel_status_flags".

FSL_ADC16_DRIVER_VERSION

ADC16 driver version 2.3.0.

enum _adc16_channel_status_flags

Channel status flags.

Values:

enumerator kADC16_ChannelConversionDoneFlag

Conversion done.

enum _adc16_status_flags

Converter status flags.

Values:

enumerator kADC16_ActiveFlag
    Converter is active.

enumerator kADC16_CalibrationFailedFlag
    Calibration is failed.

enum _adc_channel_mux_mode
    Channel multiplexer mode for each channel.

    For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0_SE4a and ADC0_SE4b are the different channels that share the same channel number.

    Values:

    enumerator kADC16_ChannelMuxA
        For channel with channel mux a.

    enumerator kADC16_ChannelMuxB
        For channel with channel mux b.

enum _adc16_clock_divider
    Clock divider for the converter.

    Values:

    enumerator kADC16_ClockDivider1
        For divider 1 from the input clock to the module.

    enumerator kADC16_ClockDivider2
        For divider 2 from the input clock to the module.

    enumerator kADC16_ClockDivider4
        For divider 4 from the input clock to the module.

    enumerator kADC16_ClockDivider8
        For divider 8 from the input clock to the module.

enum _adc16_resolution
    Converter's resolution.

    Values:

    enumerator kADC16_Resolution8or9Bit
        Single End 8-bit or Differential Sample 9-bit.

    enumerator kADC16_Resolution12or13Bit
        Single End 12-bit or Differential Sample 13-bit.

    enumerator kADC16_Resolution10or11Bit
        Single End 10-bit or Differential Sample 11-bit.

    enumerator kADC16_ResolutionSE8Bit
        Single End 8-bit.

    enumerator kADC16_ResolutionSE12Bit
        Single End 12-bit.

    enumerator kADC16_ResolutionSE10Bit
        Single End 10-bit.

    enumerator kADC16_ResolutionDF9Bit
        Differential Sample 9-bit.

enumerator kADC16_ResolutionDF13Bit
    Differential Sample 13-bit.

enumerator kADC16_ResolutionDF11Bit
    Differential Sample 11-bit.

enum __adc16_clock_source
    Clock source.

    Values:

    enumerator kADC16_ClockSourceAlt0
        Selection 0 of the clock source.

    enumerator kADC16_ClockSourceAlt1
        Selection 1 of the clock source.

    enumerator kADC16_ClockSourceAlt2
        Selection 2 of the clock source.

    enumerator kADC16_ClockSourceAlt3
        Selection 3 of the clock source.

    enumerator kADC16_ClockSourceAsynchronousClock
        Using internal asynchronous clock.

enum __adc16_long_sample_mode
    Long sample mode.

    Values:

    enumerator kADC16_LongSampleCycle24
        20 extra ADCK cycles, 24 ADCK cycles total.

    enumerator kADC16_LongSampleCycle16
        12 extra ADCK cycles, 16 ADCK cycles total.

    enumerator kADC16_LongSampleCycle10
        6 extra ADCK cycles, 10 ADCK cycles total.

    enumerator kADC16_LongSampleCycle6
        2 extra ADCK cycles, 6 ADCK cycles total.

    enumerator kADC16_LongSampleDisabled
        Disable the long sample feature.

enum __adc16_reference_voltage_source
    Reference voltage source.

    Values:

    enumerator kADC16_ReferenceVoltageSourceVref
        For external pins pair of VrefH and VrefL.

    enumerator kADC16_ReferenceVoltageSourceValt
        For alternate reference pair of ValtH and ValtL.

enum __adc16_hardware_average_mode
    Hardware average mode.

    Values:

    enumerator kADC16_HardwareAverageCount4
        For hardware average with 4 samples.

enumerator kADC16_HardwareAverageCount8
   For hardware average with 8 samples.

enumerator kADC16_HardwareAverageCount16
   For hardware average with 16 samples.

enumerator kADC16_HardwareAverageCount32
   For hardware average with 32 samples.

enumerator kADC16_HardwareAverageDisabled
   Disable the hardware average feature.

enum _adc16_hardware_compare_mode
   Hardware compare mode.

   Values:

   enumerator kADC16_HardwareCompareMode0
      x < value1.

   enumerator kADC16_HardwareCompareMode1
      x > value1.

   enumerator kADC16_HardwareCompareMode2
      if value1 <= value2, then x < value1 || x > value2; else, value1 > x > value2.

   enumerator kADC16_HardwareCompareMode3
      if value1 <= value2, then value1 <= x <= value2; else x >= value1 || x <= value2.

enum _adc16_pga_gain
   PGA's Gain mode.

   Values:

   enumerator kADC16_PGAGainValueOf1
      For amplifier gain of 1.

   enumerator kADC16_PGAGainValueOf2
      For amplifier gain of 2.

   enumerator kADC16_PGAGainValueOf4
      For amplifier gain of 4.

   enumerator kADC16_PGAGainValueOf8
      For amplifier gain of 8.

   enumerator kADC16_PGAGainValueOf16
      For amplifier gain of 16.

   enumerator kADC16_PGAGainValueOf32
      For amplifier gain of 32.

   enumerator kADC16_PGAGainValueOf64
      For amplifier gain of 64.

typedef enum *_adc_channel_mux_mode* adc16_channel_mux_mode_t
   Channel multiplexer mode for each channel.

   For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0_SE4a and ADC0_SE4b are the different channels that share the same channel number.

typedef enum *_adc16_clock_divider* adc16_clock_divider_t
   Clock divider for the converter.

typedef enum _*adc16_resolution* adc16_resolution_t
  Converter's resolution.

typedef enum _*adc16_clock_source* adc16_clock_source_t
  Clock source.

typedef enum _*adc16_long_sample_mode* adc16_long_sample_mode_t
  Long sample mode.

typedef enum _*adc16_reference_voltage_source* adc16_reference_voltage_source_t
  Reference voltage source.

typedef enum _*adc16_hardware_average_mode* adc16_hardware_average_mode_t
  Hardware average mode.

typedef enum _*adc16_hardware_compare_mode* adc16_hardware_compare_mode_t
  Hardware compare mode.

typedef enum _*adc16_pga_gain* adc16_pga_gain_t
  PGA's Gain mode.

typedef struct _*adc16_config* adc16_config_t
  ADC16 converter configuration.

typedef struct _*adc16_hardware_compare_config* adc16_hardware_compare_config_t
  ADC16 Hardware comparison configuration.

typedef struct _*adc16_channel_config* adc16_channel_config_t
  ADC16 channel conversion configuration.

typedef struct _*adc16_pga_config* adc16_pga_config_t
  ADC16 programmable gain amplifier configuration.

struct _adc16_config_
  #include <fsl_adc16.h> ADC16 converter configuration.

Public Members

adc16_reference_voltage_source_t referenceVoltageSource
  Select the reference voltage source.

adc16_clock_source_t clockSource
  Select the input clock source to converter.

bool enableAsynchronousClock
  Enable the asynchronous clock output.

adc16_clock_divider_t clockDivider
  Select the divider of input clock source.

adc16_resolution_t resolution
  Select the sample resolution mode.

adc16_long_sample_mode_t longSampleMode
  Select the long sample mode.

bool enableHighSpeed
  Enable the high-speed mode.

bool enableLowPower
  Enable low power.

bool enableContinuousConversion

Enable continuous conversion mode.

adc16_hardware_average_mode_t hardwareAverageMode

Set hardware average mode.

struct __adc16__hardware__compare__config

#include <fsl_adc16.h> ADC16 Hardware comparison configuration.

Public Members

adc16_hardware_compare_mode_t hardwareCompareMode

Select the hardware compare mode. See "adc16_hardware_compare_mode_t".

int16_t value1

Setting value1 for hardware compare mode.

int16_t value2

Setting value2 for hardware compare mode.

struct __adc16__channel__config

#include <fsl_adc16.h> ADC16 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 enableInterruptOnConversionCompleted

Generate an interrupt request once the conversion is completed.

bool enableDifferentialConversion

Using Differential sample mode.

struct __adc16__pga__config

#include <fsl_adc16.h> ADC16 programmable gain amplifier configuration.

Public Members

adc16_pga_gain_t pgaGain

Setting PGA gain.

bool enableRunInNormalMode

Enable PGA working in normal mode, or low power mode by default.

bool disablePgaChopping

Disable the PGA chopping function. The PGA employs chopping to remove/reduce offset and 1/f noise and offers an offset measurement configuration that aids the offset calibration.

bool enableRunInOffsetMeasurement

Enable the PGA working in offset measurement mode. When this feature is enabled, the PGA disconnects itself from the external inputs and auto-configures into offset measurement mode. With this field set, run the ADC in the recommended settings and enable the maximum hardware averaging to get the PGA offset number. The output is the (PGA offset * (64+1)) for the given PGA setting.

2.2   AFE: Analog Front End Driver

void AFE_Init(AFE_Type *base, const *afe_config_t* *config)

>   Initialization for the AFE module.

>   This function configures the AFE module for the configuration which are shared by all channels.

>   > **Parameters**

>   > >   • base – AFE peripheral base address.

>   > >   • config – Pointer to structure of "afe_config_t".

void AFE_Deinit(AFE_Type *base)

>   De-Initialization for the AFE module.

>   This function disables clock.

>   > **Parameters**

>   > >   • base – AFE peripheral base address.

void AFE_GetDefaultConfig(*afe_config_t* *config)

>   Fills the user configure structure.

>   This function fills the afe_config_t structure with default settings. Defaut value are:

```
config->enableLowPower   = false;
config->resultFormat     = kAFE_ResultFormatRight;
config->clockDivider     = kAFE_ClockDivider2;
config->clockSource      = kAFE_ClockSource1;
config->startupCount     = 2U;
```

>   > **Parameters**

>   > >   • config – Pointer to structure of "afe_config_t".

static inline void AFE_SoftwareReset(AFE_Type *base, bool enable)

>   Software reset the AFE module.

>   This function is to reset all the ADCs, PGAs, decimation filters and clock configuration bits. When asserted as "false", all ADCs, PGAs and decimation filters are disabled. Clock Configuration bits are reset. When asserted as "true", all ADCs, PGAs and decimation filters are enabled.

>   > **Parameters**

>   > >   • base – AFE peripheral base address.

>   > >   • enable – Assert the reset command.

static inline void AFE_Enable(AFE_Type *base, bool enable)

>   Enables all configured AFE channels.

>   This function enables AFE and filter.

>   > **Parameters**

>   > >   • base – AFE peripheral base address.

>   > >   • enable – Enable the AFE module or not.

void AFE_SetChannelConfig(AFE_Type *base, uint32_t channel, const *afe_channel_config_t* *config)

>   Configure the selected AFE channel.

>   This function configures the selected AFE channel.

Parameters

- base – AFE peripheral base address.
- channel – AFE channel index.
- config – Pointer to structure of "afe_channel_config_t".

void AFE_GetDefaultChannelConfig(*afe_channel_config_t* *config)

Fills the channel configuration structure.

This function fills the afe_channel_config_t structure with default settings. Default value are:

```
config->enableHardwareTrigger      = false;
config->enableContinuousConversion = false;
config->channelMode                = kAFE_Normal;
config->decimatorOversampleRatio   = kAFE_DecimatorOversampleRatio64;
config->pgaGainSelect              = kAFE_PgaGain1;
```

Parameters

- config – Pointer to structure of "afe_channel_config_t".

uint32_t AFE_GetChannelConversionValue(AFE_Type *base, uint32_t channel)

Reads the raw conversion value.

This function returns the raw conversion value of the selected channel.

Note: The returned value could be left or right adjusted according to the AFE module configuration.

Parameters

- base – AFE peripheral base address.
- channel – AFE channel index.

Returns

Conversion value.

static inline void AFE_DoSoftwareTriggerChannel(AFE_Type *base, uint32_t mask)

Triggers the AFE conversion by software.

This function triggers the AFE conversion by executing a software command. It starts the conversion on selected channels if the software trigger option is selected for the channels.

Parameters

- base – AFE peripheral base address.
- mask – AFE channel mask software trigger. The parameter can be combination of the following source if defined:
  - kAFE_Channel0Trigger
  - kAFE_Channel1Trigger
  - kAFE_Channel2Trigger
  - kAFE_Channel3Trigger

static inline uint32_t AFE_GetChannelStatusFlags(AFE_Type *base)

Gets the AFE status flag state.

This function gets all AFE status.

Parameters

- base – AFE peripheral base address.

Returns

the mask of these status flag bits.

void AFE_SetChannelPhaseDelayValue(AFE_Type *base, uint32_t channel, uint32_t value)

Sets phase delays value.

This function sets the phase delays for channels. This delay is inserted before the trigger response of the decimation filters. The delay is used to provide a phase compensation between AFE channels in step of prescaled modulator clock periods.

Parameters

- base – AFE peripheral base address.

- channel – AFE channel index.

- value – delay time value.

static inline void AFE_SetChannelPhasetDelayOk(AFE_Type *base)

Asserts the phase delay setting.

This function should be called after all desired channel's delay registers are loaded. Values in channel's delay registers are active after calling this function and after the conversation starts.

Parameters

- base – AFE peripheral base address.

static inline void AFE_EnableChannelInterrupts(AFE_Type *base, uint32_t mask)

Enables AFE interrupt.

This function enables one channel interrupt.

Parameters

- base – AFE peripheral base address.

- mask – AFE channel interrupt mask. The parameter can be combination of the following source if defined:

  - kAFE_Channel0InterruptEnable

  - kAFE_Channel1InterruptEnable

  - kAFE_Channel2InterruptEnable

  - kAFE_Channel3InterruptEnable

static inline void AFE_DisableChannelInterrupts(AFE_Type *base, uint32_t mask)

Disables AFE interrupt.

This function disables one channel interrupt.

Parameters

- base – AFE peripheral base address.

- mask – AFE channel interrupt mask. The parameter can be combination of the following source if defined:

  - kAFE_Channel0InterruptEnable

  - kAFE_Channel1InterruptEnable

  - kAFE_Channel2InterruptEnable

  - kAFE_Channel3InterruptEnable

static inline uint32_t AFE_GetEnabledChannelInterrupts(AFE_Type *base)

   Returns mask of all enabled AFE interrupts.

   Parameters

   - base – AFE peripheral base address.

   Returns

   Return the mask of these interrupt enable/disable bits.

void AFE_EnableChannelDMA(AFE_Type *base, uint32_t mask, bool enable)

   Enables/Disables AFE DMA.

   This function enables/disables one channel DMA request.

   Parameters

   - base – AFE peripheral base address.

   - mask – AFE channel dma mask.

   - enable – Pass true to enable interrupt, false to disable. The parameter can be combination of the following source if defined:

      – kAFE_Channel0DMAEnable

      – kAFE_Channel1DMAEnable

      – kAFE_Channel2DMAEnable

      – kAFE_Channel3DMAEnable

FSL_AFE_DRIVER_VERSION

   Version 2.0.3.

enum _afe_channel_status_flag

   Defines the type of status flags.

   Values:

   enumerator kAFE_Channel0OverflowFlag

   Channel 0 previous conversion result has not been read and new data has already arrived.

   enumerator kAFE_Channel1OverflowFlag

   Channel 1 previous conversion result has not been read and new data has already arrived.

   enumerator kAFE_Channel2OverflowFlag

   Channel 2 previous conversion result has not been read and new data has already arrived.

   enumerator kAFE_Channel0ReadyFlag
   Channel 0 is ready to conversion.

   enumerator kAFE_Channel1ReadyFlag
   Channel 1 is ready to conversion.

   enumerator kAFE_Channel2ReadyFlag
   Channel 2 is ready to conversion.

   enumerator kAFE_Channel0ConversionCompleteFlag
   Channel 0 conversion is complete.

   enumerator kAFE_Channel1ConversionCompleteFlag
   Channel 1 conversion is complete.

enumerator kAFE_Channel2ConversionCompleteFlag
    Channel 2 conversion is complete.

enumerator kAFE_Channel3OverflowFlag
    Channel 3 previous conversion result has not been read and new data has already arrived.

enumerator kAFE_Channel3ReadyFlag
    Channel 3 is ready to conversion.

enumerator kAFE_Channel3ConversionCompleteFlag
    Channel 3 conversion is complete.

Defines AFE interrupt enable.

Values:

enumerator kAFE_Channel0InterruptEnable
    Channel 0 Interrupt.

enumerator kAFE_Channel1InterruptEnable
    Channel 1 Interrupt.

enumerator kAFE_Channel2InterruptEnable
    Channel 2 Interrupt.

enumerator kAFE_Channel3InterruptEnable
    Channel 3 Interrupt.

Defines AFE DMA enable.

Values:

enumerator kAFE_Channel0DMAEnable
    Channel 0 DMA.

enumerator kAFE_Channel1DMAEnable
    Channel 1 DMA.

enumerator kAFE_Channel2DMAEnable
    Channel 2 DMA.

enumerator kAFE_Channel3DMAEnable
    Channel 3 DMA

Defines AFE channel trigger flag.

Values:

enumerator kAFE_Channel0Trigger
    Channel 0 software trigger.

enumerator kAFE_Channel1Trigger
    Channel 1 software trigger.

enumerator kAFE_Channel2Trigger
    Channel 2 software trigger.

enumerator kAFE_Channel3Trigger
    Channel 3 software trigger.

enum __afe_decimator_oversampling_ratio

AFE OSR modes.

Values:

enumerator kAFE_DecimatorOversampleRatio64
Decimator over sample ratio is 64.

enumerator kAFE_DecimatorOversampleRatio128
Decimator over sample ratio is 128.

enumerator kAFE_DecimatorOversampleRatio256
Decimator over sample ratio is 256.

enumerator kAFE_DecimatorOversampleRatio512
Decimator over sample ratio is 512.

enumerator kAFE_DecimatorOversampleRatio1024
Decimator over sample ratio is 1024.

enumerator kAFE_DecimatorOversampleRatio2048
Decimator over sample ratio is 2048.

enum __afe_result_format

Defines the AFE result format modes.

Values:

enumerator kAFE_ResultFormatLeft
Left justified result format.

enumerator kAFE_ResultFormatRight
Right justified result format.

enum __afe_clock_divider

Defines the AFE clock divider modes.

Values:

enumerator kAFE_ClockDivider1
Clock divided by 1.

enumerator kAFE_ClockDivider2
Clock divided by 2.

enumerator kAFE_ClockDivider4
Clock divided by 4.

enumerator kAFE_ClockDivider8
Clock divided by 8.

enumerator kAFE_ClockDivider16
Clock divided by 16.

enumerator kAFE_ClockDivider32
Clock divided by 32.

enumerator kAFE_ClockDivider64
Clock divided by 64.

enumerator kAFE_ClockDivider128
Clock divided by 128.

enumerator kAFE_ClockDivider256
   Clock divided by 256.

enum __afe_clock_source
   Defines the AFE clock source modes.

   Values:

   enumerator kAFE_ClockSource0
      Modulator clock source 0.

   enumerator kAFE_ClockSource1
      Modulator clock source 1.

   enumerator kAFE_ClockSource2
      Modulator clock source 2.

   enumerator kAFE_ClockSource3
      Modulator clock source 3.

enum __afe_pga_gain
   Defines the PGA's values.

   Values:

   enumerator kAFE_PgaDisable
      PGA disabled.

   enumerator kAFE_PgaGain1
      Input gained by 1.

   enumerator kAFE_PgaGain2
      Input gained by 2.

   enumerator kAFE_PgaGain4
      Input gained by 4.

   enumerator kAFE_PgaGain8
      Input gained by 8.

   enumerator kAFE_PgaGain16
      Input gained by 16.

   enumerator kAFE_PgaGain32
      Input gained by 32.

enum __afe_bypass_mode
   Defines the bypass modes.

   Values:

   enumerator kAFE_BypassInternalClockPositiveEdge
      Bypassed channel mode - internal clock selected, positive edge for registering data by the decimation filter

   enumerator kAFE_BypassExternalClockPositiveEdge
      Bypassed channel mode - external clock selected, positive edge for registering data by the decimation filter

   enumerator kAFE_BypassInternalClockNegativeEdge
      Bypassed channel mode - internal clock selected, negative edge for registering data by the decimation filter

enumerator kAFE_BypassExternalClockNegativeEdge
    Bypassed channel mode - external clock selected, negative edge for registering data by
    the decimation filter

enumerator kAFE_BypassDisable
    Normal channel mode.

typedef enum _afe_decimator_oversampling_ratio afe_decimator_oversample_ratio_t
    AFE OSR modes.

typedef enum _afe_result_format afe_result_format_t
    Defines the AFE result format modes.

typedef enum _afe_clock_divider afe_clock_divider_t
    Defines the AFE clock divider modes.

typedef enum _afe_clock_source afe_clock_source_t
    Defines the AFE clock source modes.

typedef enum _afe_pga_gain afe_pga_gain_t
    Defines the PGA's values.

typedef enum _afe_bypass_mode afe_bypass_mode_t
    Defines the bypass modes.

typedef struct _afe_channel_config afe_channel_config_t
    Defines the structure to initialize the AFE channel.

    This structure keeps the configuration for the AFE channel.

typedef struct _afe_config afe_config_t
    Defines the structure to initialize the AFE module.

    This structure keeps the configuration for the AFE module.

struct _afe_channel_config
    #include <fsl_afe.h> Defines the structure to initialize the AFE channel.

    This structure keeps the configuration for the AFE channel.

Public Members

bool enableHardwareTrigger
    Enable triggering by hardware.

bool enableContinuousConversion
    Enable continuous conversion mode.

afe_bypass_mode_t channelMode
    Select if channel is in bypassed mode.

afe_pga_gain_t pgaGainSelect
    Select the analog gain applied to the input signal.

afe_decimator_oversample_ratio_t decimatorOversampleRatio
    Select the over sampling ration.

struct _afe_config
    #include <fsl_afe.h> Defines the structure to initialize the AFE module.

    This structure keeps the configuration for the AFE module.

Public Members

bool enableLowPower
> Enable low power mode.

afe_result_format_t resultFormat
> Select the result format.

afe_clock_divider_t clockDivider
> Select the clock divider ration for the modulator clock.

afe_clock_source_t clockSource
> Select clock source for modulator clock.

uint8_t startupCount
> Select the start up delay of modulators.

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_Er32kClk
> > External reference 32K clock (ERCLK32K)
>
> enumerator kCLOCK_Osc0ErClk
> > OSC0 external reference clock (OSC0ERCLK)
>
> enumerator kCLOCK_McgFixedFreqClk
> > MCG fixed frequency clock (MCGFFCLK)
>
> enumerator kCLOCK_McgInternalRefClk
> > MCG internal reference clock (MCGIRCLK)
>
> enumerator kCLOCK_McgFllClk
> > MCGFLLCLK
>
> enumerator kCLOCK_McgPll0Clk
> > MCGPLL0CLK
>
> enumerator kCLOCK_McgExtPllClk
> > EXT_PLLCLK
>
> enumerator kCLOCK_McgPeriphClk
> > MCG peripheral clock (MCGPCLK)

enumerator kCLOCK_LpoClk
   LPO clock

enum __clock_ip_name

   Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

   Values:

   enumerator kCLOCK_IpInvalid

   enumerator kCLOCK_Ewm0

   enumerator kCLOCK_Mcg

   enumerator kCLOCK_Osc

   enumerator kCLOCK_I2c0

   enumerator kCLOCK_I2c1

   enumerator kCLOCK_Uart0

   enumerator kCLOCK_Uart1

   enumerator kCLOCK_Uart2

   enumerator kCLOCK_Uart3

   enumerator kCLOCK_Vref0

   enumerator kCLOCK_Cmp0

   enumerator kCLOCK_Cmp1

   enumerator kCLOCK_Spi0

   enumerator kCLOCK_Spi1

   enumerator kCLOCK_Slcd0

   enumerator kCLOCK_PortA

   enumerator kCLOCK_PortB

   enumerator kCLOCK_PortC

   enumerator kCLOCK_PortD

   enumerator kCLOCK_PortE

   enumerator kCLOCK_PortF

   enumerator kCLOCK_PortG

   enumerator kCLOCK_PortH

   enumerator kCLOCK_PortI

   enumerator kCLOCK_Rtc0

   enumerator kCLOCK_Rtcreg

   enumerator kCLOCK_Wdog

   enumerator kCLOCK_Xbar

enumerator kCLOCK_Tmr0

enumerator kCLOCK_Tmr1

enumerator kCLOCK_Tmr2

enumerator kCLOCK_Tmr3

enumerator kCLOCK_Ftf0

enumerator kCLOCK_Dmamux0

enumerator kCLOCK_Dmamux1

enumerator kCLOCK_Dmamux2

enumerator kCLOCK_Dmamux3

enumerator kCLOCK_Rnga0

enumerator kCLOCK_Adc0

enumerator kCLOCK_Pit0

enumerator kCLOCK_Pit1

enumerator kCLOCK_Afe0

enumerator kCLOCK_Crc0

enumerator kCLOCK_Lptmr0

enumerator kCLOCK_SimLp

enumerator kCLOCK_SimHp

enumerator kCLOCK_Sysmpu0

enumerator kCLOCK_Dma0

enum __osc_mode

OSC work mode.

Values:

enumerator kOSC_ModeExt
Use an external clock.

enumerator kOSC_ModeOscLowPower
Oscillator low power.

enumerator kOSC_ModeOscHighGain
Oscillator high gain.

enum __osc_cap_load

Oscillator capacitor load setting.

Values:

enumerator kOSC_Cap2P
2 pF capacitor load

enumerator kOSC_Cap4P
4 pF capacitor load

enumerator kOSC_Cap8P
    8 pF capacitor load

enumerator kOSC_Cap16P
    16 pF capacitor load

enum __oscer_enable_mode
    OSCERCLK enable mode.

    Values:

    enumerator kOSC_ErClkEnable
        Enable.

    enumerator kOSC_ErClkEnableInStop
        Enable in stop mode.

enum __mcg_fll_src
    MCG FLL reference clock source select.

    Values:

    enumerator kMCG_FllSrcExternal
        External reference clock is selected

    enumerator kMCG_FllSrcInternal
        The slow internal reference clock is selected

enum __mcg_irc_mode
    MCG internal reference clock select.

    Values:

    enumerator kMCG_IrcSlow
        Slow internal reference clock selected

    enumerator kMCG_IrcFast
        Fast internal reference clock selected

enum __mcg_dmx32
    MCG DCO Maximum Frequency with 32.768 kHz Reference.

    Values:

    enumerator kMCG_Dmx32Default
        DCO has a default range of 25%

    enumerator kMCG_Dmx32Fine
        DCO is fine-tuned for maximum frequency with 32.768 kHz reference

enum __mcg_drs
    MCG DCO range select.

    Values:

    enumerator kMCG_DrsLow
        Low frequency range

    enumerator kMCG_DrsMid
        Mid frequency range

    enumerator kMCG_DrsMidHigh
        Mid-High frequency range

enumerator kMCG_DrsHigh
>   High frequency range

enum __mcg_pll_ref_src
>   MCG PLL reference clock select.
>
>   *Values:*
>
>   enumerator kMCG_PllRefRtc
>   >   Selects 32k RTC oscillator.
>
>   enumerator kMCG_PllRefIrc
>   >   Selects 32k IRC.
>
>   enumerator kMCG_PllRefFllRef
>   >   Selects FLL reference clock, the clock after FRDIV.

enum __mcg_clkout_src
>   MCGOUT clock source.
>
>   *Values:*
>
>   enumerator kMCG_ClkOutSrcOut
>   >   Output of the FLL is selected (reset default)
>
>   enumerator kMCG_ClkOutSrcInternal
>   >   Internal reference clock is selected
>
>   enumerator kMCG_ClkOutSrcExternal
>   >   External reference clock is selected

enum __mcg_atm_select
>   MCG Automatic Trim Machine Select.
>
>   *Values:*
>
>   enumerator kMCG_AtmSel32k
>   >   32 kHz Internal Reference Clock selected
>
>   enumerator kMCG_AtmSel4m
>   >   4 MHz Internal Reference Clock selected

enum __mcg_oscsel
>   MCG OSC Clock Select.
>
>   *Values:*
>
>   enumerator kMCG_OscselOsc
>   >   Selects System Oscillator (OSCCLK)
>
>   enumerator kMCG_OscselRtc
>   >   Selects 32 kHz RTC Oscillator

enum __mcg_pll_clk_select
>   MCG PLLCS select.
>
>   *Values:*
>
>   enumerator kMCG_PllClkSelPll0
>   >   PLL0 output clock is selected
>
>   enumerator kMCG_PllClkSelPll1

enum __mcg__monitor__mode
    MCG clock monitor mode.

    Values:

    enumerator kMCG__MonitorNone
        Clock monitor is disabled.

    enumerator kMCG__MonitorInt
        Trigger interrupt when clock lost.

    enumerator kMCG__MonitorReset
        System reset when clock lost.

    MCG status. Enumeration _mcg_status.

    Values:

    enumerator kStatus__MCG__ModeUnreachable
        Can't switch to target mode.

    enumerator kStatus__MCG__ModeInvalid
        Current mode invalid for the specific function.

    enumerator kStatus__MCG__AtmBusClockInvalid
        Invalid bus clock for ATM.

    enumerator kStatus__MCG__AtmDesiredFreqInvalid
        Invalid desired frequency for ATM.

    enumerator kStatus__MCG__AtmIrcUsed
        IRC is used when using ATM.

    enumerator kStatus__MCG__AtmHardwareFail
        Hardware fail occurs during ATM.

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

    MCG status flags. Enumeration _mcg_status_flags_t.

    Values:

    enumerator kMCG__Osc0LostFlag
        OSC0 lost.

    enumerator kMCG__Osc0InitFlag
        OSC0 crystal initialized.

    enumerator kMCG__RtcOscLostFlag
        RTC OSC lost.

    enumerator kMCG__Pll0LostFlag
        PLL0 lost.

    enumerator kMCG__Pll0LockFlag
        PLL0 locked.

    MCG internal reference clock (MCGIRCLK) enable mode definition. Enumeration _mcg_irclk_enable_mode.

    Values:

enumerator kMCG_IrclkEnable
MCGIRCLK enable.

enumerator kMCG_IrclkEnableInStop
MCGIRCLK enable in stop mode.

MCG PLL clock enable mode definition. Enumeration _mcg_pll_enable_mode.

Values:

enumerator kMCG_PllEnableIndependent
MCGPLLCLK enable independent of the MCG clock mode. Generally, the PLL is disabled in FLL modes (FEI/FBI/FEE/FBE). Setting the PLL clock enable independent, enables the PLL in the FLL modes.

enumerator kMCG_PllEnableInStop
MCGPLLCLK enable in STOP mode.

enum _mcg_mode
MCG mode definitions.

Values:

enumerator kMCG_ModeFEI
FEI - FLL Engaged Internal

enumerator kMCG_ModeFBI
FBI - FLL Bypassed Internal

enumerator kMCG_ModeBLPI
BLPI - Bypassed Low Power Internal

enumerator kMCG_ModeFEE
FEE - FLL Engaged External

enumerator kMCG_ModeFBE
FBE - FLL Bypassed External

enumerator kMCG_ModeBLPE
BLPE - Bypassed Low Power External

enumerator kMCG_ModePBE
PBE - PLL Bypassed External

enumerator kMCG_ModePEE
PEE - PLL Engaged External

enumerator kMCG_ModePEI
PEI - PLL Engaged Internal

enumerator kMCG_ModePBI
PBI - PLL Bypassed Internal

enumerator kMCG_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 enum *_osc_mode* osc_mode_t

>   OSC work mode.

typedef struct *_oscer_config* oscer_config_t

>   OSC configuration for OSCERCLK.

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.

typedef enum *_mcg_fll_src* mcg_fll_src_t

>   MCG FLL reference clock source select.

typedef enum *_mcg_irc_mode* mcg_irc_mode_t

>   MCG internal reference clock select.

typedef enum *_mcg_dmx32* mcg_dmx32_t

>   MCG DCO Maximum Frequency with 32.768 kHz Reference.

typedef enum *_mcg_drs* mcg_drs_t

>   MCG DCO range select.

typedef enum *_mcg_pll_ref_src* mcg_pll_ref_src_t

>   MCG PLL reference clock select.

typedef enum *_mcg_clkout_src* mcg_clkout_src_t

>   MCGOUT clock source.

typedef enum *_mcg_atm_select* mcg_atm_select_t

>   MCG Automatic Trim Machine Select.

typedef enum *_mcg_oscsel* mcg_oscsel_t

>   MCG OSC Clock Select.

typedef enum *_mcg_pll_clk_select* mcg_pll_clk_select_t

>   MCG PLLCS select.

typedef enum *_mcg_monitor_mode* mcg_monitor_mode_t

>   MCG clock monitor mode.

typedef enum *_mcg_mode* mcg_mode_t

>   MCG mode definitions.

typedef struct *_mcg_pll_config* mcg_pll_config_t

>   MCG PLL configuration.

typedef struct *_mcg_config* mcg_config_t

>   MCG mode change configuration structure.

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

>   >   a. frdiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by frdiv is in the 31.25 kHz to 39.0625 kHz range.

>   >   b. The PLL reference clock divider PRDIV: PLL reference clock frequency after PRDIV should be in the FSL_FEATURE_MCG_PLL_REF_MIN to FSL_FEATURE_MCG_PLL_REF_MAX range.

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:

```
Set up the OSC0
CLOCK_InitOsc0(...);
Set the XTAL0 value to the clock driver.
CLOCK_SetXtal0Freq(80000000);
```

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.

volatile uint32_t g_xtal32Freq

External XTAL32/EXTAL32/RTC_CLKIN clock frequency.

The XTAL32/EXTAL32/RTC_CLKIN clock frequency in Hz. When the clock is set up, use the function CLOCK_SetXtal32Freq to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the CLOCK_SetXtal32Freq 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_SetEr32kClock(uint32_t src)

Set ERCLK32K source.

> **Parameters**
>
> > • src – The value to set ERCLK32K clock source.

static inline void CLOCK_SetAfeClkSrc(uint32_t src)

Set the clock selection of AFECLKSEL.

> **Parameters**
>
> > • src – The value to set AFECLKSEL clock source.

static inline void CLOCK_SetClkOutClock(uint32_t src)

Set CLKOUT source.

> **Parameters**
>
> > • src – The value to set CLKOUT source.

static inline void CLOCK_SetAdcTriggerClock(uint32_t src)

Set ADC trigger clock source.

> **Parameters**
>
> > • src – The value to set ADC trigger clock source.

uint32_t CLOCK_GetAfeFreq(**void**)

    Gets the clock frequency for AFE module.

    This function checks the current mode configurations in MISC_CTL register.

        Returns

            Clock frequency value in Hertz

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 MCG 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_GetPlatClkFreq(**void**)

    Get the platform 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_GetEr32kClkFreq(**void**)

    Get the external reference 32K clock frequency (ERCLK32K).

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

FSL_CLOCK_DRIVER_VERSION

> CLOCK driver version 2.0.1.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

DMAMUX_CLOCKS

> Clock ip name array for DMAMUX.

RTC_CLOCKS

> Clock ip name array for RTC.

SPI_CLOCKS

> Clock ip name array for SPI.

SLCD_CLOCKS

> Clock ip name array for SLCD.

EWM_CLOCKS

> Clock ip name array for EWM.

AFE_CLOCKS

> Clock ip name array for AFE.

ADC16_CLOCKS

> Clock ip name array for ADC16.

XBAR_CLOCKS

> Clock ip name array for XBAR.

SYSMPU_CLOCKS

> Clock ip name array for MPU.

VREF_CLOCKS

> Clock ip name array for VREF.

DMA_CLOCKS

> Clock ip name array for DMA.

PORT_CLOCKS

> Clock ip name array for PORT.

UART_CLOCKS

> Clock ip name array for UART.

PIT_CLOCKS

> Clock ip name array for PIT.

RNGA_CLOCKS

> Clock ip name array for RNGA.

CRC_CLOCKS

> Clock ip name array for CRC.

I2C_CLOCKS

> Clock ip name array for I2C.

LPTMR_CLOCKS
> Clock ip name array for LPTMR.

TMR_CLOCKS
> Clock ip name array for TMR.

PDB_CLOCKS
> Clock ip name array for PDB.

FTF_CLOCKS
> Clock ip name array for FTF.

CMP_CLOCKS
> Clock ip name array for CMP.

LPO_CLK_FREQ
> LPO clock frequency.

SYS_CLK
> Peripherals clock source definition.

BUS_CLK

I2C0_CLK_SRC

I2C1_CLK_SRC

SPI0_CLK_SRC

SPI1_CLK_SRC

UART0_CLK_SRC

UART1_CLK_SRC

UART2_CLK_SRC

UART3_CLK_SRC

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_GetOutClkFreq(void)
> Gets the MCG output clock (MCGOUTCLK) frequency.

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

>> **Returns**
>> The frequency of MCGOUTCLK.

uint32_t CLOCK_GetFllFreq(void)

Gets the MCG FLL clock (MCGFLLCLK) frequency.

This function gets the MCG FLL clock frequency in Hz based on the current MCG 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 MCGFLLCLK.

uint32_t CLOCK_GetInternalRefClkFreq(void)

Gets the MCG internal reference clock (MCGIRCLK) frequency.

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

Returns

The frequency of MCGIRCLK.

uint32_t CLOCK_GetFixedFreqClkFreq(void)

Gets the MCG fixed frequency clock (MCGFFCLK) frequency.

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

Returns

The frequency of MCGFFCLK.

uint32_t CLOCK_GetPll0Freq(void)

Gets the MCG PLL0 clock (MCGPLL0CLK) frequency.

This function gets the MCG PLL0 clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGPLL0CLK.

static inline void CLOCK_SetLowPowerEnable(bool enable)

Enables or disables the MCG low power.

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

Parameters

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

status_t CLOCK_SetInternalRefClkConfig(uint8_t enableMode, *mcg_irc_mode_t* ircs, uint8_t fcrdiv)

Configures the Internal Reference clock (MCGIRCLK).

This function sets the MCGIRCLK base on parameters. It also selects the IRC source. If the fast IRC is used, this function sets the fast IRC divider. This function also sets whether the MCGIRCLK is enabled in stop mode. Calling this function in FBI/PBI/BLPI modes may change the system clock. As a result, using the function in these modes it is not allowed.

Parameters

- enableMode – MCGIRCLK enable mode, OR'ed value of the enumeration _mcg_irclk_enable_mode.

- ircs – MCGIRCLK clock source, choose fast or slow.

- fcrdiv – Fast IRC divider setting (FCRDIV).

Return values

- kStatus_MCG_SourceUsed – Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.

- kStatus_Success – MCGIRCLK configuration finished successfully.

status_t CLOCK_SetExternalRefClkConfig(*mcg_oscsel_t* oscsel)

Selects the MCG external reference clock.

Selects the MCG external reference clock source, changes the MCG_C7[OSCSEL], and waits for the clock source to be stable. Because the external reference clock should not be changed in FEE/FBE/BLPE/PBE/PEE modes, do not call this function in these modes.

Parameters

- oscsel – MCG external reference clock source, MCG_C7[OSCSEL].

Return values

- kStatus_MCG_SourceUsed – Because the external reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.

- kStatus_Success – External reference clock set successfully.

static inline void CLOCK_SetFllExtRefDiv(uint8_t frdiv)

Set the FLL external reference clock divider value.

Sets the FLL external reference clock divider value, the register MCG_C1[FRDIV].

Parameters

- frdiv – The FLL external reference clock divider value, MCG_C1[FRDIV].

void CLOCK_EnablePll0(*mcg_pll_config_t* const *config)

Enables the PLL0 in FLL mode.

This function sets us the PLL0 in FLL mode and reconfigures the PLL0. Ensure that the PLL reference clock is enabled before calling this function and that the PLL0 is not used as a clock source. The function CLOCK_CalcPllDiv gets the correct PLL divider values.

Parameters

- config – Pointer to the configuration structure.

static inline void CLOCK_DisablePll0(void)

Disables the PLL0 in FLL mode.

This function disables the PLL0 in FLL mode. It should be used together with the CLOCK_EnablePll0.

void CLOCK_SetOsc0MonitorMode(*mcg_monitor_mode_t* mode)

Sets the OSC0 clock monitor mode.

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

Parameters

- mode – Monitor mode to set.

void CLOCK_SetRtcOscMonitorMode(*mcg_monitor_mode_t* mode)

Sets the RTC OSC clock monitor mode.

This function sets the RTC OSC clock monitor mode. See mcg_monitor_mode_t for details.

Parameters

- mode – Monitor mode to set.

void CLOCK_SetPll0MonitorMode(*mcg_monitor_mode_t* mode)

Sets the PLL0 clock monitor mode.

This function sets the PLL0 clock monitor mode. See mcg_monitor_mode_t for details.

Parameters

- mode – Monitor mode to set.

uint32_t CLOCK_GetStatusFlags(void)

Gets the MCG status flags.

This function gets the MCG clock status flags. All status flags are returned as a logical OR of the enumeration refer to _mcg_status_flags_t. To check a specific flag, compare the return value with the flag.

Example:

```
To check the clock lost lock status of OSC0 and PLL0.
uint32_t mcgFlags;

mcgFlags = CLOCK_GetStatusFlags();

if (mcgFlags & kMCG_Osc0LostFlag)
{
    OSC0 clock lock lost. Do something.
}
if (mcgFlags & kMCG_Pll0LostFlag)
{
    PLL0 clock lock lost. Do something.
}
```

Returns

Logical OR value of the enumeration _mcg_status_flags_t.

void CLOCK_ClearStatusFlags(uint32_t mask)

Clears the MCG status flags.

This function clears the MCG clock lock lost status. The parameter is a logical OR value of the flags to clear. See the enumeration _mcg_status_flags_t.

