



MCUXpresso SDK Documentation

Release 26.03.00



NXP
Mar 24, 2026



Table of contents

1	MC56F83000-EVK	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	13
1.3.1	Getting Started with MCUXpresso SDK Repository	13
1.4	Release Notes	37
1.4.1	MCUXpresso SDK Release Notes	37
1.5	ChangeLog	40
1.5.1	MCUXpresso SDK Changelog	40
2	MC56F83789	55
2.1	CADC: 12-bit Cyclic Analog-to-Digital Converter Driver	55
2.2	The Driver Change Log	90
2.3	CADC Peripheral and Driver Overview	90
2.4	Clock Driver	90
2.5	Driver Change Log	103
2.6	CMP: Comparator Driver	103
2.7	The Driver Change Log	112
2.8	CMP Peripheral and Driver Overview	112
2.9	COP: Computer Operating Properly(Watchdog) Driver	112
2.10	The Driver Change Log	116
2.11	COP Peripheral and Driver Overview	116
2.12	CRC: Cyclic Redundancy Check Driver	116
2.13	The Driver Change Log	120
2.14	CRC Peripheral and Driver Overview	120
2.15	DAC: 12-bit Digital-to-Analog Converter Driver	120
2.16	The Driver Change Log	131
2.17	DAC Peripheral and Driver Overview	131
2.18	DMAMUX: DMA Channel Multiplexer Driver	131
2.19	The Driver Change Log	132
2.20	DMAMUX Peripheral and Driver Overview	132
2.21	EDMA: Enhanced Direct Memory Access Driver	132
2.22	The Driver Change Log	152
2.23	EDMA Peripheral and Driver Overview	152
2.24	EVTG: Event Generator Driver	152
2.25	The Driver Change Log	159
2.26	EVTG Peripheral and Driver Overview	159
2.27	EWM: External Watchdog Monitor Driver	159
2.28	The Driver Change Log	161
2.29	EWM Peripheral and Driver Overview	161
2.30	Flash Driver	161
2.31	Driver Change Log	175
2.32	FlexCAN Driver	175
2.33	The Driver Change Log	202
2.34	FlexCAN Peripheral and Driver Overview	202

2.35	GPIO: General-Purpose Input/Output Driver	202
2.36	The Driver Change Log	214
2.37	GPIO Peripheral and Driver Overview	214
2.38	I2C: Inter-Integrated Circuit Driver	214
2.39	The Driver Change Log	237
2.40	I2C_EDMA: EDMA based I2C Driver	237
2.41	I2C Peripheral and Driver Overview	238
2.42	INTC: Interrupt Controller Driver	238
2.43	The Driver Change Log	241
2.44	INTC Peripheral and Driver Overview	241
2.45	Common Driver	241
2.46	MCM: Miscellaneous Control Module Driver	257
2.47	The Driver Change Log	265
2.48	MCM Peripheral and Driver Overview	265
2.49	PIT: Periodic Interrupt Timer (PIT) Driver	265
2.50	The Driver Change Log	271
2.51	PIT Peripheral and Driver Overview	271
2.52	PMC: Power Management Controller Driver	271
2.53	The Driver Change Log	273
2.54	PMC Peripheral and Driver Overview	273
2.55	eFlexPWM: Enhanced Flexible Pulse Width Modulator Driver	273
2.56	The Driver Change Log	311
2.57	eFlexPWM Peripheral and Driver Overview	311
2.58	QSCI: Queued Serial Communications Interface Driver	311
2.59	The Driver Change Log	327
2.60	QSCI_EDMA: EDMA based QSCI Driver	327
2.61	QSCI Peripheral and Driver Overview	330
2.62	QSPI: Queued SPI Driver	330
2.63	QSPI Peripheral and Driver Overview	348
2.64	QSPI_EDMA: EDMA based QSPI Driver	348
2.65	QTMR: Quad Timer Driver	352
2.66	The Driver Change Log	375
2.67	QTMR Peripheral and Driver Overview	375
2.68	The Driver Change Log	375
2.69	SIM: System Integration Module Driver	375
2.70	The Driver Change Log	375
2.71	SIM Peripheral and Driver Overview	387
2.72	XBAR: Inter-Peripheral Crossbar Switch Driver	387
2.73	The Driver Change Log	391
2.74	XBAR Peripheral and Driver Overview	391
3	Middleware	393
4	RTOS	395
4.1	FreeRTOS	395
4.1.1	FreeRTOS kernel	395
4.1.2	FreeRTOS drivers	395
4.1.3	backoffalgorithm	395
4.1.4	corehttp	395
4.1.5	corejson	395
4.1.6	coremqtt	396
4.1.7	corepkcs11	396
4.1.8	freertos-plus-tcp	396

This documentation contains information specific to the mc56f83000evk board.

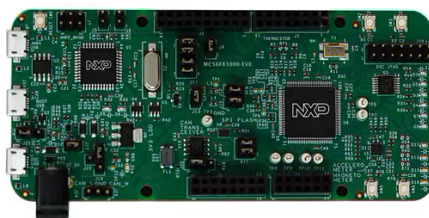
Chapter 1

MC56F83000-EVK

1.1 Overview

The MC56F83000-EVK is an ultra-low-cost development platform for digital signal controller MC56F83xxx MCU.

- The kit is form-factor compatible with the Arduino™ R3 pin layout and features ROM boot-loader supporting SCI, IIC and CAN.
- Peripherals enable rapid prototyping, including a 6-axis digital accelerometer and magnetometer to create full eCompass capabilities, 6 buffered LEDs indicating PWM signals, 4 user LEDs, 4 user push-buttons for direct interaction, an SPI interfaced Flash memory, a high speed CAN transceiver circuit, a USBOTG connector and a USB to UART bridge circuit.
- The MC56F83000-EVK features onboard debugger OSJTAG circuit enabling debugging and programming with CodeWarrior.



MCU device and part on board is shown below:

- Device: MC56F83789
- PartNumber: MC56F83789VLL

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

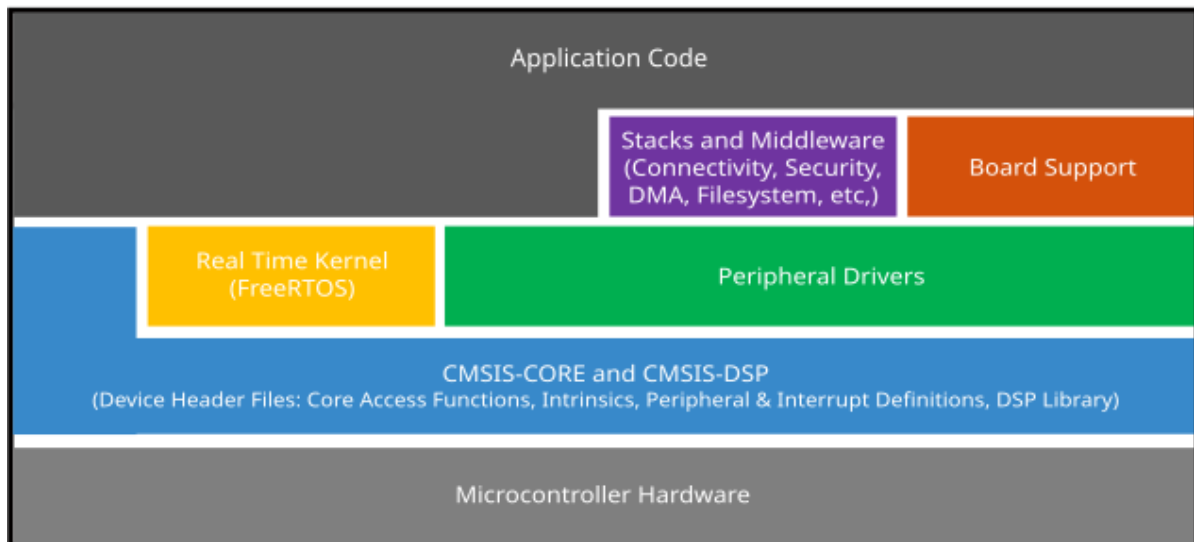
Overview

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 \(SDK\)](#).



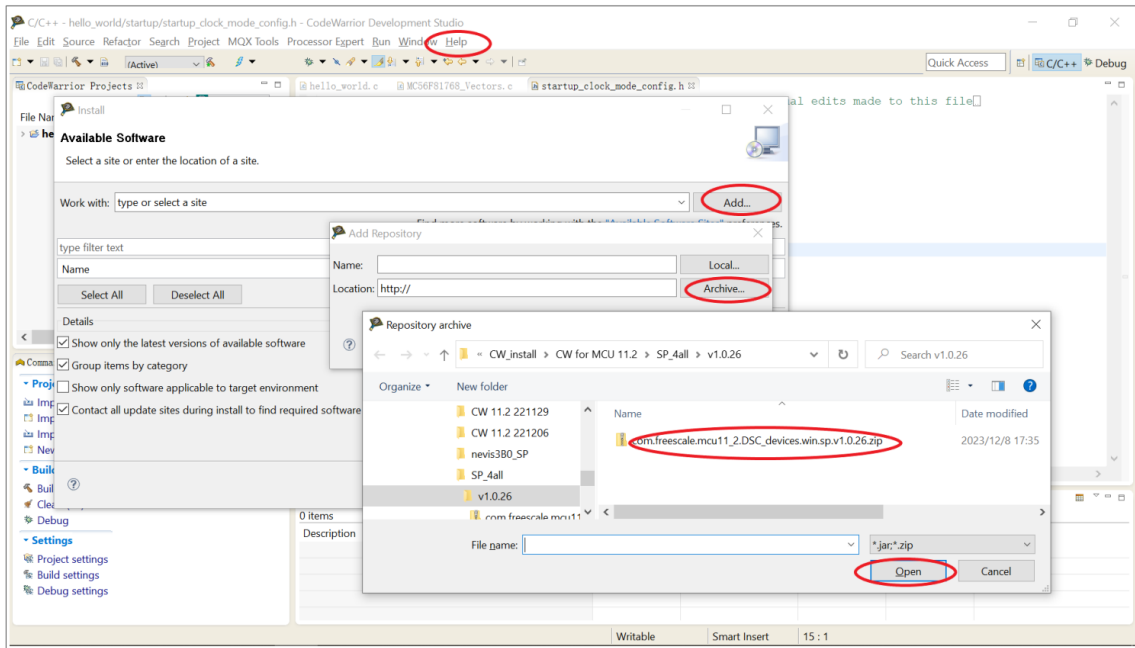
Build and run SDK example on codewarrior

Install CodeWarrior Take below codewarrior specific combination as example

- CodeWarrior Development Studio v11.2 + CodeWarrior for DSC v11.2 SP1 (Service Pack 1)

Steps to install CodeWarrior

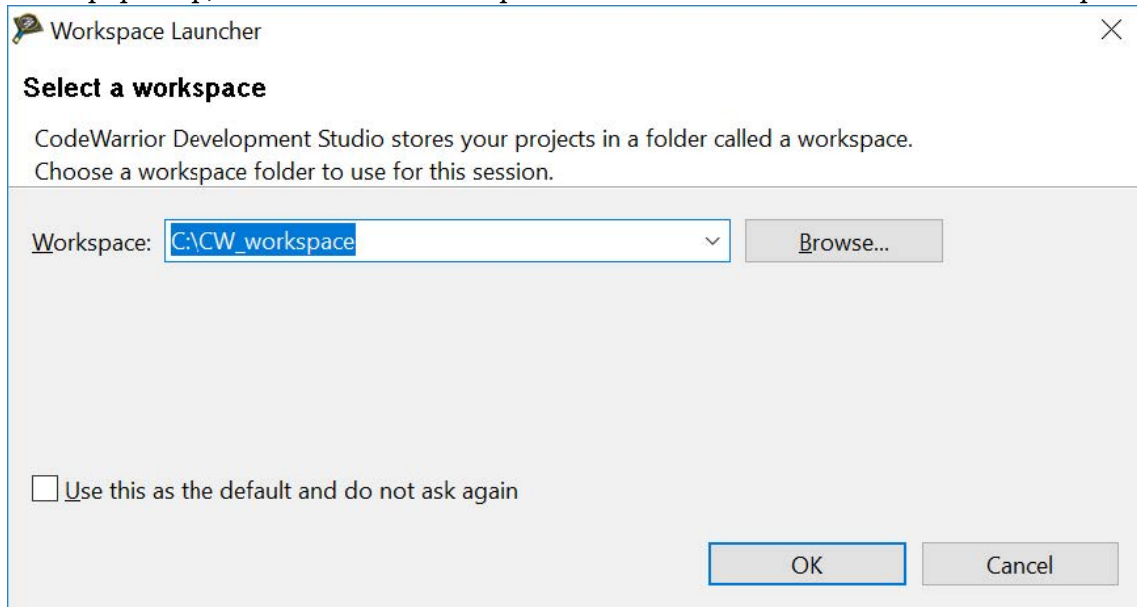
- Download the following packages from [CodeWarrior for 56800 Digital Signal Controller v11.2](#), and ensure to keep them in the same folder.
 - CodeWarrior for DSC v11.2: CW_MCU_v11.2_b221206.exe.
 - DSC support package: com.freescale.mcu11_2.dsc.update.site.zip.
 - DSC device ServicePack1: com.freescale.mcu11_2.DSC_devices.win.sp.v1.0.26.zip.
- Install CodeWarrior for DSC v11.2.
- Install ServicePack1 within CodeWarrior from the menu. Click the **Help** menu -> select **Install new software** -> **Add** -> **Archive** -> select the downloaded SP1 -> open -> check **MCU v11.2 DSC Service Packs** -> click **Next**.