Example:

```
To clear the clock lost lock status flags of OSC0 and PLL0.

CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Parameters

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

static inline void OSC_SetExtRefClkConfig(OSC_Type *base, *oscer_config_t* const *config)

Configures the OSC external reference clock (OSCERCLK).

This function configures the OSC external reference clock (OSCERCLK). This is an example to enable the OSCERCLK in normal and stop modes and also set the output divider to 1:

```
oscer_config_t config =
{
    .enableMode = kOSC_ErClkEnable | kOSC_ErClkEnableInStop,
    .erclkDiv   = 1U,
};

OSC_SetExtRefClkConfig(OSC, &config);
```

Parameters

- base – OSC peripheral address.

- config – Pointer to the configuration structure.

static inline void OSC_SetCapLoad(OSC_Type *base, uint8_t capLoad)

Sets the capacitor load configuration for the oscillator.

This function sets the specified capacitors configuration for the oscillator. This should be done in the early system level initialization function call based on the system configuration.

Example:

```
To enable only 2 pF and 8 pF capacitor load, please use like this.
OSC_SetCapLoad(OSC, kOSC_Cap2P | kOSC_Cap8P);
```

Parameters

- base – OSC peripheral address.

- capLoad – OR'ed value for the capacitor load option, see _osc_cap_load.

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_SetXtal32Freq(uint32_t freq)

Sets the XTAL32/RTC_CLKIN frequency based on board settings.

Parameters

- freq – The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.

void CLOCK_SetSlowIrcFreq(uint32_t freq)

Set the Slow IRC frequency based on the trimmed value.

Parameters

- freq – The Slow IRC frequency input clock frequency in Hz.

void CLOCK_SetFastIrcFreq(uint32_t freq)

Set the Fast IRC frequency based on the trimmed value.

Parameters

- freq – The Fast IRC frequency input clock frequency in Hz.

status_t CLOCK_TrimInternalRefClk(uint32_t extFreq, uint32_t desireFreq, uint32_t *actualFreq, *mcg_atm_select_t* atms)

Auto trims the internal reference clock.

This function trims the internal reference clock by using the external clock. If successful, it returns the kStatus_Success and the frequency after trimming is received in the parameter actualFreq. If an error occurs, the error code is returned.

Parameters

- extFreq – External clock frequency, which should be a bus clock.
- desireFreq – Frequency to trim to.
- actualFreq – Actual frequency after trimming.
- atms – Trim fast or slow internal reference clock.

Return values

- kStatus_Success – ATM success.
- kStatus_MCG_AtmBusClockInvalid – The bus clock is not in allowed range for the ATM.
- kStatus_MCG_AtmDesiredFreqInvalid – MCGIRCLK could not be trimmed to the desired frequency.
- kStatus_MCG_AtmIrcUsed – Could not trim because MCGIRCLK is used as a bus clock source.
- kStatus_MCG_AtmHardwareFail – Hardware fails while trimming.

mcg_mode_t CLOCK_GetMode(**void**)

Gets the current MCG mode.

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

Returns

Current MCG mode or error code; See mcg_mode_t.

status_t CLOCK_SetFeiMode(*mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*fllStableDelay)(void))

Sets the MCG to FEI mode.

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

Note: If dmx32 is set to kMCG_Dmx32Fine, the slow IRC must not be trimmed to a frequency above 32768 Hz.

Parameters

- dmx32 – DMX32 in FEI mode.
- drs – The DCO range selection.
- fllStableDelay – Delay function to ensure that the FLL is stable. Passing NULL does not cause a delay.

Return values

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

status_t CLOCK_SetFeeMode(uint8_t frdiv, *mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void
(*fllStableDelay)(void))

Sets the MCG to FEE mode.

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

Parameters

- frdiv – FLL reference clock divider setting, FRDIV.

- dmx32 – DMX32 in FEE mode.

- drs – The DCO range selection.

- fllStableDelay – Delay function to make sure FLL is stable. Passing NULL does not cause a delay.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetFbiMode(*mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void (*fllStableDelay)(void))

Sets the MCG to FBI mode.

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

Note: If dmx32 is set to kMCG_Dmx32Fine, the slow IRC must not be trimmed to frequency above 32768 Hz.

Parameters

- dmx32 – DMX32 in FBI mode.

- drs – The DCO range selection.

- fllStableDelay – Delay function to make sure FLL is stable. If the FLL is not used in FBI mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetFbeMode(uint8_t frdiv, *mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void
(*fllStableDelay)(void))

Sets the MCG to FBE mode.

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

Parameters

- frdiv – FLL reference clock divider setting, FRDIV.

- dmx32 – DMX32 in FBE mode.

- drs – The DCO range selection.

- fllStableDelay – Delay function to make sure FLL is stable. If the FLL is not used in FBE mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

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

status_t CLOCK_SetBlpiMode(**void**)

Sets the MCG to BLPI mode.

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

Return values

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

status_t CLOCK_SetBlpeMode(**void**)

Sets the MCG to BLPE mode.

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

Return values

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

status_t CLOCK_SetPbeMode(*mcg_pll_clk_select_t* pllcs, *mcg_pll_config_t* const *config)

Sets the MCG to PBE mode.

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

Note:

a. The parameter pllcs selects the PLL. For platforms with only one PLL, the parameter pllcs is kept for interface compatibility.

b. The parameter config is the PLL configuration structure. On some platforms, it is possible to choose the external PLL directly, which renders the configuration structure not necessary. In this case, pass in NULL. For example: CLOCK_SetPbeMode(kMCG_OscselOsc, kMCG_PllClkSelExtPll, NULL);

Parameters

- pllcs – The PLL selection, PLLCS.
- config – Pointer to the PLL configuration.

Return values

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

status_t CLOCK_SetPeeMode(**void**)

Sets the MCG to PEE mode.

This function sets the MCG to PEE mode.

Note: This function only changes the CLKS to use the PLL/FLL output. If the PRDIV/VDIV are different than in the PBE mode, set them up in PBE mode and wait. When the clock is stable, switch to PEE mode.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetPbiMode(**void**)

Sets the MCG to PBI mode.

This function sets the MCG to PBI mode.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetPeiMode(**void**)

Sets the MCG to PEI mode.

This function sets the MCG to PEI mode.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_ExternalModeToFbeModeQuick(**void**)

Switches the MCG to FBE mode from the external mode.

This function switches the MCG from external modes (PEE/PBE/BLPE/FEE) to the FBE mode quickly. The external clock is used as the system clock source and PLL is disabled. However, the FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEE mode to FEI mode:

```
CLOCK_ExternalModeToFbeModeQuick();
CLOCK_SetFeiMode(...);
```

Return values

- kStatus_Success – Switched successfully.

- kStatus_MCG_ModeInvalid – If the current mode is not an external mode, do not call this function.

status_t CLOCK_InternalModeToFbiModeQuick(**void**)

Switches the MCG to FBI mode from internal modes.

This function switches the MCG from internal modes (PEI/PBI/BLPI/FEI) to the FBI mode quickly. The MCGIRCLK is used as the system clock source and PLL is disabled. However, FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEI mode to FEE mode:

```
CLOCK_InternalModeToFbiModeQuick();
CLOCK_SetFeeMode(...);
```

Return values

- kStatus_Success – Switched successfully.

- kStatus_MCG_ModeInvalid – If the current mode is not an internal mode, do not call this function.

status_t CLOCK_BootToFeiMode(*mcg_dmx32_t* dmx32, *mcg_drs_t* drs, void
(*fllStableDelay)(void))

Sets the MCG to FEI mode during system boot up.

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

Note: If dmx32 is set to kMCG_Dmx32Fine, the slow IRC must not be trimmed to frequency above 32768 Hz.

Parameters

- dmx32 – DMX32 in FEI mode.
- drs – The DCO range selection.
- fllStableDelay – Delay function to ensure that the FLL is stable.

Return values

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

status_t CLOCK_BootToFeeMode(*mcg_oscsel_t* oscsel, uint8_t frdiv, *mcg_dmx32_t* dmx32,
mcg_drs_t drs, void (*fllStableDelay)(void))

Sets the MCG to FEE mode during system bootup.

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

Parameters

- oscsel – OSC clock select, OSCSEL.
- frdiv – FLL reference clock divider setting, FRDIV.
- dmx32 – DMX32 in FEE mode.
- drs – The DCO range selection.
- fllStableDelay – Delay function to ensure that the FLL is stable.

Return values

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

status_t CLOCK_BootToBlpiMode(uint8_t fcrdiv, *mcg_irc_mode_t* ircs, uint8_t ircEnableMode)

Sets the MCG to BLPI mode during system boot up.

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

Parameters

- fcrdiv – Fast IRC divider, FCRDIV.
- ircs – The internal reference clock to select, IRCS.
- ircEnableMode – The MCGIRCLK enable mode, OR'ed value of the enumeration _mcg_irclk_enable_mode.

Return values

- kStatus_MCG_SourceUsed – Could not change MCGIRCLK setting.
- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToBlpeMode(*mcg_oscsel_t* oscsel)

Sets the MCG to BLPE mode during system boot up.

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

Parameters

- oscsel – OSC clock select, MCG_C7[OSCSEL].

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToPeeMode(*mcg_oscsel_t* oscsel, *mcg_pll_clk_select_t* pllcs,
mcg_pll_config_t const *config)

Sets the MCG to PEE mode during system boot up.

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

Parameters

- oscsel – OSC clock select, MCG_C7[OSCSEL].

- pllcs – The PLL selection, PLLCS.

- config – Pointer to the PLL configuration.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_BootToPeiMode(**void**)

Sets the MCG to PEI mode during system boot up.

This function sets the MCG to PEI mode from the reset mode. It can be used to set up the MCG during system boot up.

Return values

- kStatus_MCG_ModeUnreachable – Could not switch to the target mode.

- kStatus_Success – Switched to the target mode successfully.

status_t CLOCK_SetMcgConfig(*mcg_config_t* const *config)

Sets the MCG to a target mode.

This function sets MCG 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 MCG mode configuration structure.

Returns

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

uint8_t er32kSrc
  ERCLK32K source selection.

uint32_t clkdiv1
  SIM_CLKDIV1.

uint8_t enableMode
  OSCERCLK enable mode. OR'ed value of _oscer_enable_mode.

uint32_t freq
  External clock frequency.

uint8_t capLoad
  Capacitor load setting.

osc_mode_t workMode
  OSC work mode setting.

oscer_config_t oscerConfig
  Configuration for OSCERCLK.

uint8_t enableMode
  Enable mode. OR'ed value of enumeration _mcg_pll_enable_mode.

mcg_pll_ref_src_t refSrc
  PLL reference clock source.

uint8_t frdiv
  FLL reference clock divider.

mcg_mode_t mcgMode
  MCG mode.

uint8_t irclkEnableMode
  MCGIRCLK enable mode.

mcg_irc_mode_t ircs
  Source, MCG_C2[IRCS].

uint8_t fcrdiv
  Divider, MCG_SC[FCRDIV].

uint8_t frdiv
  Divider MCG_C1[FRDIV].

mcg_drs_t drs
  DCO range MCG_C4[DRST_DRS].

mcg_dmx32_t dmx32
  MCG_C4[DMX32].

mcg_oscsel_t oscsel
  OSC select MCG_C7[OSCSEL].

mcg_pll_config_t pll0Config
  MCGPLL0CLK configuration.

MCG_CONFIG_CHECK_PARAM
  Configures whether to check a parameter in a function.

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

  a. MCGIRCLK settings, such as the source, divider, and the trim value should not change when MCGIRCLK is used as a system clock source.

b. MCG_C7[OSCSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BLPE/PBE modes.

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

MCG functions check the parameter and MCG 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 MCG_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 __oscer_config

> *#include <fsl_clock.h>* OSC configuration for OSCERCLK.

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.

struct __mcg_pll_config

> *#include <fsl_clock.h>* MCG PLL configuration.

struct __mcg_config

> *#include <fsl_clock.h>* MCG mode change configuration structure.

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

> a. frdiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by frdiv is in the 31.25 kHz to 39.0625 kHz range.

> b. The PLL reference clock divider PRDIV: PLL reference clock frequency after PRDIV should be in the FSL_FEATURE_MCG_PLL_REF_MIN to FSL_FEATURE_MCG_PLL_REF_MAX range.

2.4 CMP: Analog Comparator Driver

void CMP_Init(CMP_Type *base, const *cmp_config_t* *config)

> Initializes the CMP.

> This function initializes the CMP module. The operations included are as follows.

> - Enabling the clock for CMP module.
> - Configuring the comparator.

- Enabling the CMP module. Note that for some devices, multiple CMP instances share the same clock gate. In this case, to enable the clock for any instance enables all CMPs. See the appropriate MCU reference manual for the clock assignment of the CMP.

> **Parameters**
>
>> - base – CMP peripheral base address.
>>
>> - config – Pointer to the configuration structure.

void CMP_Deinit(**CMP_Type *base**)

> De-initializes the CMP module.
>
> This function de-initializes the CMP module. The operations included are as follows.

- Disabling the CMP module.

- Disabling the clock for CMP module.

> This function disables the clock for the CMP. Note that for some devices, multiple CMP instances share the same clock gate. In this case, before disabling the clock for the CMP, ensure that all the CMP instances are not used.

> **Parameters**
>
>> - base – CMP peripheral base address.

static inline void CMP_Enable(**CMP_Type *base, bool enable**)

> Enables/disables the CMP module.

> **Parameters**
>
>> - base – CMP peripheral base address.
>>
>> - enable – Enables or disables the module.

void CMP_GetDefaultConfig(*cmp_config_t* ***config**)

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

```
config->enableCmp          = true;
config->hysteresisMode     = kCMP_HysteresisLevel0;
config->enableHighSpeed    = false;
config->enableInvertOutput = false;
config->useUnfilteredOutput = false;
config->enablePinOut       = false;
config->enableTriggerMode  = false;
```

> **Parameters**
>
>> - config – Pointer to the configuration structure.

void CMP_SetInputChannels(**CMP_Type *base, uint8_t positiveChannel, uint8_t negativeChannel**)

> Sets the input channels for the comparator.
>
> This function sets the input channels for the comparator. Note that two input channels cannot be set the same way in the application. When the user selects the same input from the analog mux to the positive and negative port, the comparator is disabled automatically.

> **Parameters**
>
>> - base – CMP peripheral base address.
>>
>> - positiveChannel – Positive side input channel number. Available range is 0-7.

- negativeChannel – Negative side input channel number. Available range is 0-7.

void CMP_EnableDMA(CMP_Type *base, bool enable)

Enables/disables the DMA request for rising/falling events.

This function enables/disables the DMA request for rising/falling events. Either event triggers the generation of the DMA request from CMP if the DMA feature is enabled. Both events are ignored for generating the DMA request from the CMP if the DMA is disabled.

Parameters

- base – CMP peripheral base address.

- enable – Enables or disables the feature.

static inline void CMP_EnableWindowMode(CMP_Type *base, bool enable)

Enables/disables the window mode.

Parameters

- base – CMP peripheral base address.

- enable – Enables or disables the feature.

static inline void CMP_EnablePassThroughMode(CMP_Type *base, bool enable)

Enables/disables the pass through mode.

Parameters

- base – CMP peripheral base address.

- enable – Enables or disables the feature.

void CMP_SetFilterConfig(CMP_Type *base, const *cmp_filter_config_t* *config)

Configures the filter.

Parameters

- base – CMP peripheral base address.

- config – Pointer to the configuration structure.

void CMP_SetDACConfig(CMP_Type *base, const *cmp_dac_config_t* *config)

Configures the internal DAC.

Parameters

- base – CMP peripheral base address.

- config – Pointer to the configuration structure. "NULL" disables the feature.

void CMP_EnableInterrupts(CMP_Type *base, uint32_t mask)

Enables the interrupts.

Parameters

- base – CMP peripheral base address.

- mask – Mask value for interrupts. See "_cmp_interrupt_enable".

void CMP_DisableInterrupts(CMP_Type *base, uint32_t mask)

Disables the interrupts.

Parameters

- base – CMP peripheral base address.

- mask – Mask value for interrupts. See "_cmp_interrupt_enable".

uint32_t CMP_GetStatusFlags(CMP_Type *base)

Gets the status flags.

> **Parameters**
>
> > • base – CMP peripheral base address.
>
> **Returns**
> > Mask value for the asserted flags. See "_cmp_status_flags".

void CMP_ClearStatusFlags(CMP_Type *base, uint32_t mask)

Clears the status flags.

> **Parameters**
>
> > • base – CMP peripheral base address.
> >
> > • mask – Mask value for the flags. See "_cmp_status_flags".

FSL_CMP_DRIVER_VERSION

> CMP driver version 2.0.3.

enum _cmp_interrupt_enable

> Interrupt enable/disable mask.
>
> *Values:*
>
> enumerator kCMP_OutputRisingInterruptEnable
> > Comparator interrupt enable rising.
>
> enumerator kCMP_OutputFallingInterruptEnable
> > Comparator interrupt enable falling.

enum _cmp_status_flags

> Status flags' mask.
>
> *Values:*
>
> enumerator kCMP_OutputRisingEventFlag
> > Rising-edge on the comparison output has occurred.
>
> enumerator kCMP_OutputFallingEventFlag
> > Falling-edge on the comparison output has occurred.
>
> enumerator kCMP_OutputAssertEventFlag
> > Return the current value of the analog comparator output.

enum _cmp_hysteresis_mode

> CMP Hysteresis mode.
>
> *Values:*
>
> enumerator kCMP_HysteresisLevel0
> > Hysteresis level 0.
>
> enumerator kCMP_HysteresisLevel1
> > Hysteresis level 1.
>
> enumerator kCMP_HysteresisLevel2
> > Hysteresis level 2.
>
> enumerator kCMP_HysteresisLevel3
> > Hysteresis level 3.

enum __cmp_reference_voltage_source
    CMP Voltage Reference source.

    Values:

    enumerator kCMP_VrefSourceVin1
        Vin1 is selected as a resistor ladder network supply reference Vin.

    enumerator kCMP_VrefSourceVin2
        Vin2 is selected as a resistor ladder network supply reference Vin.

typedef enum *_cmp_hysteresis_mode* cmp_hysteresis_mode_t
    CMP Hysteresis mode.

typedef enum *_cmp_reference_voltage_source* cmp_reference_voltage_source_t
    CMP Voltage Reference source.

typedef struct *_cmp_config* cmp_config_t
    Configures the comparator.

typedef struct *_cmp_filter_config* cmp_filter_config_t
    Configures the filter.

typedef struct *_cmp_dac_config* cmp_dac_config_t
    Configures the internal DAC.

struct __cmp_config
    #include <fsl_cmp.h> Configures the comparator.

Public Members

    bool enableCmp
        Enable the CMP module.

    cmp_hysteresis_mode_t hysteresisMode
        CMP Hysteresis mode.

    bool enableHighSpeed
        Enable High-speed (HS) comparison mode.

    bool enableInvertOutput
        Enable the inverted comparator output.

    bool useUnfilteredOutput
        Set the compare output(COUT) to equal COUTA(true) or COUT(false).

    bool enablePinOut
        The comparator output is available on the associated pin.

    bool enableTriggerMode
        Enable the trigger mode.

struct __cmp_filter_config
    #include <fsl_cmp.h> Configures the filter.

Public Members

    bool enableSample
        Using the external SAMPLE as a sampling clock input or using a divided bus clock.

---

uint8_t filterCount

    Filter Sample Count. Available range is 1-7; 0 disables the filter.

uint8_t filterPeriod

    Filter Sample Period. The divider to the bus clock. Available range is 0-255.

struct __cmp__dac__config

    #include <fsl_cmp.h> Configures the internal DAC.

Public Members

cmp_reference_voltage_source_t referenceVoltageSource

    Supply voltage reference source.

uint8_t DACValue

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

2.5 CRC: Cyclic Redundancy Check Driver

FSL__CRC__DRIVER__VERSION

    CRC driver version. Version 2.0.4.

    Current version: 2.0.4

    Change log:

- 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 defeault.

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()

2.6   DMA: Direct Memory Access Controller Driver

void DMA_Init(DMA_Type *base)

Initializes the DMA peripheral.

This function ungates the DMA clock.

Parameters

- base – DMA peripheral base address.

void DMA_Deinit(DMA_Type *base)

Deinitializes the DMA peripheral.

This function gates the DMA clock.

Parameters

- base – DMA peripheral base address.

void DMA_ResetChannel(DMA_Type *base, uint32_t channel)

Resets the DMA channel.

Sets all register values to reset values and enables the cycle steal and auto stop channel request features.

Parameters

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

void DMA_SetTransferConfig(DMA_Type *base, uint32_t channel, const *dma_transfer_config_t* *config)

Configures the DMA transfer attribute.

This function configures the transfer attribute including the source address, destination address, transfer size, and so on. This example shows how to set up the dma_transfer_config_t parameters and how to call the DMA_ConfigBasicTransfer function.

```
dma_transfer_config_t transferConfig;
memset(&transferConfig, 0, sizeof(transferConfig));
transferConfig.srcAddr = (uint32_t)srcAddr;
transferConfig.destAddr = (uint32_t)destAddr;
transferConfig.enbaleSrcIncrement = true;
transferConfig.enableDestIncrement = true;
transferConfig.srcSize = kDMA_Transfersize32bits;
transferConfig.destSize = kDMA_Transfersize32bits;
transferConfig.transferSize = sizeof(uint32_t) * BUFF_LENGTH;
DMA_SetTransferConfig(DMA0, 0, &transferConfig);
```

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- config – Pointer to the DMA transfer configuration structure.

void DMA_SetChannelLinkConfig(DMA_Type *base, uint32_t channel, const *dma_channel_link_config_t* *config)

Configures the DMA channel link feature.

This function allows DMA channels to have their transfers linked. The current DMA channel triggers a DMA request to the linked channels (LCH1 or LCH2) depending on the channel link type. Perform a link to channel LCH1 after each cycle-steal

---

transfer followed by a link to LCH2 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AndChannel2. Perform a link to LCH1 after each cycle-steal transfer if the type is kDMA_ChannelLinkChannel1. Perform a link to LCH1 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AfterBCR0.

> **Parameters**
>
> - base – DMA peripheral base address.
> - channel – DMA channel number.
> - config – Pointer to the channel link configuration structure.

static inline void DMA_SetSourceAddress(DMA_Type *base, uint32_t channel, uint32_t srcAddr)

Sets the DMA source address for the DMA transfer.

> **Parameters**
>
> - base – DMA peripheral base address.
> - channel – DMA channel number.
> - srcAddr – DMA source address.

static inline void DMA_SetDestinationAddress(DMA_Type *base, uint32_t channel, uint32_t destAddr)

Sets the DMA destination address for the DMA transfer.

> **Parameters**
>
> - base – DMA peripheral base address.
> - channel – DMA channel number.
> - destAddr – DMA destination address.

static inline void DMA_SetTransferSize(DMA_Type *base, uint32_t channel, uint32_t size)

Sets the DMA transfer size for the DMA transfer.

> **Parameters**
>
> - base – DMA peripheral base address.
> - channel – DMA channel number.
> - size – The number of bytes to be transferred.

void DMA_SetModulo(DMA_Type *base, uint32_t channel, *dma_modulo_t* srcModulo, *dma_modulo_t* destModulo)

Sets the DMA modulo for the DMA transfer.

This function defines a specific address range specified to be the value after (SAR + SSIZE)/(DAR + DSIZE) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

> **Parameters**
>
> - base – DMA peripheral base address.
> - channel – DMA channel number.
> - srcModulo – source address modulo.
> - destModulo – destination address modulo.

static inline void DMA_EnableCycleSteal(DMA_Type *base, uint32_t channel, bool enable)

Enables the DMA cycle steal for the DMA transfer.

If the cycle steal feature is enabled (true), the DMA controller forces a single read/write transfer per request, or it continuously makes read/write transfers until the BCR decrements to 0.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

static inline void DMA_EnableAutoAlign(DMA_Type *base, uint32_t channel, bool enable)

Enables the DMA auto align for the DMA transfer.

If the auto align feature is enabled (true), the appropriate address register increments regardless of DINC or SINC.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

static inline void DMA_EnableAsyncRequest(DMA_Type *base, uint32_t channel, bool enable)

Enables the DMA async request for the DMA transfer.

If the async request feature is enabled (true), the DMA supports asynchronous DREQs while the MCU is in stop mode.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- enable – The command for enable (true) or disable (false).

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

Enables an interrupt for the DMA transfer.

Parameters

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

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

Disables an interrupt for the DMA transfer.

Parameters

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

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

Enables the DMA hardware channel request.

Parameters

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

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

Disables the DMA hardware channel request.

Parameters

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

---

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

> Starts the DMA transfer with a software trigger.

> This function starts only one read/write iteration.

> > **Parameters**

> > > • base – DMA peripheral base address.

> > > • channel – The DMA channel number.

static inline void DMA_EnableAutoStopRequest(DMA_Type *base, uint32_t channel, bool enable)

> Starts the DMA enable/disable auto disable request.

> > **Parameters**

> > > • base – DMA peripheral base address.

> > > • channel – The DMA channel number.

> > > • enable – true is enable, false is disable.

static inline uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)

> Gets the remaining bytes of the current DMA transfer.

> > **Parameters**

> > > • base – DMA peripheral base address.

> > > • channel – DMA channel number.

> > **Returns**

> > > The number of bytes which have not been transferred yet.

static inline uint32_t DMA_GetChannelStatusFlags(DMA_Type *base, uint32_t channel)

> Gets the DMA channel status flags.

> > **Parameters**

> > > • base – DMA peripheral base address.

> > > • channel – DMA channel number.

> > **Returns**

> > > The mask of the channel status. Use the _dma_channel_status_flags type to decode the return 32 bit variables.

static inline void DMA_ClearChannelStatusFlags(DMA_Type *base, uint32_t channel, uint32_t mask)

> Clears the DMA channel status flags.

> > **Parameters**

> > > • base – DMA peripheral base address.

> > > • channel – DMA channel number.

> > > • mask – The mask of the channel status to be cleared. Use the defined _dma_channel_status_flags type.

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

> Creates the DMA handle.

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

> > **Parameters**

> > > • handle – DMA handle pointer. The DMA handle stores callback function and parameters.

- base – DMA peripheral base address.

- channel – DMA channel number.

void DMA__SetCallback(*dma_handle_t* *handle, *dma_callback* callback, void *userData)

Sets the DMA callback function.

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

Parameters

- handle – DMA handle pointer.

- callback – DMA callback function pointer.

- userData – Parameter for callback function. If it is not needed, just set to NULL.

void DMA__PrepareTransferConfig(*dma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, *dma_addr_increment_t* srcIncrement, *dma_addr_increment_t* destIncrement)

Prepares the DMA transfer configuration structure.

This function prepares the transfer configuration structure according to the user input. The difference between this function and DMA_PrepareTransfer is that this function expose the address increment parameter to application, but in DMA_PrepareTransfer, only parts of the address increment option can be selected by dma_transfer_type_t.

Parameters

- config – Pointer to the user configuration structure of type dma_transfer_config_t.

- srcAddr – DMA transfer source address.

- srcWidth – DMA transfer source address width (byte).

- destAddr – DMA transfer destination address.

- destWidth – DMA transfer destination address width (byte).

- transferBytes – DMA transfer bytes to be transferred.

- srcIncrement – source address increment type.

- destIncrement – dest address increment type.

void DMA__PrepareTransfer(*dma_transfer_config_t* *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, *dma_transfer_type_t* type)

Prepares the DMA transfer configuration structure.

This function prepares the transfer configuration structure according to the user input.

Parameters

- config – Pointer to the user configuration structure of type dma_transfer_config_t.

- srcAddr – DMA transfer source address.

- srcWidth – DMA transfer source address width (byte).

- destAddr – DMA transfer destination address.

- destWidth – DMA transfer destination address width (byte).

- transferBytes – DMA transfer bytes to be transferred.

- type – DMA transfer type.

status_t DMA_SubmitTransfer(*dma_handle_t* *handle, const *dma_transfer_config_t* *config, uint32_t options)

Submits the DMA transfer request.

This function submits the DMA transfer request according to the transfer configuration structure.

Note: This function can't process multi transfer request.

Parameters

- handle – DMA handle pointer.

- config – Pointer to DMA transfer configuration structure.

- options – Additional configurations for transfer. Use the defined dma_transfer_options_t type.

Return values

- kStatus_DMA_Success – It indicates that the DMA submit transfer request succeeded.

- kStatus_DMA_Busy – It indicates that the DMA is busy. Submit transfer request is not allowed.

static inline void DMA_StartTransfer(*dma_handle_t* *handle)

DMA starts a transfer.

This function enables the channel request. Call this function after submitting a transfer request.

Parameters

- handle – DMA handle pointer.

Return values

- kStatus_DMA_Success – It indicates that the DMA start transfer succeed.

- kStatus_DMA_Busy – It indicates that the DMA has started a transfer.

static inline void DMA_StopTransfer(*dma_handle_t* *handle)

DMA stops a transfer.

This function disables the channel request to stop a DMA transfer. The transfer can be resumed by calling the DMA_StartTransfer.

Parameters

- handle – DMA handle pointer.

void DMA_AbortTransfer(*dma_handle_t* *handle)

DMA aborts a transfer.

This function disables the channel request and clears all status bits. Submit another transfer after calling this API.

Parameters

- handle – DMA handle pointer.

void DMA_HandleIRQ(*dma_handle_t* *handle)

> DMA IRQ handler for current transfer complete.

> This function clears the channel interrupt flag and calls the callback function if it is not NULL.

> > **Parameters**

> > > • handle – DMA handle pointer.

FSL_DMA_DRIVER_VERSION

> DMA driver version 2.1.2.

> _dma_channel_status_flags status flag for the DMA driver.

> *Values:*

> enumerator kDMA_TransactionsBCRFlag

> > Contains the number of bytes yet to be transferred for a given block

> enumerator kDMA_TransactionsDoneFlag

> > Transactions Done

> enumerator kDMA_TransactionsBusyFlag

> > Transactions Busy

> enumerator kDMA_TransactionsRequestFlag

> > Transactions Request

> enumerator kDMA_BusErrorOnDestinationFlag

> > Bus Error on Destination

> enumerator kDMA_BusErrorOnSourceFlag

> > Bus Error on Source

> enumerator kDMA_ConfigurationErrorFlag

> > Configuration Error

enum _dma_transfer_size

> DMA transfer size type.

> *Values:*

> enumerator kDMA_Transfersize32bits

> > 32 bits are transferred for every read/write

> enumerator kDMA_Transfersize8bits

> > 8 bits are transferred for every read/write

> enumerator kDMA_Transfersize16bits

> > 16b its are transferred for every read/write

enum _dma_modulo

> Configuration type for the DMA modulo.

> *Values:*

> enumerator kDMA_ModuloDisable

> > Buffer disabled

> enumerator kDMA_Modulo16Bytes

> > Circular buffer size is 16 bytes.

enumerator kDMA_Modulo32Bytes
    Circular buffer size is 32 bytes.

enumerator kDMA_Modulo64Bytes
    Circular buffer size is 64 bytes.

enumerator kDMA_Modulo128Bytes
    Circular buffer size is 128 bytes.

enumerator kDMA_Modulo256Bytes
    Circular buffer size is 256 bytes.

enumerator kDMA_Modulo512Bytes
    Circular buffer size is 512 bytes.

enumerator kDMA_Modulo1KBytes
    Circular buffer size is 1 KB.

enumerator kDMA_Modulo2KBytes
    Circular buffer size is 2 KB.

enumerator kDMA_Modulo4KBytes
    Circular buffer size is 4 KB.

enumerator kDMA_Modulo8KBytes
    Circular buffer size is 8 KB.

enumerator kDMA_Modulo16KBytes
    Circular buffer size is 16 KB.

enumerator kDMA_Modulo32KBytes
    Circular buffer size is 32 KB.

enumerator kDMA_Modulo64KBytes
    Circular buffer size is 64 KB.

enumerator kDMA_Modulo128KBytes
    Circular buffer size is 128 KB.

enumerator kDMA_Modulo256KBytes
    Circular buffer size is 256 KB.

enum __dma_channel_link_type
    DMA channel link type.

    Values:

    enumerator kDMA_ChannelLinkDisable
        No channel link.

    enumerator kDMA_ChannelLinkChannel1AndChannel2
        Perform a link to channel LCH1 after each cycle-steal transfer. followed by a link to
        LCH2 after the BCR decrements to 0.

    enumerator kDMA_ChannelLinkChannel1
        Perform a link to LCH1 after each cycle-steal transfer.

    enumerator kDMA_ChannelLinkChannel1AfterBCR0
        Perform a link to LCH1 after the BCR decrements.

enum __dma_transfer_type
    DMA transfer type.

    Values:

enumerator kDMA_MemoryToMemory
Memory to Memory transfer.

enumerator kDMA_PeripheralToMemory
Peripheral to Memory transfer.

enumerator kDMA_MemoryToPeripheral
Memory to Peripheral transfer.

enum __dma_transfer_options
DMA transfer options.

Values:

enumerator kDMA_NoOptions
Transfer without options.

enumerator kDMA_EnableInterrupt
Enable interrupt while transfer complete.

enum __dma_addr_increment
dma addre increment type

Values:

enumerator kDMA_AddrNoIncrement
Transfer address not increment.

enumerator kDMA_AddrIncrementPerTransferWidth
Transfer address increment per transfer width

_dma_transfer_status DMA transfer status

Values:

enumerator kStatus_DMA_Busy
DMA is busy.

typedef enum *_dma_transfer_size* dma_transfer_size_t
DMA transfer size type.

typedef enum *_dma_modulo* dma_modulo_t
Configuration type for the DMA modulo.

typedef enum *_dma_channel_link_type* dma_channel_link_type_t
DMA channel link type.

typedef enum *_dma_transfer_type* dma_transfer_type_t
DMA transfer type.

typedef enum *_dma_transfer_options* dma_transfer_options_t
DMA transfer options.

typedef enum *_dma_addr_increment* dma_addr_increment_t
dma addre increment type

typedef struct *_dma_transfer_config* dma_transfer_config_t
DMA transfer configuration structure.

typedef struct *_dma_channel_link_config* dma_channel_link_config_t
DMA transfer configuration structure.

typedef void (*dma_callback)(struct *_dma_handle* *handle, void *userData)
> Callback function prototype for the DMA driver.

typedef struct *_dma_handle* dma_handle_t
> DMA DMA handle structure.

struct __dma__transfer__config
> *#include <fsl_dma.h>* DMA transfer configuration structure.

Public Members

uint32_t srcAddr
> DMA transfer source address.

uint32_t destAddr
> DMA destination address.

bool enableSrcIncrement
> Source address increase after each transfer.

dma_transfer_size_t srcSize
> Source transfer size unit.

bool enableDestIncrement
> Destination address increase after each transfer.

dma_transfer_size_t destSize
> Destination transfer unit.

uint32_t transferSize
> The number of bytes to be transferred.

struct __dma__channel__link__config
> *#include <fsl_dma.h>* DMA transfer configuration structure.

Public Members

dma_channel_link_type_t linkType
> Channel link type.

uint32_t channel1
> The index of channel 1.

uint32_t channel2
> The index of channel 2.

struct __dma__handle
> *#include <fsl_dma.h>* DMA DMA handle structure.

Public Members

DMA_Type *base
> DMA peripheral address.

uint8_t channel
> DMA channel used.

dma_callback callback
> DMA callback function.

void *userData
> Callback parameter.

2.7 DMAMUX: Direct Memory Access Multiplexer Driver

void DMAMUX_Init(DMAMUX_Type *base)
> Initializes the DMAMUX peripheral.
>
> This function ungates the DMAMUX clock.
>
> > **Parameters**
> >
> > > • base – DMAMUX peripheral base address.

void DMAMUX_Deinit(DMAMUX_Type *base)
> Deinitializes the DMAMUX peripheral.
>
> This function gates the DMAMUX clock.
>
> > **Parameters**
> >
> > > • base – DMAMUX peripheral base address.

static inline void DMAMUX_EnableChannel(DMAMUX_Type *base, uint32_t channel)
> Enables the DMAMUX channel.
>
> This function enables the DMAMUX channel.
>
> > **Parameters**
> >
> > > • base – DMAMUX peripheral base address.
> > >
> > > • channel – DMAMUX channel number.

static inline void DMAMUX_DisableChannel(DMAMUX_Type *base, uint32_t channel)
> Disables the DMAMUX channel.
>
> This function disables the DMAMUX channel.
>
> ---
>
> **Note:** The user must disable the DMAMUX channel before configuring it.
>
> ---
>
> > **Parameters**
> >
> > > • base – DMAMUX peripheral base address.
> > >
> > > • channel – DMAMUX channel number.

static inline void DMAMUX_SetSource(DMAMUX_Type *base, uint32_t channel, int32_t source)
> Configures the DMAMUX channel source.
>
> > **Parameters**
> >
> > > • base – DMAMUX peripheral base address.
> > >
> > > • channel – DMAMUX channel number.
> > >
> > > • source – Channel source, which is used to trigger the DMA transfer.User
> > >   need to use the dma_request_source_t type as the input parameter.

static inline void DMAMUX_EnablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)
> Enables the DMAMUX period trigger.
>
> This function enables the DMAMUX period trigger feature.
>
> > **Parameters**

- base – DMAMUX peripheral base address.

- channel – DMAMUX channel number.

static inline void DMAMUX_DisablePeriodTrigger(DMAMUX_Type *base, uint32_t channel)

Disables the DMAMUX period trigger.

This function disables the DMAMUX period trigger.

Parameters

- base – DMAMUX peripheral base address.

- channel – DMAMUX channel number.

FSL_DMAMUX_DRIVER_VERSION

DMAMUX driver version 2.1.1.

2.8 EWM: External Watchdog Monitor Driver

void EWM_Init(EWM_Type *base, const *ewm_config_t* *config)

Initializes the EWM peripheral.

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.compareHighValue = 0xAAU;
EWM_Init(ewm_base,&config);
```

Parameters

- base – EWM peripheral base address

- config – The configuration of the EWM

void EWM_Deinit(EWM_Type *base)

Deinitializes the EWM peripheral.

This function is used to shut down the EWM.

Parameters

- base – EWM peripheral base address

void EWM_GetDefaultConfig(*ewm_config_t* *config)

Initializes the EWM configuration structure.

This function initializes the EWM configuration structure to default values. The default values are as follows.

```
ewmConfig->enableEwm = true;
ewmConfig->enableEwmInput = false;
ewmConfig->setInputAssertLogic = false;
ewmConfig->enableInterrupt = false;
ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
ewmConfig->prescaler = 0;
ewmConfig->compareLowValue = 0;
ewmConfig->compareHighValue = 0xFEU;
```

See also:

ewm_config_t

Parameters

- config – Pointer to the EWM configuration structure.

static inline void EWM_EnableInterrupts(EWM_Type *base, uint32_t mask)

Enables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address

- mask – The interrupts to enable The parameter can be combination of the following source if defined

  – kEWM_InterruptEnable

static inline void EWM_DisableInterrupts(EWM_Type *base, uint32_t mask)

Disables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address

- mask – The interrupts to disable The parameter can be combination of the following source if defined

  – kEWM_InterruptEnable

static inline uint32_t EWM_GetStatusFlags(EWM_Type *base)

Gets all status flags.

This function gets all status flags.

This is an example for getting the running flag.

```
uint32_t status;
status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
```

See also:

_ewm_status_flags_t

- True: a related status flag has been set.
- False: a related status flag is not set.

Parameters

- base – EWM peripheral base address

Returns

State of the status flag: asserted (true) or not-asserted (false).

void EWM_Refresh(EWM_Type *base)

Services the EWM.

This function resets the EWM counter to zero.

Parameters

- base – EWM peripheral base address

FSL_EWM_DRIVER_VERSION
    EWM driver version 2.0.4.

enum _ewm_lpo_clock_source
    Describes EWM clock source.

    Values:

    enumerator kEWM_LpoClockSource0
        EWM clock sourced from lpo_clk[0]

    enumerator kEWM_LpoClockSource1
        EWM clock sourced from lpo_clk[1]

    enumerator kEWM_LpoClockSource2
        EWM clock sourced from lpo_clk[2]

    enumerator kEWM_LpoClockSource3
        EWM clock sourced from lpo_clk[3]

enum _ewm_interrupt_enable_t
    EWM interrupt configuration structure with default settings all disabled.

    This structure contains the settings for all of EWM interrupt configurations.

    Values:

    enumerator kEWM_InterruptEnable
        Enable the EWM to generate an interrupt

enum _ewm_status_flags_t
    EWM status flags.

    This structure contains the constants for the EWM status flags for use in the EWM functions.

    Values:

    enumerator kEWM_RunningFlag
        Running flag, set when EWM is enabled

typedef enum _ewm_lpo_clock_source ewm_lpo_clock_source_t
    Describes EWM clock source.

typedef struct _ewm_config ewm_config_t
    Data structure for EWM configuration.

    This structure is used to configure the EWM.

struct _ewm_config
    #include <fsl_ewm.h> Data structure for EWM configuration.

    This structure is used to configure the EWM.

Public Members

    bool enableEwm
        Enable EWM module

    bool enableEwmInput
        Enable EWM_in input

    bool setInputAssertLogic
        EWM_in signal assertion state

bool enableInterrupt
    Enable EWM interrupt

ewm_lpo_clock_source_t clockSource
    Clock source select

uint8_t prescaler
    Clock prescaler value

uint8_t compareLowValue
    Compare low-register value

uint8_t compareHighValue
    Compare high-register value

2.9   FGPIO Driver

2.10   C90TFS Flash Driver

2.11   ftfx adapter

2.12   Ftftx CACHE Driver

enum __ftfx__cache__ram__func__constants
    Constants for execute-in-RAM flash function.

    Values:

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

typedef struct _*flash_prefetch_speculation_status* ftfx__prefetch__speculation__status__t
    FTFx prefetch speculation status.

typedef struct _*ftfx_cache_config* ftfx__cache__config__t
    FTFx cache driver state information.

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

status_t FTFx__CACHE__Init(*ftfx_cache_config_t* *config)
    Initializes the global FTFx cache structure members.

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

    Parameters

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

    Return values

    - kStatus__FTFx__Success – API was executed successfully.

    - kStatus__FTFx__InvalidArgument – An invalid argument is provided.

    - kStatus__FTFx__ExecuteInRamFunctionNotReady – Execute-in-RAM function
      is not available.

status_t FTFx_CACHE_ClearCachePrefetchSpeculation(*ftfx_cache_config_t* *config, bool
isPreProcess)

Process the cache/prefetch/speculation to the flash.

Parameters

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

- isPreProcess – The possible option used to control flash cache/prefetch/speculation

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – Invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

status_t FTFx_CACHE_PflashSetPrefetchSpeculation(*ftfx_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidSpeculationOption – An invalid speculation option argument is provided.

status_t FTFx_CACHE_PflashGetPrefetchSpeculation(*ftfx_prefetch_speculation_status_t*
*speculationStatus)

Gets the PFlash prefetch speculation status.

Parameters

- speculationStatus – Speculation status returned by the PFlash IP.

Return values

kStatus_FTFx_Success – API was executed successfully.

struct __flash_prefetch_speculation_status
#include <fsl_ftfx_cache.h> FTFx prefetch speculation status.

Public Members

bool instructionOff
Instruction speculation.

bool dataOff
Data speculation.

union function_bit_operation_ptr_t
#include <fsl_ftfx_cache.h>

Public Members

uint32_t commadAddr

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

struct __ftfx_cache_config

#include <fsl_ftfx_cache.h> FTFx cache 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

uint8_t flashMemoryIndex
    0 - primary flash; 1 - secondary flash

function_bit_operation_ptr_t bitOperFuncAddr
    An buffer point to the flash execute-in-RAM function.

2.13   ftfx controller

FTFx driver status codes.