**NOTE**

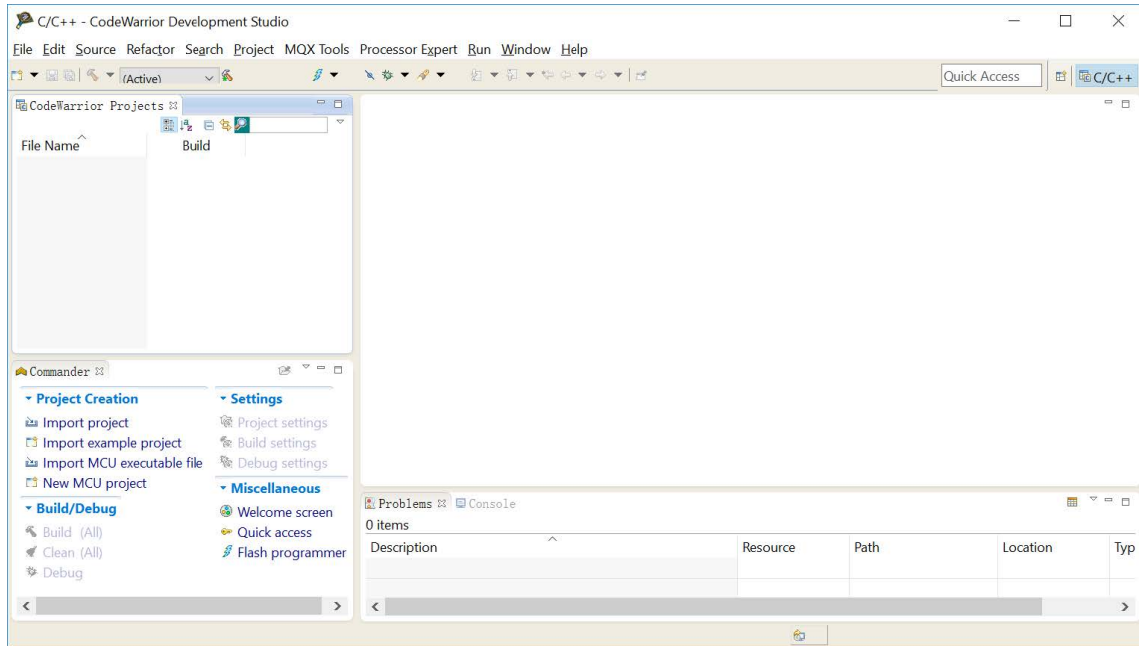
- CodeWarrior for DSC only support Windows.
- Check the corresponding board release note for specific requirement of codewarrior and service pack version.

Build an example application To build the hello_world example application, perform the following example steps:

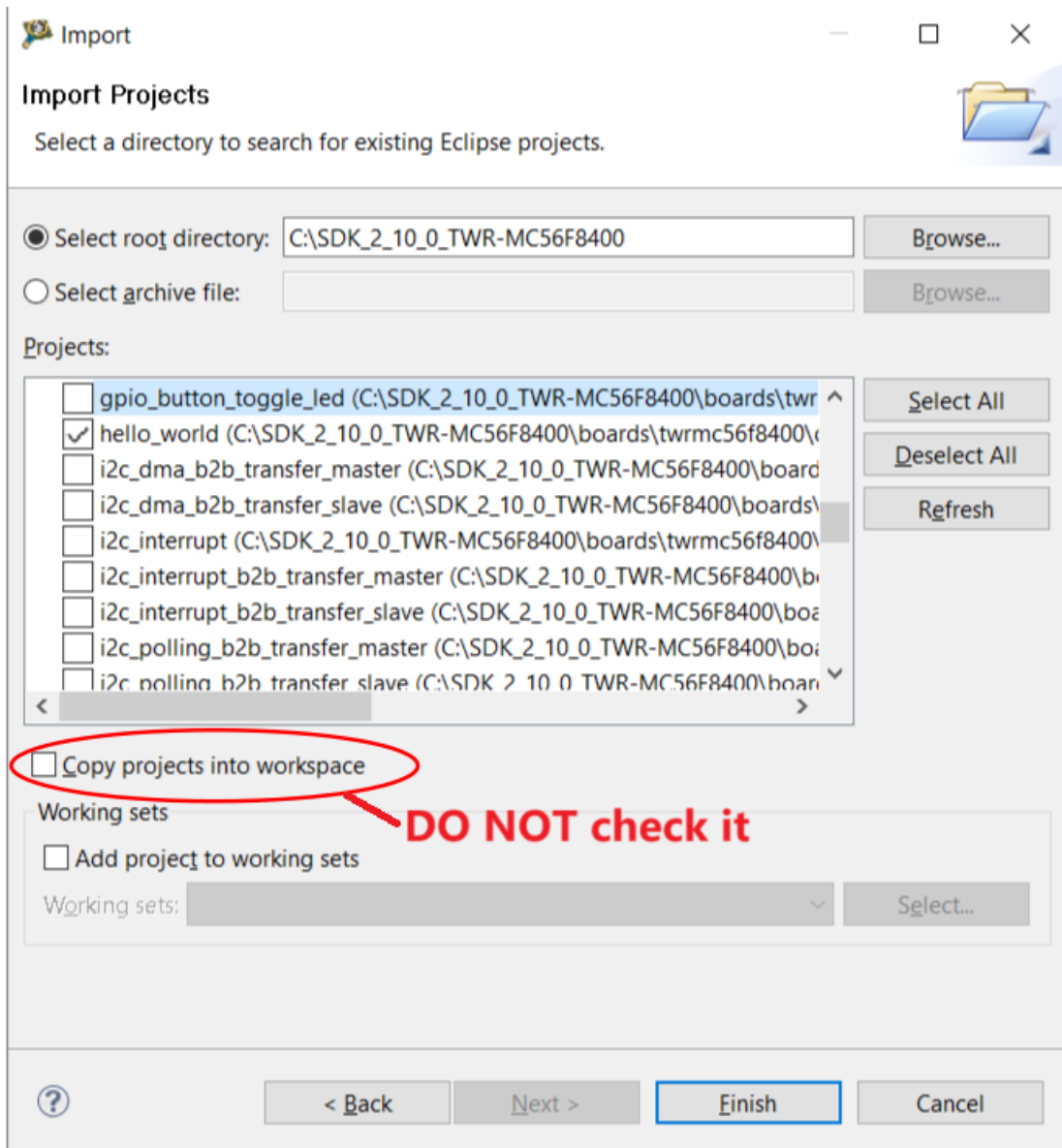
1. Launch CodeWarrior and in the workspace launcher, choose a workspace which holds the projects to use. If the dialogue box does not pop up, enter a workspace folder and create one workspace.



Then the CodeWarrior Development Studio workspace with empty projects appears.



2. Import the project into the workspace. Click **Import project** in the **Commander** pane. A form pops up. Click **Browse** to the SDK install directory. Take TWR-MC56F8400 SDK as an example, all available demo projects are shown. Select the `hello_world` project in the list and click **Finish**.

**NOTE**

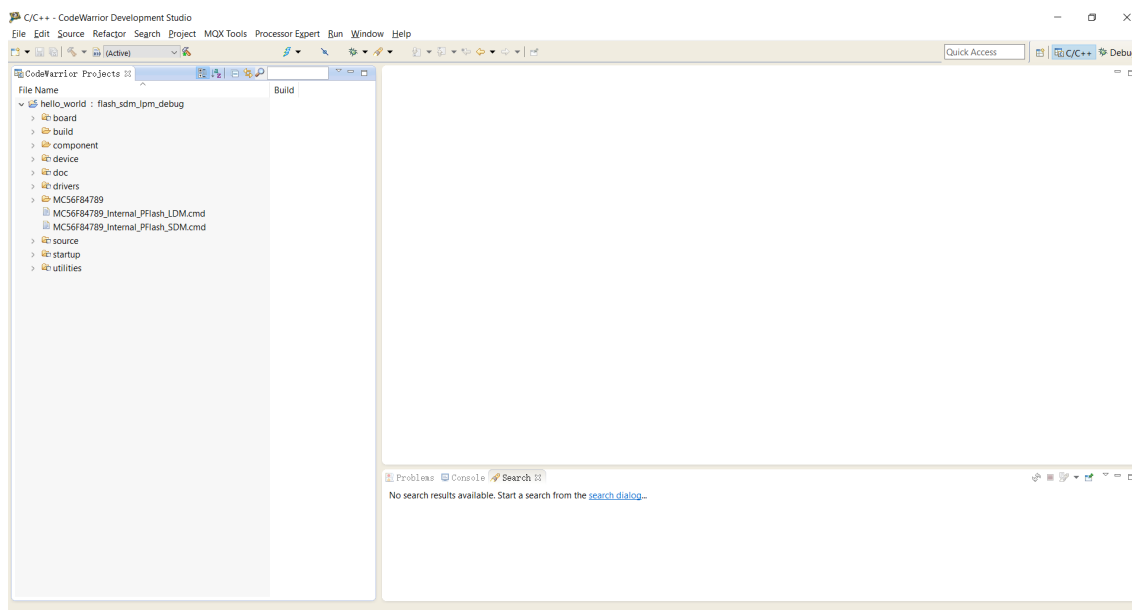
- If you already know the project location, navigate to the folder when clicking **Browse**, and only one project can be seen. To locate most example application workspace files, use the following path

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

Take TWR-MC56F8400 SDK as an example, the hello_world workspace is located in

```
<install_dir>/boards/twrmc56f8400/demo_apps/hello_world/codewarrior
```

3. Select the desired build target from the drop-down menu. For this example, select **hello_world** – **flash_sdm_lpm_debug**



4. To build the demo application, click **Build (All)** in the **Commander** pane.
5. The build completes without errors.

Board debugger setup Board debugger info:

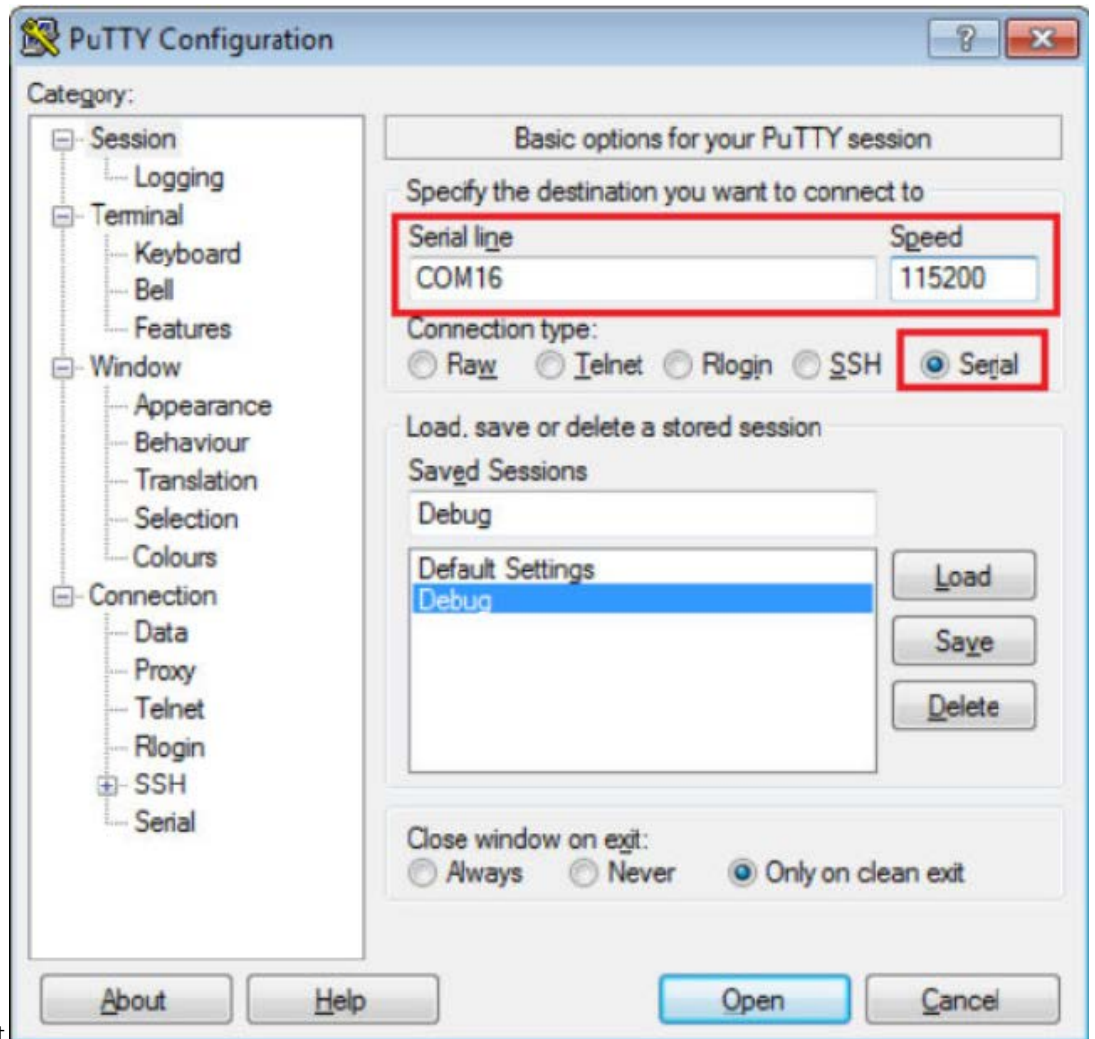
- Default debugger is OSJTAG.
- Onboard debugger USB port is J8, which set the debugger and COM port.
- Onboard debugger firmware update jumper is J6.