Values:

enumerator kStatus_FTFx_Success
    API is executed successfully

enumerator kStatus_FTFx_InvalidArgument
    Invalid argument

enumerator kStatus_FTFx_SizeError
    Error size

enumerator kStatus_FTFx_AlignmentError
    Parameter is not aligned with the specified baseline

enumerator kStatus_FTFx_AddressError
    Address is out of range

enumerator kStatus_FTFx_AccessError
    Invalid instruction codes and out-of bound addresses

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

enumerator kStatus_FTFx_CommandFailure
    Run-time error during command execution.

enumerator kStatus_FTFx_UnknownProperty
    Unknown property.

enumerator kStatus_FTFx_EraseKeyError
    API erase key is invalid.

enumerator kStatus_FTFx_RegionExecuteOnly
    The current region is execute-only.

enumerator kStatus_FTFx_ExecuteInRamFunctionNotReady
    Execute-in-RAM function is not available.

enumerator kStatus_FTFx_PartitionStatusUpdateFailure
    Failed to update partition status.

enumerator kStatus_FTFx_SetFlexramAsEepromError
    Failed to set FlexRAM as EEPROM.

enumerator kStatus_FTFx_RecoverFlexramAsRamError
    Failed to recover FlexRAM as RAM.

enumerator kStatus_FTFx_SetFlexramAsRamError
    Failed to set FlexRAM as RAM.

enumerator kStatus_FTFx_RecoverFlexramAsEepromError
    Failed to recover FlexRAM as EEPROM.

enumerator kStatus_FTFx_CommandNotSupported
    Flash API is not supported.

enumerator kStatus_FTFx_SwapSystemNotInUninitialized
    Swap system is not in an uninitialzed state.

enumerator kStatus_FTFx_SwapIndicatorAddressError
    The swap indicator address is invalid.

enumerator kStatus_FTFx_ReadOnlyProperty
    The flash property is read-only.

enumerator kStatus_FTFx_InvalidPropertyValue
    The flash property value is out of range.

enumerator kStatus_FTFx_InvalidSpeculationOption
    The option of flash prefetch speculation is invalid.

enumerator kStatus_FTFx_CommandOperationInProgress
    The option of flash command is processing.

enum __ftfx_driver_api_keys
    Enumeration for FTFx 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 kFTFx_ApiEraseKey
    Key value used to validate all FTFx erase APIs.

void FTFx_API_Init(*ftfx_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.

status_t FTFx_API_UpdateFlexnvmPartitionStatus(*ftfx_config_t* *config)
    Updates FlexNVM memory partition status according to data flash 0 IFR.

    This function updates FlexNVM memory partition status.

    Parameters

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

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FTFx_CMD_Erase(*ftfx_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.

- kStatus_FTFx_AddressError – The address is out of range.

- kStatus_FTFx_EraseKeyError – The API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_EraseSectorNonBlocking(*ftfx_config_t* *config, uint32_t start, uint32_t key)

Erases the flash sectors encompassed by parameters passed into function.

This function erases one flash sector size based on the start address.

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.

- key – The value used to validate all flash erase APIs.

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – The address is out of range.
- kStatus_FTFx_EraseKeyError – The API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

status_t FTFx_CMD_EraseAll(*ftfx_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_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_EraseKeyError – API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during command execution.
- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FTFx_CMD_EraseAllUnsecure(*ftfx_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_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_EraseKeyError – API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FTFx_CMD_EraseAllExecuteOnlySegments(*ftfx_config_t* *config, uint32_t key)

Erases all program flash execute-only segments defined by the FXACC registers.

Parameters

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

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_EraseKeyError – API erase key is invalid.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_Program(*ftfx_config_t* *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters.

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

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be programmed. Must be 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_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_ProgramOnce(*ftfx_config_t* *config, uint32_t index, const uint8_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_ProgramSection(*ftfx_config_t* *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory 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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_SetFlexramAsRamError – Failed to set flexram as RAM.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_RecoverFlexramAsEepromError – Failed to recover FlexRAM as EEPROM.

status_t FTFx_CMD_ProgramPartition(*ftfx_config_t* *config, *ftfx_partition_flexram_load_opt_t* option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode, uint8_t CSEcKeySize, uint8_t CFE)

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Parameters

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

- option – The option used to set FlexRAM load behavior during reset.

- eepromDataSizeCode – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.

- flexnvmPartitionCode – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – Invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FTFx_CMD_ReadOnce(*ftfx_config_t* *config, uint32_t index, uint8_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_ReadResource(*ftfx_config_t* *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, *ftfx_read_resource_opt_t* option)

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location 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.

- 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 read. Must be word-aligned.

- option – The resource option which indicates which area should be read back.

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_VerifyErase(*ftfx_config_t* *config, uint32_t start, uint32_t lengthInBytes, *ftfx_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.
>
> - 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.
>
> - margin – Read margin choice.
>
> **Return values**
>
> - kStatus_FTFx_Success – API was executed successfully.
>
> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.
>
> - kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.
>
> - kStatus_FTFx_AddressError – Address is out of range.
>
> - kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
>
> - kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
>
> - kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
>
> - kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_VerifyEraseAll(*ftfx_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully.
>
> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.
>
> - kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
>
> - kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.
>
> - kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
>
> - kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_VerifyEraseAllExecuteOnlySegments(*ftfx_config_t* *config, *ftfx_margin_value_t* margin)

Verifies whether the program flash execute-only segments have been 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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_VerifyProgram(*ftfx_config_t* *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, *ftfx_margin_value_t* margin, uint32_t *failedAddress, uint32_t *failedData)

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

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

Parameters

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

- start – The start address of the desired flash memory to be verified. Must be word-aligned.

- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.

- expectedData – A pointer to the expected data that is to be verified against.

- margin – Read margin choice.

- failedAddress – A pointer to the returned failing address.

- failedData – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is re-
  quested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during the command ex-
  ecution.

status_t FTFx_REG_GetSecurityState(*ftfx_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

status_t FTFx_CMD_SecurityBypass(*ftfx_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_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function
  is not available.
- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds
  addresses.
- kStatus_FTFx_ProtectionViolation – The program/erase operation is re-
  quested to execute on protected areas.
- kStatus_FTFx_CommandFailure – Run-time error during the command ex-
  ecution.

status_t FTFx_CMD_SetFlexramFunction(*ftfx_config_t* *config, *ftfx_flexram_func_opt_t* option)

Sets the FlexRAM function command.

Parameters

- config – A pointer to the storage for the driver runtime state.
- option – The option used to set the work mode of FlexRAM.

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function
  is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FTFx_CMD_SwapControl(*ftfx_config_t* *config, uint32_t address, *ftfx_swap_control_opt_t* option, *ftfx_swap_state_config_t* *returnInfo)

Configures the Swap function or checks the swap state of the Flash module.

Parameters

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

- address – Address used to configure the flash Swap function.

- option – The possible option used to configure Flash Swap function or check the flash Swap status

- returnInfo – A pointer to the data which is used to return the information of flash Swap.

Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_SwapIndicatorAddressError – Swap indicator address is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

enum _ftfx_partition_flexram_load_option

Enumeration for the FlexRAM load during reset option.

Values:

enumerator kFTFx_PartitionFlexramLoadOptLoadedWithValidEepromData

FlexRAM is loaded with valid EEPROM data during reset sequence.

enumerator kFTFx_PartitionFlexramLoadOptNotLoaded

FlexRAM is not loaded during reset sequence.

enum _ftfx_read_resource_opt

Enumeration for the two possible options of flash read resource command.

Values:

enumerator kFTFx_ResourceOptionFlashIfr

Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR

enumerator kFTFx_ResourceOptionVersionId
    Select code for the version ID

enum __ftfx_margin_value
    Enumeration for supported FTFx margin levels.

    Values:

    enumerator kFTFx_MarginValueNormal
        Use the 'normal' read level for 1s.

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

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

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

enum __ftfx_security_state
    Enumeration for the three possible FTFx security states.

    Values:

    enumerator kFTFx_SecurityStateNotSecure
        Flash is not secure.

    enumerator kFTFx_SecurityStateBackdoorEnabled
        Flash backdoor is enabled.

    enumerator kFTFx_SecurityStateBackdoorDisabled
        Flash backdoor is disabled.

enum __ftfx_flexram_function_option
    Enumeration for the two possilbe options of set FlexRAM function command.

    Values:

    enumerator kFTFx_FlexramFuncOptAvailableAsRam
        An option used to make FlexRAM available as RAM

    enumerator kFTFx_FlexramFuncOptEepromQuickWriteRecovery
        An option used to complete interrupted EEPROM quick write process

    enumerator kFTFx_FlexramFuncOptEepromQuickWriteStatus
        An option used to make EEPROM quick write status query

    enumerator kFTFx_FlexramFuncOptAvailableForEepromQuickWrite
        An option used to make FlexRAM available for EEPROM in Quick Write mode

    enumerator kFTFx_FlexramFuncOptAvailableForEeprom
        An option used to make FlexRAM available for EEPROM

enum __flash_acceleration_ram_property
    Enumeration for acceleration ram property.

    Values:

    enumerator kFLASH_AccelerationRamSize

enum __ftfx_swap_control_option
    Enumeration for the possible options of Swap control commands.

    Values:

enumerator kFTFx_SwapControlOptionIntializeSystem
    An option used to initialize the Swap system

enumerator kFTFx_SwapControlOptionSetInUpdateState
    An option used to set the Swap in an update state

enumerator kFTFx_SwapControlOptionSetInCompleteState
    An option used to set the Swap in a complete state

enumerator kFTFx_SwapControlOptionReportStatus
    An option used to report the Swap status

enumerator kFTFx_SwapControlOptionDisableSystem
    An option used to disable the Swap status

enum _ftfx_swap_state
    Enumeration for the possible flash Swap status.

    Values:

    enumerator kFTFx_SwapStateUninitialized
        Flash Swap system is in an uninitialized state.

    enumerator kFTFx_SwapStateReady
        Flash Swap system is in a ready state.

    enumerator kFTFx_SwapStateUpdate
        Flash Swap system is in an update state.

    enumerator kFTFx_SwapStateUpdateErased
        Flash Swap system is in an updateErased state.

    enumerator kFTFx_SwapStateComplete
        Flash Swap system is in a complete state.

    enumerator kFTFx_SwapStateDisabled
        Flash Swap system is in a disabled state.

enum _ftfx_swap_block_status
    Enumeration for the possible flash Swap block status.

    Values:

    enumerator kFTFx_SwapBlockStatusLowerHalfProgramBlocksAtZero
        Swap block status is that lower half program block at zero.

    enumerator kFTFx_SwapBlockStatusUpperHalfProgramBlocksAtZero
        Swap block status is that upper half program block at zero.

enum _ftfx_memory_type
    Enumeration for FTFx memory type.

    Values:

    enumerator kFTFx_MemTypePflash

    enumerator kFTFx_MemTypeFlexnvm

typedef enum *_ftfx_partition_flexram_load_option* ftfx_partition_flexram_load_opt_t
    Enumeration for the FlexRAM load during reset option.

typedef enum *_ftfx_read_resource_opt* ftfx_read_resource_opt_t
    Enumeration for the two possible options of flash read resource command.

typedef enum _*ftfx_margin_value* ftfx_margin_value_t

Enumeration for supported FTFx margin levels.

typedef enum _*ftfx_security_state* ftfx_security_state_t

Enumeration for the three possible FTFx security states.

typedef enum _*ftfx_flexram_function_option* ftfx_flexram_func_opt_t

Enumeration for the two possilbe options of set FlexRAM function command.

typedef enum _*ftfx_swap_control_option* ftfx_swap_control_opt_t

Enumeration for the possible options of Swap control commands.

typedef enum _*ftfx_swap_state* ftfx_swap_state_t

Enumeration for the possible flash Swap status.

typedef enum _*ftfx_swap_block_status* ftfx_swap_block_status_t

Enumeration for the possible flash Swap block status.

typedef struct _*ftfx_swap_state_config* ftfx_swap_state_config_t

Flash Swap information.

typedef struct _*ftfx_special_mem* ftfx_spec_mem_t

ftfx special memory access information.

typedef struct _*ftfx_mem_descriptor* ftfx_mem_desc_t

Flash memory descriptor.

typedef struct _*ftfx_ops_config* ftfx_ops_config_t

Active FTFx information for the current operation.

typedef struct _*ftfx_ifr_descriptor* ftfx_ifr_desc_t

Flash IFR memory descriptor.

typedef struct _*ftfx_config* ftfx_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 __ftfx_swap_state_config

#include <fsl_ftfx_controller.h> Flash Swap information.

Public Members

ftfx_swap_state_t flashSwapState

The current Swap system status.

ftfx_swap_block_status_t currentSwapBlockStatus

The current Swap block status.

ftfx_swap_block_status_t nextSwapBlockStatus

The next Swap block status.

struct __ftfx_special_mem

#include <fsl_ftfx_controller.h> ftfx special memory access information.

Public Members

uint32_t base

Base address of flash special memory.

uint32_t size
>   size of flash special memory.

uint32_t count
>   flash special memory count.

struct __ftfx__mem__descriptor
>   *#include <fsl_ftfx_controller.h>* Flash memory descriptor.

Public Members

uint32_t blockBase
>   A base address of the flash block

uint32_t aliasBlockBase
>   A base address of the alias flash block

uint32_t totalSize
>   The size of the flash block.

uint32_t sectorSize
>   The size in bytes of a sector of flash.

uint32_t blockCount
>   A number of flash blocks.

struct __ftfx__ops__config
>   *#include <fsl_ftfx_controller.h>* Active FTFx information for the current operation.

Public Members

uint32_t convertedAddress
>   A converted address for the current flash type.

struct __ftfx__ifr__descriptor
>   *#include <fsl_ftfx_controller.h>* Flash IFR memory descriptor.

union function__ptr__t
>   *#include <fsl_ftfx_controller.h>*

Public Members

uint32_t commadAddr

void (*callFlashCommand)(volatile uint8_t *FTMRx_fstat)

struct __ftfx__config
>   *#include <fsl_ftfx_controller.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 flexramBlockBase
>   The base address of the FlexRAM/acceleration RAM

uint32_t flexramTotalSize
> The size of the FlexRAM/acceleration RAM

uint16_t eepromTotalSize
> The size of EEPROM area which was partitioned from FlexRAM

function_ptr_t runCmdFuncAddr
> An buffer point to the flash execute-in-RAM function.

struct ___unnamed12___

Public Members

uint8_t type
> Type of flash block.

uint8_t index
> Index of flash block.

struct feature

struct addrAligment

struct feature

struct resRange

Public Members

uint8_t versionIdStart
> Version ID start address

uint32_t pflashIfrStart
> Program Flash 0 IFR start address

uint32_t dflashIfrStart
> Data Flash 0 IFR start address

uint32_t pflashSwapIfrStart
> Program Flash Swap IFR start address

struct idxInfo

2.14 ftfx feature

FTFx_DRIVER_IS_FLASH_RESIDENT
> Flash driver location.
>
> Used for the flash resident application.

FTFx_DRIVER_IS_EXPORTED
> Flash Driver Export option.
>
> Used for the MCUXpresso SDK application.

FTFx_FLASH1_HAS_PROT_CONTROL
> Indicates whether the secondary flash has its own protection register in flash module.

FTFx_FLASH1_HAS_XACC_CONTROL

Indicates whether the secondary flash has its own Execute-Only access register in flash module.

FTFx_DRIVER_HAS_FLASH1_SUPPORT

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

FTFx_FLASH_COUNT

FTFx_FLASH1_IS_INDEPENDENT_BLOCK

2.15   Ftftx FLASH Driver

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_FTFx_Success – API was executed successfully.
> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.
> - kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.
> - kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

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

Erases the Dflash 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_FTFx_Success – API was executed successfully; the appropriate number of flash sectors based on the desired start address and length were erased successfully.
> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.
> - kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.
> - kStatus_FTFx_AddressError – The address is out of range.
> - kStatus_FTFx_EraseKeyError – The API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_EraseSectorNonBlocking(*flash_config_t* *config, uint32_t start, uint32_t key)

    Erases the Dflash sectors encompassed by parameters passed into function.

    This function erases one flash sector size based on the start address, and it is executed asynchronously.

    NOTE: This function can only erase one flash sector at a time, and the other commands can be executed after the previous command has been completed.

    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.

- key – The value used to validate all flash erase APIs.

    Return values

- kStatus_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.

- kStatus_FTFx_AddressError – The address is out of range.

- kStatus_FTFx_EraseKeyError – The API erase key is invalid.

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

    Erases entire flexnvm.

    Parameters

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

- key – A value used to validate all flash erase APIs.

    Return values

- kStatus_FTFx_Success – API was executed successfully; the all pflash and flexnvm were erased successfully, the swap and eeprom have been reset to unconfigured state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_EraseKeyError – API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

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

Erases the entire flexnvm, 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_FTFx_Success – API was executed successfully; the protected sectors of flash were reset to unprotected status.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_EraseKeyError – API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

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

Programs flash with data at locations passed in through parameters.

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

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_FTFx_Success – API was executed successfully; the desired data were programed successfully into flash based on desired start address and length.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_ProgramOnce(*flash_config_t* *config, uint32_t index, uint8_t *src, uint32_t lengthInBytes)

Program the Program-Once-Field through parameters.

This function Program the Program-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.

- src – A pointer to the source buffer of data that is used to store data to be write.

- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

- kStatus_FTFx_Success – API was executed successfully; The index indicating the area of program once field was programed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

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

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory 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_FTFx_Success – API was executed successfully; the desired data have been programed successfully into flash based on start address and length.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_SetFlexramAsRamError – Failed to set flexram as RAM.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_RecoverFlexramAsEepromError – Failed to recover FlexRAM as EEPROM.

status_t FLASH_ReadResource(*flash_config_t* *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, *ftfx_read_resource_opt_t* option)

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location 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.

- 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 read. Must be word-aligned.

- option – The resource option which indicates which area should be read back.

Return values

- kStatus_FTFx_Success – API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_ReadOnce(*flash_config_t* *config, uint32_t index, uint8_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_FTFx_Success – API was executed successfully; the data have been successfuly read form Program flash0 IFR map and Program Once field based on index and length.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_VerifyErase(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes, *ftfx_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.

- 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.

- margin – Read margin choice.

Return values

- kStatus_FTFx_Success – API was executed successfully; the specified FLASH region has been erased.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_VerifyEraseAll(*flash_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully; all program flash and flexnvm were in erased state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

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

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

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

Parameters

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

- start – The start address of the desired flash memory to be verified. Must be word-aligned.

- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.

- expectedData – A pointer to the expected data that is to be verified against.

- margin – Read margin choice.

- failedAddress – A pointer to the returned failing address.

- failedData – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- kStatus_FTFx_Success – API was executed successfully; the desired data have been successfully programed into specified FLASH region.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_GetSecurityState(*flash_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully; the security state of flash was stored to state.

- kStatus_FTFx_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_SetFlexramFunction(*flash_config_t* *config, *ftfx_flexram_func_opt_t* option)

> Sets the FlexRAM function command.

>> **Parameters**

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

>>> - option – The option used to set the work mode of FlexRAM.

>> **Return values**

>>> - kStatus_FTFx_Success – API was executed successfully; the FlexRAM has been successfully configured as RAM or EEPROM.

>>> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.

>>> - kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

>>> - kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

>>> - kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

>>> - kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLASH_Swap(*flash_config_t* *config, uint32_t address, bool isSetEnable)

> Swaps the lower half flash with the higher half flash.

>> **Parameters**

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

>>> - address – Address used to configure the flash swap function

>>> - isSetEnable – The possible option used to configure the Flash Swap function or check the flash Swap status.

>> **Return values**

>>> - kStatus_FTFx_Success – API was executed successfully; the lower half flash and higher half flash have been swaped.

>>> - kStatus_FTFx_InvalidArgument – An invalid argument is provided.

>>> - kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

>>> - kStatus_FTFx_SwapIndicatorAddressError – Swap indicator address is invalid.

>>> - kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

>>> - kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

>>> - kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

>>> - kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_SwapSystemNotInUninitialized – Swap system is not in an uninitialized state.

status_t FLASH_IsProtected(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes,
  flash_prot_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_FTFx_Success – API was executed successfully; the protection state of specified FLASH region was stored to protection_state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – The address is out of range.

status_t FLASH_IsExecuteOnly(*flash_config_t* *config, uint32_t start, uint32_t lengthInBytes,
  flash_xacc_state_t *access_state)

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

This function retrieves the current flash access 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.

- access_state – A pointer to the value returned for the current access status code for the desired flash area.

Return values

- kStatus_FTFx_Success – API was executed successfully; the executeOnly state of specified FLASH region was stored to access_state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – The parameter is not aligned to the specified baseline.

- kStatus_FTFx_AddressError – The address is out of range.

status_t FLASH_PflashSetProtection(*flash_config_t* *config, *pflash_prot_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. Each bit is corresponding to protection of 1/32(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- kStatus_FTFx_Success – API was executed successfully; the specified FLASH region is protected.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLASH_PflashGetProtection(*flash_config_t* *config, *pflash_prot_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. Each bit is corresponding to the protection of 1/32(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- kStatus_FTFx_Success – API was executed successfully; the Protection state was stored to protectStatus;

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

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_FTFx_Success – API was executed successfully; the flash property was stored to value.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_UnknownProperty – An unknown property tag.

status_t FLASH_GetCommandState(**void**)

> Get previous command status.

> This function is used to obtain the execution status of the previous command.

>> **Return values**

>>> • kStatus_FTFx_Success – The previous command is executed successfully.

>>> • kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

>>> • kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

>>> • kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

>>> • kStatus_FTFx_CommandFailure – Run-time error during the command execution.

FSL_FLASH_DRIVER_VERSION

> Flash driver version for SDK.

> Version 3.3.0.

FSL_FLASH_DRIVER_VERSION_ROM

> Flash driver version for ROM.

> Version 3.0.0.

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_execute_only_access_state

> Enumeration for the three possible flash execute access levels.

> *Values:*

> enumerator kFLASH_AccessStateUnLimited
>> Flash region is unlimited.

> enumerator kFLASH_AccessStateExecuteOnly
>> Flash region is execute only.

> enumerator kFLASH_AccessStateMixed
>> Flash is mixed with unlimited and execute only region.

enum _flash_property_tag

> Enumeration for various flash properties.

> *Values:*

> enumerator kFLASH_PropertyPflash0SectorSize
>> Pflash sector size property.

enumerator kFLASH_PropertyPflash0TotalSize
   Pflash total size property.

enumerator kFLASH_PropertyPflash0BlockSize
   Pflash block size property.

enumerator kFLASH_PropertyPflash0BlockCount
   Pflash block count property.

enumerator kFLASH_PropertyPflash0BlockBaseAddr
   Pflash block base address property.

enumerator kFLASH_PropertyPflash0FacSupport
   Pflash fac support property.

enumerator kFLASH_PropertyPflash0AccessSegmentSize
   Pflash access segment size property.

enumerator kFLASH_PropertyPflash0AccessSegmentCount
   Pflash access segment count property.

enumerator kFLASH_PropertyPflash1SectorSize
   Pflash sector size property.

enumerator kFLASH_PropertyPflash1TotalSize
   Pflash total size property.

enumerator kFLASH_PropertyPflash1BlockSize
   Pflash block size property.

enumerator kFLASH_PropertyPflash1BlockCount
   Pflash block count property.

enumerator kFLASH_PropertyPflash1BlockBaseAddr
   Pflash block base address property.

enumerator kFLASH_PropertyPflash1FacSupport
   Pflash fac support property.

enumerator kFLASH_PropertyPflash1AccessSegmentSize
   Pflash access segment size property.

enumerator kFLASH_PropertyPflash1AccessSegmentCount
   Pflash access segment count property.

enumerator kFLASH_PropertyFlexRamBlockBaseAddr
   FlexRam block base address property.

enumerator kFLASH_PropertyFlexRamTotalSize
   FlexRam total size property.

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

typedef union _pflash_protection_status pflash_prot_status_t
   PFlash protection status.

typedef enum _flash_execute_only_access_state flash_xacc_state_t
   Enumeration for the three possible flash execute access levels.

typedef enum _flash_property_tag flash_property_tag_t
   Enumeration for various flash properties.

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.

kStatus_FLASH_Success

kFLASH_ApiEraseKey

union __pflash_protection_status

 #include <fsl_ftfx_flash.h> PFlash protection status.

 Public Members

 uint32_t protl
  PROT[31:0] .

 uint32_t proth
  PROT[63:32].

 uint8_t protsl
  PROTS[7:0] .

 uint8_t protsh
  PROTS[15:8] .

 uint8_t reserved[2]

struct __flash_config

 #include <fsl_ftfx_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.

2.16 Ftftx FLEXNVM Driver

status_t FLEXNVM_Init(*flexnvm_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_FTFx_Success – API was executed successfully.

   • kStatus_FTFx_InvalidArgument – An invalid argument is provided.

   • kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

   • kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FLEXNVM_DflashErase(*flexnvm_config_t* *config, uint32_t start, uint32_t lengthInBytes,
uint32_t key)

Erases the Dflash 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_FTFx_Success – API was executed successfully; the appropriate number of date flash sectors based on the desired start address and length were erased successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – The parameter is not aligned with the specified baseline.

- kStatus_FTFx_AddressError – The address is out of range.

- kStatus_FTFx_EraseKeyError – The API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_EraseAll(*flexnvm_config_t* *config, uint32_t key)

Erases entire flexnvm.

Parameters

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

- key – A value used to validate all flash erase APIs.

Return values

- kStatus_FTFx_Success – API was executed successfully; the entire flexnvm has been erased successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_EraseKeyError – API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FLEXNVM_EraseAllUnsecure(*flexnvm_config_t* *config, uint32_t key)

Erases the entire flexnvm, 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_FTFx_Success – API was executed successfully; the flexnvm is not in securityi state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_EraseKeyError – API erase key is invalid.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_PartitionStatusUpdateFailure – Failed to update the partition status.

status_t FLEXNVM_DflashProgram(*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters.

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

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_FTFx_Success – API was executed successfully; the desired date have been successfully programed into specified date flash region.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_DflashProgramSection(*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters via the Program Section command.

This function programs the flash memory 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_FTFx_Success – API was executed successfully; the desired date have been successfully programed into specified date flash area.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_SetFlexramAsRamError – Failed to set flexram as RAM.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

- kStatus_FTFx_RecoverFlexramAsEepromError – Failed to recover FlexRAM as EEPROM.

status_t FLEXNVM_ProgramPartition(*flexnvm_config_t* *config, *ftfx_partition_flexram_load_opt_t* option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Parameters

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

- option – The option used to set FlexRAM load behavior during reset.

- eepromDataSizeCode – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.

- flexnvmPartitionCode – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

- kStatus_FTFx_Success – API was executed successfully; the FlexNVM block for use as data flash, EEPROM backup, or a combination of both have been Prepared.

- kStatus_FTFx_InvalidArgument – Invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLEXNVM_ProgramPartition_CSE(*flexnvm_config_t* *config,
ftfx_partition_flexram_load_opt_t option, uint32_t
eepromDataSizeCode, uint32_t
flexnvmPartitionCode, uint8_t CSEcKeySize, uint8_t
SFE)

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM. This is the CSE enabled version for IP's like FTFC.

Parameters

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

- option – The option used to set FlexRAM load behavior during reset.

- eepromDataSizeCode – Determines the amount of FlexRAM used in each of the available EEPROM subsystems.

- flexnvmPartitionCode – Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

- CSEcKeySize – CSEc/SHE key size, see RM for details and possible values

- SFE – Security Flag Extension (SFE), see RM for details and possible values

Return values

- kStatus_FTFx_Success – API was executed successfully; the FlexNVM block for use as data flash, EEPROM backup, or a combination of both have been Prepared.

- kStatus_FTFx_InvalidArgument – Invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLEXNVM_ReadResource(*flexnvm_config_t* *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, *ftfx_read_resource_opt_t* option)

Reads the resource with data at locations passed in through parameters.

This function reads the flash memory with the desired location 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.

- 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 read. Must be word-aligned.

- option – The resource option which indicates which area should be read back.

Return values

- kStatus_FTFx_Success – API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with the specified baseline.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_DflashVerifyErase(*flexnvm_config_t* *config, uint32_t start, uint32_t lengthInBytes, *ftfx_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.

- 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.

- margin – Read margin choice.

Return values

- kStatus_FTFx_Success – API was executed successfully; the specified data flash region is in erased state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_VerifyEraseAll(*flexnvm_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully; the entire flexnvm region is in erased state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_DflashVerifyProgram(*flexnvm_config_t* *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, *ftfx_margin_value_t* margin, uint32_t *failedAddress, uint32_t *failedData)

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

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

Parameters

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

- start – The start address of the desired flash memory to be verified. Must be word-aligned.

- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.

- expectedData – A pointer to the expected data that is to be verified against.

- margin – Read margin choice.

- failedAddress – A pointer to the returned failing address.

- failedData – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- kStatus_FTFx_Success – API was executed successfully; the desired data hve been programed successfully into specified data flash region.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AlignmentError – Parameter is not aligned with specified baseline.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_GetSecurityState(*flexnvm_config_t* *config, *ftfx_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_FTFx_Success – API was executed successfully; the security state of flexnvm was stored to state.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

status_t FLEXNVM_SecurityBypass(*flexnvm_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_FTFx_Success – API was executed successfully.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_SetFlexramFunction(*flexnvm_config_t* *config, *ftfx_flexram_func_opt_t* option)

　　Sets the FlexRAM function command.

Parameters

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

- option – The option used to set the work mode of FlexRAM.

Return values

- kStatus_FTFx_Success – API was executed successfully; the FlexRAM has been successfully configured as RAM or EEPROM

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_ExecuteInRamFunctionNotReady – Execute-in-RAM function is not available.

- kStatus_FTFx_AccessError – Invalid instruction codes and out-of bounds addresses.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_CommandFailure – Run-time error during the command execution.

status_t FLEXNVM_DflashSetProtection(*flexnvm_config_t* *config, uint8_t protectStatus)

　　Sets the DFlash 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 DFlash protection register. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

- kStatus_FTFx_Success – API was executed successfully; the specified DFlash region is protected.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_CommandNotSupported – Flash API is not supported.

• kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLEXNVM_DflashGetProtection(*flexnvm_config_t* *config, uint8_t *protectStatus)

Gets the DFlash protection status.

Parameters

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

• protectStatus – DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash, and so on. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

• kStatus_FTFx_Success – API was executed successfully.

• kStatus_FTFx_InvalidArgument – An invalid argument is provided.

• kStatus_FTFx_CommandNotSupported – Flash API is not supported.

status_t FLEXNVM_EepromSetProtection(*flexnvm_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. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of EEPROM, and so on. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

• kStatus_FTFx_Success – API was executed successfully.

• kStatus_FTFx_InvalidArgument – An invalid argument is provided.

• kStatus_FTFx_CommandNotSupported – Flash API is not supported.

• kStatus_FTFx_CommandFailure – Run-time error during command execution.

status_t FLEXNVM_EepromGetProtection(*flexnvm_config_t* *config, uint8_t *protectStatus)

Gets the EEPROM protection status.

Parameters

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

• protectStatus – DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of the EEPROM. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

• kStatus_FTFx_Success – API was executed successfully.

• kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_CommandNotSupported – Flash API is not supported.

status_t FLEXNVM_GetProperty(*flexnvm_config_t* *config, *flexnvm_property_tag_t* whichProperty, uint32_t *value)

Returns the desired flexnvm property.

Parameters

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

Return values

- kStatus_FTFx_Success – API was executed successfully.
- kStatus_FTFx_InvalidArgument – An invalid argument is provided.
- kStatus_FTFx_UnknownProperty – An unknown property tag.

enum _flexnvm_property_tag

Enumeration for various flexnvm properties.

Values:

enumerator kFLEXNVM_PropertyDflashSectorSize
Dflash sector size property.

enumerator kFLEXNVM_PropertyDflashTotalSize
Dflash total size property.

enumerator kFLEXNVM_PropertyDflashBlockSize
Dflash block size property.

enumerator kFLEXNVM_PropertyDflashBlockCount
Dflash block count property.

enumerator kFLEXNVM_PropertyDflashBlockBaseAddr
Dflash block base address property.

enumerator kFLEXNVM_PropertyAliasDflashBlockBaseAddr
Dflash block base address Alias property.

enumerator kFLEXNVM_PropertyFlexRamBlockBaseAddr
FlexRam block base address property.

enumerator kFLEXNVM_PropertyFlexRamTotalSize
FlexRam total size property.

enumerator kFLEXNVM_PropertyEepromTotalSize
EEPROM total size property.

typedef enum *_flexnvm_property_tag* flexnvm_property_tag_t
Enumeration for various flexnvm properties.

typedef struct *_flexnvm_config* flexnvm_config_t
Flexnvm driver state information.

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

status_t FLEXNVM__EepromWrite(*flexnvm_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_FTFx_Success – API was executed successfully; the desires data have been successfully programed into specified eeprom region.

- kStatus_FTFx_InvalidArgument – An invalid argument is provided.

- kStatus_FTFx_AddressError – Address is out of range.

- kStatus_FTFx_SetFlexramAsEepromError – Failed to set flexram as eeprom.

- kStatus_FTFx_ProtectionViolation – The program/erase operation is requested to execute on protected areas.

- kStatus_FTFx_RecoverFlexramAsRamError – Failed to recover the FlexRAM as RAM.

struct __flexnvm_config

#include <fsl_ftfx_flexnvm.h> Flexnvm driver state information.

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

2.17 ftfx utilities

ALIGN__DOWN(x, a)

Alignment(down) utility.

ALIGN__UP(x, a)

Alignment(up) utility.

MAKE__VERSION(major, minor, bugfix)

Constructs the version number for drivers.

MAKE__STATUS(group, code)

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

FOUR__CHAR__CODE(a, b, c, d)

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

B1P4(b)

bytes2word utility.

B1P3(**b**)

B1P2(**b**)

B1P1(**b**)

B2P3(**b**)

B2P2(**b**)

B2P1(**b**)

B3P2(**b**)

B3P1(**b**)

BYTE2WORD_1_3(**x**, **y**)

BYTE2WORD_2_2(**x**, **y**)

BYTE2WORD_3_1(**x**, **y**)

BYTE2WORD_1_1_2(**x**, **y**, **z**)

BYTE2WORD_1_2_1(**x**, **y**, **z**)

BYTE2WORD_2_1_1(**x**, **y**, **z**)

BYTE2WORD_1_1_1_1(**x**, **y**, **z**, **w**)

2.18   GPIO: General-Purpose Input/Output Driver

FSL_GPIO_DRIVER_VERSION
   GPIO driver version.

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

enum __gpio_checker_attribute
   GPIO checker attribute.

   Values:

   enumerator kGPIO_UsernonsecureRWUsersecureRWPrivilegedsecureRW
      User nonsecure:Read+Write; User Secure:Read+Write; Privileged Secure:Read+Write

   enumerator kGPIO_UsernonsecureRUsersecureRWPrivilegedsecureRW
      User nonsecure:Read; User Secure:Read+Write; Privileged Secure:Read+Write

   enumerator kGPIO_UsernonsecureNUsersecureRWPrivilegedsecureRW
      User nonsecure:None; User Secure:Read+Write; Privileged Secure:Read+Write

   enumerator kGPIO_UsernonsecureRUsersecureRPrivilegedsecureRW
      User nonsecure:Read; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureRPrivilegedsecureRW
     User nonsecure:None; User Secure:Read; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureRW
     User nonsecure:None; User Secure:None; Privileged Secure:Read+Write

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureR
     User nonsecure:None; User Secure:None; Privileged Secure:Read

enumerator kGPIO_UsernonsecureNUsersecureNPrivilegedsecureN
     User nonsecure:None; User Secure:None; Privileged Secure:None

enumerator kGPIO_IgnoreAttributeCheck
     Ignores the attribute check

typedef enum *_gpio_pin_direction* gpio_pin_direction_t
     GPIO direction definition.

typedef enum *_gpio_checker_attribute* gpio_checker_attribute_t
     GPIO checker attribute.

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().

GPIO_FIT_REG(value)

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

2.19 GPIO Driver

void GPIO_PortInit(GPIO_Type *base)
     Initializes the GPIO peripheral.

     This function ungates the GPIO clock.

          Parameters

               • base – GPIO peripheral base pointer.

void GPIO_PortDenit(GPIO_Type *base)
     Denitializes the GPIO peripheral.