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

- Connect USB cable between the host PC and the debugger USB port.
- Install the debugger driver and USB CDC driver as PC hint if it is the first time you run it on the PC. The debugger and USB CDC driver are provided by CodeWarrior by default.
- The CodeWarrior may prompt to update the debugger firmware, which requires to connect the firmware update jumper on board and then follow the instruction by CodeWarrior to finish the firmware update.

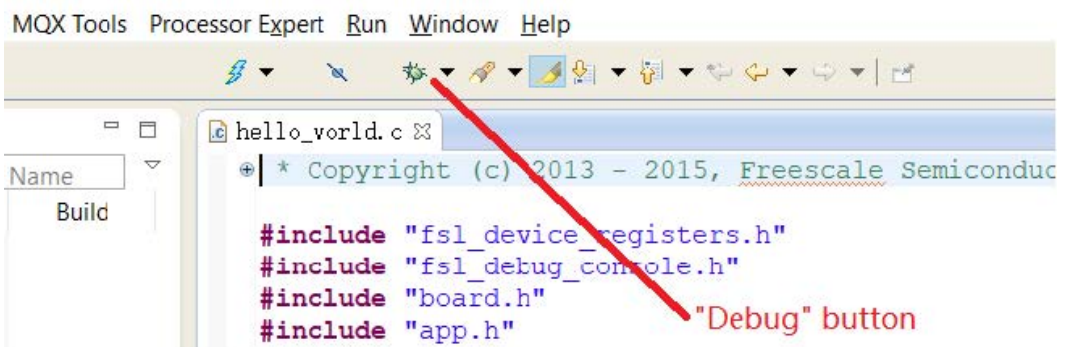
Run an example application To download and run the application, perform the following steps:

1. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (see [How to determine COM port](#)). Configure the terminal with these settings:
 - 115200, defined by BOARD_DEBUG_UART_BAUDRATE in the *board.h* file
 - No parity
 - 8 data bits

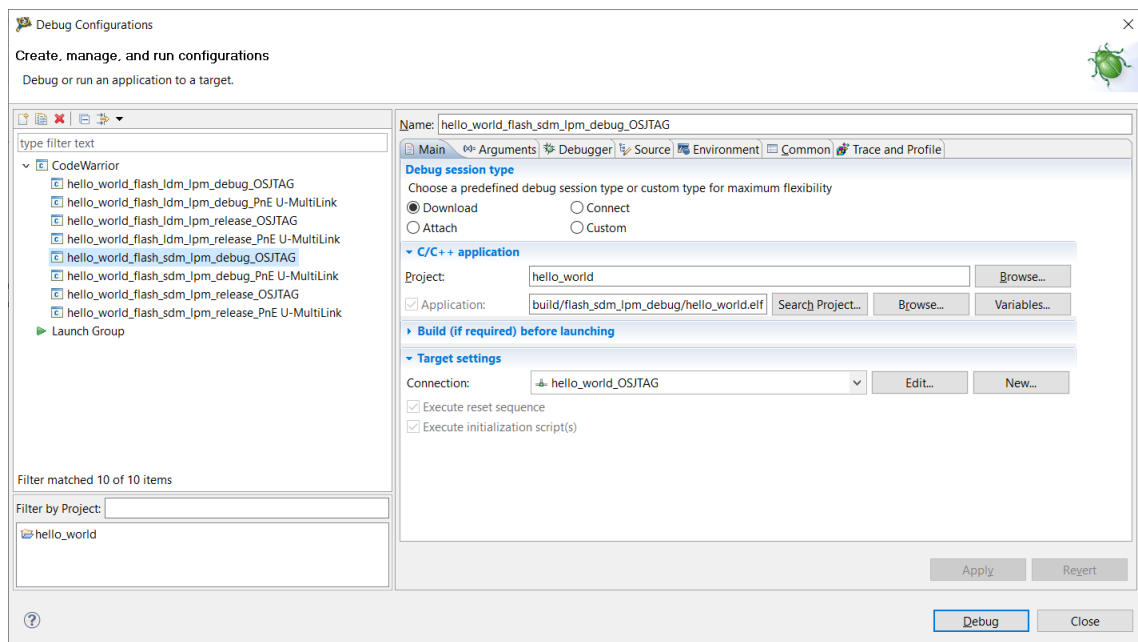


- 1 stop bit

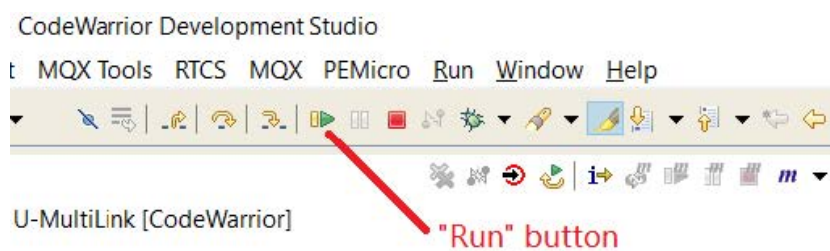
2. For this example (TWR-MC56F8400 hello_world), click **Debug** in the **Commander** pane, and select the hello_world_flash_sdm_lpm_debug_OSJTAG launch configuration. Then the application is downloaded onto target board and automatically runs to the main() function in CodeWarrior Development Studio



tion.