          Parameters

- base – GPIO peripheral base pointer.

void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const *gpio_pin_config_t* *config)

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, 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,
}
Define a digital output pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalOutput,
  0,
}
```

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO port pin number
- config – GPIO pin configuration pointer

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t pin, uint8_t output)

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

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- pin – GPIO pin number
- output – GPIO pin output logic level.
  - 0: corresponding pin output low-logic level.
  - 1: corresponding pin output high-logic level.

static inline void GPIO_PortSet(GPIO_Type *base, uint32_t mask)

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

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortClear(GPIO_Type *base, uint32_t mask)

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

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- mask – GPIO pin number macro

static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t mask)

Reverses the current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)

- mask – GPIO pin number macro

static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t pin)

Reads the current input value of the GPIO port.

> **Parameters**
>
>> - base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
>>
>> - pin – GPIO pin number
>
> **Return values**
>
>> GPIO – port input value
>>
>> - 0: corresponding pin input low-logic level.
>>
>> - 1: corresponding pin input high-logic level.

uint32_t GPIO_PortGetInterruptFlags(GPIO_Type *base)

Reads the GPIO port interrupt status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

> **Parameters**
>
>> - base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
>
> **Return values**
>
>> The – current GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

void GPIO_PortClearInterruptFlags(GPIO_Type *base, uint32_t mask)

Clears multiple GPIO pin interrupt status flags.

> **Parameters**
>
>> - base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
>>
>> - mask – GPIO pin number macro

void GPIO_CheckAttributeBytes(GPIO_Type *base, *gpio_checker_attribute_t* attribute)

brief The GPIO module supports a device-specific number of data ports, organized as 32-bit words/8-bit Bytes. Each 32-bit/8-bit data port includes a GACR register, which defines the byte-level attributes required for a successful access to the GPIO programming model. If the GPIO module's GACR register organized as 32-bit words, the attribute controls for the 4 data bytes in the GACR follow a standard little endian data convention.

> **Parameters**
>
>> - base – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
>>
>> - attribute – GPIO checker attribute

2.20   I2C: Inter-Integrated Circuit Driver

2.21   I2C DMA Driver

void I2C_MasterTransferCreateHandleDMA(I2C_Type *base, *i2c_master_dma_handle_t* *handle, *i2c_master_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *dmaHandle)

Initializes the I2C handle which is used in transactional functions.

Parameters

- base – I2C peripheral base address
- handle – Pointer to the i2c_master_dma_handle_t structure
- callback – Pointer to the user callback function
- userData – A user parameter passed to the callback function
- dmaHandle – DMA handle pointer

status_t I2C_MasterTransferDMA(I2C_Type *base, *i2c_master_dma_handle_t* *handle,
i2c_master_transfer_t *xfer)

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

Parameters

- base – I2C peripheral base address
- handle – A pointer to the i2c_master_dma_handle_t structure
- xfer – A pointer to the transfer structure of the i2c_master_transfer_t

Return values

- kStatus_Success – Successfully completes the data transmission.
- kStatus_I2C_Busy – A previous transmission is still not finished.
- kStatus_I2C_Timeout – A transfer error, waits for the signal timeout.
- kStatus_I2C_ArbitrationLost – A transfer error, arbitration lost.
- kStataus_I2C_Nak – A transfer error, receives NAK during transfer.

status_t I2C_MasterTransferGetCountDMA(I2C_Type *base, *i2c_master_dma_handle_t* *handle,
size_t *count)

Gets a master transfer status during a DMA non-blocking transfer.

Parameters

- base – I2C peripheral base address
- handle – A pointer to the i2c_master_dma_handle_t structure
- count – A number of bytes transferred so far by the non-blocking transaction.

void I2C_MasterTransferAbortDMA(I2C_Type *base, *i2c_master_dma_handle_t* *handle)

Aborts a master DMA non-blocking transfer early.

Parameters

- base – I2C peripheral base address
- handle – A pointer to the i2c_master_dma_handle_t structure.

FSL_I2C_DMA_DRIVER_VERSION

I2C DMA driver version.

typedef struct *_i2c_master_dma_handle* i2c_master_dma_handle_t

Retry times for waiting flag.

I2C master DMA handle typedef.

typedef void (*i2c_master_dma_transfer_callback_t)(I2C_Type *base, *i2c_master_dma_handle_t*
*handle, *status_t* status, void *userData)

I2C master DMA transfer callback typedef.

struct _i2c_master_dma_handle

#include <fsl_i2c_dma.h> I2C master DMA transfer structure.

Public Members

i2c_master_transfer_t transfer
> I2C master transfer struct.

size_t transferSize
> Total bytes to be transferred.

uint8_t state
> I2C master transfer status.

dma_handle_t *dmaHandle
> The DMA handler used.

i2c_master_dma_transfer_callback_t completionCallback
> A callback function called after the DMA transfer finished.

void *userData
> A callback parameter passed to the callback function.

2.22 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_IntPendingFlagFlag

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.

>   >   >   • kStataus_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.
- kStataus_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.
>
> • kStataus_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_SlaveGenaralcallEvent

    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
>   Mater 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.

2.23 IRTC: IRTC Driver

status_t IRTC_Init(RTC_Type *base, const *irtc_config_t* *config)
> Ungates the IRTC clock and configures the peripheral for basic operation.

> This function initiates a soft-reset of the IRTC module, this has not effect on DST, calendaring, standby time and tamper detect registers.

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

> **Parameters**
> - base – IRTC peripheral base address
> - config – Pointer to user's IRTC config structure.
>
> **Returns**
> kStatus_Success If the driver is initialized successfully.
>
> **Returns**
> kStatus_Fail if we cannot disable register write protection

Returns

kStatus_InvalidArgument If the input parameters are wrong.

status_t IRTC_Deinit(RTC_Type *base)

Gate the IRTC clock.

Parameters

- base – IRTC peripheral base address

Returns

kStatus_Success If the driver is initialized successfully.

Returns

kStatus_InvalidArgument If the input parameters are wrong.

void IRTC_GetDefaultConfig(*irtc_config_t* *config)

Fill in the IRTC config struct with the default settings.

The default values are:

```
config->wakeupSelect = true;
config->timerStdMask = false;
config->alrmMatch = kRTC_MatchSecMinHr;
```

Parameters

- config – Pointer to user's IRTC config structure.

status_t IRTC_SetDatetime(RTC_Type *base, const *irtc_datetime_t* *datetime)

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

The IRTC counter is started after the time is set.

Parameters

- base – IRTC peripheral base address

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

Returns

kStatus_Success: success in setting the time and starting the IRTC kStatus_InvalidArgument: failure. An error occurs because the datetime format is incorrect.

void IRTC_GetDatetime(RTC_Type *base, *irtc_datetime_t* *datetime)

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

Parameters

- base – IRTC peripheral base address

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

status_t IRTC_SetAlarm(RTC_Type *base, const *irtc_datetime_t* *alarmTime)

Sets the IRTC alarm time.

Note: weekDay field of alarmTime is not used during alarm match and should be set to 0

Parameters

- base – RTC peripheral base address

- alarmTime – Pointer to structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the alarm kStatus_InvalidArgument: error in setting the alarm. Error occurs because the alarm datetime format is incorrect.

void IRTC_GetAlarm(RTC_Type *base, *irtc_datetime_t* *datetime)

Returns the IRTC alarm time.

Parameters

- base – RTC peripheral base address

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

static inline void IRTC_EnableInterrupts(RTC_Type *base, uint32_t mask)

Enables the selected IRTC interrupts.

Parameters

- base – IRTC peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration irtc_interrupt_enable_t

static inline void IRTC_DisableInterrupts(RTC_Type *base, uint32_t mask)

Disables the selected IRTC interrupts.

Parameters

- base – IRTC peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration irtc_interrupt_enable_t

static inline uint32_t IRTC_GetEnabledInterrupts(RTC_Type *base)

Gets the enabled IRTC interrupts.

Parameters

- base – IRTC peripheral base address

Returns

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

static inline uint32_t IRTC_GetStatusFlags(RTC_Type *base)

Gets the IRTC status flags.

Parameters

- base – IRTC peripheral base address

Returns

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

static inline void IRTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)

Clears the IRTC status flags.

Parameters

- base – IRTC peripheral base address

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

void IRTC_SetDaylightTime(RTC_Type *base, const *irtc_daylight_time_t* *datetime)

Sets the IRTC daylight savings start and stop date and time.

It also enables the daylight saving bit in the IRTC control register

> **Parameters**
>
> - base – IRTC peripheral base address
> - datetime – Pointer to a structure where the date and time details are stored.

void IRTC_GetDaylightTime(RTC_Type *base, *irtc_daylight_time_t* *datetime)

Gets the IRTC daylight savings time and stores it in the given time structure.

> **Parameters**
>
> - base – IRTC peripheral base address
> - datetime – Pointer to a structure where the date and time details are stored.

void IRTC_SetCoarseCompensation(RTC_Type *base, uint8_t compensationValue, uint8_t compensationInterval)

Enables the coarse compensation and sets the value in the IRTC compensation register.

> **Parameters**
>
> - base – IRTC peripheral base address
> - compensationValue – Compensation value is a 2's complement value.
> - compensationInterval – Compensation interval.

void IRTC_SetFineCompensation(RTC_Type *base, uint8_t integralValue, uint8_t fractionValue, bool accumulateFractional)

Enables the fine compensation and sets the value in the IRTC compensation register.

> **Parameters**
>
> - base – The IRTC peripheral base address
> - integralValue – Compensation integral value; twos complement value of the integer part
> - fractionValue – Compensation fraction value expressed as number of clock cycles of a fixed 4.194304Mhz clock that have to be added.
> - accumulateFractional – Flag indicating if we want to add to previous fractional part; true: Add to previously accumulated fractional part, false: Start afresh and overwrite current value

void IRTC_SetTamperParams(RTC_Type *base, *irtc_tamper_pins_t* tamperNumber, const *irtc_tamper_config_t* *tamperConfig)

This function allows configuring the four tamper inputs.

The function configures the filter properties for the three external tampers. It also sets up active/passive and direction of the tamper bits, which are not available on all platforms.

Note: This function programs the tamper filter parameters. The user must gate the 32K clock to the RTC before calling this function. It is assumed that the time and date are set after this and the tamper parameters do not require to be changed again later.

> **Parameters**
>
> - base – The IRTC peripheral base address
> - tamperNumber – The IRTC tamper input to configure

> • tamperConfig – The IRTC tamper properties

uint8_t IRTC_ReadTamperQueue(RTC_Type *base, *irtc_datetime_t* *tamperTimestamp)

> This function reads the tamper timestamp and returns the associated tamper pin.

> The tamper timestamp has month, day, hour, minutes, and seconds. Ignore the year field as this information is not available in the tamper queue. The user should look at the RTC_YEARMON register for this because the expectation is that the queue is read at least once a year. Return the tamper pin number associated with the timestamp.

> **Parameters**
>
> > • base – The IRTC peripheral base address
> >
> > • tamperTimestamp – The tamper timestamp
>
> **Returns**
> > The tamper pin number

static inline bool IRTC_GetTamperQueueFullStatus(RTC_Type *base)

> Gets the IRTC Tamper queue full status.

> **Parameters**
>
> > • base – IRTC peripheral base address
>
> **Return values**
>
> > • true – Tamper queue is full.
> >
> > • false – Tamper queue is not full.

static inline void IRTC_ClearTamperQueueFullStatus(RTC_Type *base)

> Clear the IRTC Tamper queue full status.

> **Parameters**
>
> > • base – IRTC peripheral base address

FSL_IRTC_DRIVER_VERSION

enum _irtc_filter_clock_source

> IRTC filter clock source options.

> *Values:*

> enumerator kIRTC_32K
> > Use 32 kHz clock source for the tamper filter.

> enumerator kIRTC_512
> > Use 512 Hz clock source for the tamper filter.

> enumerator kIRTC_128
> > Use 128 Hz clock source for the tamper filter.

> enumerator kIRTC_64
> > Use 64 Hz clock source for the tamper filter.

> enumerator kIRTC_16
> > Use 16 Hz clock source for the tamper filter.

> enumerator kIRTC_8
> > Use 8 Hz clock source for the tamper filter.

> enumerator kIRTC_4
> > Use 4 Hz clock source for the tamper filter.

enumerator kIRTC_2
    Use 2 Hz clock source for the tamper filter.

enum _irtc_tamper_pins
    IRTC Tamper pins.

    Values:

    enumerator kIRTC_Tamper_0
        External Tamper 0

    enumerator kIRTC_Tamper_1
        External Tamper 1

    enumerator kIRTC_Tamper_2
        External Tamper 2

    enumerator kIRTC_Tamper_3
        Internal tamper, does not have filter configuration

enum _irtc_interrupt_enable
    List of IRTC interrupts.

    Values:

    enumerator kIRTC_TamperInterruptEnable
        Tamper Interrupt Enable

    enumerator kIRTC_AlarmInterruptEnable
        Alarm Interrupt Enable

    enumerator kIRTC_DayInterruptEnable
        Days Interrupt Enable

    enumerator kIRTC_HourInterruptEnable
        Hours Interrupt Enable

    enumerator kIRTC_MinInterruptEnable
        Minutes Interrupt Enable

    enumerator kIRTC_1hzInterruptEnable
        1 Hz interval Interrupt Enable

    enumerator kIRTC_2hzInterruptEnable
        2 Hz interval Interrupt Enable

    enumerator kIRTC_4hzInterruptEnable
        4 Hz interval Interrupt Enable

    enumerator kIRTC_8hzInterruptEnable
        8 Hz interval Interrupt Enable

    enumerator kIRTC_16hzInterruptEnable
        16 Hz interval Interrupt Enable

    enumerator kIRTC_32hzInterruptEnable
        32 Hz interval Interrupt Enable

    enumerator kIRTC_64hzInterruptEnable
        64 Hz interval Interrupt Enable

    enumerator kIRTC_128hzInterruptEnable
        128 Hz interval Interrupt Enable

enumerator kIRTC_256hzInterruptEnable
    256 Hz interval Interrupt Enable

enumerator kIRTC_512hzInterruptEnable
    512 Hz interval Interrupt Enable

enumerator kIRTC_TamperQueueFullInterruptEnable
    Tamper queue full Interrupt Enable

enum _irtc_status_flags
    List of IRTC flags.

    Values:

    enumerator kIRTC_TamperFlag
        Tamper Status flag

    enumerator kIRTC_AlarmFlag
        Alarm Status flag

    enumerator kIRTC_DayFlag
        Days Status flag

    enumerator kIRTC_HourFlag
        Hour Status flag

    enumerator kIRTC_MinFlag
        Minutes Status flag

    enumerator kIRTC_1hzFlag
        1 Hz interval status flag

    enumerator kIRTC_2hzFlag
        2 Hz interval status flag

    enumerator kIRTC_4hzFlag
        4 Hz interval status flag

    enumerator kIRTC_8hzFlag
        8 Hz interval status flag

    enumerator kIRTC_16hzFlag
        16 Hz interval status flag

    enumerator kIRTC_32hzFlag
        32 Hz interval status flag

    enumerator kIRTC_64hzFlag
        64 Hz interval status flag

    enumerator kIRTC_128hzFlag
        128 Hz interval status flag

    enumerator kIRTC_256hzFlag
        256 Hz interval status flag

    enumerator kIRTC_512hzFlag
        512 Hz interval status flag

    enumerator kIRTC_InvalidFlag
        Indicates if time/date counters are invalid

enumerator kIRTC_WriteProtFlag

Write protect enable status flag

enumerator kIRTC_CpuLowVoltFlag

CPU low voltage warning flag

enumerator kIRTC_ResetSrcFlag

Reset source flag

enumerator kIRTC_CmpIntFlag

Compensation interval status flag

enumerator kIRTC_CmpDoneFlag

Compensation done flag

enumerator kIRTC_BusErrFlag

Bus error flag

enum _irtc_alarm_match

IRTC alarm match options.

Values:

enumerator kRTC_MatchSecMinHr

Only match second, minute and hour

enumerator kRTC_MatchSecMinHrDay

Only match second, minute, hour and day

enumerator kRTC_MatchSecMinHrDayMnth

Only match second, minute, hour, day and month

enumerator kRTC_MatchSecMinHrDayMnthYr

Only match second, minute, hour, day, month and year

enum _irtc_osc_cap_load

List of RTC Oscillator capacitor load settings.

Values:

enumerator kIRTC_Capacitor2p

2pF capacitor load

enumerator kIRTC_Capacitor4p

4pF capacitor load

enumerator kIRTC_Capacitor8p

8pF capacitor load

enumerator kIRTC_Capacitor16p

16pF capacitor load

enum _irtc_clockout_sel

IRTC clockout select.

Values:

enumerator kIRTC_ClkoutNo

No clock out

enumerator kIRTC_ClkoutFine1Hz

clock out fine 1Hz

enumerator kIRTC_Clkout32kHz
    clock out 32.768kHz

enumerator kIRTC_ClkoutCoarse1Hz
    clock out coarse 1Hz

typedef enum _irtc_filter_clock_source irtc_filter_clock_source_t
    IRTC filter clock source options.

typedef enum _irtc_tamper_pins irtc_tamper_pins_t
    IRTC Tamper pins.

typedef enum _irtc_interrupt_enable irtc_interrupt_enable_t
    List of IRTC interrupts.

typedef enum _irtc_status_flags irtc_status_flags_t
    List of IRTC flags.

typedef enum _irtc_alarm_match irtc_alarm_match_t
    IRTC alarm match options.

typedef enum _irtc_osc_cap_load irtc_osc_cap_load_t
    List of RTC Oscillator capacitor load settings.

typedef enum _irtc_clockout_sel irtc_clockout_sel_t
    IRTC clockout select.

typedef struct _irtc_datetime irtc_datetime_t
    Structure is used to hold the date and time.

typedef struct _irtc_daylight_time irtc_daylight_time_t
    Structure is used to hold the daylight saving time.

typedef struct _irtc_tamper_config irtc_tamper_config_t
    Structure is used to define the parameters to configure a RTC tamper event.

typedef struct _irtc_config irtc_config_t
    RTC config structure.

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

    The config struct can be made const so it resides in flash

static inline void IRTC_SetOscCapLoad(RTC_Type *base, uint16_t capLoad)
    This function sets the specified capacitor configuration for the RTC oscillator.

        Parameters

            • base – IRTC peripheral base address

            • capLoad – Oscillator loads to enable. This is a logical OR of members of the enumeration irtc_osc_cap_load_t

status_t IRTC_SetWriteProtection(RTC_Type *base, bool lock)
    Locks or unlocks IRTC registers for write access.

Note: When the registers are unlocked, they remain in unlocked state for 2 seconds, after which they are locked automatically. After power-on-reset, the registers come out unlocked and they are locked automatically 15 seconds after power on.

        Parameters

- base – IRTC peripheral base address

- lock – true: Lock IRTC registers; false: Unlock IRTC registers.

>   **Returns**
>   kStatus_Success: if lock or unlock operation is successful kStatus_Fail: if lock
>   or unlock operation fails even after multiple retry attempts

static inline void IRTC_Reset(RTC_Type *base)

>   Performs a software reset on the IRTC module.

>   Clears contents of alarm, interrupt (status and enable except tamper interrupt enable bit)
>   registers, STATUS[CMP_DONE] and STATUS[BUS_ERR]. This has no effect on DST, calendaring, standby time and tamper detect registers.

>   **Parameters**
>   - base – IRTC peripheral base address

static inline void IRTC_Enable32kClkDuringRegisterWrite(RTC_Type *base, bool enable)

>   Enable/disable 32 kHz RTC OSC clock during RTC register write.

>   **Parameters**
>   - base – IRTC peripheral base address

>   - enable – Enable/disable 32 kHz RTC OSC clock.

>   - true: Enables the oscillator.

>   - false: Disables the oscillator.

void IRTC_ConfigClockOut(RTC_Type *base, *irtc_clockout_sel_t* clkOut)

>   Select which clock to output from RTC.

>   Select which clock to output from RTC for other modules to use inside SoC, for example,
>   RTC subsystem needs RTC to output 1HZ clock for sub-second counter.

>   **Parameters**
>   - base – IRTC peripheral base address

>   - clkOut – select clock to use for output,

static inline uint8_t IRTC_GetTamperStatusFlag(RTC_Type *base)

>   Gets the IRTC Tamper status flags.

>   **Parameters**
>   - base – IRTC peripheral base address

>   **Returns**
>   The Tamper status value.

static inline void IRTC_ClearTamperStatusFlag(RTC_Type *base)

>   Gets the IRTC Tamper status flags.

>   **Parameters**
>   - base – IRTC peripheral base address

static inline void IRTC_SetTamperConfigurationOver(RTC_Type *base)

>   Set tamper configuration over.

>   Note that this API is neeeded after call IRTC_SetTamperParams to configure tamper events
>   to notify IRTC module that tamper configuration process is over.

>   **Parameters**
>   - base – IRTC peripheral base address

IRTC_STATUS_W1C_BITS

struct _irtc_datetime
    #include <fsl_irtc.h> Structure is used to hold the date and time.

Public Members

uint16_t year
    Range from 1984 to 2239.

uint8_t month
    Range from 1 to 12.

uint8_t day
    Range from 1 to 31 (depending on month).

uint8_t weekDay
    Range from 0(Sunday) to 6(Saturday).

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 _irtc_daylight_time
    #include <fsl_irtc.h> Structure is used to hold the daylight saving time.

Public Members

uint8_t startMonth
    Range from 1 to 12

uint8_t endMonth
    Range from 1 to 12

uint8_t startDay
    Range from 1 to 31 (depending on month)

uint8_t endDay
    Range from 1 to 31 (depending on month)

uint8_t startHour
    Range from 0 to 23

uint8_t endHour
    Range from 0 to 23

struct _irtc_tamper_config
    #include <fsl_irtc.h> Structure is used to define the parameters to configure a RTC tamper
    event.

Public Members

bool activePassive

true: configure tamper as active; false: passive tamper

bool direction

true: configure tamper direction as output; false: configure as input; this is only used if a tamper pin is defined as active

bool pinPolarity

true: tamper has active low polarity; false: active high polarity

irtc_filter_clock_source_t filterClk

Clock source for the tamper filter

uint8_t filterDuration

Tamper filter duration.

struct __irtc_config

#include <fsl_irtc.h> RTC config structure.

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

The config struct can be made const so it resides in flash

Public Members

bool wakeupSelect

true: Tamper pin 0 is used to wakeup the chip; false: Tamper pin 0 is used as the tamper pin

bool timerStdMask

true: Sampling clocks gated in standby mode; false: Sampling clocks not gated

irtc_alarm_match_t alrmMatch

Pick one option from enumeration :: irtc_alarm_match_t

2.24 Common Driver

FSL_COMMON_DRIVER_VERSION

common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE

No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART

Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART

Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI

Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM
    Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART
    Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART
    Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART
    Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO
    Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI
    Debug console based on QSCI.

MIN(a, b)
    Computes the minimum of *a* and *b*.

MAX(a, b)
    Computes the maximum of *a* and *b*.

UINT16_MAX
    Max value of uint16_t type.

UINT32_MAX
    Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(addr, val)
    Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(addr, val)
    Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(addr, bits)
    Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(addr, bits)
    Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(addr, bits)
    Toggle the bits specified by *bits* to the variable at address *address*.

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

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

SDK_ATOMIC_LOCAL_TEST_AND_SET(addr, newValue)
    For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(us, clockFreqInHz)
    Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(count, clockFreqInHz)
    Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(ms, clockFreqInHz)

> Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(count, clockFreqInHz)

> Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_SIZEALIGN(var, alignbytes)

> Macro to define a variable with L1 d-cache line size alignment

> Macro to define a variable with L2 cache line size alignment

> Macro to change a value to a given size aligned value

AT_NONCACHEABLE_SECTION(var)

> Define a variable *var,* and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

> Define a variable *var,* and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes.*

AT_NONCACHEABLE_SECTION_INIT(var)

> Define a variable *var* with initial value, and place it in non-cacheable section.

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)

> Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes.*

enum __status_groups

> Status group numbers.

> *Values:*

> enumerator kStatusGroup_Generic
>
>> Group number for generic status codes.

> enumerator kStatusGroup_FLASH
>
>> Group number for FLASH status codes.

> enumerator kStatusGroup_LPSPI
>
>> Group number for LPSPI status codes.

> enumerator kStatusGroup_FLEXIO_SPI
>
>> Group number for FLEXIO SPI status codes.

> enumerator kStatusGroup_DSPI
>
>> Group number for DSPI status codes.

> enumerator kStatusGroup_FLEXIO_UART
>
>> Group number for FLEXIO UART status codes.

> enumerator kStatusGroup_FLEXIO_I2C
>
>> Group number for FLEXIO I2C status codes.

> enumerator kStatusGroup_LPI2C
>
>> Group number for LPI2C status codes.

> enumerator kStatusGroup_UART
>
>> Group number for UART status codes.

> enumerator kStatusGroup_I2C
>
>> Group number for UART status codes.

enumerator kStatusGroup_LPSCI
    Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART
    Group number for LPUART status codes.

enumerator kStatusGroup_SPI
    Group number for SPI status code.

enumerator kStatusGroup_XRDC
    Group number for XRDC status code.

enumerator kStatusGroup_SEMA42
    Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC
    Group number for SDHC status code

enumerator kStatusGroup_SDMMC
    Group number for SDMMC status code

enumerator kStatusGroup_SAI
    Group number for SAI status code

enumerator kStatusGroup_MCG
    Group number for MCG status codes.

enumerator kStatusGroup_SCG
    Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
    Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
    Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
    Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
    Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
    Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
    Group number for I2S status codes

enumerator kStatusGroup_IUART
    Group number for IUART status codes

enumerator kStatusGroup_CSI
    Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
    Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
    Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
    Group number for POWER status codes.

enumerator kStatusGroup_ENET
> Group number for ENET status codes.

enumerator kStatusGroup_PHY
> Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
> Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
> Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
> Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
> Group number for QSPI status codes.

enumerator kStatusGroup_DMA
> Group number for DMA status codes.

enumerator kStatusGroup_EDMA
> Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
> Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
> Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
> Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
> Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
> Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
> Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
> Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
> Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
> Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
> Group number for OTP status codes.

enumerator kStatusGroup_MCAN
> Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
> Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
> Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
> Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
> Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
> Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
> Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
> Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
> Group number for FLEXSPI status codes.

enumerator kStatusGroup_MMDC
> Group number for MMDC status codes.

enumerator kStatusGroup_PDM
> Group number for MIC status codes.

enumerator kStatusGroup_SDMA
> Group number for SDMA status codes.

enumerator kStatusGroup_ICS
> Group number for ICS status codes.

enumerator kStatusGroup_SPDIF
> Group number for SPDIF status codes.

enumerator kStatusGroup_LPC_MINISPI
> Group number for LPC_MINISPI status codes.

enumerator kStatusGroup_HASHCRYPT
> Group number for Hashcrypt status codes

enumerator kStatusGroup_LPC_SPI_SSP
> Group number for LPC_SPI_SSP status codes.

enumerator kStatusGroup_I3C
> Group number for I3C status codes

enumerator kStatusGroup_LPC_I2C_1
> Group number for LPC_I2C_1 status codes.

enumerator kStatusGroup_NOTIFIER
> Group number for NOTIFIER status codes.

enumerator kStatusGroup_DebugConsole
> Group number for debug console status codes.

enumerator kStatusGroup_SEMC
> Group number for SEMC status codes.

enumerator kStatusGroup_ApplicationRangeStart
> Starting number for application groups.

enumerator kStatusGroup_IAP
> Group number for IAP status codes

enumerator kStatusGroup_SFA
    Group number for SFA status codes

enumerator kStatusGroup_SPC
    Group number for SPC status codes.

enumerator kStatusGroup_PUF
    Group number for PUF status codes.

enumerator kStatusGroup_TOUCH_PANEL
    Group number for touch panel status codes

enumerator kStatusGroup_VBAT
    Group number for VBAT status codes

enumerator kStatusGroup_XSPI
    Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
    Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
    Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
    Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
    Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
    Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
    Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
    Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
    Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
    Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
    Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
    Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
    Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
    Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
    Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
    Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
  Group number for LED status codes.

enumerator kStatusGroup_BUTTON
  Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
  Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
  Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
  Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
  Group number for List status codes.

enumerator kStatusGroup_OSA
  Group number for OSA status codes.

enumerator kStatusGroup_COMMON_TASK
  Group number for Common task status codes.

enumerator kStatusGroup_MSG
  Group number for messaging status codes.

enumerator kStatusGroup_SDK_OCOTP
  Group number for OCOTP status codes.

enumerator kStatusGroup_SDK_FLEXSPINOR
  Group number for FLEXSPINOR status codes.

enumerator kStatusGroup_CODEC
  Group number for codec status codes.

enumerator kStatusGroup_ASRC
  Group number for codec status ASRC.

enumerator kStatusGroup_OTFAD
  Group number for codec status codes.

enumerator kStatusGroup_SDIOSLV
  Group number for SDIOSLV status codes.

enumerator kStatusGroup_MECC
  Group number for MECC status codes.

enumerator kStatusGroup_ENET_QOS
  Group number for ENET_QOS status codes.

enumerator kStatusGroup_LOG
  Group number for LOG status codes.

enumerator kStatusGroup_I3CBUS
  Group number for I3CBUS status codes.

enumerator kStatusGroup_QSCI
  Group number for QSCI status codes.

enumerator kStatusGroup_ELEMU
  Group number for ELEMU status codes.

enumerator kStatusGroup_QUEUEDSPI
    Group number for QSPI status codes.

enumerator kStatusGroup_POWER_MANAGER
    Group number for POWER_MANAGER status codes.

enumerator kStatusGroup_IPED
    Group number for IPED status codes.

enumerator kStatusGroup_ELS_PKC
    Group number for ELS PKC status codes.

enumerator kStatusGroup_CSS_PKC
    Group number for CSS PKC status codes.

enumerator kStatusGroup_HOSTIF
    Group number for HOSTIF status codes.

enumerator kStatusGroup_CLIF
    Group number for CLIF status codes.

enumerator kStatusGroup_BMA
    Group number for BMA status codes.

enumerator kStatusGroup_NETC
    Group number for NETC status codes.

enumerator kStatusGroup_ELE
    Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
    Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
    Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
    Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
    Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
    Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
    Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
    Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
    Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
    Group number for A-format status codes.

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 wil be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, "SUPPRESS_FALL_THROUGH_WARNING();" need to be added at the end of each case section which misses "break;"statement.

MSDK_REG_SECURE_ADDR(x)

> Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

> Convert the register address to the one used in non-secure mode.

MSDK_INVALID_IRQ_HANDLER

> Invalid IRQ handler address.

2.25 LLWU: Low-Leakage Wakeup Unit Driver

static inline void LLWU_GetVersionId(LLWU_Type *base, *llwu_version_id_t* *versionId)

> Gets the LLWU version ID.
>
> This function gets the LLWU version ID, including the major version number, the minor version number, and the feature specification number.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> > - versionId – A pointer to the version ID structure.

static inline void LLWU_GetParam(LLWU_Type *base, *llwu_param_t* *param)

> Gets the LLWU parameter.
>
> This function gets the LLWU parameter, including a wakeup pin number, a module number, a DMA number, and a pin filter number.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> > - param – A pointer to the LLWU parameter structure.

void LLWU_SetExternalWakeupPinMode(LLWU_Type *base, uint32_t pinIndex,
                                    llwu_external_pin_mode_t pinMode)

> Sets the external input pin source mode.
>
> This function sets the external input pin source mode that is used as a wake up source.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> >
> > - pinIndex – A pin index to be enabled as an external wakeup source starting from 1.
> >
> > - pinMode – A pin configuration mode defined in the llwu_external_pin_modes_t.

bool LLWU_GetExternalWakeupPinFlag(LLWU_Type *base, uint32_t pinIndex)

> Gets the external wakeup source flag.
>
> This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> >
> > - pinIndex – A pin index, which starts from 1.
>
> > **Returns**
> >
> > True if the specific pin is a wakeup source.

void LLWU_ClearExternalWakeupPinFlag(LLWU_Type *base, uint32_t pinIndex)

> Clears the external wakeup source flag.
>
> This function clears the external wakeup source flag for a specific pin.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> >
> > - pinIndex – A pin index, which starts from 1.

static inline void LLWU_EnableInternalModuleInterruptWakup(LLWU_Type *base, uint32_t
                                    moduleIndex, bool enable)

> Enables/disables the internal module source.
>
> This function enables/disables the internal module source mode that is used as a wake up source.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> >
> > - moduleIndex – A module index to be enabled as an internal wakeup source starting from 1.
> >
> > - enable – An enable or a disable setting

static inline void LLWU_EnableInternalModuleDmaRequestWakup(LLWU_Type *base, uint32_t
                                    moduleIndex, bool enable)

> Enables/disables the internal module DMA wakeup source.
>
> This function enables/disables the internal DMA that is used as a wake up source.
>
> > **Parameters**
> >
> > - base – LLWU peripheral base address.
> >
> > - moduleIndex – An internal module index which is used as a DMA request source, starting from 1.

- enable – Enable or disable the DMA request source

void LLWU_SetPinFilterMode(LLWU_Type *base, uint32_t filterIndex,
                                        llwu_external_pin_filter_mode_t filterMode)

Sets the pin filter configuration.

This function sets the pin filter configuration.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index used to enable/disable the digital filter, starting from 1.
- filterMode – A filter mode configuration

bool LLWU_GetPinFilterFlag(LLWU_Type *base, uint32_t filterIndex)

Gets the pin filter configuration.

This function gets the pin filter flag.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

void LLWU_ClearPinFilterFlag(LLWU_Type *base, uint32_t filterIndex)

Clears the pin filter configuration.

This function clears the pin filter flag.

Parameters

- base – LLWU peripheral base address.
- filterIndex – A pin filter index to clear the flag, starting from 1.

void LLWU_SetResetPinMode(LLWU_Type *base, bool pinEnable, bool pinFilterEnable)

Sets the reset pin mode.

This function determines how the reset pin is used as a low leakage mode exit source.

Parameters

- base – LLWU peripheral base address.
- pinEnable – Enable reset the pin filter
- pinFilterEnable – Specify whether the pin filter is enabled in Low-Leakage power mode.

FSL_LLWU_DRIVER_VERSION

LLWU driver version.

enum __llwu_external_pin_mode

External input pin control modes.

Values:

enumerator kLLWU_ExternalPinDisable

Pin disabled as a wakeup input.

enumerator kLLWU_ExternalPinRisingEdge

Pin enabled with the rising edge detection.

2.25. LLWU: Low-Leakage Wakeup Unit Driver   259

enumerator kLLWU_ExternalPinFallingEdge
Pin enabled with the falling edge detection.

enumerator kLLWU_ExternalPinAnyEdge
Pin enabled with any change detection.

enum __llwu_pin_filter_mode
Digital filter control modes.

Values:

enumerator kLLWU_PinFilterDisable
Filter disabled.

enumerator kLLWU_PinFilterRisingEdge
Filter positive edge detection.

enumerator kLLWU_PinFilterFallingEdge
Filter negative edge detection.

enumerator kLLWU_PinFilterAnyEdge
Filter any edge detection.

typedef enum *_llwu_external_pin_mode* llwu_external_pin_mode_t
External input pin control modes.

typedef enum *_llwu_pin_filter_mode* llwu_pin_filter_mode_t
Digital filter control modes.

typedef struct *_llwu_version_id* llwu_version_id_t
IP version ID definition.

typedef struct *_llwu_param* llwu_param_t
IP parameter definition.

typedef struct *_llwu_external_pin_filter_mode* llwu_external_pin_filter_mode_t
An external input pin filter control structure.

LLWU_REG_VAL(x)

struct __llwu_version_id
#include <fsl_llwu.h> IP version ID definition.

Public Members

uint16_t feature
A feature specification number.

uint8_t minor
The minor version number.

uint8_t major
The major version number.

struct __llwu_param
#include <fsl_llwu.h> IP parameter definition.

Public Members

uint8_t filters
> A number of the pin filter.

uint8_t dmas
> A number of the wakeup DMA.

uint8_t modules
> A number of the wakeup module.

uint8_t pins
> A number of the wake up pin.

struct __llwu__external__pin__filter__mode
> *#include <fsl_llwu.h>* An external input pin filter control structure.

Public Members

uint32_t pinIndex
> A pin number

llwu_pin_filter_mode_t filterMode
> Filter mode

2.26 LPTMR: Low-Power Timer

void LPTMR_Init(LPTMR_Type *base, const *lptmr_config_t* *config)
> Ungates the LPTMR clock and configures the peripheral for a basic operation.

> **Note:** This API should be called at the beginning of the application using the LPTMR driver.

> **Parameters**
> - base – LPTMR peripheral base address
> - config – A pointer to the LPTMR configuration structure.

void LPTMR_Deinit(LPTMR_Type *base)
> Gates the LPTMR clock.
> **Parameters**
> - base – LPTMR peripheral base address

void LPTMR_GetDefaultConfig(*lptmr_config_t* *config)
> Fills in the LPTMR configuration structure with default settings.

> The default values are as follows.

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

- config – A pointer to the LPTMR configuration structure.

static inline void LPTMR_EnableInterrupts(LPTMR_Type *base, uint32_t mask)

Enables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t

static inline void LPTMR_DisableInterrupts(LPTMR_Type *base, uint32_t mask)

Disables the selected LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

- mask – The interrupts to disable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t.

static inline uint32_t LPTMR_GetEnabledInterrupts(LPTMR_Type *base)

Gets the enabled LPTMR interrupts.

Parameters

- base – LPTMR peripheral base address

Returns

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

static inline uint32_t LPTMR_GetStatusFlags(LPTMR_Type *base)

Gets the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address

Returns

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

static inline void LPTMR_ClearStatusFlags(LPTMR_Type *base, uint32_t mask)

Clears the LPTMR status flags.

Parameters

- base – LPTMR peripheral base address

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

static inline void LPTMR_SetTimerPeriod(LPTMR_Type *base, uint32_t ticks)

Sets the timer period in units of count.

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

Note:

a. The TCF flag is set with the CNR equals the count provided here and then increments.

b. Call the utility macros provided in the fsl_common.h to convert to ticks.

Parameters

- base – LPTMR peripheral base address
- ticks – A timer period in units of ticks

static inline uint32_t LPTMR_GetCurrentTimerCount(LPTMR_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 – LPTMR peripheral base address

Returns

The current counter value in ticks

static inline void LPTMR_StartTimer(LPTMR_Type *base)

Starts the timer.

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

- base – LPTMR peripheral base address

static inline void LPTMR_StopTimer(LPTMR_Type *base)

Stops the timer.

This function stops the timer and resets the timer's counter register.

Parameters

- base – LPTMR peripheral base address

FSL_LPTMR_DRIVER_VERSION

Driver Version

enum _lptmr_pin_select

LPTMR pin selection used in pulse counter mode.

Values:

enumerator kLPTMR_PinSelectInput_0

Pulse counter input 0 is selected

enumerator kLPTMR_PinSelectInput_1

Pulse counter input 1 is selected

enumerator kLPTMR_PinSelectInput_2

Pulse counter input 2 is selected

enumerator kLPTMR_PinSelectInput_3

Pulse counter input 3 is selected

enum _lptmr_pin_polarity

LPTMR pin polarity used in pulse counter mode.

Values:

---

enumerator kLPTMR_PinPolarityActiveHigh
> Pulse Counter input source is active-high

enumerator kLPTMR_PinPolarityActiveLow
> Pulse Counter input source is active-low

enum __lptmr_timer_mode
> LPTMR timer mode selection.
>
> *Values:*

enumerator kLPTMR_TimerModeTimeCounter
> Time Counter mode

enumerator kLPTMR_TimerModePulseCounter
> Pulse Counter mode

enum __lptmr_prescaler_glitch_value
> LPTMR prescaler/glitch filter values.
>
> *Values:*

enumerator kLPTMR_Prescale_Glitch_0
> Prescaler divide 2, glitch filter does not support this setting

enumerator kLPTMR_Prescale_Glitch_1
> Prescaler divide 4, glitch filter 2

enumerator kLPTMR_Prescale_Glitch_2
> Prescaler divide 8, glitch filter 4

enumerator kLPTMR_Prescale_Glitch_3
> Prescaler divide 16, glitch filter 8

enumerator kLPTMR_Prescale_Glitch_4
> Prescaler divide 32, glitch filter 16

enumerator kLPTMR_Prescale_Glitch_5
> Prescaler divide 64, glitch filter 32

enumerator kLPTMR_Prescale_Glitch_6
> Prescaler divide 128, glitch filter 64

enumerator kLPTMR_Prescale_Glitch_7
> Prescaler divide 256, glitch filter 128

enumerator kLPTMR_Prescale_Glitch_8
> Prescaler divide 512, glitch filter 256

enumerator kLPTMR_Prescale_Glitch_9
> Prescaler divide 1024, glitch filter 512

enumerator kLPTMR_Prescale_Glitch_10
> Prescaler divide 2048 glitch filter 1024

enumerator kLPTMR_Prescale_Glitch_11
> Prescaler divide 4096, glitch filter 2048

enumerator kLPTMR_Prescale_Glitch_12
> Prescaler divide 8192, glitch filter 4096

enumerator kLPTMR_Prescale_Glitch_13
> Prescaler divide 16384, glitch filter 8192

enumerator kLPTMR_Prescale_Glitch_14
    Prescaler divide 32768, glitch filter 16384

enumerator kLPTMR_Prescale_Glitch_15
    Prescaler divide 65536, glitch filter 32768

enum _lptmr_prescaler_clock_select
    LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

Values:

enumerator kLPTMR_PrescalerClock_0
    Prescaler/glitch filter clock 0 selected.

enumerator kLPTMR_PrescalerClock_1
    Prescaler/glitch filter clock 1 selected.

enumerator kLPTMR_PrescalerClock_2
    Prescaler/glitch filter clock 2 selected.

enumerator kLPTMR_PrescalerClock_3
    Prescaler/glitch filter clock 3 selected.

enum _lptmr_interrupt_enable
    List of the LPTMR interrupts.
    Values:

enumerator kLPTMR_TimerInterruptEnable
    Timer interrupt enable

enum _lptmr_status_flags
    List of the LPTMR status flags.
    Values:

enumerator kLPTMR_TimerCompareFlag
    Timer compare flag

typedef enum *_lptmr_pin_select* lptmr_pin_select_t
    LPTMR pin selection used in pulse counter mode.

typedef enum *_lptmr_pin_polarity* lptmr_pin_polarity_t
    LPTMR pin polarity used in pulse counter mode.

typedef enum *_lptmr_timer_mode* lptmr_timer_mode_t
    LPTMR timer mode selection.

typedef enum *_lptmr_prescaler_glitch_value* lptmr_prescaler_glitch_value_t
    LPTMR prescaler/glitch filter values.

typedef enum *_lptmr_prescaler_clock_select* lptmr_prescaler_clock_select_t
    LPTMR prescaler/glitch filter clock select.

Note: Clock connections are SoC-specific

typedef enum *_lptmr_interrupt_enable* lptmr_interrupt_enable_t
    List of the LPTMR interrupts.

typedef enum *_lptmr_status_flags* lptmr_status_flags_t
> List of the LPTMR status flags.

typedef struct *_lptmr_config* lptmr_config_t
> LPTMR config structure.

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

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

static inline void LPTMR_EnableTimerDMA(LPTMR_Type *base, bool enable)
> Enable or disable timer DMA request.

> #### Parameters
> - base – base LPTMR peripheral base address
> - enable – Switcher of timer DMA feature. "true" means to enable, "false" means to disable.

struct _lptmr_config
> #include <fsl_lptmr.h> LPTMR config structure.

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

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

Public Members

lptmr_timer_mode_t timerMode
> Time counter mode or pulse counter mode

lptmr_pin_select_t pinSelect
> LPTMR pulse input pin select; used only in pulse counter mode

lptmr_pin_polarity_t pinPolarity
> LPTMR pulse input pin polarity; used only in pulse counter mode

bool enableFreeRunning
> True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set

bool bypassPrescaler
> True: bypass prescaler; false: use clock from prescaler

lptmr_prescaler_clock_select_t prescalerClockSource
> LPTMR clock source

lptmr_prescaler_glitch_value_t value
> Prescaler or glitch filter value

2.27 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_ClearCacheWriteBufferErroStatus(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.

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.

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_attribut* attribute_memory

struct __mcm_buffer_fault_attribut

#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_attribut* attribute_memory

struct __mcm__lmem__fault__attribut

#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.

2.28   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

2.29 PMC: Power Management Controller

static inline void PMC_GetVersionId(PMC_Type *base, *pmc_version_id_t* *versionId)

> Gets the PMC version ID.

> This function gets the PMC version ID, including major version number, minor version number, and a feature specification number.

> > **Parameters**

> > > • base – PMC peripheral base address.

> > > • versionId – Pointer to version ID structure.

void PMC_GetParam(PMC_Type *base, *pmc_param_t* *param)

> Gets the PMC parameter.

> This function gets the PMC parameter including the VLPO enable and the HVD enable.

> > **Parameters**

> > > • base – PMC peripheral base address.

> > > • param – Pointer to PMC param structure.

void PMC_ConfigureLowVoltDetect(PMC_Type *base, const *pmc_low_volt_detect_config_t* *config)

> Configures the low-voltage detect setting.

> This function configures the low-voltage detect setting, including the trip point voltage setting, enables or disables the interrupt, enables or disables the system reset.

> > **Parameters**

> > > • base – PMC peripheral base address.

> > > • config – Low-voltage detect configuration structure.

static inline bool PMC_GetLowVoltDetectFlag(PMC_Type *base)

> Gets the Low-voltage Detect Flag status.

> This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

> > **Parameters**

> > > • base – PMC peripheral base address.

> > **Returns**

> > > Current low-voltage detect flag

> > > • true: Low-voltage detected

> > > • false: Low-voltage not detected

static inline void PMC_ClearLowVoltDetectFlag(PMC_Type *base)

> Acknowledges clearing the Low-voltage Detect flag.

> This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

> > **Parameters**

> > > • base – PMC peripheral base address.

void PMC_ConfigureLowVoltWarning(PMC_Type *base, const *pmc_low_volt_warning_config_t* *config)

> Configures the low-voltage warning setting.

> This function configures the low-voltage warning setting, including the trip point voltage setting and enabling or disabling the interrupt.

> > **Parameters**

> > > • base – PMC peripheral base address.

- config – Low-voltage warning configuration structure.

static inline bool PMC_GetLowVoltWarningFlag(PMC_Type *base)

Gets the Low-voltage Warning Flag status.

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

- base – PMC peripheral base address.

Returns

Current LVWF status

- true: Low-voltage Warning Flag is set.

- false: the Low-voltage Warning does not happen.

static inline void PMC_ClearLowVoltWarningFlag(PMC_Type *base)

Acknowledges the Low-voltage Warning flag.

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

- base – PMC peripheral base address.

void PMC_ConfigureHighVoltDetect(PMC_Type *base, const *pmc_high_volt_detect_config_t* *config)

Configures the high-voltage detect setting.

This function configures the high-voltage detect setting, including the trip point voltage setting, enabling or disabling the interrupt, enabling or disabling the system reset.

Parameters

- base – PMC peripheral base address.

- config – High-voltage detect configuration structure.

static inline bool PMC_GetHighVoltDetectFlag(PMC_Type *base)

Gets the High-voltage Detect Flag status.

This function reads the current HVDF status. If it returns 1, a low voltage event is detected.

Parameters

- base – PMC peripheral base address.

Returns

Current high-voltage detect flag

- true: High-voltage detected

- false: High-voltage not detected

static inline void PMC_ClearHighVoltDetectFlag(PMC_Type *base)

Acknowledges clearing the High-voltage Detect flag.

This function acknowledges the high-voltage detection errors (write 1 to clear HVDF).

Parameters

- base – PMC peripheral base address.

void PMC_ConfigureBandgapBuffer(PMC_Type *base, const *pmc_bandgap_buffer_config_t*
*config)

Configures the PMC bandgap.

This function configures the PMC bandgap, including the drive select and behavior in low-power mode.

Parameters

- base – PMC peripheral base address.

- config – Pointer to the configuration structure

static inline bool PMC_GetPeriphIOIsolationFlag(PMC_Type *base)

Gets the acknowledge Peripherals and I/O pads isolation flag.

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

- base – PMC peripheral base address.

- base – Base address for current PMC instance.

Returns

ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

static inline void PMC_ClearPeriphIOIsolationFlag(PMC_Type *base)

Acknowledges the isolation flag to Peripherals and I/O pads.

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

- base – PMC peripheral base address.

static inline bool PMC_IsRegulatorInRunRegulation(PMC_Type *base)

Gets the regulator regulation status.

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

Parameters

- base – PMC peripheral base address.

- base – Base address for current PMC instance.

Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

FSL_PMC_DRIVER_VERSION

PMC driver version.

Version 2.0.3.

enum __pmc_low_volt_detect_volt_select

Low-voltage Detect Voltage Select.

Values:

enumerator kPMC_LowVoltDetectLowTrip

Low-trip point selected (VLVD = VLVDL)

enumerator kPMC_LowVoltDetectHighTrip
    High-trip point selected (VLVD = VLVDH)

enum __pmc_low_volt_warning_volt_select
    Low-voltage Warning Voltage Select.

    Values:

    enumerator kPMC_LowVoltWarningLowTrip
        Low-trip point selected (VLVW = VLVW1)

    enumerator kPMC_LowVoltWarningMid1Trip
        Mid 1 trip point selected (VLVW = VLVW2)

    enumerator kPMC_LowVoltWarningMid2Trip
        Mid 2 trip point selected (VLVW = VLVW3)

    enumerator kPMC_LowVoltWarningHighTrip
        High-trip point selected (VLVW = VLVW4)

enum __pmc_high_volt_detect_volt_select
    High-voltage Detect Voltage Select.

    Values:

    enumerator kPMC_HighVoltDetectLowTrip
        Low-trip point selected (VHVD = VHVDL)

    enumerator kPMC_HighVoltDetectHighTrip
        High-trip point selected (VHVD = VHVDH)

enum __pmc_bandgap_buffer_drive_select
    Bandgap Buffer Drive Select.

    Values:

    enumerator kPMC_BandgapBufferDriveLow
        Low-drive.

    enumerator kPMC_BandgapBufferDriveHigh
        High-drive.

enum __pmc_vlp_freq_option
    VLPx Option.

    Values:

    enumerator kPMC_FreqRestrict
        Frequency is restricted in VLPx mode.

    enumerator kPMC_FreqUnrestrict
        Frequency is unrestricted in VLPx mode.

typedef enum *_pmc_low_volt_detect_volt_select* pmc_low_volt_detect_volt_select_t
    Low-voltage Detect Voltage Select.

typedef enum *_pmc_low_volt_warning_volt_select* pmc_low_volt_warning_volt_select_t
    Low-voltage Warning Voltage Select.

typedef enum *_pmc_high_volt_detect_volt_select* pmc_high_volt_detect_volt_select_t
    High-voltage Detect Voltage Select.

typedef enum *_pmc_bandgap_buffer_drive_select* pmc_bandgap_buffer_drive_select_t
    Bandgap Buffer Drive Select.

typedef enum *_pmc_vlp_freq_option* pmc_vlp_freq_mode_t
    VLPx Option.

typedef struct *_pmc_version_id* pmc_version_id_t
    IP version ID definition.

typedef struct *_pmc_param* pmc_param_t
    IP parameter definition.

typedef struct *_pmc_low_volt_detect_config* pmc_low_volt_detect_config_t
    Low-voltage Detect Configuration Structure.

typedef struct *_pmc_low_volt_warning_config* pmc_low_volt_warning_config_t
    Low-voltage Warning Configuration Structure.

typedef struct *_pmc_high_volt_detect_config* pmc_high_volt_detect_config_t
    High-voltage Detect Configuration Structure.

typedef struct *_pmc_bandgap_buffer_config* pmc_bandgap_buffer_config_t
    Bandgap Buffer configuration.

struct _pmc_version_id
    #include <fsl_pmc.h> IP version ID definition.

Public Members

uint16_t feature
    Feature Specification Number.

uint8_t minor
    Minor version number.

uint8_t major
    Major version number.

struct _pmc_param
    #include <fsl_pmc.h> IP parameter definition.

Public Members

bool vlpoEnable
    VLPO enable.

bool hvdEnable
    HVD enable.

struct _pmc_low_volt_detect_config
    #include <fsl_pmc.h> Low-voltage Detect Configuration Structure.

Public Members

bool enableInt
    Enable interrupt when Low-voltage detect

bool enableReset
    Enable system reset when Low-voltage detect

pmc_low_volt_detect_volt_select_t voltSelect
    Low-voltage detect trip point voltage selection

struct __pmc__low__volt__warning__config
    #include <fsl_pmc.h> Low-voltage Warning Configuration Structure.

Public Members

bool enableInt
    Enable interrupt when low-voltage warning

pmc_low_volt_warning_volt_select_t voltSelect
    Low-voltage warning trip point voltage selection

struct __pmc__high__volt__detect__config
    #include <fsl_pmc.h> High-voltage Detect Configuration Structure.

Public Members

bool enableInt
    Enable interrupt when high-voltage detect

bool enableReset
    Enable system reset when high-voltage detect

pmc_high_volt_detect_volt_select_t voltSelect
    High-voltage detect trip point voltage selection

struct __pmc__bandgap__buffer__config
    #include <fsl_pmc.h> Bandgap Buffer configuration.

Public Members

bool enable
    Enable bandgap buffer.

bool enableInLowPowerMode
    Enable bandgap buffer in low-power mode.

pmc_bandgap_buffer_drive_select_t drive
    Bandgap buffer drive select.

2.30 PORT: Port Control and Interrupts

static inline void PORT_SetPinConfig(PORT_Type *base, uint32_t pin, const *port_pin_config_t*
                                                    *config)
    Sets the port PCR register.

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

```
// Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.

- pin – PORT pin number.

- config – PORT PCR register configuration structure.

static inline void PORT_SetMultiplePinsConfig(PORT_Type *base, uint32_t mask, const
port_pin_config_t *config)

Sets the port PCR register for multiple pins.

This is an example to define input pins or output pins PCR configuration.

```
Define a digital input pin PCR configuration
port_pin_config_t config = {
    kPORT_PullUp ,
    kPORT_PullEnable,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnlockRegister,
};
```

Parameters

- base – PORT peripheral base pointer.

- mask – PORT pin number macro.

- config – PORT PCR register configuration structure.

static inline void PORT_SetMultipleInterruptPinsConfig(PORT_Type *base, uint32_t mask,
port_interrupt_t config)

Sets the port interrupt configuration in PCR register for multiple pins.

Parameters

- base – PORT peripheral base pointer.

- mask – PORT pin number macro.

- config – PORT pin interrupt configuration.

  – kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled.

  – kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).

  – kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).

  – kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).

  – kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).

  – kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).

  – kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).

  – kPORT_InterruptLogicZero : Interrupt when logic zero.

  – kPORT_InterruptRisingEdge : Interrupt on rising edge.

  – kPORT_InterruptFallingEdge: Interrupt on falling edge.

  – kPORT_InterruptEitherEdge : Interrupt on either edge.

– kPORT_InterruptLogicOne : Interrupt when logic one.

– kPORT_ActiveHighTriggerOutputEnable :   Enable active high-trigger output (if the trigger states exit).

– kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit)..

static inline void PORT_SetPinMux(PORT_Type *base, uint32_t pin, *port_mux_t* mux)

Configures the pin muxing.

Note: : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

---

Parameters

- base – PORT peripheral base pointer.

- pin – PORT pin number.

- mux – pin muxing slot selection.

  – kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.

  – kPORT_MuxAsGpio : Set as GPIO.

  – kPORT_MuxAlt2 : chip-specific.

  – kPORT_MuxAlt3 : chip-specific.

  – kPORT_MuxAlt4 : chip-specific.

  – kPORT_MuxAlt5 : chip-specific.

  – kPORT_MuxAlt6 : chip-specific.

  – kPORT_MuxAlt7 : chip-specific.

static inline void PORT_EnablePinsDigitalFilter(PORT_Type *base, uint32_t mask, bool enable)

Enables the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.

- mask – PORT pin number macro.

- enable – PORT digital filter configuration.

static inline void PORT_SetDigitalFilterConfig(PORT_Type *base, const *port_digital_filter_config_t* *config)

Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Parameters

- base – PORT peripheral base pointer.

- config – PORT digital filter configuration structure.

static inline void PORT_SetPinInterruptConfig(PORT_Type *base, uint32_t pin, *port_interrupt_t* config)

Configures the port pin interrupt/DMA request.

Parameters

- base – PORT peripheral base pointer.

- pin – PORT pin number.
- config – PORT pin interrupt configuration.
  - kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled.
  - kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit).
  - kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit).
  - kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit).
  - kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit).
  - kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit).
  - kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit).
  - kPORT_InterruptLogicZero : Interrupt when logic zero.
  - kPORT_InterruptRisingEdge : Interrupt on rising edge.
  - kPORT_InterruptFallingEdge: Interrupt on falling edge.
  - kPORT_InterruptEitherEdge : Interrupt on either edge.
  - kPORT_InterruptLogicOne : Interrupt when logic one.
  - kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit).
  - kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

static inline void PORT_SetPinDriveStrength(PORT_Type *base, uint32_t pin, uint8_t strength)

Configures the port pin drive strength.

Parameters

- base – PORT peripheral base pointer.
- pin – PORT pin number.
- strength – PORT pin drive strength
  - kPORT_LowDriveStrength = 0U - Low-drive strength is configured.
  - kPORT_HighDriveStrength = 1U - High-drive strength is configured.

static inline uint32_t PORT_GetPinsInterruptFlags(PORT_Type *base)

Reads the whole port status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

- base – PORT peripheral base pointer.

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

static inline void PORT_ClearPinsInterruptFlags(PORT_Type *base, uint32_t mask)

Clears the multiple pin interrupt status flag.

Parameters

- base – PORT peripheral base pointer.

- mask – PORT pin number macro.

FSL_PORT_DRIVER_VERSION
    PORT driver version.

enum __port_pull
    Internal resistor pull feature selection.

    Values:

    enumerator kPORT_PullDisable
        Internal pull-up/down resistor is disabled.

    enumerator kPORT_PullDown
        Internal pull-down resistor is enabled.

    enumerator kPORT_PullUp
        Internal pull-up resistor is enabled.

enum __port_slew_rate
    Slew rate selection.

    Values:

    enumerator kPORT_FastSlewRate
        Fast slew rate is configured.

    enumerator kPORT_SlowSlewRate
        Slow slew rate is configured.

enum __port_open_drain_enable
    Open Drain feature enable/disable.

    Values:

    enumerator kPORT_OpenDrainDisable
        Open drain output is disabled.

    enumerator kPORT_OpenDrainEnable
        Open drain output is enabled.

enum __port_passive_filter_enable
    Passive filter feature enable/disable.

    Values:

    enumerator kPORT_PassiveFilterDisable
        Passive input filter is disabled.

    enumerator kPORT_PassiveFilterEnable
        Passive input filter is enabled.

enum __port_drive_strength
    Configures the drive strength.

    Values:

    enumerator kPORT_LowDriveStrength
        Low-drive strength is configured.

    enumerator kPORT_HighDriveStrength
        High-drive strength is configured.

enum __port_lock_register
    Unlock/lock the pin control register field[15:0].

    Values:

    enumerator kPORT_UnlockRegister
        Pin Control Register fields [15:0] are not locked.

    enumerator kPORT_LockRegister
        Pin Control Register fields [15:0] are locked.

enum __port_mux
    Pin mux selection.

    Values:

    enumerator kPORT_PinDisabledOrAnalog
        Corresponding pin is disabled, but is used as an analog pin.

    enumerator kPORT_MuxAsGpio
        Corresponding pin is configured as GPIO.

    enumerator kPORT_MuxAlt0
        Chip-specific

    enumerator kPORT_MuxAlt1
        Chip-specific

    enumerator kPORT_MuxAlt2
        Chip-specific

    enumerator kPORT_MuxAlt3
        Chip-specific

    enumerator kPORT_MuxAlt4
        Chip-specific

    enumerator kPORT_MuxAlt5
        Chip-specific

    enumerator kPORT_MuxAlt6
        Chip-specific

    enumerator kPORT_MuxAlt7
        Chip-specific

    enumerator kPORT_MuxAlt8
        Chip-specific

    enumerator kPORT_MuxAlt9
        Chip-specific

    enumerator kPORT_MuxAlt10
        Chip-specific

    enumerator kPORT_MuxAlt11
        Chip-specific

    enumerator kPORT_MuxAlt12
        Chip-specific

    enumerator kPORT_MuxAlt13
        Chip-specific

enumerator kPORT_MuxAlt14
    Chip-specific

enumerator kPORT_MuxAlt15
    Chip-specific

enum __port_interrupt
    Configures the interrupt generation condition.

    Values:

    enumerator kPORT_InterruptOrDMADisabled
        Interrupt/DMA request is disabled.

    enumerator kPORT_DMARisingEdge
        DMA request on rising edge.

    enumerator kPORT_DMAFallingEdge
        DMA request on falling edge.

    enumerator kPORT_DMAEitherEdge
        DMA request on either edge.

    enumerator kPORT_FlagRisingEdge
        Flag sets on rising edge.

    enumerator kPORT_FlagFallingEdge
        Flag sets on falling edge.

    enumerator kPORT_FlagEitherEdge
        Flag sets on either edge.

    enumerator kPORT_InterruptLogicZero
        Interrupt when logic zero.

    enumerator kPORT_InterruptRisingEdge
        Interrupt on rising edge.

    enumerator kPORT_InterruptFallingEdge
        Interrupt on falling edge.

    enumerator kPORT_InterruptEitherEdge
        Interrupt on either edge.

    enumerator kPORT_InterruptLogicOne
        Interrupt when logic one.

    enumerator kPORT_ActiveHighTriggerOutputEnable
        Enable active high-trigger output.

    enumerator kPORT_ActiveLowTriggerOutputEnable
        Enable active low-trigger output.

enum __port_digital_filter_clock_source
    Digital filter clock source selection.

    Values:

    enumerator kPORT_BusClock
        Digital filters are clocked by the bus clock.

    enumerator kPORT_LpoClock
        Digital filters are clocked by the 1 kHz LPO clock.

2.30. **PORT: Port Control and Interrupts** 287

typedef enum *_port_mux* port_mux_t
    Pin mux selection.

typedef enum *_port_interrupt* port_interrupt_t
    Configures the interrupt generation condition.

typedef enum *_port_digital_filter_clock_source* port_digital_filter_clock_source_t
    Digital filter clock source selection.

typedef struct *_port_digital_filter_config* port_digital_filter_config_t
    PORT digital filter feature configuration definition.

typedef struct *_port_pin_config* port_pin_config_t
    PORT pin configuration structure.

FSL_COMPONENT_ID

struct _port_digital_filter_config
    #include <fsl_port.h> PORT digital filter feature configuration definition.

Public Members

uint32_t digitalFilterWidth
    Set digital filter width

port_digital_filter_clock_source_t clockSource
    Set digital filter clockSource

struct _port_pin_config
    #include <fsl_port.h> PORT pin configuration structure.

Public Members

uint16_t pullSelect
    No-pull/pull-down/pull-up select

uint16_t slewRate
    Fast/slow slew rate Configure

uint16_t passiveFilterEnable
    Passive filter enable/disable

uint16_t openDrainEnable
    Open drain enable/disable

uint16_t driveStrength
    Fast/slow drive strength configure

uint16_t lockRegister
    Lock/unlock the PCR field[15:0]

2.31 QTMR: Quad Timer Driver

void QTMR_Init(TMR_Type *base, const *qtmr_config_t* *config)

Ungates the Quad Timer clock and configures the peripheral for basic operation.

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

Parameters

- base – Quad Timer peripheral base address

- config – Pointer to user's Quad Timer config structure

void QTMR_Deinit(TMR_Type *base)

Stops the counter and gates the Quad Timer clock.

Parameters

- base – Quad Timer peripheral base address

void QTMR_GetDefaultConfig(*qtmr_config_t* *config)

Fill in the Quad Timer config struct with the default settings.

The default values are:

```
config->debugMode = kQTMR_RunNormalInDebug;
config->enableExternalForce = false;
config->enableMasterMode = false;
config->faultFilterCount = 0;
config->faultFilterPeriod = 0;
config->primarySource = kQTMR_ClockDivide_2;
config->secondarySource = kQTMR_Counter0InputPin;
```

Parameters

- config – Pointer to user's Quad Timer config structure.

void QTMR_EnableInterrupts(TMR_Type *base, uint32_t mask)

Enables the selected Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration qtmr_interrupt_enable_t

void QTMR_DisableInterrupts(TMR_Type *base, uint32_t mask)

Disables the selected Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address

- mask – The interrupts to enable. This is a logical OR of members of the enumeration qtmr_interrupt_enable_t

uint32_t QTMR_GetEnabledInterrupts(TMR_Type *base)

Gets the enabled Quad Timer interrupts.

Parameters

- base – Quad Timer peripheral base address

Returns

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

---

uint32_t QTMR_GetStatus(TMR_Type *base)

>   Gets the Quad Timer status flags.

>   >   **Parameters**

>   >   >   • base – Quad Timer peripheral base address

>   >   **Returns**

>   >   >   The status flags. This is the logical OR of members of the enumeration qtmr_status_flags_t

void QTMR_ClearStatusFlags(TMR_Type *base, uint32_t mask)

>   Clears the Quad Timer status flags.

>   >   **Parameters**

>   >   >   • base – Quad Timer peripheral base address

>   >   >   • mask – The status flags to clear. This is a logical OR of members of the enumeration qtmr_status_flags_t

void QTMR_SetTimerPeriod(TMR_Type *base, uint16_t ticks)

>   Sets the timer period in ticks.

>   Timers counts from initial value till it equals the count value set here. The counter will then reinitialize to the value specified in the Load register.

>   **Note:**

>   >   a. This function will write the time period in ticks to COMP1 or COMP2 register depending on the count direction

>   >   b. User can call the utility macros provided in fsl_common.h to convert to ticks

>   >   c. This function supports cases, providing only primary source clock without secondary source clock.

>   >   **Parameters**

>   >   >   • base – Quad Timer peripheral base address

>   >   >   • ticks – Timer period in units of ticks

static inline uint16_t QTMR_GetCurrentTimerCount(TMR_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:** User can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

---

>   >   **Parameters**

>   >   >   • base – Quad Timer peripheral base address

>   >   **Returns**

>   >   >   Current counter value in ticks

static inline void QTMR_StartTimer(TMR_Type *base, *qtmr_counting_mode_t* clockSource)

>   Starts the Quad Timer counter.

>   >   **Parameters**

>   >   >   • base – Quad Timer peripheral base address

- clockSource – Quad Timer clock source

static inline void QTMR_StopTimer(TMR_Type *base)

Stops the Quad Timer counter.

Parameters

- base – Quad Timer peripheral base address

FSL_QTMR_DRIVER_VERSION

Version.

enum __qtmr_primary_count_source

Quad Timer primary clock source selection.

Values:

enumerator kQTMR_ClockCounter0InputPin

Use counter 0 input pin

enumerator kQTMR_ClockCounter1InputPin

Use counter 1 input pin

enumerator kQTMR_ClockCounter2InputPin

Use counter 2 input pin

enumerator kQTMR_ClockCounter3InputPin

Use counter 3 input pin

enumerator kQTMR_ClockCounter0Output

Use counter 0 output

enumerator kQTMR_ClockCounter1Output

Use counter 1 output

enumerator kQTMR_ClockCounter2Output

Use counter 2 output

enumerator kQTMR_ClockCounter3Output

Use counter 3 output

enumerator kQTMR_ClockDivide_1

IP bus clock divide by 1 prescaler

enumerator kQTMR_ClockDivide_2

IP bus clock divide by 2 prescaler

enumerator kQTMR_ClockDivide_4

IP bus clock divide by 4 prescaler

enumerator kQTMR_ClockDivide_8

IP bus clock divide by 8 prescaler

enumerator kQTMR_ClockDivide_16

IP bus clock divide by 16 prescaler

enumerator kQTMR_ClockDivide_32

IP bus clock divide by 32 prescaler

enumerator kQTMR_ClockDivide_64

IP bus clock divide by 64 prescaler

enumerator kQTMR_ClockDivide_128

IP bus clock divide by 128 prescaler

enum __qtmr_input_source

Quad Timer input sources selection.

Values:

enumerator kQTMR_Counter0InputPin
Use counter 0 input pin

enumerator kQTMR_Counter1InputPin
Use counter 1 input pin

enumerator kQTMR_Counter2InputPin
Use counter 2 input pin

enumerator kQTMR_Counter3InputPin
Use counter 3 input pin

enum __qtmr_counting_mode

Quad Timer counting mode selection.

Values:

enumerator kQTMR_NoOperation
No operation

enumerator kQTMR_PriSrcRiseEdge
Count rising edges of primary source

enumerator kQTMR_PriSrcRiseAndFallEdge
Count rising and falling edges of primary source

enumerator kQTMR_PriSrcRiseEdgeSecInpHigh
Count rise edges of pri SRC while sec inp high active

enumerator kQTMR_QuadCountMode
Quadrature count mode, uses pri and sec sources

enumerator kQTMR_PriSrcRiseEdgeSecDir
Count rising edges of pri SRC; sec SRC specifies dir

enumerator kQTMR_SecSrcTrigPriCnt
Edge of sec SRC trigger primary count until compare

enumerator kQTMR_CascadeCount
Cascaded count mode (up/down)

enum __qtmr_output_mode

Quad Timer output mode selection.

Values:

enumerator kQTMR_AssertWhenCountActive
Assert OFLAG while counter is active

enumerator kQTMR_ClearOnCompare
Clear OFLAG on successful compare

enumerator kQTMR_SetOnCompare
Set OFLAG on successful compare

enumerator kQTMR_ToggleOnCompare
Toggle OFLAG on successful compare

enumerator kQTMR_ToggleOnAltCompareReg
    Toggle OFLAG using alternating compare registers

enumerator kQTMR_SetOnCompareClearOnSecSrcInp
    Set OFLAG on compare, clear on sec SRC input edge

enumerator kQTMR_SetOnCompareClearOnCountRoll
    Set OFLAG on compare, clear on counter rollover

enumerator kQTMR_EnableGateClock
    Enable gated clock output while count is active

enum __qtmr_input_capture_edge
    Quad Timer input capture edge mode, rising edge, or falling edge.

    Values:

    enumerator kQTMR_NoCapture
        Capture is disabled

    enumerator kQTMR_RisingEdge
        Capture on rising edge (IPS=0) or falling edge (IPS=1)

    enumerator kQTMR_FallingEdge
        Capture on falling edge (IPS=0) or rising edge (IPS=1)

    enumerator kQTMR_RisingAndFallingEdge
        Capture on both edges

enum __qtmr_preload_control
    Quad Timer input capture edge mode, rising edge, or falling edge.

    Values:

    enumerator kQTMR_NoPreload
        Never preload

    enumerator kQTMR_LoadOnComp1
        Load upon successful compare with value in COMP1

    enumerator kQTMR_LoadOnComp2
        Load upon successful compare with value in COMP2

enum __qtmr_debug_action
    List of Quad Timer run options when in Debug mode.

    Values:

    enumerator kQTMR_RunNormalInDebug
        Continue with normal operation

    enumerator kQTMR_HaltCounter
        Halt counter

    enumerator kQTMR_ForceOutToZero
        Force output to logic 0

    enumerator kQTMR_HaltCountForceOutZero
        Halt counter and force output to logic 0

enum __qtmr_interrupt_enable
    List of Quad Timer interrupts.

    Values:

enumerator kQTMR_CompareInterruptEnable
        Compare interrupt.

enumerator kQTMR_Compare1InterruptEnable
        Compare 1 interrupt.

enumerator kQTMR_Compare2InterruptEnable
        Compare 2 interrupt.

enumerator kQTMR_OverflowInterruptEnable
        Timer overflow interrupt.

enumerator kQTMR_EdgeInterruptEnable
        Input edge interrupt.

enum _qtmr_status_flags
        List of Quad Timer flags.

        Values:

        enumerator kQTMR_CompareFlag
                Compare flag

        enumerator kQTMR_Compare1Flag
                Compare 1 flag

        enumerator kQTMR_Compare2Flag
                Compare 2 flag

        enumerator kQTMR_OverflowFlag
                Timer overflow flag

        enumerator kQTMR_EdgeFlag
                Input edge flag

typedef enum *_qtmr_primary_count_source* qtmr_primary_count_source_t
        Quad Timer primary clock source selection.

typedef enum *_qtmr_input_source* qtmr_input_source_t
        Quad Timer input sources selection.

typedef enum *_qtmr_counting_mode* qtmr_counting_mode_t
        Quad Timer counting mode selection.

typedef enum *_qtmr_output_mode* qtmr_output_mode_t
        Quad Timer output mode selection.

typedef enum *_qtmr_input_capture_edge* qtmr_input_capture_edge_t
        Quad Timer input capture edge mode, rising edge, or falling edge.

typedef enum *_qtmr_preload_control* qtmr_preload_control_t
        Quad Timer input capture edge mode, rising edge, or falling edge.

typedef enum *_qtmr_debug_action* qtmr_debug_action_t
        List of Quad Timer run options when in Debug mode.

typedef enum *_qtmr_interrupt_enable* qtmr_interrupt_enable_t
        List of Quad Timer interrupts.

typedef enum *_qtmr_status_flags* qtmr_status_flags_t
        List of Quad Timer flags.

typedef struct _*qtmr_config* qtmr_config_t

Quad Timer config structure.

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

The config struct can be made const so it resides in flash

status_t QTMR_SetupPwm(TMR_Type *base, uint32_t pwmFreqHz, uint8_t dutyCyclePercent, bool outputPolarity, uint32_t srcClock_Hz)

Sets up Quad timer module for PWM signal output.

The function initializes the timer module according to the parameters passed in by the user. The function also sets up the value compare registers to match the PWM signal requirements.

Parameters

- base – Quad Timer peripheral base address

- pwmFreqHz – PWM signal frequency in Hz

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

- outputPolarity – true: invert polarity of the output signal, false: no inversion

- srcClock_Hz – Main counter clock in Hz.

Returns

Returns an error if there was error setting up the signal.

void QTMR_SetupInputCapture(TMR_Type *base, *qtmr_input_source_t* capturePin, bool inputPolarity, bool reloadOnCapture, *qtmr_input_capture_edge_t* captureMode)

Allows the user to count the source clock cycles until a capture event arrives.

The count is stored in the capture register.

Parameters

- base – Quad Timer peripheral base address

- capturePin – Pin through which we receive the input signal to trigger the capture

- inputPolarity – true: invert polarity of the input signal, false: no inversion

- reloadOnCapture – true: reload the counter when an input capture occurs, false: no reload

- captureMode – Specifies which edge of the input signal triggers a capture

struct __qtmr_config

#include <fsl_qtmr.h> Quad Timer config structure.

This structure holds the configuration settings for the Quad Timer peripheral. To initialize this structure to reasonable defaults, call the QTMR_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

> *qtmr_primary_count_source_t* primarySource
>> Specify the primary count source

> *qtmr_input_source_t* secondarySource
>> Specify the secondary count source

> bool enableMasterMode
>> true: Broadcast compare function output to other counters; false no broadcast

> bool enableExternalForce
>> true: Compare from another counter force state of OFLAG signal false: OFLAG controlled by local counter

> uint8_t faultFilterCount
>> Fault filter count

> uint8_t faultFilterPeriod
>> Fault filter period;value of 0 will bypass the filter

> *qtmr_debug_action_t* debugMode
>> Operation in Debug mode

2.32 RCM: Reset Control Module Driver

static inline void RCM_GetVersionId(RCM_Type *base, *rcm_version_id_t* *versionId)
> Gets the RCM version ID.

> This function gets the RCM version ID including the major version number, the minor version number, and the feature specification number.

> **Parameters**
>> • base – RCM peripheral base address.
>>
>> • versionId – Pointer to the version ID structure.

static inline uint32_t RCM_GetResetSourceImplementedStatus(RCM_Type *base)
> Gets the reset source implemented status.

> This function gets the RCM parameter that indicates whether the corresponding reset source is implemented. Use source masks defined in the rcm_reset_source_t to get the desired source status.

> This is an example.

```
uint32_t status;

To test whether the MCU is reset using Watchdog.
status = RCM_GetResetSourceImplementedStatus(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

> **Parameters**
>> • base – RCM peripheral base address.

> **Returns**
>> All reset source implemented status bit map.

static inline uint32_t RCM_GetPreviousResetSources(RCM_Type *base)
> Gets the reset source status which caused a previous reset.

> This function gets the current reset source status. Use source masks defined in the rcm_reset_source_t to get the desired source status.

This is an example.

```
uint32_t resetStatus;

To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceWdog;

To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

Returns

All reset source status bit map.

static inline uint32_t RCM_GetStickyResetSources(RCM_Type *base)

Gets the sticky reset source status.

This function gets the current reset source status that has not been cleared by software for a specific source.

This is an example.

```
uint32_t resetStatus;

To get all reset source statuses.
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceAll;

To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceWdog;

To test multiple reset sources.
resetStatus = RCM_GetStickyResetSources(RCM) & (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

Returns

All reset source status bit map.

static inline void RCM_ClearStickyResetSources(RCM_Type *base, uint32_t sourceMasks)

Clears the sticky reset source status.

This function clears the sticky system reset flags indicated by source masks.

This is an example.

```
Clears multiple reset sources.
RCM_ClearStickyResetSources(kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

- base – RCM peripheral base address.

- sourceMasks – reset source status bit map

void RCM_ConfigureResetPinFilter(RCM_Type *base, const *rcm_reset_pin_filter_config_t* *config)

Configures the reset pin filter.

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

- base – RCM peripheral base address.
- config – Pointer to the configuration structure.

static inline bool RCM_GetEasyPortModePinStatus(RCM_Type *base)

Gets the EZP_MS_B pin assert status.

This function gets the easy port mode status (EZP_MS_B) pin assert status.

Parameters

- base – RCM peripheral base address.

Returns

status true - asserted, false - reasserted

static inline *rcm_boot_rom_config_t* RCM_GetBootRomSource(RCM_Type *base)

Gets the ROM boot source.

This function gets the ROM boot source during the last chip reset.

Parameters

- base – RCM peripheral base address.

Returns

The ROM boot source.

static inline void RCM_ClearBootRomSource(RCM_Type *base)

Clears the ROM boot source flag.

This function clears the ROM boot source flag.

Parameters

- base – Register base address of RCM

void RCM_SetForceBootRomSource(RCM_Type *base, *rcm_boot_rom_config_t* config)

Forces the boot from ROM.

This function forces booting from ROM during all subsequent system resets.

Parameters

- base – RCM peripheral base address.
- config – Boot configuration.

static inline void RCM_SetSystemResetInterruptConfig(RCM_Type *base, uint32_t intMask,
rcm_reset_delay_t delay)

Sets the system reset interrupt configuration.

For a graceful shut down, the RCM supports delaying the assertion of the system reset for a period of time when the reset interrupt is generated. This function can be used to enable the interrupt and the delay period. The interrupts are passed in as bit mask. See rcm_int_t for details. For example, to delay a reset for 512 LPO cycles after the WDOG timeout or loss-of-clock occurs, configure as follows: RCM_SetSystemResetInterruptConfig(kRCM_IntWatchDog | kRCM_IntLossOfClk, kRCM_ResetDelay512Lpo);

Parameters

- base – RCM peripheral base address.
- intMask – Bit mask of the system reset interrupts to enable. See rcm_interrupt_enable_t for details.
- delay – Bit mask of the system reset interrupts to enable.

FSL_RCM_DRIVER_VERSION
    RCM driver version 2.0.4.

enum _rcm_reset_source
    System Reset Source Name definitions.

    Values:

    enumerator kRCM_SourceWakeup
        Low-leakage wakeup reset

    enumerator kRCM_SourceLvd
        Low-voltage detect reset

    enumerator kRCM_SourceLoc
        Loss of clock reset

    enumerator kRCM_SourceLol
        Loss of lock reset

    enumerator kRCM_SourceWdog
        Watchdog reset

    enumerator kRCM_SourcePin
        External pin reset

    enumerator kRCM_SourcePor
        Power on reset

    enumerator kRCM_SourceJtag
        JTAG generated reset

    enumerator kRCM_SourceLockup
        Core lock up reset

    enumerator kRCM_SourceSw
        Software reset

    enumerator kRCM_SourceMdmap
        MDM-AP system reset

    enumerator kRCM_SourceEzpt
        EzPort reset

    enumerator kRCM_SourceSackerr
        Parameter could get all reset flags

    enumerator kRCM_SourceAll

enum _rcm_run_wait_filter_mode
    Reset pin filter select in Run and Wait modes.

    Values:

    enumerator kRCM_FilterDisable
        All filtering disabled

    enumerator kRCM_FilterBusClock
        Bus clock filter enabled

    enumerator kRCM_FilterLpoClock
        LPO clock filter enabled

enum __rcm__boot__rom__config

    Boot from ROM configuration.

    Values:

    enumerator kRCM__BootFlash

        Boot from flash

    enumerator kRCM__BootRomCfg0

        Boot from boot ROM due to BOOTCFG0

    enumerator kRCM__BootRomFopt

        Boot from boot ROM due to FOPT[7]

    enumerator kRCM__BootRomBoth

        Boot from boot ROM due to both BOOTCFG0 and FOPT[7]

enum __rcm__reset__delay

    Maximum delay time from interrupt asserts to system reset.

    Values:

    enumerator kRCM__ResetDelay8Lpo

        Delay 8 LPO cycles.

    enumerator kRCM__ResetDelay32Lpo

        Delay 32 LPO cycles.

    enumerator kRCM__ResetDelay128Lpo

        Delay 128 LPO cycles.

    enumerator kRCM__ResetDelay512Lpo

        Delay 512 LPO cycles.

enum __rcm__interrupt__enable

    System reset interrupt enable bit definitions.

    Values:

    enumerator kRCM__IntNone

        No interrupt enabled.

    enumerator kRCM__IntLossOfClk

        Loss of clock interrupt.

    enumerator kRCM__IntLossOfLock

        Loss of lock interrupt.

    enumerator kRCM__IntWatchDog

        Watch dog interrupt.

    enumerator kRCM__IntExternalPin

        External pin interrupt.

    enumerator kRCM__IntGlobal

        Global interrupts.

    enumerator kRCM__IntCoreLockup

        Core lock up interrupt

    enumerator kRCM__IntSoftware

        software interrupt

enumerator kRCM_IntStopModeAckErr
    Stop mode ACK error interrupt.

enumerator kRCM_IntCore1
    Core 1 interrupt.

enumerator kRCM_IntAll
    Enable all interrupts.

typedef enum _rcm_reset_source rcm_reset_source_t
    System Reset Source Name definitions.

typedef enum _rcm_run_wait_filter_mode rcm_run_wait_filter_mode_t
    Reset pin filter select in Run and Wait modes.

typedef enum _rcm_boot_rom_config rcm_boot_rom_config_t
    Boot from ROM configuration.

typedef enum _rcm_reset_delay rcm_reset_delay_t
    Maximum delay time from interrupt asserts to system reset.

typedef enum _rcm_interrupt_enable rcm_interrupt_enable_t
    System reset interrupt enable bit definitions.

typedef struct _rcm_version_id rcm_version_id_t
    IP version ID definition.

typedef struct _rcm_reset_pin_filter_config rcm_reset_pin_filter_config_t
    Reset pin filter configuration.

struct _rcm_version_id
    #include <fsl_rcm.h> IP version ID definition.

Public Members

uint16_t feature
    Feature Specification Number.

uint8_t minor
    Minor version number.

uint8_t major
    Major version number.

struct _rcm_reset_pin_filter_config
    #include <fsl_rcm.h> Reset pin filter configuration.

Public Members

bool enableFilterInStop
    Reset pin filter select in stop mode.

rcm_run_wait_filter_mode_t filterInRunWait
    Reset pin filter in run/wait mode.

uint8_t busClockFilterCount
    Reset pin bus clock filter width.

2.33   RNGA: Random Number Generator Accelerator Driver

FSL_RNGA_DRIVER_VERSION
  RNGA driver version 2.0.2.

enum _rnga_mode
  RNGA working mode.

  Values:

  enumerator kRNGA_ModeNormal
    Normal Mode. The ring-oscillator clocks are active; RNGA generates entropy (randomness) from the clocks and stores it in shift registers.

  enumerator kRNGA_ModeSleep
    Sleep Mode. The ring-oscillator clocks are inactive; RNGA does not generate entropy.

typedef enum *_rnga_mode* rnga_mode_t
  RNGA working mode.

void RNGA_Init(RNG_Type *base)
  Initializes the RNGA.

  This function initializes the RNGA. When called, the RNGA entropy generation starts immediately.

    Parameters

        • base – RNGA base address

void RNGA_Deinit(RNG_Type *base)
  Shuts down the RNGA.

  This function shuts down the RNGA.

    Parameters

        • base – RNGA base address

status_t RNGA_GetRandomData(RNG_Type *base, void *data, size_t data_size)
  Gets random data.

  This function gets random data from the RNGA.

    Parameters

        • base – RNGA base address

        • data – pointer to user buffer to be filled by random data

        • data_size – size of data in bytes

    Returns
        RNGA status

void RNGA_Seed(RNG_Type *base, uint32_t seed)
  Feeds the RNGA module.

  This function inputs an entropy value that the RNGA uses to seed its pseudo-random algorithm.

    Parameters

        • base – RNGA base address

        • seed – input seed value

void RNGA_SetMode(RNG_Type *base, *rnga_mode_t* mode)

    Sets the RNGA in normal mode or sleep mode.

    This function sets the RNGA in sleep mode or normal mode.

        Parameters

- base – RNGA base address
- mode – normal mode or sleep mode

rnga_mode_t RNGA_GetMode(RNG_Type *base)

    Gets the RNGA working mode.

    This function gets the RNGA working mode.

        Parameters

- base – RNGA base address

        Returns

        normal mode or sleep mode

2.34   SIM: System Integration Module Driver

FSL_SIM_DRIVER_VERSION

    Driver version.

enum _sim_usb_volt_reg_enable_mode

    USB voltage regulator enable setting.

    Values:

    enumerator kSIM_UsbVoltRegEnable

        Enable voltage regulator.

    enumerator kSIM_UsbVoltRegEnableInLowPower

        Enable voltage regulator in VLPR/VLPW modes.

    enumerator kSIM_UsbVoltRegEnableInStop

        Enable voltage regulator in STOP/VLPS/LLS/VLLS modes.

    enumerator kSIM_UsbVoltRegEnableInAllModes

        Enable voltage regulator in all power modes.

enum _sim_flash_mode

    Flash enable mode.

    Values:

    enumerator kSIM_FlashDisableInWait

        Disable flash in wait mode.

    enumerator kSIM_FlashDisable

        Disable flash in normal mode.

typedef struct _sim_uid sim_uid_t

    Unique ID.

void SIM_SetUsbVoltRegulatorEnableMode(uint32_t mask)

Sets the USB voltage regulator setting.

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP/VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of _sim_usb_volt_reg_enable_mode. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable | kSIM_UsbVoltRegEnableInLowPower);

Parameters

- mask – USB voltage regulator enable setting.

void SIM_GetUniqueId(*sim_uid_t* *uid)

Gets the unique identification register value.

Parameters

- uid – Pointer to the structure to save the UID value.

static inline void SIM_SetFlashMode(uint8_t mode)

Sets the flash enable mode.

Parameters

- mode – The mode to set; see _sim_flash_mode for mode details.

struct __sim_uid

#include <fsl_sim.h> Unique ID.

Public Members

uint32_t H

UIDH.

uint32_t M

SIM_UIDM.

uint32_t L

UIDL.

2.35 SLCD: Segment LCD Driver

void SLCD_Init(LCD_Type *base, *slcd_config_t* *configure)

Initializes the SLCD, ungates the module clock, initializes the power setting, enables all used plane pins, and sets with interrupt and work mode with the configuration.

Parameters

- base – SLCD peripheral base address.

- configure – SLCD configuration pointer. For the configuration structure, many parameters have the default setting and the SLCD_Getdefaultconfig() is provided to get them. Use it verified for their applications. The others have no default settings, such as "clkConfig", and must be provided by the application before calling the SLCD_Init() API.

void SLCD_Deinit(LCD_Type *base)

    Deinitializes the SLCD module, gates the module clock, disables an interrupt, and displays the SLCD.

        Parameters

                • base – SLCD peripheral base address.

void SLCD_GetDefaultConfig(*slcd_config_t* *configure)

    Gets the SLCD default configuration structure. The purpose of this API is to get default parameters of the configuration structure for the SLCD_Init(). Use these initialized parameters unchanged in SLCD_Init() or modify fields of the structure before the calling SLCD_Init(). All default parameters of the configure structuration are listed.

```
config.displayMode        = kSLCD_NormalMode;
config.powerSupply        = kSLCD_InternalVll3UseChargePump;
config.voltageTrim        = kSLCD_RegulatedVolatgeTrim00;
config.lowPowerBehavior   = kSLCD_EnabledInWaitStop;
config.interruptSrc       = 0;
config.faultConfig        = NULL;
config.frameFreqIntEnable =  false;
```

        Parameters

                • configure – The SLCD configuration structure pointer.

static inline void SLCD_StartDisplay(LCD_Type *base)

    Enables the SLCD controller, starts generation, and displays the front plane and back plane waveform.

        Parameters

                • base – SLCD peripheral base address.

static inline void SLCD_StopDisplay(LCD_Type *base)

    Stops the SLCD controller. There is no waveform generator and all enabled pins only output a low value.

        Parameters

                • base – SLCD peripheral base address.

void SLCD_StartBlinkMode(LCD_Type *base, *slcd_blink_mode_t* mode, *slcd_blink_rate_t* rate)

    Starts the SLCD blink mode.

        Parameters

                • base – SLCD peripheral base address.

                • mode – SLCD blink mode.

                • rate – SLCD blink rate.

static inline void SLCD_StopBlinkMode(LCD_Type *base)

    Stops the SLCD blink mode.

        Parameters

                • base – SLCD peripheral base address.

static inline void SLCD_SetBackPlanePhase(LCD_Type *base, uint32_t pinIndx, *slcd_phase_type_t* phase)

    Sets the SLCD back plane pin phase.

2.35. SLCD: Segment LCD Driver

This function sets the SLCD back plane pin phase. "kSLCD_PhaseXActivate" setting means the phase X is active for the back plane pin. "kSLCD_NoPhaseActivate" setting means there is no phase active for the back plane pin. For example, set the back plane pin 20 for phase A.

```
SLCD_SetBackPlanePhase(LCD, 20, kSLCD_PhaseAActivate);
```

Parameters

- base – SLCD peripheral base address.

- pinIndx – SLCD back plane pin index. Range from 0 to 63.

- phase – The phase activates for the back plane pin.

static inline void SLCD_SetFrontPlaneSegments(LCD_Type *base, uint32_t pinIndx, uint8_t operation)

Sets the SLCD front plane segment operation for a front plane pin.

This function sets the SLCD front plane segment on or off operation. Each bit turns on or off the segments associated with the front plane pin in the following pattern: HGFEDCBA (most significant bit controls segment H and least significant bit controls segment A). For example, turn on the front plane pin 20 for phase B and phase C.

```
SLCD_SetFrontPlaneSegments(LCD, 20, (kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));
```

Parameters

- base – SLCD peripheral base address.

- pinIndx – SLCD back plane pin index. Range from 0 to 63.

- operation – The operation for the segment on the front plane pin. This is a logical OR of the enumeration :: slcd_phase_type_t.

static inline void SLCD_SetFrontPlaneOnePhase(LCD_Type *base, uint32_t pinIndx, slcd_phase_index_t phaseIndx, bool enable)

Sets one SLCD front plane pin for one phase.

This function can be used to set one phase on or off for the front plane pin. It can be call many times to set the plane pin for different phase indexes. For example, turn on the front plane pin 20 for phase B and phase C.

```
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseBIndex, true);
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseCIndex, true);
```

Parameters

- base – SLCD peripheral base address.

- pinIndx – SLCD back plane pin index. Range from 0 to 63.

- phaseIndx – The phase bit index slcd_phase_index_t.

- enable – True to turn on the segment for phaseIndx phase false to turn off the segment for phaseIndx phase.

static inline void SLCD_EnablePadSafeState(LCD_Type *base, bool enable)

Enables/disables the SLCD pad safe state.

Forces the safe state on the LCD pad controls. All LCD front plane and backplane functions are disabled.

Parameters

- base – SLCD peripheral base address.

- enable – True enable, false disable.

static inline uint32_t SLCD_GetFaultDetectCounter(LCD_Type *base)

Gets the SLCD fault detect counter.

This function gets the number of samples inside the fault detection sample window.

Parameters

- base – SLCD peripheral base address.

Returns

The fault detect counter. The maximum return value is 255. If the maximum 255 returns, the overflow may happen. Reconfigure the fault detect sample window and fault detect clock prescaler for proper sampling.

void SLCD_EnableInterrupts(LCD_Type *base, uint32_t mask)

Enables the SLCD interrupt. For example, to enable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD_EnableInterrupts(LCD,kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);
```

Parameters

- base – SLCD peripheral base address.

- mask – SLCD interrupts to enable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

void SLCD_DisableInterrupts(LCD_Type *base, uint32_t mask)

Disables the SLCD interrupt. For example, to disable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD_DisableInterrupts(LCD,kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);
```

Parameters

- base – SLCD peripheral base address.

- mask – SLCD interrupts to disable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

uint32_t SLCD_GetInterruptStatus(LCD_Type *base)

Gets the SLCD interrupt status flag.

Parameters

- base – SLCD peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

void SLCD_ClearInterruptStatus(LCD_Type *base, uint32_t mask)

Clears the SLCD interrupt events status flag.

Parameters

- base – SLCD peripheral base address.

- mask – SLCD interrupt source to be cleared. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

2.35. SLCD: Segment LCD Driver 307

FSL_SLCD_DRIVER_VERSION
   SLCD driver version.

enum _slcd_clock_prescaler
   SLCD clock prescaler to generate frame frequency.

   Values:

   enumerator kSLCD_ClkPrescaler00
      Prescaler 0.

   enumerator kSLCD_ClkPrescaler01
      Prescaler 1.

   enumerator kSLCD_ClkPrescaler02
      Prescaler 2.

   enumerator kSLCD_ClkPrescaler03
      Prescaler 3.

   enumerator kSLCD_ClkPrescaler04
      Prescaler 4.

   enumerator kSLCD_ClkPrescaler05
      Prescaler 5.

   enumerator kSLCD_ClkPrescaler06
      Prescaler 6.

   enumerator kSLCD_ClkPrescaler07
      Prescaler 7.

enum _slcd_blink_rate
   SLCD blink rate.

   Values:

   enumerator kSLCD_BlinkRate00
      SLCD blink rate is LCD clock/$((2^{12}))$.

   enumerator kSLCD_BlinkRate01
      SLCD blink rate is LCD clock/$((2^{13}))$.

   enumerator kSLCD_BlinkRate02
      SLCD blink rate is LCD clock/$((2^{14}))$.

   enumerator kSLCD_BlinkRate03
      SLCD blink rate is LCD clock/$((2^{15}))$.

   enumerator kSLCD_BlinkRate04
      SLCD blink rate is LCD clock/$((2^{16}))$.

   enumerator kSLCD_BlinkRate05
      SLCD blink rate is LCD clock/$((2^{17}))$.

   enumerator kSLCD_BlinkRate06
      SLCD blink rate is LCD clock/$((2^{18}))$.

   enumerator kSLCD_BlinkRate07
      SLCD blink rate is LCD clock/$((2^{19}))$.

enum _slcd_power_supply_option
   SLCD power supply option.

   Values:

enumerator kSLCD_InternalVll3UseChargePump

VLL3 connected to VDD internally, charge pump is used to generate VLL1 and VLL2.

enumerator kSLCD_ExternalVll3UseResistorBiasNetwork

VLL3 is driven externally and resistor bias network is used to generate VLL1 and VLL2.

enumerator kSLCD_ExteranlVll3UseChargePump

VLL3 is driven externally and charge pump is used to generate VLL1 and VLL2.

enumerator kSLCD_InternalVll1UseChargePump

VIREG is connected to VLL1 internally and charge pump is used to generate VLL2 and VLL3.

enum __slcd_regulated_voltage_trim

SLCD regulated voltage trim parameter, be used to meet the desired contrast.

Values:

enumerator kSLCD_RegulatedVolatgeTrim00

Increase the voltage to 0.91 V.

enumerator kSLCD_RegulatedVolatgeTrim01

Increase the voltage to 1.01 V.

enumerator kSLCD_RegulatedVolatgeTrim02

Increase the voltage to 0.96 V.

enumerator kSLCD_RegulatedVolatgeTrim03

Increase the voltage to 1.06 V.

enumerator kSLCD_RegulatedVolatgeTrim04

Increase the voltage to 0.93 V.

enumerator kSLCD_RegulatedVolatgeTrim05

Increase the voltage to 1.03 V.

enumerator kSLCD_RegulatedVolatgeTrim06

Increase the voltage to 0.98 V.

enumerator kSLCD_RegulatedVolatgeTrim07

Increase the voltage to 1.07 V.

enumerator kSLCD_RegulatedVolatgeTrim08

Increase the voltage to 0.92 V.

enumerator kSLCD_RegulatedVolatgeTrim09

Increase the voltage to 1.02 V.

enumerator kSLCD_RegulatedVolatgeTrim10

Increase the voltage to 0.97 V.

enumerator kSLCD_RegulatedVolatgeTrim11

Increase the voltage to 1.08 V.

enumerator kSLCD_RegulatedVolatgeTrim12

Increase the voltage to 0.94 V.

enumerator kSLCD_RegulatedVolatgeTrim13

Increase the voltage to 1.05 V.

enumerator kSLCD_RegulatedVolatgeTrim14

Increase the voltage to 0.99 V.

enumerator kSLCD_RegulatedVolatgeTrim15
> Increase the voltage to 1.09 V.

enum __slcd_load_adjust
> SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source. Adjust the LCD glass capacitance if resistor bias network is enabled: kSLCD_LowLoadOrFastestClkSrc - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_LowLoadOrIntermediateClkSrc - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_HighLoadOrIntermediateClkSrc - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) kSLCD_HighLoadOrSlowestClkSrc - high load (LCD glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: kSLCD_LowLoadOrFastestClkSrc - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) kSLCD_LowLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 4000pF or 2000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrSlowestClkSrc - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

Values:

enumerator kSLCD_LowLoadOrFastestClkSrc
> Adjust in low load or selects fastest clock.

enumerator kSLCD_LowLoadOrIntermediateClkSrc
> Adjust in low load or selects intermediate clock.

enumerator kSLCD_HighLoadOrIntermediateClkSrc
> Adjust in high load or selects intermediate clock.

enumerator kSLCD_HighLoadOrSlowestClkSrc
> Adjust in high load or selects slowest clock.

enum __slcd_clock_src
> SLCD clock source.

Values:

enumerator kSLCD_DefaultClk
> Select default clock ERCLK32K.

enumerator kSLCD_AlternateClk1
> Select alternate clock source 1 : MCGIRCLK.

enumerator kSLCD_AlternateClk2
> Select alternate clock source 2 : OSCERCLK.

enum __slcd_alt_clock_div
> SLCD alternate clock divider.

Values:

enumerator kSLCD_AltClkDivFactor1
> No divide for alternate clock.

enumerator kSLCD_AltClkDivFactor64
> Divide alternate clock with factor 64.

enumerator kSLCD_AltClkDivFactor256
> Divide alternate clock with factor 256.

enumerator kSLCD__AltClkDivFactor512
  Divide alternate clock with factor 512.

enum __slcd_duty_cycle
  SLCD duty cycle.

  Values:

  enumerator kSLCD__1Div1DutyCycle
    LCD use 1 BP 1/1 duty cycle.

  enumerator kSLCD__1Div2DutyCycle
    LCD use 2 BP 1/2 duty cycle.

  enumerator kSLCD__1Div3DutyCycle
    LCD use 3 BP 1/3 duty cycle.

  enumerator kSLCD__1Div4DutyCycle
    LCD use 4 BP 1/4 duty cycle.

  enumerator kSLCD__1Div5DutyCycle
    LCD use 5 BP 1/5 duty cycle.

  enumerator kSLCD__1Div6DutyCycle
    LCD use 6 BP 1/6 duty cycle.

  enumerator kSLCD__1Div7DutyCycle
    LCD use 7 BP 1/7 duty cycle.

  enumerator kSLCD__1Div8DutyCycle
    LCD use 8 BP 1/8 duty cycle.

enum __slcd_phase_type
  SLCD segment phase type.

  Values:

  enumerator kSLCD__NoPhaseActivate
    LCD wareform no phase activates.

  enumerator kSLCD__PhaseAActivate
    LCD waveform phase A activates.

  enumerator kSLCD__PhaseBActivate
    LCD waveform phase B activates.

  enumerator kSLCD__PhaseCActivate
    LCD waveform phase C activates.

  enumerator kSLCD__PhaseDActivate
    LCD waveform phase D activates.

  enumerator kSLCD__PhaseEActivate
    LCD waveform phase E activates.

  enumerator kSLCD__PhaseFActivate
    LCD waveform phase F activates.

  enumerator kSLCD__PhaseGActivate
    LCD waveform phase G activates.

  enumerator kSLCD__PhaseHActivate
    LCD waveform phase H activates.

enum __slcd_phase_index
   SLCD segment phase bit index.

   Values:

   enumerator kSLCD_PhaseAIndex
      LCD phase A bit index.

   enumerator kSLCD_PhaseBIndex
      LCD phase B bit index.

   enumerator kSLCD_PhaseCIndex
      LCD phase C bit index.

   enumerator kSLCD_PhaseDIndex
      LCD phase D bit index.

   enumerator kSLCD_PhaseEIndex
      LCD phase E bit index.

   enumerator kSLCD_PhaseFIndex
      LCD phase F bit index.

   enumerator kSLCD_PhaseGIndex
      LCD phase G bit index.

   enumerator kSLCD_PhaseHIndex
      LCD phase H bit index.

enum __slcd_display_mode
   SLCD display mode.

   Values:

   enumerator kSLCD_NormalMode
      LCD Normal display mode.

   enumerator kSLCD_AlternateMode
      LCD Alternate display mode. For four back planes or less.

   enumerator kSLCD_BlankMode
      LCD Blank display mode.

enum __slcd_blink_mode
   SLCD blink mode.

   Values:

   enumerator kSLCD_BlankDisplayBlink
      Display blank during the blink period.

   enumerator kSLCD_AltDisplayBlink
      Display alternate display during the blink period if duty cycle is lower than 5.

enum __slcd_fault_detect_clock_prescaler
   SLCD fault detect clock prescaler.

   Values:

   enumerator kSLCD_FaultSampleFreqDivider1
      Fault detect sample clock frequency is 1/1 bus clock.

   enumerator kSLCD_FaultSampleFreqDivider2
      Fault detect sample clock frequency is 1/2 bus clock.

enumerator kSLCD_FaultSampleFreqDivider4
    Fault detect sample clock frequency is 1/4 bus clock.

enumerator kSLCD_FaultSampleFreqDivider8
    Fault detect sample clock frequency is 1/8 bus clock.

enumerator kSLCD_FaultSampleFreqDivider16
    Fault detect sample clock frequency is 1/16 bus clock.

enumerator kSLCD_FaultSampleFreqDivider32
    Fault detect sample clock frequency is 1/32 bus clock.

enumerator kSLCD_FaultSampleFreqDivider64
    Fault detect sample clock frequency is 1/64 bus clock.

enumerator kSLCD_FaultSampleFreqDivider128
    Fault detect sample clock frequency is 1/128 bus clock.

enum __slcd_fault_detect_sample_window_width
    SLCD fault detect sample window width.

Values:

enumerator kSLCD_FaultDetectWindowWidth4SampleClk
    Sample window width is 4 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth8SampleClk
    Sample window width is 8 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth16SampleClk
    Sample window width is 16 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth32SampleClk
    Sample window width is 32 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth64SampleClk
    Sample window width is 64 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth128SampleClk
    Sample window width is 128 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth256SampleClk
    Sample window width is 256 sample clock cycles.

enumerator kSLCD_FaultDetectWindowWidth512SampleClk
    Sample window width is 512 sample clock cycles.

enum __slcd_interrupt_enable
    SLCD interrupt source.

Values:

enumerator kSLCD_FaultDetectCompleteInterrupt
    SLCD fault detection complete interrupt source.

enumerator kSLCD_FrameFreqInterrupt
    SLCD frame frequency interrupt source. Not available in all low-power modes.

enum __slcd_lowpower_behavior
    SLCD behavior in low power mode.

Values:

enumerator kSLCD__EnabledInWaitStop

SLCD works in wait and stop mode.

enumerator kSLCD__EnabledInWaitOnly

SLCD works in wait mode and is disabled in stop mode.

enumerator kSLCD__EnabledInStopOnly

SLCD works in stop mode and is disabled in wait mode.

enumerator kSLCD__DisabledInWaitStop

SLCD is disabled in stop mode and wait mode.

typedef enum *_slcd_clock_prescaler* slcd__clock__prescaler__t

SLCD clock prescaler to generate frame frequency.

typedef enum *_slcd_blink_rate* slcd__blink__rate__t

SLCD blink rate.

typedef enum *_slcd_power_supply_option* slcd__power__supply__option__t

SLCD power supply option.

typedef enum *_slcd_regulated_voltage_trim* slcd__regulated__voltage__trim__t

SLCD regulated voltage trim parameter, be used to meet the desired contrast.

typedef enum *_slcd_load_adjust* slcd__load__adjust__t

SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source. Adjust the LCD glass capacitance if resistor bias network is enabled: kSLCD_LowLoadOrFastestClkSrc - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_LowLoadOrIntermediateClkSrc - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD_HighLoadOrIntermediateClkSrc - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) kSLCD_HighLoadOrSlowestClkSrc - high load (LCD glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: kSLCD_LowLoadOrFastestClkSrc - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) kSLCD_LowLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 4000pF or 2000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrSlowestClkSrc - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

typedef enum *_slcd_clock_src* slcd__clock__src__t

SLCD clock source.

typedef enum *_slcd_alt_clock_div* slcd__alt__clock__div__t

SLCD alternate clock divider.

typedef struct *_slcd_clock_config* slcd__clock__config__t

SLCD clock configuration structure.

typedef enum *_slcd_duty_cycle* slcd__duty__cycle__t

SLCD duty cycle.

typedef enum *_slcd_phase_type* slcd__phase__type__t

SLCD segment phase type.

typedef enum *_slcd_phase_index* slcd__phase__index__t

SLCD segment phase bit index.

typedef enum *_slcd_display_mode* slcd__display__mode__t
    SLCD display mode.

typedef enum *_slcd_blink_mode* slcd__blink__mode__t
    SLCD blink mode.

typedef enum *_slcd_fault_detect_clock_prescaler* slcd__fault__detect__clock__prescaler__t
    SLCD fault detect clock prescaler.

typedef enum *_slcd_fault_detect_sample_window_width*
slcd__fault__detect__sample__window__width__t
    SLCD fault detect sample window width.

typedef enum *_slcd_interrupt_enable* slcd__interrupt__enable__t
    SLCD interrupt source.

typedef enum *_slcd_lowpower_behavior* slcd__lowpower__behavior
    SLCD behavior in low power mode.

typedef struct *_slcd_fault_detect_config* slcd__fault__detect__config__t
    SLCD fault frame detection configuration structure.

typedef struct *_slcd_config* slcd__config__t
    SLCD configuration structure.

struct __slcd__clock__config
    #include <fsl_slcd.h> SLCD clock configuration structure.

Public Members

slcd_clock_src_t clkSource
    Clock source. "slcd_clock_src_t" is recommended to be used. The SLCD is optimized to operate using a 32.768kHz clock input.

slcd_alt_clock_div_t altClkDivider
    The divider to divide the alternate clock used for alternate clock source.

slcd_clock_prescaler_t clkPrescaler
    Clock prescaler.

bool fastFrameRateEnable
    Fast frame rate enable flag.

struct __slcd__fault__detect__config
    #include <fsl_slcd.h> SLCD fault frame detection configuration structure.

Public Members

bool faultDetectIntEnable
    Fault frame detection interrupt enable flag.

bool faultDetectBackPlaneEnable
    True means the pin id fault detected is back plane otherwise front plane.

uint8_t faultDetectPinIndex
    Fault detected pin id from 0 to 63.

slcd_fault_detect_clock_prescaler_t faultPrescaler
    Fault detect clock prescaler.

slcd_fault_detect_sample_window_width_t width
> Fault detect sample window width.

struct __slcd_config
> *#include <fsl_slcd.h>* SLCD configuration structure.

Public Members

slcd_power_supply_option_t powerSupply
> Power supply option.

slcd_regulated_voltage_trim_t voltageTrim
> Regulated voltage trim used for the internal regulator VIREG to adjust to facilitate contrast control.

slcd_clock_config_t *clkConfig
> Clock configure.

slcd_load_adjust_t loadAdjust
> Load adjust to handle glass capacitance.

slcd_display_mode_t displayMode
> SLCD display mode.

slcd_duty_cycle_t dutyCycle
> Duty cycle.

slcd_lowpower_behavior lowPowerBehavior
> SLCD behavior in low power mode.

bool frameFreqIntEnable
> Frame frequency interrupt enable flag.

uint32_t slcdLowPinEnabled
> Setting enabled SLCD pin 0 ~ pin 31. Setting bit n to 1 means enable pin n.

uint32_t slcdHighPinEnabled
> Setting enabled SLCD pin 32 ~ pin 63. Setting bit n to 1 means enable pin (n + 32).

uint32_t backPlaneLowPin
> Setting back plane pin 0 ~ pin 31. Setting bit n to 1 means setting pin n as back plane. It should never have the same bit setting as the frontPlane Pin.

uint32_t backPlaneHighPin
> Setting back plane pin 32 ~ pin 63. Setting bit n to 1 means setting pin (n + 32) as back plane. It should never have the same bit setting as the frontPlane Pin.

slcd_fault_detect_config_t *faultConfig
> Fault frame detection configure. If not requirement, set to NULL.

2.36 Smart Card

FSL_SMARTCARD_DRIVER_VERSION
> Smart card driver version 2.3.0.

> Smart card Error codes.

> *Values:*

enumerator kStatus_SMARTCARD_Success
    Transfer ends successfully

enumerator kStatus_SMARTCARD_TxBusy
    Transmit in progress

enumerator kStatus_SMARTCARD_RxBusy
    Receiving in progress

enumerator kStatus_SMARTCARD_NoTransferInProgress
    No transfer in progress

enumerator kStatus_SMARTCARD_Timeout
    Transfer ends with time-out

enumerator kStatus_SMARTCARD_Initialized
    Smart card driver is already initialized

enumerator kStatus_SMARTCARD_PhyInitialized
    Smart card PHY drive is already initialized

enumerator kStatus_SMARTCARD_CardNotActivated
    Smart card is not activated

enumerator kStatus_SMARTCARD_InvalidInput
    Function called with invalid input arguments

enumerator kStatus_SMARTCARD_OtherError
    Some other error occur

enum __smartcard_control
    Control codes for the Smart card protocol timers and misc.

    Values:

    enumerator kSMARTCARD_EnableADT

    enumerator kSMARTCARD_DisableADT

    enumerator kSMARTCARD_EnableGTV

    enumerator kSMARTCARD_DisableGTV

    enumerator kSMARTCARD_ResetWWT

    enumerator kSMARTCARD_EnableWWT

    enumerator kSMARTCARD_DisableWWT

    enumerator kSMARTCARD_ResetCWT

    enumerator kSMARTCARD_EnableCWT

    enumerator kSMARTCARD_DisableCWT

    enumerator kSMARTCARD_ResetBWT

    enumerator kSMARTCARD_EnableBWT

    enumerator kSMARTCARD_DisableBWT

    enumerator kSMARTCARD_EnableInitDetect

    enumerator kSMARTCARD_EnableAnack

enumerator kSMARTCARD_DisableAnack

enumerator kSMARTCARD_ConfigureBaudrate

enumerator kSMARTCARD_SetupATRMode

enumerator kSMARTCARD_SetupT0Mode

enumerator kSMARTCARD_SetupT1Mode

enumerator kSMARTCARD_EnableReceiverMode

enumerator kSMARTCARD_DisableReceiverMode

enumerator kSMARTCARD_EnableTransmitterMode

enumerator kSMARTCARD_DisableTransmitterMode

enumerator kSMARTCARD_ResetWaitTimeMultiplier

enum __smartcard_card_voltage_class

Defines Smart card interface voltage class values.

Values:

enumerator kSMARTCARD_VoltageClassUnknown

enumerator kSMARTCARD_VoltageClassA5_0V

enumerator kSMARTCARD_VoltageClassB3_3V

enumerator kSMARTCARD_VoltageClassC1_8V

enum __smartcard_transfer_state

Defines Smart card I/O transfer states.

Values:

enumerator kSMARTCARD_IdleState

enumerator kSMARTCARD_WaitingForTSState

enumerator kSMARTCARD_InvalidTSDetecetedState

enumerator kSMARTCARD_ReceivingState

enumerator kSMARTCARD_TransmittingState

enum __smartcard_reset_type

Defines Smart card reset types.

Values:

enumerator kSMARTCARD_ColdReset

enumerator kSMARTCARD_WarmReset

enumerator kSMARTCARD_NoColdReset

enumerator kSMARTCARD_NoWarmReset

enum __smartcard_transport_type

Defines Smart card transport protocol types.

Values:

enumerator kSMARTCARD_T0Transport

enumerator kSMARTCARD_T1Transport

enum _smartcard_parity_type
Defines Smart card data parity types.

Values:

enumerator kSMARTCARD_EvenParity

enumerator kSMARTCARD_OddParity

enum _smartcard_card_convention
Defines data Convention format.

Values:

enumerator kSMARTCARD_DirectConvention

enumerator kSMARTCARD_InverseConvention

enum _smartcard_interface_control
Defines Smart card interface IC control types.

Values:

enumerator kSMARTCARD_InterfaceSetVcc

enumerator kSMARTCARD_InterfaceSetClockToResetDelay

enumerator kSMARTCARD_InterfaceReadStatus

enum _smartcard_direction
Defines transfer direction.

Values:

enumerator kSMARTCARD_Receive

enumerator kSMARTCARD_Transmit

typedef enum _smartcard_control smartcard_control_t
Control codes for the Smart card protocol timers and misc.

typedef enum _smartcard_card_voltage_class smartcard_card_voltage_class_t
Defines Smart card interface voltage class values.

typedef enum _smartcard_transfer_state smartcard_transfer_state_t
Defines Smart card I/O transfer states.

typedef enum _smartcard_reset_type smartcard_reset_type_t
Defines Smart card reset types.

typedef enum _smartcard_transport_type smartcard_transport_type_t
Defines Smart card transport protocol types.

typedef enum _smartcard_parity_type smartcard_parity_type_t
Defines Smart card data parity types.

typedef enum _smartcard_card_convention smartcard_card_convention_t
Defines data Convention format.

typedef enum _smartcard_interface_control smartcard_interface_control_t
Defines Smart card interface IC control types.

typedef enum *_smartcard_direction* smartcard_direction_t
>   Defines transfer direction.

typedef void (*smartcard_interface_callback_t)(void *smartcardContext, void *param)
>   Smart card interface interrupt callback function type.

typedef void (*smartcard_transfer_callback_t)(void *smartcardContext, void *param)
>   Smart card transfer interrupt callback function type.

typedef void (*smartcard_time_delay_t)(uint32_t us)
>   Time Delay function used to passive waiting using RTOS [us].

typedef struct *_smartcard_card_params* smartcard_card_params_t
>   Defines card-specific parameters for Smart card driver.

typedef struct *_smartcard_timers_state* smartcard_timers_state_t
>   Smart card defines the state of the EMV timers in the Smart card driver.

typedef struct *_smartcard_interface_config* smartcard_interface_config_t
>   Defines user specified configuration of Smart card interface.

typedef struct *_smartcard_xfer* smartcard_xfer_t
>   Defines user transfer structure used to initialize transfer.

typedef struct *_smartcard_context* smartcard_context_t
>   Runtime state of the Smart card driver.

SMARTCARD_INIT_DELAY_CLOCK_CYCLES
>   Smart card global define which specify number of clock cycles until initial 'TS' character has to be received.

SMARTCARD_EMV_ATR_DURATION_ETU
>   Smart card global define which specify number of clock cycles during which ATR string has to be received.

SMARTCARD_TS_DIRECT_CONVENTION
>   Smart card specification initial TS character definition of direct convention.

SMARTCARD_TS_INVERSE_CONVENTION
>   Smart card specification initial TS character definition of inverse convention.

struct _smartcard_card_params
>   *#include <fsl_smartcard.h>* Defines card-specific parameters for Smart card driver.

Public Members

uint16_t Fi
>   4 bits Fi - clock rate conversion integer

uint8_t fMax
>   Maximum Smart card frequency in MHz

uint8_t WI
>   8 bits WI - work wait time integer

uint8_t Di
>   4 bits DI - baud rate divisor

uint8_t BWI
>   4 bits BWI - block wait time integer

uint8_t CWI

    4 bits CWI - character wait time integer

uint8_t BGI

    4 bits BGI - block guard time integer

uint8_t GTN

    8 bits GTN - extended guard time integer

uint8_t IFSC

    Indicates IFSC value of the card

uint8_t modeNegotiable

    Indicates if the card acts in negotiable or a specific mode.

uint8_t currentD

    4 bits DI - current baud rate divisor

uint8_t status

    Indicates smart card status

bool t0Indicated

    Indicates ff T=0 indicated in TD1 byte

bool t1Indicated

    Indicates if T=1 indicated in TD2 byte

bool atrComplete

    Indicates whether the ATR received from the card was complete or not

bool atrValid

    Indicates whether the ATR received from the card was valid or not

bool present

    Indicates if a smart card is present

bool active

    Indicates if the smart card is activated

bool faulty

    Indicates whether smart card/interface is faulty

smartcard_card_convention_t convention

    Card convention, kSMARTCARD_DirectConvention for direct convention, kSMART-CARD_InverseConvention for inverse convention

struct __smartcard_timers_state

    #include <fsl_smartcard.h> Smart card defines the state of the EMV timers in the Smart card driver.

Public Members

volatile bool adtExpired

    Indicates whether ADT timer expired

volatile bool wwtExpired

    Indicates whether WWT timer expired

volatile bool cwtExpired

    Indicates whether CWT timer expired

volatile bool bwtExpired

Indicates whether BWT timer expired

volatile bool initCharTimerExpired

Indicates whether reception timer for initialization character (TS) after the RST has expired

struct __smartcard_interface_config

#include <fsl_smartcard.h> Defines user specified configuration of Smart card interface.

Public Members

uint32_t smartCardClock

Smart card interface clock [Hz]

uint32_t clockToResetDelay

Indicates clock to RST apply delay [smart card clock cycles]

uint8_t clockModule

Smart card clock module number

uint8_t clockModuleChannel

Smart card clock module channel number

uint8_t clockModuleSourceClock

Smart card clock module source clock [e.g., BusClk]

smartcard_card_voltage_class_t vcc

Smart card voltage class

uint8_t controlPort

Smart card PHY control port instance

uint8_t controlPin

Smart card PHY control pin instance

uint8_t irqPort

Smart card PHY Interrupt port instance

uint8_t irqPin

Smart card PHY Interrupt pin instance

uint8_t resetPort

Smart card reset port instance

uint8_t resetPin

Smart card reset pin instance

uint8_t vsel0Port

Smart card PHY Vsel0 control port instance

uint8_t vsel0Pin

Smart card PHY Vsel0 control pin instance

uint8_t vsel1Port

Smart card PHY Vsel1 control port instance

uint8_t vsel1Pin

Smart card PHY Vsel1 control pin instance

uint8_t dataPort
    Smart card PHY data port instance

uint8_t dataPin
    Smart card PHY data pin instance

uint8_t dataPinMux
    Smart card PHY data pin mux option

uint8_t tsTimerId
    Numerical identifier of the External HW timer for Initial character detection

struct __smartcard__xfer
    #include <fsl_smartcard.h> Defines user transfer structure used to initialize transfer.

Public Members

smartcard_direction_t direction
    Direction of communication. (RX/TX)

uint8_t *buff
    The buffer of data.

size_t size
    The number of transferred units.

struct __smartcard__context
    #include <fsl_smartcard.h> Runtime state of the Smart card driver.

Public Members

void *base
    Smart card module base address

smartcard_direction_t direction
    Direction of communication. (RX/TX)

uint8_t *xBuff
    The buffer of data being transferred.

volatile size_t xSize
    The number of bytes to be transferred.

volatile bool xIsBusy
    True if there is an active transfer.

uint8_t txFifoEntryCount
    Number of data word entries in transmit FIFO.

uint8_t rxFifoThreshold
    The max value of the receiver FIFO threshold.

smartcard_interface_callback_t interfaceCallback
    Callback to invoke after interface IC raised interrupt.

smartcard_transfer_callback_t transferCallback
    Callback to invoke after transfer event occur.

void *interfaceCallbackParam
    Interface callback parameter pointer.

---

void *transferCallbackParam

Transfer callback parameter pointer.

smartcard_time_delay_t timeDelay

Function which handles time delay defined by user or RTOS.

smartcard_reset_type_t resetType

Indicates whether a Cold reset or Warm reset was requested.

smartcard_transport_type_t tType

Indicates current transfer protocol (T0 or T1)

volatile *smartcard_transfer_state_t* transferState

Indicates the current transfer state

smartcard_timers_state_t timersState

Indicates the state of different protocol timers used in driver

smartcard_card_params_t cardParams

Smart card parameters(ATR and current) and interface slots states(ATR and current)

uint8_t IFSD

Indicates the terminal IFSD

smartcard_parity_type_t parity

Indicates current parity even/odd

volatile bool rxtCrossed

Indicates whether RXT thresholds has been crossed

volatile bool txtCrossed

Indicates whether TXT thresholds has been crossed

volatile bool wtxRequested

Indicates whether WTX has been requested or not

volatile bool parityError

Indicates whether a parity error has been detected

uint8_t statusBytes[2]

Used to store Status bytes SW1, SW2 of the last executed card command response

smartcard_interface_config_t interfaceConfig

Smart card interface configuration structure

bool abortTransfer

Used to abort transfer.

2.37 Smart Card UART Driver

void SMARTCARD_UART_GetDefaultConfig(*smartcard_card_params_t* *cardParams)

Fills in the smartcard_card_params structure with default values according to the EMV 4.3 specification.

Parameters

- cardParams – The configuration structure of type smartcard_interface_config_t. Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.

status_t SMARTCARD_UART_Init(UART_Type *base, *smartcard_context_t* *context, uint32_t srcClock_Hz)

Initializes a UART peripheral for the Smart card/ISO-7816 operation.

This function un-gates the UART clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core, and initializes the driver context.

Parameters

- base – The UART peripheral base address.

- context – A pointer to a smart card driver context structure.

- srcClock_Hz – Smart card clock generation module source clock.

Returns

An error code or kStatus_SMARTCARD_Success.

void SMARTCARD_UART_Deinit(UART_Type *base)

This function disables the UART interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates UART clock in SIM.

Parameters

- base – The UART peripheral base address.

int32_t SMARTCARD_UART_GetTransferRemainingBytes(UART_Type *base, *smartcard_context_t* *context)

Returns whether the previous UART transfer has finished.

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred by reading xSize of smart card context structure.

Parameters

- base – The UART peripheral base address.

- context – A pointer to a Smart card driver context structure.

Returns

The number of bytes not transferred.

status_t SMARTCARD_UART_AbortTransfer(UART_Type *base, *smartcard_context_t* *context)

Terminates an asynchronous UART transfer early.

During an async UART transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

- base – The UART peripheral base address.

- context – A pointer to a Smart card driver context structure.

Return values

- kStatus_SMARTCARD_Success – The transfer abort was successful.

- kStatus_SMARTCARD_NoTransmitInProgress – No transmission is currently in progress.

status_t SMARTCARD_UART_TransferNonBlocking(UART_Type *base, *smartcard_context_t* *context, *smartcard_xfer_t* *xfer)

Transfers data using interrupts.

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling non-blocking (asynchronous) transfer function, the application must get the transfer status to check if transmit is completed or not.

> **Parameters**
>
> - base – The UART peripheral base address.
>
> - context – A pointer to a Smart card driver context structure.
>
> - xfer – A pointer to Smart card transfer structure where the linked buffers and sizes are stored.
>
> **Returns**
>
> An error code or kStatus_SMARTCARD_Success.

status_t SMARTCARD_UART_Control(UART_Type *base, *smartcard_context_t* *context, *smartcard_control_t* control, uint32_t param)

Controls the UART module per different user requests.

return An kStatus_SMARTCARD_OtherError in case of error return kStatus_SMARTCARD_Success in success

> **Parameters**
>
> - base – The UART peripheral base address.
>
> - context – A pointer to a smart card driver context structure.
>
> - control – Smart card command type.
>
> - param – Integer value specific to a control command.

void SMARTCARD_UART_IRQHandler(UART_Type *base, *smartcard_context_t* *context)

Interrupt handler for UART.

This handler uses the buffers stored in the smartcard_context_t structures to transfer data. The Smart card driver requires this function to call when the UART interrupt occurs.

> **Parameters**
>
> - base – The UART peripheral base address.
>
> - context – A pointer to a Smart card driver context structure.

void SMARTCARD_UART_ErrIRQHandler(UART_Type *base, *smartcard_context_t* *context)

Error interrupt handler for UART.

This function handles error conditions during a transfer.

> **Parameters**
>
> - base – The UART peripheral base address.
>
> - context – A pointer to a Smart card driver context structure.

void SMARTCARD_UART_TSExpiryCallback(UART_Type *base, *smartcard_context_t* *context)

Handles initial TS character timer time-out event.

> **Parameters**
>
> - base – The UART peripheral base address.
>
> - context – A pointer to a Smart card driver context structure.

void smartcard_uart_TimerStart(uint8_t channel, uint32_t time)

> Initializes timer specific channel with input period, enable channel interrupt and start counter.

> **Parameters**
> > - channel – The timer channel.
> > - time – The time period.

SMARTCARD_EMV_RX_NACK_THRESHOLD

> EMV RX NACK interrupt generation threshold.

SMARTCARD_EMV_TX_NACK_THRESHOLD

> EMV TX NACK interrupt generation threshold.

SMARTCARD_EMV_RX_TO_TX_GUARD_TIME_T0

> EMV TX & RX GUART TIME default value.

SBR_CAL_ADJUST_D1_T0

BRFA_CAL_ADJUST_D1_T0

SBR_CAL_ADJUST_D2_T0

BRFA_CAL_ADJUST_D2_T0

SBR_CAL_ADJUST_D4_T0

BRFA_CAL_ADJUST_D4_T0

SBR_CAL_ADJUST_D1_T1

BRFA_CAL_ADJUST_D1_T1

SBR_CAL_ADJUST_D2_T1

BRFA_CAL_ADJUST_D2_T1

SBR_CAL_ADJUST_D4_T1

BRFA_CAL_ADJUST_D4_T1

2.38 SMC: System Mode Controller Driver

static inline void SMC_GetVersionId(SMC_Type *base, *smc_version_id_t* *versionId)

> Gets the SMC version ID.

> This function gets the SMC version ID, including major version number, minor version number, and feature specification number.

> **Parameters**
> > - base – SMC peripheral base address.
> > - versionId – Pointer to the version ID structure.

void SMC_GetParam(SMC_Type *base, *smc_param_t* *param)

> Gets the SMC parameter.

> This function gets the SMC parameter including the enabled power mdoes.

> **Parameters**
> > - base – SMC peripheral base address.

- param – Pointer to the SMC param structure.

static inline void SMC_SetPowerModeProtection(SMC_Type *base, uint8_t allowedModes)

Configures all power mode protection settings.

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the smc_power_mode_protection_t. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map. For example, to allow LLS and VLLS, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps). To allow all modes, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll).

Parameters

- base – SMC peripheral base address.
- allowedModes – Bitmap of the allowed power modes.

static inline *smc_power_state_t* SMC_GetPowerModeState(SMC_Type *base)

Gets the current power mode status.

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc_power_state_t for information about the power status.

Parameters

- base – SMC peripheral base address.

Returns

Current power mode status.

void SMC_PreEnterStopModes(void)

Prepares to enter stop modes.

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

void SMC_PostExitStopModes(void)

Recovers after wake up from stop modes.

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with SMC_PreEnterStopModes.

void SMC_PreEnterWaitModes(void)

Prepares to enter wait modes.

This function should be called before entering WAIT/VLPW modes.

void SMC_PostExitWaitModes(void)

Recovers after wake up from stop modes.

This function should be called after wake up from WAIT/VLPW modes. It is used with SMC_PreEnterWaitModes.

status_t SMC_SetPowerModeRun(SMC_Type *base)

Configures the system to RUN power mode.

Parameters

- base – SMC peripheral base address.

> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeHsrun(SMC_Type *base)

Configures the system to HSRUN power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeWait(SMC_Type *base)

Configures the system to WAIT power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeStop(SMC_Type *base, *smc_partial_stop_option_t* option)

Configures the system to Stop power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> - option – Partial Stop mode option.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeVlpr(SMC_Type *base, bool wakeupMode)

Configures the system to VLPR power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> - wakeupMode – Enter Normal Run mode if true, else stay in VLPR mode.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeVlpw(SMC_Type *base)

Configures the system to VLPW power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC__SetPowerModeVlps(SMC_Type *base)

Configures the system to VLPS power mode.

> **Parameters**
>
> - base – SMC peripheral base address.
>
> **Returns**
>
> SMC configuration error code.

status_t SMC_SetPowerModeLls(SMC_Type *base, const *smc_power_mode_lls_config_t* *config)
    Configures the system to LLS power mode.

>   **Parameters**

>   - base – SMC peripheral base address.

>   - config – The LLS power mode configuration structure

>   **Returns**
>   SMC configuration error code.

status_t SMC_SetPowerModeVlls(SMC_Type *base, const *smc_power_mode_vlls_config_t* *config)
    Configures the system to VLLS power mode.

>   **Parameters**

>   - base – SMC peripheral base address.

>   - config – The VLLS power mode configuration structure.

>   **Returns**
>   SMC configuration error code.

FSL_SMC_DRIVER_VERSION
    SMC driver version.

enum __smc_power_mode_protection
    Power Modes Protection.

    Values:

    enumerator kSMC_AllowPowerModeVlls
        Allow Very-low-leakage Stop Mode.

    enumerator kSMC_AllowPowerModeLls
        Allow Low-leakage Stop Mode.

    enumerator kSMC_AllowPowerModeVlp
        Allow Very-Low-power Mode.

    enumerator kSMC_AllowPowerModeHsrun
        Allow High-speed Run mode.

    enumerator kSMC_AllowPowerModeAll
        Allow all power mode.

enum __smc_power_state
    Power Modes in PMSTAT.

    Values:

    enumerator kSMC_PowerStateRun
        0000_0001 - Current power mode is RUN

    enumerator kSMC_PowerStateStop
        0000_0010 - Current power mode is STOP

    enumerator kSMC_PowerStateVlpr
        0000_0100 - Current power mode is VLPR

    enumerator kSMC_PowerStateVlpw
        0000_1000 - Current power mode is VLPW

    enumerator kSMC_PowerStateVlps
        0001_0000 - Current power mode is VLPS

enumerator kSMC_PowerStateLls
: 0010_0000 - Current power mode is LLS

enumerator kSMC_PowerStateVlls
: 0100_0000 - Current power mode is VLLS

enumerator kSMC_PowerStateHsrun
: 1000_0000 - Current power mode is HSRUN

enum __smc_run_mode
: Run mode definition.

Values:

enumerator kSMC_RunNormal
: Normal RUN mode.

enumerator kSMC_RunVlpr
: Very-low-power RUN mode.

enumerator kSMC_Hsrun
: High-speed Run mode (HSRUN).

enum __smc_stop_mode
: Stop mode definition.

Values:

enumerator kSMC_StopNormal
: Normal STOP mode.

enumerator kSMC_StopVlps
: Very-low-power STOP mode.

enumerator kSMC_StopLls
: Low-leakage Stop mode.

enumerator kSMC_StopVlls
: Very-low-leakage Stop mode.

enum __smc_stop_submode
: VLLS/LLS stop sub mode definition.

Values:

enumerator kSMC_StopSub0
: Stop submode 0, for VLLS0/LLS0.

enumerator kSMC_StopSub1
: Stop submode 1, for VLLS1/LLS1.

enumerator kSMC_StopSub2
: Stop submode 2, for VLLS2/LLS2.

enumerator kSMC_StopSub3
: Stop submode 3, for VLLS3/LLS3.

enum __smc_partial_stop_mode
: Partial STOP option.

Values:

enumerator kSMC_PartialStop
: STOP - Normal Stop mode

---

enumerator kSMC_PartialStop1
> Partial Stop with both system and bus clocks disabled

enumerator kSMC_PartialStop2
> Partial Stop with system clock disabled and bus clock enabled

_smc_status, SMC configuration status.

Values:

enumerator kStatus_SMC_StopAbort
> Entering Stop mode is abort

typedef enum *_smc_power_mode_protection* smc_power_mode_protection_t
> Power Modes Protection.

typedef enum *_smc_power_state* smc_power_state_t
> Power Modes in PMSTAT.

typedef enum *_smc_run_mode* smc_run_mode_t
> Run mode definition.

typedef enum *_smc_stop_mode* smc_stop_mode_t
> Stop mode definition.

typedef enum *_smc_stop_submode* smc_stop_submode_t
> VLLS/LLS stop sub mode definition.

typedef enum *_smc_partial_stop_mode* smc_partial_stop_option_t
> Partial STOP option.

typedef struct *_smc_version_id* smc_version_id_t
> IP version ID definition.

typedef struct *_smc_param* smc_param_t
> IP parameter definition.

typedef struct *_smc_power_mode_lls_config* smc_power_mode_lls_config_t
> SMC Low-Leakage Stop power mode configuration.

typedef struct *_smc_power_mode_vlls_config* smc_power_mode_vlls_config_t
> SMC Very Low-Leakage Stop power mode configuration.

struct _smc_version_id
> *#include <fsl_smc.h>* IP version ID definition.

Public Members

uint16_t feature
> Feature Specification Number.

uint8_t minor
> Minor version number.

uint8_t major
> Major version number.

struct _smc_param
> *#include <fsl_smc.h>* IP parameter definition.

Public Members

bool hsrunEnable
    HSRUN mode enable.

bool llsEnable
    LLS mode enable.

bool lls2Enable
    LLS2 mode enable.

bool vlls0Enable
    VLLS0 mode enable.

struct __smc_power_mode_lls_config
    #include <fsl_smc.h> SMC Low-Leakage Stop power mode configuration.

Public Members

smc_stop_submode_t subMode
    Low-leakage Stop sub-mode

bool enableLpoClock
    Enable LPO clock in LLS mode

struct __smc_power_mode_vlls_config
    #include <fsl_smc.h> SMC Very Low-Leakage Stop power mode configuration.

Public Members

smc_stop_submode_t subMode
    Very Low-leakage Stop sub-mode

bool enablePorDetectInVlls0
    Enable Power on reset detect in VLLS mode

bool enableRam2InVlls2
    Enable RAM2 power in VLLS2

bool enableLpoClock
    Enable LPO clock in VLLS mode

2.39   SPI: Serial Peripheral Interface Driver

2.40   SPI DMA Driver

void SPI_MasterTransferCreateHandleDMA(SPI_Type *base, *spi_dma_handle_t* *handle,
                                       spi_dma_callback_t callback, void *userData,
                                       dma_handle_t *txHandle, *dma_handle_t* *rxHandle)
    Initialize the SPI master DMA handle.

    This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.

- handle – SPI handle pointer.

- callback – User callback function called at the end of a transfer.

- userData – User data for callback.

- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.

- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

status_t SPI_MasterTransferDMA(SPI_Type *base, *spi_dma_handle_t* *handle, *spi_transfer_t* *xfer)

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.

- handle – SPI DMA handle pointer.

- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.

- kStatus_InvalidArgument – Input argument is invalid.

- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

void SPI_MasterTransferAbortDMA(SPI_Type *base, *spi_dma_handle_t* *handle)

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.

- handle – SPI DMA handle pointer.

status_t SPI_MasterTransferGetCountDMA(SPI_Type *base, *spi_dma_handle_t* *handle, size_t *count)

Get the transferred bytes for SPI slave DMA.

Parameters

- base – SPI peripheral base address.

- handle – SPI DMA handle pointer.

- count – Transferred bytes.

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_SlaveTransferCreateHandleDMA(SPI_Type *base, *spi_dma_handle_t* *handle, *spi_dma_callback_t* callback, void *userData, *dma_handle_t* *txHandle, *dma_handle_t* *rxHandle)

Initialize the SPI slave DMA handle.

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

static inline *status_t* SPI_SlaveTransferDMA(SPI_Type *base, *spi_dma_handle_t* *handle, *spi_transfer_t* *xfer)

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

static inline void SPI_SlaveTransferAbortDMA(SPI_Type *base, *spi_dma_handle_t* *handle)

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

static inline *status_t* SPI_SlaveTransferGetCountDMA(SPI_Type *base, *spi_dma_handle_t* *handle, size_t *count)

Get the transferred bytes for SPI slave DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

- count – Transferred bytes.

Return values

- kStatus_SPI_Success – Succeed get the transfer count.

- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

FSL_SPI_DMA_DRIVER_VERSION

SPI DMA driver version.

typedef struct _spi_dma_handle spi_dma_handle_t

typedef void (*spi_dma_callback_t)(SPI_Type *base, *spi_dma_handle_t* *handle, *status_t* status, void *userData)

SPI DMA callback called at the end of transfer.

struct __spi_dma_handle

#include <fsl_spi_dma.h> SPI DMA transfer handle, users should not touch the content of the handle.

Public Members

bool txInProgress

Send transfer finished

bool rxInProgress

Receive transfer finished

dma_handle_t *txHandle

DMA handler for SPI send

dma_handle_t *rxHandle

DMA handler for SPI receive

uint8_t bytesPerFrame

Bytes in a frame for SPI transfer

spi_dma_callback_t callback

Callback for SPI DMA transfer

void *userData

User Data for SPI DMA callback

uint32_t state

Internal state of SPI DMA transfer

size_t transferSize

Bytes need to be transfer

2.41 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 SPSCK occurs at the middle of the first cycle of a data transfer.

    enumerator kSPI_ClockPhaseSecondEdge
        First edge on SPSCK 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

2.42 SYSMPU: System Memory Protection Unit

void SYSMPU_Init(SYSMPU_Type *base, const *sysmpu_config_t* *config)
    Initializes the SYSMPU with the user configuration structure.

    This function configures the SYSMPU module with the user-defined configuration.

        Parameters

            • base – SYSMPU peripheral base address.

            • config – The pointer to the configuration structure.

void SYSMPU_Deinit(SYSMPU_Type *base)

>   Deinitializes the SYSMPU regions.

>   >   **Parameters**

>   >   >   • base – SYSMPU peripheral base address.

static inline void SYSMPU_Enable(SYSMPU_Type *base, bool enable)

>   Enables/disables the SYSMPU globally.

>   Call this API to enable or disable the SYSMPU module.

>   >   **Parameters**

>   >   >   • base – SYSMPU peripheral base address.

>   >   >   • enable – True enable SYSMPU, false disable SYSMPU.

static inline void SYSMPU_RegionEnable(SYSMPU_Type *base, uint32_t number, bool enable)

>   Enables/disables the SYSMPU for a special region.

>   When SYSMPU is enabled, call this API to disable an unused region of an enabled SYSMPU. Call this API to minimize the power dissipation.

>   >   **Parameters**

>   >   >   • base – SYSMPU peripheral base address.

>   >   >   • number – SYSMPU region number.

>   >   >   • enable – True enable the special region SYSMPU, false disable the special region SYSMPU.

void SYSMPU_GetHardwareInfo(SYSMPU_Type *base, *sysmpu_hardware_info_t* *hardwareInform*)

>   Gets the SYSMPU basic hardware information.

>   >   **Parameters**

>   >   >   • base – SYSMPU peripheral base address.

>   >   >   • hardwareInform – The pointer to the SYSMPU hardware information structure. See "sysmpu_hardware_info_t".

void SYSMPU_SetRegionConfig(SYSMPU_Type *base, const *sysmpu_region_config_t* *regionConfig*)

>   Sets the SYSMPU region.

>   Note: Due to the SYSMPU protection, the region number 0 does not allow writes from core to affect the start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters.

>   >   **Parameters**

>   >   >   • base – SYSMPU peripheral base address.

>   >   >   • regionConfig – The pointer to the SYSMPU user configuration structure. See "sysmpu_region_config_t".

void SYSMPU_SetRegionAddr(SYSMPU_Type *base, uint32_t regionNum, uint32_t startAddr, uint32_t endAddr)

>   Sets the region start and end address.

>   Memory region start address. Note: bit0 ~ bit4 is always marked as 0 by SYSMPU. The actual start address by SYSMPU is 0-modulo-32 byte address. Memory region end address. Note: bit0 ~ bit4 always be marked as 1 by SYSMPU. The end address used by the SYSMPU is 31-modulo-32 byte address. Note: Due to the SYSMPU protection, the startAddr and endAddr can't be changed by the core when regionNum is 0.

Parameters

- base – SYSMPU peripheral base address.

- regionNum – SYSMPU region number. The range is from 0 to FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1.

- startAddr – Region start address.

- endAddr – Region end address.

void SYSMPU_SetRegionRwxMasterAccessRights(SYSMPU_Type *base, uint32_t regionNum, uint32_t masterNum, const *sysmpu_rwxrights_master_access_control_t* *accessRights)

Sets the SYSMPU region access rights for masters with read, write, and execute rights. The SYSMPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The privilege rights masters have the read, write, and execute access rights. Except the normal read and write rights, the execute rights are also allowed for these masters. The privilege rights masters normally range from bus masters 0 - 3. However, the maximum master number is device-specific. See the "SYSMPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX". The normal rights masters access rights control see "SYSMPU_SetRegionRwMasterAccessRights()".

Parameters

- base – SYSMPU peripheral base address.

- regionNum – SYSMPU region number. Should range from 0 to FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1.

- masterNum – SYSMPU bus master number. Should range from 0 to SYSMPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX.

- accessRights – The pointer to the SYSMPU access rights configuration. See "sysmpu_rwxrights_master_access_control_t".

bool SYSMPU_GetSlavePortErrorStatus(SYSMPU_Type *base, *sysmpu_slave_t* slaveNum)

Gets the numbers of slave ports where errors occur.

Parameters

- base – SYSMPU peripheral base address.

- slaveNum – SYSMPU slave port number.

Returns

The slave ports error status. true - error happens in this slave port. false - error didn't happen in this slave port.

void SYSMPU_GetDetailErrorAccessInfo(SYSMPU_Type *base, *sysmpu_slave_t* slaveNum, *sysmpu_access_err_info_t* *errInform)

Gets the SYSMPU detailed error access information.

Parameters

- base – SYSMPU peripheral base address.

- slaveNum – SYSMPU slave port number.

- errInform – The pointer to the SYSMPU access error information. See "sysmpu_access_err_info_t".

FSL_SYSMPU_DRIVER_VERSION

SYSMPU driver version 2.2.3.

enum __sysmpu_region_total_num
> Describes the number of SYSMPU regions.
>
> *Values:*
>
> enumerator kSYSMPU_8Regions
>> SYSMPU supports 8 regions.
>
> enumerator kSYSMPU_12Regions
>> SYSMPU supports 12 regions.
>
> enumerator kSYSMPU_16Regions
>> SYSMPU supports 16 regions.

enum __sysmpu_slave
> SYSMPU slave port number.
>
> *Values:*
>
> enumerator kSYSMPU_Slave0
>> SYSMPU slave port 0.
>
> enumerator kSYSMPU_Slave1
>> SYSMPU slave port 1.
>
> enumerator kSYSMPU_Slave2
>> SYSMPU slave port 2.
>
> enumerator kSYSMPU_Slave3
>> SYSMPU slave port 3.
>
> enumerator kSYSMPU_Slave4
>> SYSMPU slave port 4.

enum __sysmpu_err_access_control
> SYSMPU error access control detail.
>
> *Values:*
>
> enumerator kSYSMPU_NoRegionHit
>> No region hit error.
>
> enumerator kSYSMPU_NoneOverlappRegion
>> Access single region error.
>
> enumerator kSYSMPU_OverlappRegion
>> Access overlapping region error.

enum __sysmpu_err_access_type
> SYSMPU error access type.
>
> *Values:*
>
> enumerator kSYSMPU_ErrTypeRead
>> SYSMPU error access type — read.
>
> enumerator kSYSMPU_ErrTypeWrite
>> SYSMPU error access type — write.

enum __sysmpu_err_attributes
> SYSMPU access error attributes.
>
> *Values:*

enumerator kSYSMPU_InstructionAccessInUserMode
  Access instruction error in user mode.

enumerator kSYSMPU_DataAccessInUserMode
  Access data error in user mode.

enumerator kSYSMPU_InstructionAccessInSupervisorMode
  Access instruction error in supervisor mode.

enumerator kSYSMPU_DataAccessInSupervisorMode
  Access data error in supervisor mode.

enum __sysmpu_supervisor_access_rights
  SYSMPU access rights in supervisor mode for bus master 0 ~ 3.

  Values:

  enumerator kSYSMPU_SupervisorReadWriteExecute
    Read write and execute operations are allowed in supervisor mode.

  enumerator kSYSMPU_SupervisorReadExecute
    Read and execute operations are allowed in supervisor mode.

  enumerator kSYSMPU_SupervisorReadWrite
    Read write operations are allowed in supervisor mode.

  enumerator kSYSMPU_SupervisorEqualToUsermode
    Access permission equal to user mode.

enum __sysmpu_user_access_rights
  SYSMPU access rights in user mode for bus master 0 ~ 3.

  Values:

  enumerator kSYSMPU_UserNoAccessRights
    No access allowed in user mode.

  enumerator kSYSMPU_UserExecute
    Execute operation is allowed in user mode.

  enumerator kSYSMPU_UserWrite
    Write operation is allowed in user mode.

  enumerator kSYSMPU_UserWriteExecute
    Write and execute operations are allowed in user mode.

  enumerator kSYSMPU_UserRead
    Read is allowed in user mode.

  enumerator kSYSMPU_UserReadExecute
    Read and execute operations are allowed in user mode.

  enumerator kSYSMPU_UserReadWrite
    Read and write operations are allowed in user mode.

  enumerator kSYSMPU_UserReadWriteExecute
    Read write and execute operations are allowed in user mode.

typedef enum *_sysmpu_region_total_num* sysmpu_region_total_num_t
  Describes the number of SYSMPU regions.

typedef enum *_sysmpu_slave* sysmpu_slave_t
  SYSMPU slave port number.

---

typedef enum _*sysmpu_err_access_control* sysmpu_err_access_control_t
>   SYSMPU error access control detail.

typedef enum _*sysmpu_err_access_type* sysmpu_err_access_type_t
>   SYSMPU error access type.

typedef enum _*sysmpu_err_attributes* sysmpu_err_attributes_t
>   SYSMPU access error attributes.

typedef enum _*sysmpu_supervisor_access_rights* sysmpu_supervisor_access_rights_t
>   SYSMPU access rights in supervisor mode for bus master 0 ~ 3.

typedef enum _*sysmpu_user_access_rights* sysmpu_user_access_rights_t
>   SYSMPU access rights in user mode for bus master 0 ~ 3.

typedef struct _*sysmpu_hardware_info* sysmpu_hardware_info_t
>   SYSMPU hardware basic information.

typedef struct _*sysmpu_access_err_info* sysmpu_access_err_info_t
>   SYSMPU detail error access information.

typedef struct _*sysmpu_rwxrights_master_access_control*
sysmpu_rwxrights_master_access_control_t
>   SYSMPU read/write/execute rights control for bus master 0 ~ 3.

typedef struct _*sysmpu_rwrights_master_access_control*
sysmpu_rwrights_master_access_control_t
>   SYSMPU read/write access control for bus master 4 ~ 7.

typedef struct _*sysmpu_region_config* sysmpu_region_config_t
>   SYSMPU region configuration structure.
>
>   This structure is used to configure the regionNum region. The accessRights1[0] ~ access-Rights1[3] are used to configure the bus master 0 ~ 3 with the privilege rights setting. The accessRights2[0] ~ accessRights2[3] are used to configure the high master 4 ~ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note that the SYSMPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. SYSMPU protection does not allow writes from the core to affect the "regionNum 0" start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantees that the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

typedef struct _*sysmpu_config* sysmpu_config_t
>   The configuration structure for the SYSMPU initialization.
>
>   This structure is used when calling the SYSMPU_Init function.

SYSMPU_MASTER_RWATTRIBUTE_START_PORT
>   define the start master port with read and write attributes.

SYSMPU_REGION_RWXRIGHTS_MASTER_SHIFT(n)
>   SYSMPU the bit shift for masters with privilege rights: read write and execute.

SYSMPU_REGION_RWXRIGHTS_MASTER_MASK(n)
>   SYSMPU masters with read, write and execute rights bit mask.

SYSMPU_REGION_RWXRIGHTS_MASTER_WIDTH
>   SYSMPU masters with read, write and execute rights bit width.

SYSMPU_REGION_RWXRIGHTS_MASTER(n, x)
    SYSMPU masters with read, write and execute rights priority setting.

SYSMPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n)
    SYSMPU masters with read, write and execute rights process enable bit shift.

SYSMPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n)
    SYSMPU masters with read, write and execute rights process enable bit mask.

SYSMPU_REGION_RWXRIGHTS_MASTER_PE(n, x)
    SYSMPU masters with read, write and execute rights process enable setting.

SYSMPU_REGION_RWRIGHTS_MASTER_SHIFT(n)
    SYSMPU masters with normal read write permission bit shift.

SYSMPU_REGION_RWRIGHTS_MASTER_MASK(n)
    SYSMPU masters with normal read write rights bit mask.

SYSMPU_REGION_RWRIGHTS_MASTER(n, x)
    SYSMPU masters with normal read write rights priority setting.

struct __sysmpu_hardware_info
    #include <fsl_sysmpu.h> SYSMPU hardware basic information.

Public Members

uint8_t hardwareRevisionLevel
    Specifies the SYSMPU's hardware and definition reversion level.

uint8_t slavePortsNumbers
    Specifies the number of slave ports connected to SYSMPU.

sysmpu_region_total_num_t regionsNumbers
    Indicates the number of region descriptors implemented.

struct __sysmpu_access_err_info
    #include <fsl_sysmpu.h> SYSMPU detail error access information.

Public Members

uint32_t master
    Access error master.

sysmpu_err_attributes_t attributes
    Access error attributes.

sysmpu_err_access_type_t accessType
    Access error type.

sysmpu_err_access_control_t accessControl
    Access error control.

uint32_t address
    Access error address.

uint8_t processorIdentification
    Access error processor identification.

struct __sysmpu_rwxrights_master_access_control
    #include <fsl_sysmpu.h> SYSMPU read/write/execute rights control for bus master 0 ~ 3.

Public Members

sysmpu_supervisor_access_rights_t superAccessRights
    Master access rights in supervisor mode.

sysmpu_user_access_rights_t userAccessRights
    Master access rights in user mode.

bool processIdentifierEnable
    Enables or disables process identifier.

struct __sysmpu_rwrights_master_access_control
    #include <fsl_sysmpu.h> SYSMPU read/write access control for bus master 4 ~ 7.

Public Members

bool writeEnable
    Enables or disables write permission.

bool readEnable
    Enables or disables read permission.

struct __sysmpu_region_config
    #include <fsl_sysmpu.h> SYSMPU region configuration structure.

This structure is used to configure the regionNum region. The accessRights1[0] ~ accessRights1[3] are used to configure the bus master 0 ~ 3 with the privilege rights setting. The accessRights2[0] ~ accessRights2[3] are used to configure the high master 4 ~ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note that the SYSMPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. SYSMPU protection does not allow writes from the core to affect the "regionNum 0" start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantees that the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

Public Members

uint32_t regionNum
    SYSMPU region number, range form 0 ~ FSL_FEATURE_SYSMPU_DESCRIPTOR_COUNT - 1.

uint32_t startAddress
    Memory region start address. Note: bit0 ~ bit4 always be marked as 0 by SYSMPU. The actual start address is 0-modulo-32 byte address.

uint32_t endAddress
    Memory region end address. Note: bit0 ~ bit4 always be marked as 1 by SYSMPU. The actual end address is 31-modulo-32 byte address.

sysmpu_rwxrights_master_access_control_t accessRights1[4]
    Masters with read, write and execute rights setting.

sysmpu_rwrights_master_access_control_t accessRights2[4]
    Masters with normal read write rights setting.

uint8_t processIdentifier

> Process identifier used when "processIdentifierEnable" set with true.

uint8_t processIdMask

> Process identifier mask. The setting bit will ignore the same bit in process identifier.

struct __sysmpu_config

> *#include <fsl_sysmpu.h>* The configuration structure for the SYSMPU initialization.

> This structure is used when calling the SYSMPU_Init function.

Public Members

sysmpu_region_config_t regionConfig

> Region access permission.

struct *_sysmpu_config* *next

> Pointer to the next structure.

2.43 UART: Universal Asynchronous Receiver/Transmitter Driver

2.44 UART DMA Driver

void UART_TransferCreateHandleDMA(UART_Type *base, *uart_dma_handle_t* *handle, *uart_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *txDmaHandle, *dma_handle_t* *rxDmaHandle)

> Initializes the UART handle which is used in transactional functions and sets the callback.

> #### Parameters
>
> - base – UART peripheral base address.
> - handle – Pointer to the uart_dma_handle_t structure.
> - callback – UART callback, NULL means no callback.
> - userData – User callback function data.
> - rxDmaHandle – User requested DMA handle for the RX DMA transfer.
> - txDmaHandle – User requested DMA handle for the TX DMA transfer.

status_t UART_TransferSendDMA(UART_Type *base, *uart_dma_handle_t* *handle, *uart_transfer_t* *xfer)

> Sends data using DMA.

> This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

> #### Parameters
>
> - base – UART peripheral base address.
> - handle – UART handle pointer.
> - xfer – UART DMA transfer structure. See uart_transfer_t.

> #### Return values
>
> - kStatus_Success – if succeeded; otherwise failed.

- kStatus_UART_TxBusy – Previous transfer ongoing.

- kStatus_InvalidArgument – Invalid argument.

status_t UART_TransferReceiveDMA(UART_Type *base, *uart_dma_handle_t* *handle, *uart_transfer_t* *xfer)

Receives data using DMA.

This function receives data using DMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – UART peripheral base address.

- handle – Pointer to the uart_dma_handle_t structure.

- xfer – UART DMA transfer structure. See uart_transfer_t.

Return values

- kStatus_Success – if succeeded; otherwise failed.

- kStatus_UART_RxBusy – Previous transfer on going.

- kStatus_InvalidArgument – Invalid argument.

void UART_TransferAbortSendDMA(UART_Type *base, *uart_dma_handle_t* *handle)

Aborts the send data using DMA.

This function aborts the sent data using DMA.

Parameters

- base – UART peripheral base address.

- handle – Pointer to uart_dma_handle_t structure.

void UART_TransferAbortReceiveDMA(UART_Type *base, *uart_dma_handle_t* *handle)

Aborts the received data using DMA.

This function abort receive data which using DMA.

Parameters

- base – UART peripheral base address.

- handle – Pointer to uart_dma_handle_t structure.

status_t UART_TransferGetSendCountDMA(UART_Type *base, *uart_dma_handle_t* *handle, uint32_t *count)

Gets the number of bytes written to UART TX register.

This function gets the number of bytes written to UART TX register by DMA.

Parameters

- base – UART peripheral base address.

- handle – UART handle pointer.

- count – Send bytes count.

Return values

- kStatus_NoTransferInProgress – No send in progress.

- kStatus_InvalidArgument – Parameter is invalid.

- kStatus_Success – Get successfully through the parameter count;

status_t UART_TransferGetReceiveCountDMA(UART_Type *base, *uart_dma_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;

void UART_TransferDMAHandleIRQ(UART_Type *base, void *uartDmaHandle)

UART DMA IRQ handle function.

This function handles the UART transmit complete IRQ request and invoke user callback.

Parameters

- base – UART peripheral base address.
- uartDmaHandle – UART handle pointer.

FSL_UART_DMA_DRIVER_VERSION

UART DMA driver version.

typedef struct *_uart_dma_handle* uart_dma_handle_t

typedef void (*uart_dma_transfer_callback_t)(UART_Type *base, *uart_dma_handle_t* *handle, *status_t* status, void *userData)

UART transfer callback function.

struct __uart_dma_handle

#include <fsl_uart_dma.h> UART DMA handle.

Public Members

UART_Type *base

UART peripheral base address.

uart_dma_transfer_callback_t callback

Callback function.

void *userData

UART callback function parameter.

size_t rxDataSizeAll

Size of the data to receive.

size_t txDataSizeAll

Size of the data to send out.

dma_handle_t *txDmaHandle

The DMA TX channel used.

dma_handle_t *rxDmaHandle
>   The DMA RX channel used.

volatile uint8_t txState
>   TX transfer state.

volatile uint8_t rxState
>   RX transfer state

2.45   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, 20000000U);
```

>   **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, flase 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 receices 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 ___unnamed27___

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.

2.46 VREF: Voltage Reference Driver

status_t VREF_Init(VREF_Type *base, const *vref_config_t* *config)

Enables the clock gate and configures the VREF module according to the configuration structure.

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up vref_config_t parameters and how to call the VREF_Init function by passing in these parameters. This is an example.

```
vref_config_t vrefConfig;
vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
vrefConfig.enableExternalVoltRef = false;
vrefConfig.enableLowRef = false;
VREF_Init(VREF, &vrefConfig);
```

Parameters

- base – VREF peripheral address.
- config – Pointer to the configuration structure.

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

void VREF_Deinit(VREF_Type *base)

Stops and disables the clock for the VREF module.

This function should be called to shut down the module. This is an example.

```
vref_config_t vrefUserConfig;
VREF_Init(VREF);
VREF_GetDefaultConfig(&vrefUserConfig);
…
VREF_Deinit(VREF);
```

Parameters

- base – VREF peripheral address.

void VREF_GetDefaultConfig(*vref_config_t* *config)

Initializes the VREF configuration structure.

This function initializes the VREF configuration structure to default values. This is an example.

```
vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
vrefConfig->enableExternalVoltRef = false;
vrefConfig->enableLowRef = false;
```

Parameters

- config – Pointer to the initialization structure.

status_t VREF_SetTrimVal(VREF_Type *base, uint8_t trimValue)

Sets a TRIM value for the reference voltage.

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

static inline uint8_t VREF_GetTrimVal(VREF_Type *base)

Reads the value of the TRIM meaning output voltage.

This function gets the TRIM value from the TRM register.

Parameters

- base – VREF peripheral address.

Returns

Six-bit value of trim setting.

status_t VREF_SetLowReferenceTrimVal(VREF_Type *base, uint8_t trimValue)

Sets the TRIM value for the low voltage reference.

This function sets the TRIM value for low reference voltage. Note the following.

- The TRIM value maximum is 0x05U
- The values 111b and 110b are not valid/allowed.

Parameters

- base – VREF peripheral address.
- trimValue – Value of the trim register to set output low reference voltage (maximum 0x05U (3-bit)).

Return values

- kStatus_Success – run success.
- kStatus_Timeout – timeout occurs.

static inline uint8_t VREF_GetLowReferenceTrimVal(VREF_Type *base)

Reads the value of the TRIM meaning output voltage.

This function gets the TRIM value from the VREFL_TRM register.

Parameters

- base – VREF peripheral address.

Returns

Three-bit value of the trim setting.

FSL_VREF_DRIVER_VERSION

Version 2.1.3.

VREF_INTERNAL_VOLTAGE_STABLE_TIMEOUT

Max loops to wait for VREF internal voltage stable.

This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

enum _vref_buffer_mode

VREF modes.

Values:

enumerator kVREF_ModeBandgapOnly

Bandgap on only, for stabilization and startup

enumerator kVREF_ModeHighPowerBuffer

High-power buffer mode enabled

enumerator kVREF_ModeLowPowerBuffer

Low-power buffer mode enabled

typedef enum *_vref_buffer_mode* vref_buffer_mode_t

VREF modes.

typedef struct *_vref_config* vref_config_t

The description structure for the VREF module.

VREF_SC_MODE_LV

VREF_SC_REGEN

VREF_SC_VREFEN

VREF_SC_ICOMPEN

VREF_SC_REGEN_MASK

VREF_SC_VREFST_MASK

VREF_SC_VREFEN_MASK

VREF_SC_MODE_LV_MASK

VREF_SC_ICOMPEN_MASK

TRM

VREF_TRM_TRIM

VREF_TRM_CHOPEN_MASK

VREF_TRM_TRIM_MASK

2.46. VREF: Voltage Reference Driver

VREF_TRM_CHOPEN_SHIFT

VREF_TRM_TRIM_SHIFT

VREF_SC_MODE_LV_SHIFT

VREF_SC_REGEN_SHIFT

VREF_SC_VREFST_SHIFT

VREF_SC_ICOMPEN_SHIFT

struct __vref_config
> *#include <fsl_vref.h>* The description structure for the VREF module.

> ### Public Members

> *vref_buffer_mode_t* bufferMode
> > Buffer mode selection

> bool enableLowRef
> > Set VREFL (0.4 V) reference buffer enable or disable

> bool enableExternalVoltRef
> > Select external voltage reference or not (internal)

2.47 WDOG: Watchdog Timer Driver

void WDOG_GetDefaultConfig(*wdog_config_t* *config)
> Initializes the WDOG configuration structure.

> This function initializes the WDOG configuration structure to default values. The default values are as follows.

```
wdogConfig->enableWdog = true;
wdogConfig->clockSource = kWDOG_LpoClockSource;
wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
wdogConfig->workMode.enableWait = true;
wdogConfig->workMode.enableStop = false;
wdogConfig->workMode.enableDebug = false;
wdogConfig->enableUpdate = true;
wdogConfig->enableInterrupt = false;
wdogConfig->enableWindowMode = false;
wdogConfig->windowValue = 0;
wdogConfig->timeoutValue = 0xFFFFU;
```

> **See also:**

> wdog_config_t

> > ### Parameters

> > - config – Pointer to the WDOG configuration structure.

void WDOG_Init(WDOG_Type *base, const *wdog_config_t* *config)

> Initializes the WDOG.

> This function initializes the WDOG. When called, the WDOG runs according to the configuration. To reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in the configuration.

> This is an example.

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG_Init(wdog_base,&config);
```

> **Parameters**
> - base – WDOG peripheral base address
> - config – The configuration of WDOG

void WDOG_Deinit(WDOG_Type *base)

> Shuts down the WDOG.

> This function shuts down the WDOG. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which indicates that the register update is enabled.

void WDOG_SetTestModeConfig(WDOG_Type *base, *wdog_test_config_t* *config)

> Configures the WDOG functional test.

> This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

> This is an example.

```
wdog_test_config_t test_config;
test_config.testMode = kWDOG_QuickTest;
test_config.timeoutValue = 0xffffu;
WDOG_SetTestModeConfig(wdog_base, &test_config);
```

> **Parameters**
> - base – WDOG peripheral base address
> - config – The functional test configuration of WDOG

static inline void WDOG_Enable(WDOG_Type *base)

> Enables the WDOG module.

> This function write value into WDOG_STCTRLH register to enable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

> **Parameters**
> - base – WDOG peripheral base address

static inline void WDOG_Disable(WDOG_Type *base)

> Disables the WDOG module.

> This function writes a value into the WDOG_STCTRLH register to disable the WDOG. It is a write-once register. Ensure that the WCT window is still open and that register has not been written to in this WCT while the function is called.

> **Parameters**

> • base – WDOG peripheral base address

static inline void WDOG_EnableInterrupts(WDOG_Type *base, uint32_t mask)

> Enables the WDOG interrupt.
>
> This function writes a value into the WDOG_STCTRLH register to enable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.
>
> ### Parameters
>
> > • base – WDOG peripheral base address
> >
> > • mask – The interrupts to enable The parameter can be combination of the following source if defined.
> >
> > > – kWDOG_InterruptEnable

static inline void WDOG_DisableInterrupts(WDOG_Type *base, uint32_t mask)

> Disables the WDOG interrupt.
>
> This function writes a value into the WDOG_STCTRLH register to disable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.
>
> ### Parameters
>
> > • base – WDOG peripheral base address
> >
> > • mask – The interrupts to disable The parameter can be combination of the following source if defined.
> >
> > > – kWDOG_InterruptEnable

uint32_t WDOG_GetStatusFlags(WDOG_Type *base)

> Gets the WDOG all status flags.
>
> This function gets all status flags.
>
> This is an example for getting the Running Flag.

```
uint32_t status;
status = WDOG_GetStatusFlags (wdog_base) & kWDOG_RunningFlag;
```

> ### See also:
>
> _wdog_status_flags_t
>
> • true: a related status flag has been set.
>
> • false: a related status flag is not set.
>
> ### Parameters
>
> > • base – WDOG peripheral base address
>
> ### Returns
>
> > State of the status flag: asserted (true) or not-asserted (false).

void WDOG_ClearStatusFlags(WDOG_Type *base, uint32_t mask)

> Clears the WDOG flag.
>
> This function clears the WDOG status flag.
>
> This is an example for clearing the timeout (interrupt) flag.

```
WDOG_ClearStatusFlags(wdog_base,kWDOG_TimeoutFlag);
```

> ### Parameters

- base – WDOG peripheral base address

- mask – The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag

static inline void WDOG_SetTimeoutValue(WDOG_Type *base, uint32_t timeoutCount)

Sets the WDOG timeout value.

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than 2xWCT time + 20 bus clock cycles. This function writes a value into WDOG_TOVALH and WDOG_TOVALL registers which are wirte-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address

- timeoutCount – WDOG timeout value; count of WDOG clock tick.

static inline void WDOG_SetWindowValue(WDOG_Type *base, uint32_t windowValue)

Sets the WDOG window value.

This function sets the WDOG window value. This function writes a value into WDOG_WINH and WDOG_WINL registers which are wirte-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

- base – WDOG peripheral base address

- windowValue – WDOG window value.

static inline void WDOG_Unlock(WDOG_Type *base)

Unlocks the WDOG register written.

This function unlocks the WDOG register written. Before starting the unlock sequence and following configuration, disable the global interrupts. Otherwise, an interrupt may invalidate the unlocking sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

- base – WDOG peripheral base address

void WDOG_Refresh(WDOG_Type *base)

Refreshes the WDOG timer.

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WDOG peripheral base address

static inline uint16_t WDOG_GetResetCount(WDOG_Type *base)

Gets the WDOG reset count.

This function gets the WDOG reset count value.

Parameters

- base – WDOG peripheral base address

Returns

WDOG reset count value.

static inline void WDOG_ClearResetCount(WDOG_Type *base)

    Clears the WDOG reset count.

    This function clears the WDOG reset count value.

        Parameters

                • base – WDOG peripheral base address

FSL_WDOG_DRIVER_VERSION

    Defines WDOG driver version 2.0.2.

WDOG_FIRST_WORD_OF_UNLOCK

    First word of unlock sequence

WDOG_SECOND_WORD_OF_UNLOCK

    Second word of unlock sequence

WDOG_FIRST_WORD_OF_REFRESH

    First word of refresh sequence

WDOG_SECOND_WORD_OF_REFRESH

    Second word of refresh sequence

enum __wdog_clock_source

    Describes WDOG clock source.

    Values:

    enumerator kWDOG_LpoClockSource

        WDOG clock sourced from LPO

    enumerator kWDOG_AlternateClockSource

        WDOG clock sourced from alternate clock source

enum __wdog_clock_prescaler

    Describes the selection of the clock prescaler.

    Values:

    enumerator kWDOG_ClockPrescalerDivide1

        Divided by 1

    enumerator kWDOG_ClockPrescalerDivide2

        Divided by 2

    enumerator kWDOG_ClockPrescalerDivide3

        Divided by 3

    enumerator kWDOG_ClockPrescalerDivide4

        Divided by 4

    enumerator kWDOG_ClockPrescalerDivide5

        Divided by 5

    enumerator kWDOG_ClockPrescalerDivide6

        Divided by 6

    enumerator kWDOG_ClockPrescalerDivide7

        Divided by 7

    enumerator kWDOG_ClockPrescalerDivide8

        Divided by 8

enum __wdog__test__mode
>   Describes WDOG test mode.
>
>   *Values:*
>
>   enumerator kWDOG__QuickTest
>   >   Selects quick test
>
>   enumerator kWDOG__ByteTest
>   >   Selects byte test

enum __wdog__tested__byte
>   Describes WDOG tested byte selection in byte test mode.
>
>   *Values:*
>
>   enumerator kWDOG__TestByte0
>   >   Byte 0 selected in byte test mode
>
>   enumerator kWDOG__TestByte1
>   >   Byte 1 selected in byte test mode
>
>   enumerator kWDOG__TestByte2
>   >   Byte 2 selected in byte test mode
>
>   enumerator kWDOG__TestByte3
>   >   Byte 3 selected in byte test mode

enum __wdog__interrupt__enable__t
>   WDOG interrupt configuration structure, default settings all disabled.
>
>   This structure contains the settings for all of the WDOG interrupt configurations.
>
>   *Values:*
>
>   enumerator kWDOG__InterruptEnable
>   >   WDOG timeout generates an interrupt before reset

enum __wdog__status__flags__t
>   WDOG status flags.
>
>   This structure contains the WDOG status flags for use in the WDOG functions.
>
>   *Values:*
>
>   enumerator kWDOG__RunningFlag
>   >   Running flag, set when WDOG is enabled
>
>   enumerator kWDOG__TimeoutFlag
>   >   Interrupt flag, set when an exception occurs

typedef enum *_wdog_clock_source* wdog__clock__source__t
>   Describes WDOG clock source.

typedef struct *_wdog_work_mode* wdog__work__mode__t
>   Defines WDOG work mode.

typedef enum *_wdog_clock_prescaler* wdog__clock__prescaler__t
>   Describes the selection of the clock prescaler.

typedef struct *_wdog_config* wdog__config__t
>   Describes WDOG configuration structure.

typedef enum *_wdog_test_mode* wdog__test__mode__t
>   Describes WDOG test mode.

---

typedef enum _*wdog_tested_byte* wdog_tested_byte_t
>   Describes WDOG tested byte selection in byte test mode.

typedef struct _*wdog_test_config* wdog_test_config_t
>   Describes WDOG test mode configuration structure.

WDOG_WCT_INSTRUCITON_COUNT
>   < Watchdog configuration time window

struct __wdog_work_mode
>   *#include <fsl_wdog.h>* Defines WDOG work mode.

Public Members

bool enableWait
>   Enables or disables WDOG in wait mode

bool enableStop
>   Enables or disables WDOG in stop mode

bool enableDebug
>   Enables or disables WDOG in debug mode

struct __wdog_config
>   *#include <fsl_wdog.h>* Describes WDOG configuration structure.

Public Members

bool enableWdog
>   Enables or disables WDOG

wdog_clock_source_t clockSource
>   Clock source select

wdog_clock_prescaler_t prescaler
>   Clock prescaler value

wdog_work_mode_t workMode
>   Configures WDOG work mode in debug stop and wait mode

bool enableUpdate
>   Update write-once register enable

bool enableInterrupt
>   Enables or disables WDOG interrupt

bool enableWindowMode
>   Enables or disables WDOG window mode

uint32_t windowValue
>   Window value

uint32_t timeoutValue
>   Timeout value

struct __wdog_test_config
>   *#include <fsl_wdog.h>* Describes WDOG test mode configuration structure.

Public Members

wdog_test_mode_t testMode
    Selects test mode

wdog_tested_byte_t testedByte
    Selects tested byte in byte test mode

uint32_t timeoutValue
    Timeout value

2.48 XBAR: Inter-Peripheral Crossbar Switch

void XBAR_Init(XBAR_Type *base)
    Initializes the XBAR modules.

    This function un-gates the XBAR clock.

      Parameters

        • base – XBAR peripheral address.

void XBAR_Deinit(XBAR_Type *base)
    Shutdown the XBAR modules.

    This function disables XBAR clock.

      Parameters

        • base – XBAR peripheral address.

void XBAR_SetSignalsConnection(XBAR_Type *base, xbar_input_signal_t input,
                         xbar_output_signal_t output)
    Set connection between the selected XBAR_IN[*] input and the XBAR_OUT[*] output signal.

    This function connects the XBAR input to the selected XBAR output. If more than one XBAR module is available, only the inputs and outputs from the same module can be connected.

    Example:

```
XBAR_SetSignalsConnection(XBAR, kXBAR_InputTMR_CH0_Output, kXBAR_OutputXB_DMA_
↪INT2);
```

      Parameters

        • base – XBAR peripheral address

        • input – XBAR input signal.

        • output – XBAR output signal.

void XBAR_ClearStatusFlags(XBAR_Type *base, uint32_t mask)
    Clears the edge detection status flags of relative mask.

      Parameters

        • base – XBAR peripheral address

        • mask – the status flags to clear.

uint32_t XBAR_GetStatusFlags(XBAR_Type *base)

Gets the active edge detection status.

This function gets the active edge detect status of all XBAR_OUTs. If the active edge occurs, the return value is asserted. When the interrupt or the DMA functionality is enabled for the XBAR_OUTx, this field is 1 when the interrupt or DMA request is asserted and 0 when the interrupt or DMA request has been cleared.

Example:

```
uint32_t status;

status = XBAR_GetStatusFlags(XBAR);
```

Parameters

- base – XBAR peripheral address.

Returns

the mask of these status flag bits.

void XBAR_SetOutputSignalConfig(XBAR_Type *base, xbar_output_signal_t output, const *xbar_control_config_t* *controlConfig)

Configures the XBAR control register.

This function configures an XBAR control register. The active edge detection and the DMA/IRQ function on the corresponding XBAR output can be set.

Example:

```
xbar_control_config_t userConfig;
userConfig.activeEdge = kXBAR_EdgeRising;
userConfig.requestType = kXBAR_RequestInterruptEnalbe;
XBAR_SetOutputSignalConfig(XBAR, kXBAR_OutputXB_DMA_INT0, &userConfig);
```

Parameters

- base – XBAR peripheral address

- output – XBAR output number.

- controlConfig – Pointer to structure that keeps configuration of control register.

enum __xbar_active_edge

XBAR active edge for detection.

Values:

enumerator kXBAR_EdgeNone

Edge detection status bit never asserts.

enumerator kXBAR_EdgeRising

Edge detection status bit asserts on rising edges.

enumerator kXBAR_EdgeFalling

Edge detection status bit asserts on falling edges.

enumerator kXBAR_EdgeRisingAndFalling

Edge detection status bit asserts on rising and falling edges.

enum __xbar_request

Defines the XBAR DMA and interrupt configurations.

Values:

enumerator kXBAR_RequestDisable
   Interrupt and DMA are disabled.

enumerator kXBAR_RequestDMAEnable
   DMA enabled, interrupt disabled.

enumerator kXBAR_RequestInterruptEnalbe
   Interrupt enabled, DMA disabled.

enum _xbar_status_flag_t
   XBAR status flags.

   This provides constants for the XBAR status flags for use in the XBAR functions.

   Values:

   enumerator kXBAR_EdgeDetectionOut0
      XBAR_OUT0 active edge interrupt flag, sets when active edge detected.

typedef enum *_xbar_active_edge* xbar_active_edge_t
   XBAR active edge for detection.

typedef enum *_xbar_request* xbar_request_t
   Defines the XBAR DMA and interrupt configurations.

typedef enum *_xbar_status_flag_t* xbar_status_flag_t
   XBAR status flags.

   This provides constants for the XBAR status flags for use in the XBAR functions.

typedef struct *_xbar_control_config* xbar_control_config_t
   Defines the configuration structure of the XBAR control register.

   This structure keeps the configuration of XBAR control register for one output. Control registers are available only for a few outputs. Not every XBAR module has control registers.

FSL_XBAR_DRIVER_VERSION

XBAR_SELx(base, output)

XBAR_WR_SELx_SELx(base, input, output)

struct _xbar_control_config
   #include <fsl_xbar.h> Defines the configuration structure of the XBAR control register.

   This structure keeps the configuration of XBAR control register for one output. Control registers are available only for a few outputs. Not every XBAR module has control registers.

   Public Members

   xbar_active_edge_t activeEdge
      Active edge to be detected.

   xbar_request_t requestType
      Selects DMA/Interrupt request.

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.

3.1. Motor Control   **391**

- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.

- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.

- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.

- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.

- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.

- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.

- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.

- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample spplications 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 FM-STR_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 FreeMAS-TER 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 FM-STR_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_RQUEUE_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-langiage 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_UINT*n*	Unsigned integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_SINT*n*	Signed integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_FRAC*n*	Fractional number of size *n* bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(*m,n*)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(*m,n*)	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(*name*)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()

FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *fmstr:* protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)

/* Directory entry applies to all subsequent MEMFILE entries */
FMSTR_TSA_DIRECTORY("/text_files")     /* entering a new virtual directory */
```

<div align="right">(continues on next page)</div>

```
/* 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
FM-STR_ADDR	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.
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.	
FM-STR_SIZE	Data type used to hold the memory block size.
It is required that this type is unsigned and at least 16 bits wide integer.	
FM-STR_BOOL	Data type used as a general boolean type.
This type is used only in zero/non-zero conditions in the driver code.	
FM-STR_APPCM	Data type used to hold the Application Command code.
Generally, this is an unsigned 8-bit value.	
FM-STR_APPCM	Data type used to create the Application Command data buffer.
Generally, this is an unsigned 8-bit value.	
FM-STR_APPCM	Data type used to hold the Application Command result code.
Generally, this is an unsigned 8-bit value.	

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.

FM-STR_TSA_TII By default, this is defined as FM-STR_SIZE.	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables.
FM-STR_TSA_TS. By default, this is defined as FM-STR_SIZE.	Data type used to hold a memory block size, as used in the TSA descriptors.

Public Pipes types   The table describes the data types used by the FreeMASTER Pipes API:

FM-STR_HPIPE Generally, this is a pointer to a void type.	Pipe handle that identifies the open-pipe object.
FM-STR_PIPE_P(Generally, this is an unsigned 8-bit or 16-bit type.	Integer type required to hold at least 7 bits of data.
FM-STR_PIPE_SI This is used to store the data buffer sizes.	Integer type required to hold at least 16 bits of data.
FM-STR_PPIPEF(See *FM-STR_PipeOpen* for more details.	Pointer to the pipe handler function.

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.

FMSTR_U8	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.	
FMSTR_U16	Unsigned 16-bit integer.
FMSTR_U32	Unsigned 32-bit integer.
FMSTR_S8	Signed 8-bit integer.
FMSTR_S16	Signed 16-bit integer.
FMSTR_S32	Signed 32-bit integer.
FMSTR_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_FLAGS	Data type forming a union with a structure of flag bit-fields.
FMSTR_SIZE8	Data type holding a general size value, at least 8 bits wide.
FMSTR_INDEX	General for-loop index. Must be signed, at least 16 bits wide.
FMSTR_BCHR	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.	
FMSTR_BPTR	A pointer to the communication buffer (an array of FMSTR_BCHR).

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

Overview   The purpose of this document is to describes the FreeRTOS kernel repo integration into the NXP MCUXpresso Software Development Kit: mcuxsdk. MCUXpresso SDK provides a comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on MCUs from NXP. This project involves the FreeRTOS kernel repo fork with:

- cmake and Kconfig support to allow the configuration and build in MCUXpresso SDK ecosystem

- FreeRTOS OS additions, such as FreeRTOS driver wrappers, RTOS ready FatFs file system, and the implementation of FreeRTOS tickless mode

The history of changes in FreeRTOS kernel repo for MCUXpresso SDK are summarized in *CHANGELOG_mcuxsdk.md* file.

The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

FreeRTOS example applications   The FreeRTOS examples are written to demonstrate basic FreeRTOS features and the interaction between peripheral drivers and the RTOS.

List of examples   The list of freertos_examples, their description and availability for individual supported MCUXpresso SDK development boards can be obtained here: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos_examples/index.html

Location of examples   The FreeRTOS examples are located in mcuxsdk-examples repository, see the freertos_examples folder.

Once using MCUXpresso SDK zip packages created via the MCUXpresso SDK Builder the FreeRTOS kernel library and associated freertos_examples are added into final zip package once FreeRTOS components is selected on the Developer Environment Settings page:
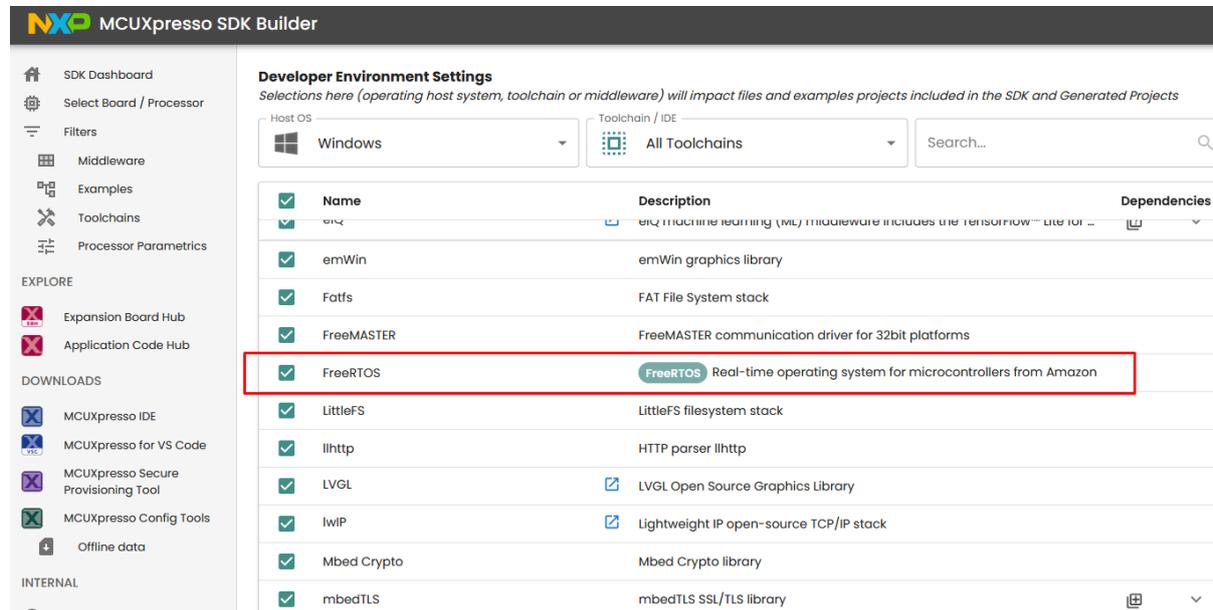
The FreeRTOS examples in MCUXpresso SDK zip packages are located in <MCUXpressoSDK_install_dir>/boards/<board_name>/freertos_examples/ subfolders.

Building a FreeRTOS example application   For information how to use the cmake and Kconfig based build and configuration system and how to build freertos_examples visit: MCUXpresso SDK documentation for Build And Configuration MCUXpresso SDK Getting Start Guide

Tip: To list all FreeRTOS example projects and targets that can be built via the west build command, use this west list_project command in mcuxsdk workspace:

```
west list_project -p examples/freertos_examples
```

FreeRTOS aware debugger plugin   NXP provides FreeRTOS task aware debugger for GDB. The plugin is compatible with Eclipse-based (MCUXpressoIDE) and is available after the installation.

FreeRTOS kernel for MCUXpresso SDK ChangeLog

Changelog FreeRTOS kernel for MCUXpresso SDK   All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[Unreleased]

Added

- Kconfig added CONFIG_FREERTOS_USE_CUSTOM_CONFIG_FRAGMENT config to optionally include custom FreeRTOSConfig fragment
  include file FreeRTOSConfig_frag.h. File must be provided by application.

- Added missing Kconfig option for configUSE_PICOLIBC_TLS.

- Add correct header files to build when configUSE_NEWLIB_REENTRANT and configUSE_PICOLIBC_TLS is selected in config.

[11.1.0_rev0]

- update amazon freertos version

[11.0.1_rev0]

- update amazon freertos version

[10.5.1_rev0]

- update amazon freertos version

[10.4.3_rev1]

- Apply CM33 security fix from 10.4.3-LTS-Patch-2. See rtos\freertos\freertos_kernel\History.txt
- Apply CM33 security fix from 10.4.3-LTS-Patch-1. See rtos\freertos\freertos_kernel\History.txt

[10.4.3_rev0]

- update amazon freertos version.

[10.4.3_rev0]

- update amazon freertos version.

[9.0.0_rev3]

- New features:
  - Tickless idle mode support for Cortex-A7. Add fsl_tickless_epit.c and fsl_tickless_generic.h in portable/IAR/ARM_CA9 folder.
  - Enabled float context saving in IAR for Cortex-A7. Added configUSE_TASK_FPU_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM_CA9 folder.
- Other changes:
  - Transformed ARM_CM core specific tickless low power support into generic form under freertos/Source/portable/low_power_tickless/.

[9.0.0_rev2]

- New features:

    - Enabled MCUXpresso thread aware debugging. Add freertos_tasks_c_additions.h and configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H and configFR-TOS_MEMORY_SCHEME macros.

[9.0.0_rev1]

- New features:

    - Enabled -flto optimization in GCC by adding **attribute**((used)) for vTaskSwitchContext.

    - Enabled KDS Task Aware Debugger. Apply FreeRTOS patch to enable configgRECORD_STACK_HIGH_ADDRESS macro. Modified files are task.c and FreeRTOS.h.

[9.0.0_rev0]

- New features:

    - Example freertos_sem_static.

    - Static allocation support RTOS driver wrappers.

- Other changes:

    - Tickless idle rework. Support for different timers is in separated files (fsl_tickless_systick.c, fsl_tickless_lptmr.c).

    - Removed configuration option configSYSTICK_USE_LOW_POWER_TIMER. Low power timer is now selected by linking of apropriate file fsl_tickless_lptmr.c.

    - Removed configOVERRIDE_DEFAULT_TICK_CONFIGURATION in RVDS port. Use of **attribute**((weak)) is the preferred solution. Not same as _weak!

[8.2.3]

- New features:

    - Tickless idle mode support.

    - Added template application for Kinetis Expert (KEx) tool (template_application).

- Other changes:

    - Folder structure reduction. Keep only Kinetis related parts.

FreeRTOS kernel Readme

MCUXpresso SDK: FreeRTOS kernel This repository is a fork of FreeRTOS kernel (https://github.com/FreeRTOS/FreeRTOS-Kernel)(11.1.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS kernel repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

For more information about the FreeRTOS kernel repo adoption see README_mcuxsdk.md: FreeRTOS kernel for MCUXpresso SDK Readme document.

Getting started This repository contains FreeRTOS kernel source/header files and kernel ports only. This repository is referenced as a submodule in FreeRTOS/FreeRTOS repository, which contains pre-configured demo application projects under FreeRTOS/Demo directory.

The easiest way to use FreeRTOS is to start with one of the pre-configured demo application projects. That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the FreeRTOS Kernel Quick Start Guide for detailed instructions and other useful links.

Additionally, for FreeRTOS kernel feature information refer to the Developer Documentation, and API Reference.

Also for contributing and creating a Pull Request please refer to *the instructions here.*

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the FreeRTOS Community Support Forum.

To consume FreeRTOS-Kernel

Consume with CMake If using CMake, it is recommended to use this repository using Fetch-Content. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_kernel
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Kernel.git
  GIT_TAG        main #Note: Best practice to use specific git-hash or tagged version
)
```

In case you prefer to add it as a git submodule, do:

```
git submodule add https://github.com/FreeRTOS/FreeRTOS-Kernel.git <path of the submodule>
git submodule update --init
```

- Add a freertos_config library (typically an INTERFACE library) The following assumes the directory structure:
    - include/FreeRTOSConfig.h

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
INTERFACE
    include
)

target_compile_definitions(freertos_config
  INTERFACE
    projCOVERAGE_TEST=0
)
```

In case you installed FreeRTOS-Kernel as a submodule, you will have to add it as a subdirectory:

```
add_subdirectory(${FREERTOS_PATH})
```

- Configure the FreeRTOS-Kernel and make it available
    - this particular example supports a native and cross-compiled build option.

```
set( FREERTOS_HEAP "4" CACHE STRING "" FORCE)
# Select the native compile PORT
set( FREERTOS_PORT "GCC_POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  set(FREERTOS_PORT "GCC_ARM_CA9" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_kernel)
```

- In case of cross compilation, you should also add the following to freertos_config:

```
target_compile_definitions(freertos_config INTERFACE ${definitions})
target_compile_options(freertos_config INTERFACE ${options})
```

Consuming stand-alone - Cloning this repository   To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
```

Repository structure

- The root of this repository contains the three files that are common to every port - list.c, queue.c and tasks.c. The kernel is contained within these three files. croutine.c implements the optional co-routine functionality - which is normally only used on very memory limited systems.

- The ./portable directory contains the files that are specific to a particular microcontroller and/or compiler. See the readme file in the ./portable directory for more information.

- The ./include directory contains the real time kernel header files.

- The ./template_configuration directory contains a sample FreeRTOSConfig.h to help jumpstart a new project. See the *FreeRTOSConfig.h* file for instructions.

Code Formatting   FreeRTOS files are formatted using the "uncrustify" tool. The configuration file used by uncrustify can be found in the FreeRTOS/CI-CD-GitHub-Actions's uncrustify.cfg file.

Line Endings   File checked into the FreeRTOS-Kernel repository use unix-style LF line endings for the best compatibility with git.

For optimal compatibility with Microsoft Windows tools, it is best to enable the git autocrlf feature. You can enable this setting for the current repository using the following command:

```
git config core.autocrlf true
```

Git History Optimizations   Some commits in this repository perform large refactors which touch many lines and lead to unwanted behavior when using the git blame command. You can configure git to ignore the list of large refactor commits in this repository with the following command:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

Spelling and Formatting   We recommend using Visual Studio Code, commonly referred to as VSCode, when working on the FreeRTOS-Kernel. The FreeRTOS-Kernel also uses cSpell as part of its spelling check. The config file for which can be found at *cspell.config.yaml* There is additionally a cSpell plugin for VSCode that can be used as well. .cSpellWords.txt contains words that are not traditionally found in an English dictionary. It is used by the spellchecker to verify the various jargon, variable names, and other odd words used in the FreeRTOS code base are correct. If your pull request fails to pass the spelling and you believe this is a mistake, then add the word to .cSpellWords.txt. When adding a word please then sort the list, which can be done by running the bash command: `sort -u .cSpellWords.txt -o .cSpellWords.txt` Note that only the FreeRTOS-Kernel Source Files, *include*, *portable/MemMang*, and *portable/Common* files are checked for proper spelling, and formatting at this time.

4.1.2   FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3   backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

Readme

MCUXpresso SDK: backoffAlgorithm Library   This repository is a fork of backoffAlgorithm library (https://github.com/FreeRTOS/backoffalgorithm)(1.3.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable backoffAlgorithm repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

backoffAlgorithm Library   This repository contains the backoffAlgorithm library, a utility library to calculate backoff period using an exponential backoff with jitter algorithm for retrying network operations (like failed network connection with server). This library uses the "Full Jitter" strategy for the exponential backoff with jitter algorithm. More information about the algorithm can be seen in the Exponential Backoff and Jitter AWS blog.

The backoffAlgorithm library is distributed under the *MIT Open Source License*.

Exponential backoff with jitter is typically used when retrying a failed network connection or operation request with the server. An exponential backoff with jitter helps to mitigate failed network operations with servers, that are caused due to network congestion or high request load on the server, by spreading out retry requests across multiple devices attempting network operations. Besides, in an environment with poor connectivity, a client can get disconnected at any time. A backoff strategy helps the client to conserve battery by not repeatedly attempting reconnections when they are unlikely to succeed.

See memory requirements for this library *here*.

backoffAlgorithm v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

backoffAlgorithm v1.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example The example below shows how to use the backoffAlgorithm library on a POSIX platform to retry a DNS resolution query for amazon.com.

```c
#include "backoff_algorithm.h"
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>

/* The maximum number of retries for the example code. */
#define RETRY_MAX_ATTEMPTS            ( 5U )

/* The maximum back-off delay (in milliseconds) for between retries in the example. */
#define RETRY_MAX_BACKOFF_DELAY_MS    ( 5000U )

/* The base back-off delay (in milliseconds) for retry configuration in the example. */
#define RETRY_BACKOFF_BASE_MS        ( 500U )

int main()
{
    /* Variables used in this example. */
    BackoffAlgorithmStatus_t retryStatus = BackoffAlgorithmSuccess;
    BackoffAlgorithmContext_t retryParams;
    char serverAddress[] = "amazon.com";
    uint16_t nextRetryBackoff = 0;

    int32_t dnsStatus = -1;
    struct addrinfo hints;
    struct addrinfo ** pListHead = NULL;
    struct timespec tp;

    /* Add hints to retrieve only TCP sockets in getaddrinfo. */
    ( void ) memset( &hints, 0, sizeof( hints ) );

    /* Address family of either IPv4 or IPv6. */
    hints.ai_family = AF_UNSPEC;
    /* TCP Socket. */
    hints.ai_socktype = ( int32_t ) SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    /* Initialize reconnect attempts and interval. */
    BackoffAlgorithm_InitializeParams( &retryParams,
                            RETRY_BACKOFF_BASE_MS,
                            RETRY_MAX_BACKOFF_DELAY_MS,
                            RETRY_MAX_ATTEMPTS );


    /* Seed the pseudo random number generator used in this example (with call to
     * rand() function provided by ISO C standard library) for use in backoff period
     * calculation when retrying failed DNS resolution. */

    /* Get current time to seed pseudo random number generator. */
    ( void ) clock_gettime( CLOCK_REALTIME, &tp );
    /* Seed pseudo random number generator with seconds. */
    srand( tp.tv_sec );

    do
    {
        /* Perform a DNS lookup on the given host name. */
        dnsStatus = getaddrinfo( serverAddress, NULL, &hints, pListHead );
```

(continues on next page)

```
        /* Retry if DNS resolution query failed. */
        if( dnsStatus != 0 )
        {
            /* Generate a random number and get back-off value (in milliseconds) for the next retry.
             * Note: It is recommended to use a random number generator that is seeded with
             * device-specific entropy source so that backoff calculation across devices is different
             * and possibility of network collision between devices attempting retries can be avoided.
             *
             * For the simplicity of this code example, the pseudo random number generator, rand()
             * function is used. */
            retryStatus = BackoffAlgorithm_GetNextBackoff( &retryParams, rand(), &nextRetryBackoff );

            /* Wait for the calculated backoff period before the next retry attempt of querying DNS.
             * As usleep() takes nanoseconds as the parameter, we multiply the backoff period by 1000. */
            ( void ) usleep( nextRetryBackoff * 1000U );
        }
    } while( ( dnsStatus != 0 ) && ( retryStatus != BackoffAlgorithmRetriesExhausted ) );

    return dnsStatus;
}
```

Building the library   A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses a header file introduced in ISO C99, stdint.h. For compilers that do not provide this header file, the *source/include* directory contains *stdint.readme*, which can be renamed to stdint.h to build the backoffAlgorithm library.

For instance, if the example above is copied to a file named example.c, *gcc* can be used like so:

```
gcc -I source/include example.c source/backoff_algorithm.c -o example
./example
```

gcc can also produce an output file to be linked:

```
gcc -I source/include -c source/backoff_algorithm.c
```

Building unit tests

Checkout Unity Submodule   By default, the submodules in this repository are configured with update=none in *.gitmodules*, to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

Platform Prerequisites

- For running unit tests
    - C89 or later compiler like gcc
    - CMake 3.13.0 or later
- For running the coverage target, gcov is additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described *above*.)

2. Create build directory: mkdir build && cd build

3. Run *cmake* while inside build directory: cmake -S ../test

4. Run this command to build the library and unit tests: make all

5. The generated test executables will be present in build/bin/tests folder.

6. Run ctest to execute all tests and view the test run summary.

Contributing　See *CONTRIBUTING.md* for information on contributing.

4.1.4　corehttp

C language HTTP client library designed for embedded platforms.

MCUXpresso SDK: coreHTTP Client Library

This repository is a fork of coreHTTP Client library (https://github.com/FreeRTOS/corehttp)(3.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreHTTP Client repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreHTTP Client Library

This repository contains a C language HTTP client library designed for embedded platforms. It has no dependencies on any additional libraries other than the standard C library, llhttp, and a customer-implemented transport interface. This library is distributed under the *MIT Open Source License.*

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety and data structure invariance through the CBMC automated reasoning tool.

See memory requirements for this library *here.*

coreHTTP v3.0.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreHTTP v2.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

coreHTTP Config File　The HTTP client library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_http_config_defaults.h.* To provide custom values for the configuration macros, a custom config file named core_http_config.h can be provided by the user application to the library.

By default, a core_http_config.h custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide HTTP_DO_NOT_USE_CUSTOM_CONFIG as a compile time preprocessor macro.

The HTTP client library can be built by either:

- Defining a core_http_config.h file in the application, and adding it to the include directories for the library build. **OR**

- Defining the HTTP_DO_NOT_USE_CUSTOM_CONFIG preprocessor macro for the library build.

Building the Library The *httpFilePaths.cmake* file contains the information of all source files and header include paths required to build the HTTP client library.

As mentioned in the *previous section,* either a custom config file (i.e. core_http_config.h) OR HTTP_DO_NOT_USE_CUSTOM_CONFIG macro needs to be provided to build the HTTP client library.

For a CMake example of building the HTTP library with the httpFilePaths.cmake file, refer to the coverity_analysis library target in *test/CMakeLists.txt* file.

Building Unit Tests

Platform Prerequisites

- For running unit tests, the following are required:
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is required for this repository's CMock test framework.
- For running the coverage target, the following are required:
  - **gcov**
  - **lcov**

Steps to build Unit Tests

1. Go to the root directory of this repository.
2. Run the *cmake* command: cmake -S test -B build -DBUILD_CLONE_SUBMODULES=ON
3. Run this command to build the library and unit tests: make -C build all
4. The generated test executables will be present in build/bin/tests folder.
5. Run cd build && ctest to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The test/cbmc/proofs directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples The AWS IoT Device SDK for Embedded C repository contains demos of using the HTTP client library here on a POSIX platform. These can be used as reference examples for the library API.

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of coreHTTP may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

4.1.5 corejson

JSON parser.

Readme

MCUXpresso SDK: coreJSON Library This repository is a fork of coreJSON library (https://github.com/FreeRTOS/corejson)(3.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreJSON repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreJSON Library This repository contains the coreJSON library, a parser that strictly enforces the ECMA-404 JSON standard and is suitable for low memory footprint embedded devices. The coreJSON library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

coreJSON v3.2.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreJSON v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example

```
#include <stdio.h>
#include "core_json.h"

int main()
{
    // Variables used in this example.
    JSONStatus_t result;
    char buffer[] = "{\"foo\":\"abc\",\"bar\":{\"foo\":\"xyz\"}}";
    size_t bufferLength = sizeof( buffer ) - 1;
    char queryKey[] = "bar.foo";
    size_t queryKeyLength = sizeof( queryKey ) - 1;
    char * value;
    size_t valueLength;

    // Calling JSON_Validate() is not necessary if the document is guaranteed to be valid.
    result = JSON_Validate( buffer, bufferLength );

    if( result == JSONSuccess )
    {
        result = JSON_Search( buffer, bufferLength, queryKey, queryKeyLength,
                        &value, &valueLength );
    }

    if( result == JSONSuccess )
    {
        // The pointer "value" will point to a location in the "buffer".
        char save = value[ valueLength ];
        // After saving the character, set it to a null byte for printing.
        value[ valueLength ] = '\0';
        // "Found: bar.foo -> xyz" will be printed.
        printf( "Found: %s -> %s\n", queryKey, value );
        // Restore the original character.
        value[ valueLength ] = save;
    }

    return 0;
}
```

A search may descend through nested objects when the queryKey contains matching key strings joined by a separator, .. In the example above, bar has the value {"foo":"xyz"}. Therefore, a search for query key bar.foo would output xyz.

Building coreJSON A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses 2 header files introduced in ISO C99, stdbool.h and stdint.h. For compilers that do not provide this header file, the *source/include* directory contains *stdbool.readme* and *stdint.readme*, which can be renamed to stdbool.h and stdint.h respectively.

For instance, if the example above is copied to a file named example.c, *gcc* can be used like so:

```
gcc -I source/include example.c source/core_json.c -o example
./example
```

gcc can also produce an output file to be linked:

```
gcc -I source/include -c source/core_json.c
```

Documentation

Existing documentation   For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of the coreJSON library may differ across repositories.

Generating documentation   The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Building unit tests

Checkout Unity Submodule   By default, the submodules in this repository are configured with update=none in *.gitmodules*, to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

Platform Prerequisites

- For running unit tests
  - C90 compiler like gcc
  - CMake 3.13.0 or later
  - Ruby 2.0.0 or later is additionally required for the Unity test framework (that we use).
- For running the coverage target, gcov is additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described *above*.)
2. Create build directory: mkdir build && cd build
3. Run *cmake* while inside build directory: cmake -S ../test
4. Run this command to build the library and unit tests: make all
5. The generated test executables will be present in build/bin/tests folder.
6. Run ctest to execute all tests and view the test run summary.

CBMC   To learn more about CBMC and proofs specifically, review the training material here.

The test/cbmc/proofs directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Contributing   See *CONTRIBUTING.md* for information on contributing.

4.1.6   coremqtt

MQTT publish/subscribe messaging library.

MCUXpresso SDK: coreMQTT Library

This repository is a fork of coreMQTT library (https://github.com/FreeRTOS/coremqtt)(2.1.1). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreMQTT Client Library

This repository contains the coreMQTT library that has been optimized for a low memory footprint. The coreMQTT library is compliant with the MQTT 3.1.1 standard. It has no dependencies on any additional libraries other than the standard C library, a customer-implemented network transport interface, and *optionally* a user-implemented platform time function. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

coreMQTT v2.1.1 source code is part of the FreeRTOS 202210.01 LTS release.

MQTT Config File   The MQTT client library exposes build configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_mqtt_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named core_mqtt_config.h can be provided by the application to the library.

By default, a core_mqtt_config.h custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide MQTT_DO_NOT_USE_CUSTOM_CONFIG as a compile time preprocessor macro.

Thus, the MQTT library can be built by either:

- Defining a core_mqtt_config.h file in the application, and adding it to the include directories list of the library
  OR

- Defining the MQTT_DO_NOT_USE_CUSTOM_CONFIG preprocessor macro for the library build.

Sending metrics to AWS IoT When establishing a connection with AWS IoT, users can optionally report the Operating System, Hardware Platform and MQTT client version information of their device to AWS. This information can help AWS IoT provide faster issue resolution and technical support. If users want to report this information, they can send a specially formatted string (see below) in the username field of the MQTT CONNECT packet.

Format

The format of the username string with metrics is:

```
<Actual_Username>?SDK=<OS_Name>&Version=<OS_Version>&Platform=<Hardware_Platform>&
↪MQTTLib=<MQTT_Library_name>@<MQTT_Library_version>
```

Where

- <Actual_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.

- <OS_Name> is the Operating System the application is running on (e.g. FreeRTOS)

- <OS_Version> is the version number of the Operating System (e.g. V10.4.3)

- <Hardware_Platform> is the Hardware Platform the application is running on (e.g. WinSim)

- <MQTT_Library_name> is the MQTT Client library being used (e.g. coreMQTT)

- <MQTT_Library_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual_Username = "iotuser", OS_Name = FreeRTOS, OS_Version = V10.4.3, Hardware_Platform_Name = WinSim, MQTT_Library_Name = coremqtt, MQTT_Library_version = 2.1.1. If username is not used, then "iotuser" can be removed.

```
/* Username string:
 * iotuser?SDK=FreeRTOS&Version=v10.4.3&Platform=WinSim&MQTTLib=coremqtt@2.1.1
 */

#define OS_NAME                    "FreeRTOS"
#define OS_VERSION                 "V10.4.3"
#define HARDWARE_PLATFORM_NAME     "WinSim"
#define MQTT_LIB                   "coremqtt@2.1.1"

#define USERNAME_STRING            "iotuser?SDK=" OS_NAME "&Version=" OS_VERSION "&
↪Platform=" HARDWARE_PLATFORM_NAME "&MQTTLib=" MQTT_LIB
#define USERNAME_STRING_LENGTH     ( ( uint16_t ) ( sizeof( USERNAME_STRING ) - 1 ) )

MQTTConnectInfo_t connectInfo;
connectInfo.pUserName = USERNAME_STRING;
connectInfo.userNameLength = USERNAME_STRING_LENGTH;
mqttStatus = MQTT_Connect( pMqttContext, &connectInfo, NULL, CONNACK_RECV_TIMEOUT_MS,
↪pSessionPresent );
```

Upgrading to v2.0.0 and above With coreMQTT versions >=v2.0.0, there are breaking changes. Please refer to the *coreMQTT version >=v2.0.0 Migration Guide*.

Building the Library The *mqttFilePaths.cmake* file contains the information of all source files and the header include path required to build the MQTT library.

Additionally, the MQTT library requires two header files that are not part of the ISO C90 standard library, stdbool.h and stdint.h. For compilers that do not provide these header files, the

source/include directory contains the files *stdbool.readme* and *stdint.readme*, which can be renamed to stdbool.h and stdint.h, respectively, to provide the type definitions required by MQTT.

As mentioned in the previous section, either a custom config file (i.e. core_mqtt_config.h) OR MQTT_DO_NOT_USE_CUSTOM_CONFIG macro needs to be provided to build the MQTT library.

For a CMake example of building the MQTT library with the mqttFilePaths.cmake file, refer to the coverity_analysis library target in *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule By default, the submodules in this repository are configured with update=none in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

Platform Prerequisites

- Docker

or the following:

- For running unit tests
    - **C90 compiler** like gcc
    - **CMake 3.13.0 or later**
    - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build Unit Tests

1. If using docker, launch the container:
    1. docker build -t coremqtt .
    2. docker run -it -v "$PWD":/workspaces/coreMQTT -w /workspaces/coreMQTT coremqtt
2. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*)
3. Run the *cmake* command: cmake -S test -B build
4. Run this command to build the library and unit tests: make -C build all
5. The generated test executables will be present in build/bin/tests folder.
6. Run cd build && ctest to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The test/cbmc/proofs directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples Please refer to the demos of the MQTT client library in the following locations for reference examples on POSIX and FreeRTOS platforms:

Platform	Location	Transport Interface Implementation
POSIX	AWS IoT Device SDK for Embedded C	POSIX sockets for TCP/IP and OpenSSL for TLS stack
FreeRTOS	FreeRTOS/FreeRTOS	FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack
FreeRTOS	FreeRTOS AWS Reference Integrations	Based on Secure Sockets Abstraction

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of coreMQTT may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

4.1.7 coremqtt-agent

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

Readme

MCUXpresso SDK: coreMQTT Agent Library This repository is a fork of coreMQTT Agent library (https://github.com/FreeRTOS/coremqtt-agent)(1.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT Agent repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreMQTT Agent Library   The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library. The library provides thread safe equivalents to the coreMQTT's APIs, greatly simplifying its use in multi-threaded environments. The coreMQTT Agent library manages the MQTT connection by serializing the access to the coreMQTT library and reducing implementation overhead (e.g., removing the need for the application to repeatedly call to MQTT_ProcessLoop). This allows your multi-threaded applications to share the same MQTT connection, and enables you to design an embedded application without having to worry about coreMQTT thread safety.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

Cloning this repository   This repo uses Git Submodules to bring in dependent components.

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

Using SSH:

```
git clone git@github.com:FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

If you have downloaded the repo without using the --recurse-submodules argument, you need to run:

```
git submodule update --init --recursive
```

coreMQTT Agent Library Configurations   The MQTT Agent library uses the same core_mqtt_config.h configuration file as coreMQTT, with the addition of configuration constants listed at the top of *core_mqtt_agent.h* and *core_mqtt_agent_command_functions.h*. Documentation for these configurations can be found here.

To provide values for these configuration values, they must be either:

- Defined in core_mqtt_config.h used by coreMQTT **OR**
- Passed as compile time preprocessor macros

Porting the coreMQTT Agent Library   In order to use the MQTT Agent library on a platform, you need to supply thread safe functions for the agent's *messaging interface*.

Messaging Interface   Each of the following functions must be thread safe.

Function Pointer	Description
MQTTA-gentMes-sage-Send_t	A function that sends commands (as MQTTAgentCommand_t * pointers) to be received by MQTTAgent_CommandLoop. This can be implemented by pushing to a thread safe queue.
MQTTA-gentMes-sageRecv_t	A function used by MQTTAgent_CommandLoop to receive MQTTAgentCommand_t * pointers that were sent by API functions. This can be implemented by receiving from a thread safe queue.
MQTTA-gentCom-mand-Get_t	A function that returns a pointer to an allocated MQTTAgentCommand_t structure, which is used to hold information and arguments for a command to be executed in MQTTAgent_CommandLoop(). If using dynamic memory, this can be implemented using malloc().
MQT-TAgent-Comman-dRelease_t	A function called to indicate that a command structure that had been allocated with the MQTTAgentCommandGet_t function pointer will no longer be used by the agent, so it may be freed or marked as not in use. If using dynamic memory, this can be implemented with free().

Reference implementations for the interface functions can be found in the *reference examples* below.

Additional Considerations

Static Memory   If only static allocation is used, then the MQTTAgentCommandGet_t and MQT-TAgentCommandRelease_t could instead be implemented with a pool of MQTTAgentCommand_t structures, with a queue or semaphore used to control access and provide thread safety. The below *reference examples* use static memory with a command pool.

Subscription Management   The MQTT Agent does not track subscriptions for MQTT topics. The receipt of any incoming PUBLISH packet will result in the invocation of a single MQTTA-gentIncomingPublishCallback_t callback, which is passed to MQTTAgent_Init() for initialization. If it is desired for different handlers to be invoked for different incoming topics, then the publish callback will have to manage subscriptions and fan out messages. A platform independent subscription manager example is implemented in the *reference examples* below.

Building the Library   You can build the MQTT Agent source files that are in the *source* directory, and add *source/include* to your compiler's include path. Additionally, the MQTT Agent library requires the coreMQTT library, whose files follow the same source/ and source/include pattern as the agent library; its build instructions can be found here.

If using CMake, the *mqttAgentFilePaths.cmake* file contains the above information of the source files and the header include path from this repository. The same information is found for coreMQTT from mqttFilePaths.cmake in the *coreMQTT submodule*.

For a CMake example of building the MQTT Agent library with the mqttAgentFilePaths.cmake file, refer to the coverity_analysis library target in *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule   To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

Unit Test Platform Prerequisites

- For running unit tests
    - **C90 compiler** like gcc
    - **CMake 3.13.0 or later**
    - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*)
2. Run the *cmake* command: `cmake -S test -B build`
3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC   To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples   Please refer to the demos of the MQTT Agent library in the following locations for reference examples on FreeRTOS platforms:

Location
coreMQTT Agent Demos
FreeRTOS/FreeRTOS

Documentation   The MQTT Agent API documentation can be found here.

Generating documentation   The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages yourself, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Getting help   You can use your Github login to get support from both the FreeRTOS community and directly from the primary FreeRTOS developers on our active support forum. You can find a list of frequently asked questions here.

Contributing   See *CONTRIBUTING.md* for information on contributing.

License   This library is licensed under the MIT License. See the *LICENSE* file.

4.1.8   corepkcs11

PKCS #11 key management library.

Readme

MCUXpresso SDK: corePKCS11 Library   This repository is a fork of PKCS #11 key management library (https://github.com/FreeRTOS/corePKCS11/tree/v3.5.0)(v3.5.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable corepkcs11 repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

corePKCS11 Library   PKCS #11 is a standardized and widely used API for manipulating common cryptographic objects. It is important because the functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to, among other things, access the secret (private) key necessary to create a network connection that is authenticated and secured by the Transport Layer Security (TLS) protocol – without the application ever 'seeing' the key.

The Cryptoki or PKCS #11 standard defines a platform-independent API to manage and use cryptographic tokens. The name, "PKCS #11", is used interchangeably to refer to the API itself and the standard which defines it.

This repository contains a software based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. Using a software mock enables rapid development and flexibility, but it is expected that the mock be replaced by an implementation specific to your chosen secure key storage in production devices.

Only a subset of the PKCS #11 standard is implemented, with a focus on operations involving asymmetric keys, random number generation, and hashing.

The targeted use cases include certificate and key management for TLS authentication and code-sign signature verification, on small embedded devices.

corePKCS11 is implemented on PKCS #11 v2.4.0, the full PKCS #11 standard can be found on the oasis website.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

corePKCS11 v3.5.0 source code is part of the FreeRTOS 202210.00 LTS release.

corePKCS11 v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Purpose   Generally vendors for secure cryptoprocessors such as Trusted Platform Module (TPM), Hardware Security Module (HSM), Secure Element, or any other type of secure hardware enclave, distribute a PKCS #11 implementation with the hardware. The purpose of the corePKCS11 software only mock library is therefore to provide a non hardware specific PKCS #11 implementation that allows for rapid prototyping and development before switching to a cryptoprocessor specific PKCS #11 implementation in production devices.

Since the PKCS #11 interface is defined as part of the PKCS #11 specification replacing this library with another implementation should require little porting effort, as the interface will not change. The system tests distributed in this repository can be leveraged to verify the behavior of a different implementation is similar to corePKCS11.

corePKCS11 Configuration   The corePKCS11 library exposes preprocessor macros which must be defined prior to building the library. A list of all the configurations and their default values are defined in the doxygen documentation for this library.

Build Prerequisites

Library Usage   For building the library the following are required:

- **A C99 compiler**

- **mbedcrypto** library from mbedtls version 2.x or 3.x.

- **pkcs11 API header(s)** available from OASIS or OpenSC

Optionally, variables from the pkcsFilePaths.cmake file may be referenced if your project uses cmake.

Integration and Unit Tests   In order to run the integration and unit test suites the following are dependencies are necessary:

- **C Compiler**

- **CMake 3.13.0 or later**

- **Ruby 2.0.0 or later** required by CMock.

- **Python 3** required for configuring mbedtls.

- **git** required for fetching dependencies.

- **GNU Make** or **Ninja**

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

Coverage Measurement and Instrumentation   The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.

- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.

- **gcov** binary corresponding to your chosen compiler

- **lcov** from the Linux Test Project

- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.

2. Run **cmake** to construct a build tree: `cmake -S test -B build`

   - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.

   - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.

3. Build the test binaries: `cmake --build ./build --target all`

4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.

5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

CBMC   To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples   The FreeRTOS-Labs repository contains demos using the PKCS #11 library here using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

Porting Guide   Documentation for porting corePKCS11 to a new platform can be found on the AWS docs web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

Related Example Implementations   These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's Platform Security Architecture.

- Microchip's cryptoauthlib.

- Infineon's Optiga Trust X.

Documentation

Existing Documentation   For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of corePKCS11 may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Security See *CONTRIBUTING* for more information.

License This library is licensed under the MIT-0 License. See the LICENSE file.

4.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme

MCUXpresso SDK: FreeRTOS-Plus-TCP Library This repository is a fork of FreeRTOS-Plus-TCP library (https://github.com/FreeRTOS/freertos-plus-tcp)(4.3.3). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

FreeRTOS-Plus-TCP Library FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the MISRA coding standard. Any deviations from the MISRA C:2012 guidelines are documented under MISRA Deviations. The library is validated for memory safety and data structure invariance through the CBMC automated reasoning tool for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

FreeRTOS-Plus-TCP Library V4.2.2 source code is part of the FreeRTOS 202406.01 LTS release.

Getting started The easiest way to use version 4.0.0 and later of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found here) Another way is to start with the pre-configured IPv4 Windows Simulator demo (found in this directory) or IPv6 Multi-endpoint Windows Simulator demo (found in this directory). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the FreeRTOS Kernel Quick Start Guide for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the Documentation, and API Reference.

Getting help　If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the FreeRTOS Community Support Forum. Please also refer to FAQ for frequently asked questions.

Also see the Submitting a bugs/feature request section of CONTRIBUTING.md for more details.

Note: All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

Upgrading to V4.3.0 and above　For users of STM32 network interfaces:

Starting from version V4.3.0, the STM32 network interfaces have been consolidated into a single unified implementation located at source/portable/NetworkInterface/STM32/NetworkInterface.c, supporting STM32 F4, F7, and H7 series microcontrollers, with newly added support for STM32 H5. The new interface has been tested with the STM32 HAL Ethernet (ETH) drivers, available at source/portable/NetworkInterface/STM32/Drivers. For compatibility, the legacy interfaces (STM32Fxx and STM32Hxx) have been retained and relocated to source/portable/NetworkInterface/STM32/Legacy.

Upgrading to V3.0.0 and V3.1.0　In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a source directory. This change requires modification of any existing project(s) to include the modified source files and directories.

FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 202210.00 LTS release.

Generating pre V3.0.0 folder structure for backward compatibility:　If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to this.

Note: After running the script, while the .c files will have same names as the pre V3.0.0 source, the files in the include directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from here.

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing python --version.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory Then run python <Path/to/the/script>/GenerateOriginalFiles.py.

To consume FreeRTOS+TCP

Consume with CMake　If using CMake, it is recommended to use this repository using Fetch-Content. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_plus_tcp
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
  GIT_TAG        main #Note: Best practice to use specific git-hash or tagged version
```

```
  GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

- Configure the FreeRTOS-Kernel and make it available
    - this particular example supports a native and cross-compiled build option.

```
# Select the native compile PORT
set( FREERTOS_PLUS_TCP_NETWORK_IF "POSIX" CACHE STRING "" FORCE)
# Or: select a cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  # Eg. STM32Hxx version of port
  set(FREERTOS_PLUS_TCP_NETWORK_IF "STM32HXX" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)
```

Consuming stand-alone This repository uses Git Submodules to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel
```

Porting The porting guide is available on this page.

Repository structure This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
    - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
    - This directory contains all the tests (unit tests and CBMC) and the dependencies (FreeRTOS-Kernel/Litani-port) the tests require.
- source/portable
    - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
    - The include directory has all the 'core' header files of FreeRTOS-Plus-TCP source.
- source

– This directory contains all the [.c] source files.

Note   At this time it is recommended to use BufferAllocation_2.c in which case it is essential to use the heap_4.c memory allocation scheme. See memory management.

Kernel sources   The FreeRTOS Kernel Source is in FreeRTOS/FreeRTOS-Kernel repository, and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at ./test/FreeRTOS-Kernel directory.

CBMC   The test/cbmc/proofs directory contains CBMC proofs.

To learn more about CBMC and proofs specifically, review the training material here.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.