**Note:**

- Generally there are four build configurations for DSC SDK examples: flash_sdm_lpm_debug, flash_sdm_lpm_release, flash_ldm_lpm_debug, and flash_ldm_lpm_release.
 - *_debug: uses optimization level 1
 - *_release: uses optimization level 4
 - sdm: small data memory model
 - ldm: large data memory model
 - lpm: large program memory model
- Select corresponding launch configuration based on build target and debugger type.
- Some examples may require specific hardware settings, check each demo readme document, which includes detail instructions for HW and SW settings.



3. To run the code, click **Run** on the toolbar.
4. The hello_world application is now running and a banner is displayed on the terminal.



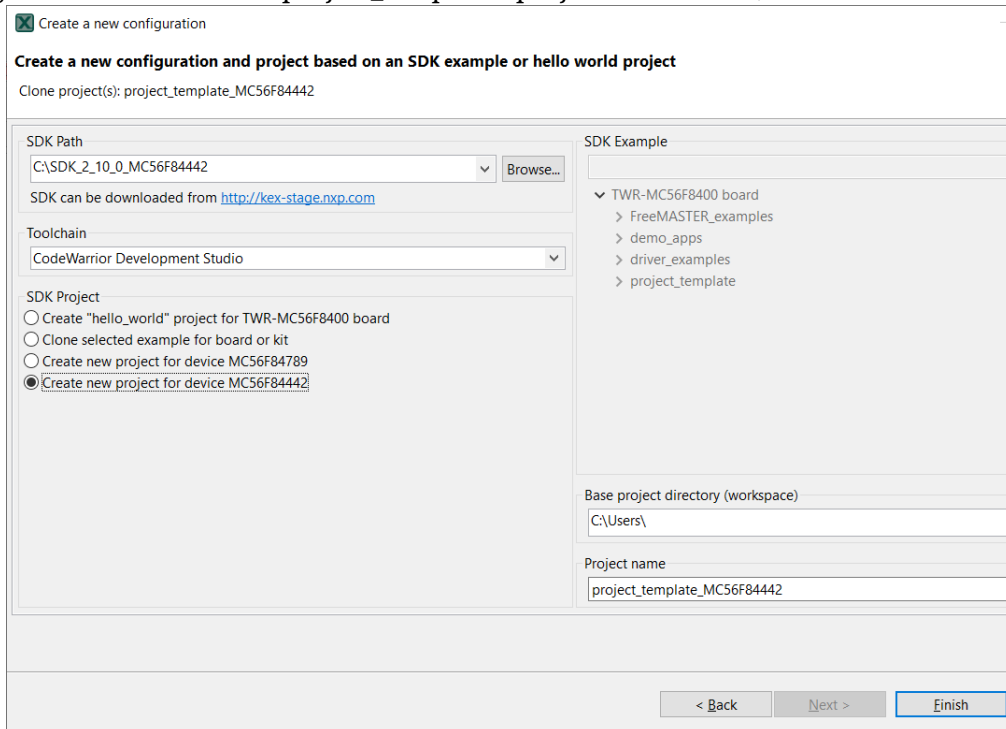
Project template for a specific DSC part

For device with specific part number, the easiest way to set up customer own project based on MCUXpresso DSC SDK peripheral driver, is the project_template. MCUXpresso Config Tool is used to generate the project_template.

The project_template provides basic MCUXpresso DSC SDK software framework, including startup, linker file, device header file, debug setting, peripheral driver, FreeMASTER, and so on.

Steps to generate the project_template for specific derivative part number by MCUXpresso Config Tool

1. Download the specific device SDK package and unzip it. note: The project template requires FreeMASTER, middleware FreeMASTER selection is a must when downloading DSC SDK from nxp website
2. Use MCUXpresso Config Tool to create a project_template project as below(take



MC56F84442 as example).

3. Import the generated template_project into CodeWarrior IDE and start the development.

NOTE

- The default created project template by Config Tool is `project_template_{part_number}`. User could modify the default name in **Project name** textbox.
- All peripheral drivers files are included in the generated `project_template` project. They are same as the peripheral drivers within SDK package. If some drivers are not used or required, users may delete them in CodeWarrior, or delete them directly under folder `${project_path}/drivers`.

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.

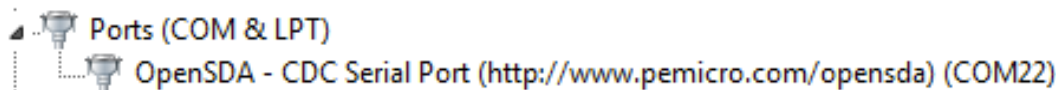
2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

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.

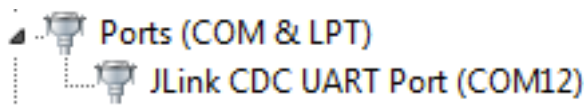
1. **CMSIS-DAP/mbed/DAPLink** interface:



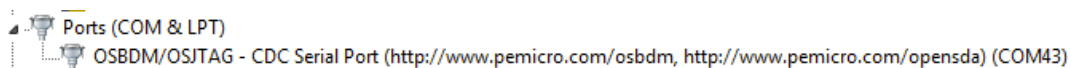
2. **P&E Micro:**



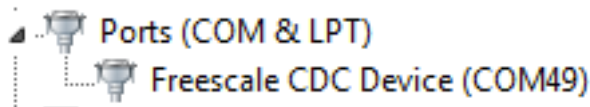
3. **J-Link:**



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



5. **MRB-KW01:**



1.3 Getting Started with MCUXpresso SDK GitHub

1.3.1 Getting Started with MCUXpresso SDK Repository

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

Overview

The GitHub Repository SDK approach offers:

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

Benefits of the Multi-Repository Approach

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

Setup and Configuration

Follow these steps to prepare your development environment:

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

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

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

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

Essential Tools

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

Installation:

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

Setup:

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

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

Installation:

- Install Python **3.10 or newer** from python.org
- **Important:** Check “Add Python to PATH” during installation

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

Installation:

```
pip install -U west
```

Minimum version: 1.2.0 or newer

Build System Tools

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

Recommended version: 3.30.0 or newer

Installation:

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

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

Minimum version: 1.12.1 or newer

Installation:

- **Windows:** Usually included, or download from ninja-build.org
- **Linux:** sudo apt install ninja-build or download binary
- **macOS:** brew install ninja or download binary

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

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

Installation: Follow the Ruby environment setup guide

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

Toolchain	Best For	Download Link	Environment Variable
ARM GCC (Recommended)	Most users, free	ARM GNU Toolchain	ARMGCC_DIR
IAR EWARM	Professional development	IAR Systems	IAR_DIR
Keil MDK ARM Compiler	ARM ecosystem Advanced optimization	ARM Developer ARM Developer	MDK_DIR ARMCLANG_DIR

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

Windows:

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

Linux/macOS:

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

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

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

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

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

Python/pip issues:

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

Slow downloads:

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

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

Prerequisites Verify the requirements:

System Requirements:

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

Verification Commands:

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

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

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

Get the latest SDK from main branch:

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

Get SDK at specific revision:

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

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

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

```
cd mcuxpresso-sdk
```

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

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

```
west update
```

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

Best for:

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

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

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

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

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

Best for:

- Single board development
- Faster setup and reduced disk usage
- Focused development workflows

Examples:

```
# Update only repositories for FRDM-MCXW23 board
west update_board --set board frdmxcw23

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

Step 3: Verify Installation Confirm successful setup:

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

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

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

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

```
# Update only repositories for specific board, e.g., frdmxcw23
west update_board --set board frdmxcw23

# List available repositories for the board before updating
west update_board --set board frdmxcw23 --list-repo
```

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

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

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

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

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

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

Benefits of Targeted Setup **Reduced Download Size**

- Download only components needed for your target board or device
- Significantly faster initial setup for focused development
- Typical reduction from 7 GB to 2GB

Optimized Workspace

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

Flexible Development

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

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

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

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

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

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

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

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

Workspace Management

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

For Complete SDK Workspace:

```
# Update manifest repository
cd manifests
git pull

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

For Targeted Workspace:

```
# Update manifest repository
cd manifests
git pull

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

Workspace Status Check workspace synchronization status:

```
# Show status of all repositories
west status

# Show detailed information about repositories
west list
```

Troubleshooting Network Issues:

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

Permission Issues:

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

Disk Space:

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

Repository Management Issues:

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

Next Steps With your workspace initialized:

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

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

Explore SDK Structure and Content

Learn about the organization of the SDK and its components:

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

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

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

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

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

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

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

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

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

- `nxp-mcuxpresso`
- `NXP`
- `nxp-zephyr`

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

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

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

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

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

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

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

Top-Level Organization

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

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

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

Directory	Contents	Purpose
arch/	Architecture-specific files	ARM CMSIS, build configurations
cmake/	Build system modules	CMake configuration and build rules
compo	Software components	Reusable software libraries and utilities
devices	Device support packages	MCU-specific headers, startup code, linker scripts
drivers	Peripheral drivers	Hardware abstraction layer for MCU peripherals
examp	Sample applications	Demonstration code and reference implementations
middle	Optional software stacks	Networking, graphics, security, and other libraries
rtos/	Operating system support	FreeRTOS integration
scripts	Build and utility scripts	West extensions and development tools
svd	Svd files for devices, this is optional because of large size. Customers run <code>west manifest config group.filter +optional</code> and <code>west update mcux-soc-svd</code> to get this folder.	

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

Common Example Files

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

Board-Specific Files

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

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

```

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

```

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

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

Documentation Structure SDK documentation is distributed across multiple locations:

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

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

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

1. General README: examples/demo_apps/hello_world/readme.md

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

2. Board-Specific README: examples/_boards/{board_name}/demo_apps/hello_world/example_board_readme.md

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

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

Development Workflows

Get started with building and running projects:

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

Prerequisites

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

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

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

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

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

Basic Build Process

Simple Build Build the hello_world example with default settings:

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

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

Specifying Configuration

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

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

Alternative Toolchains

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

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

Multicore Applications For multicore devices, specify the core ID:

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

For multicore projects using sysbuild:

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

Flash an Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Start a debug session:

```
west debug -r linkserver
```

Common Build Options

Clean Build Force a complete rebuild:

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

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

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

Device Variants For boards supporting multiple device variants:

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

Project Configuration

CMake Configuration Only Run configuration without building:

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

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

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

Getting Help View the help information for west build:

```
west build -h
```

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

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

Next Steps

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

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

Prerequisites

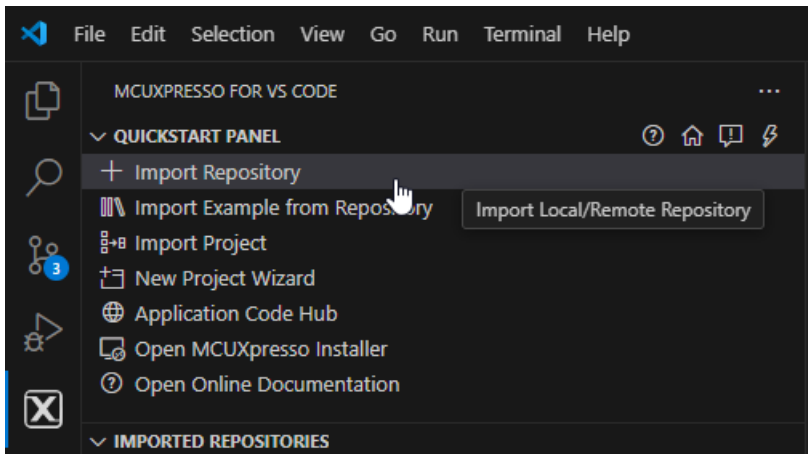
- SDK workspace initialized (GitHub Repository SDK or Repository-Layout SDK Package)
- Development tools installed per [Installation Guide](#)
- Visual Studio Code installed
- MCUXpresso for VS Code extension installed

Extension Installation

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

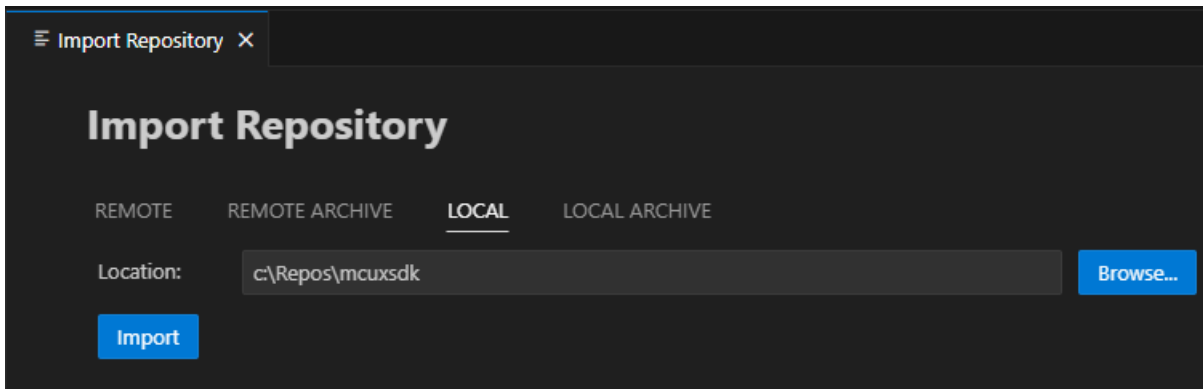
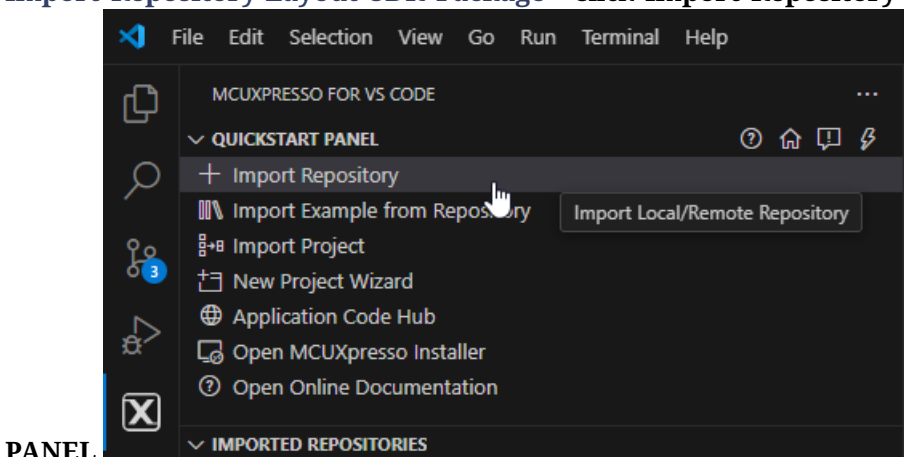
SDK Import and Setup

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

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

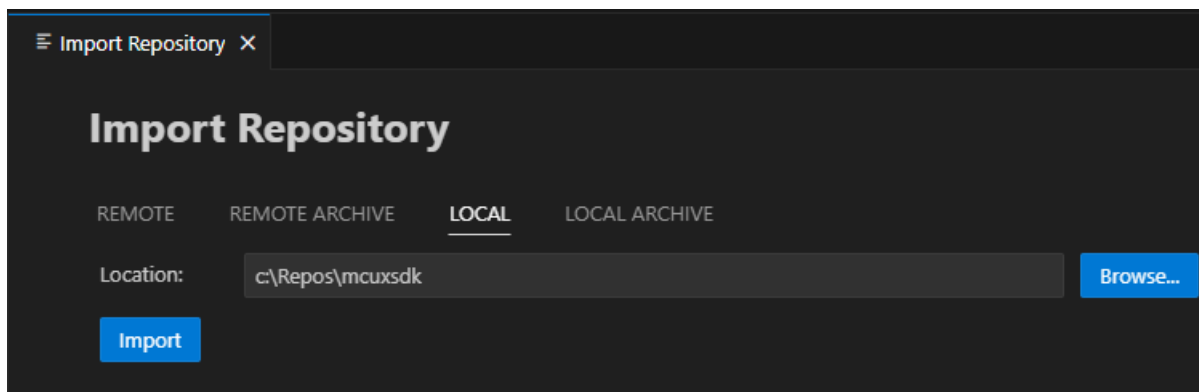
Note: You can import the SDK in several ways. Refer to the [MCUXpresso for VS Code documentation](#) for details.

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

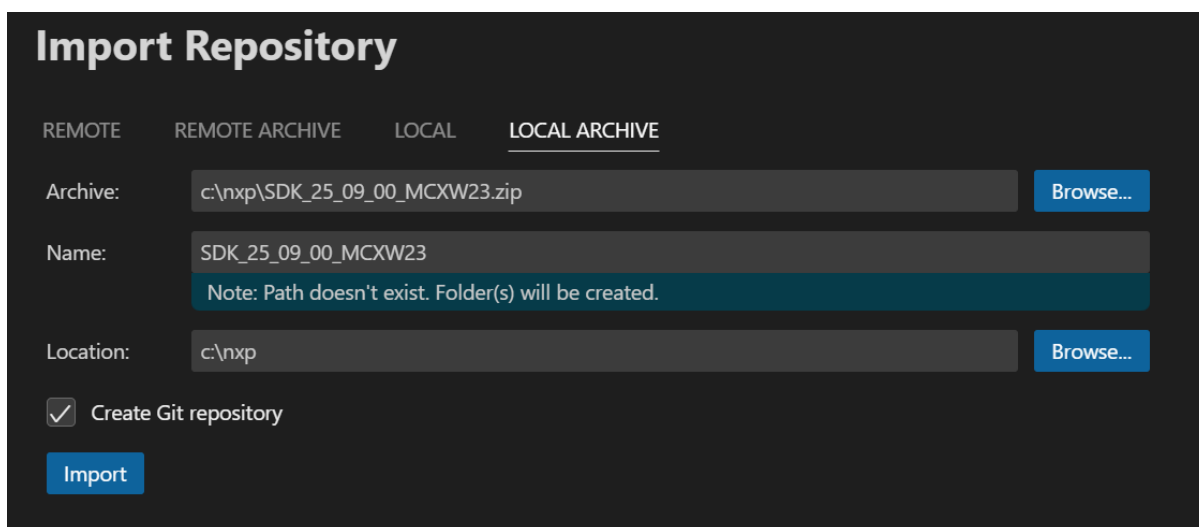
**Import Repository-Layout SDK Package** Click **Import Repository** from the **QUICKSTART**

PANEL

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



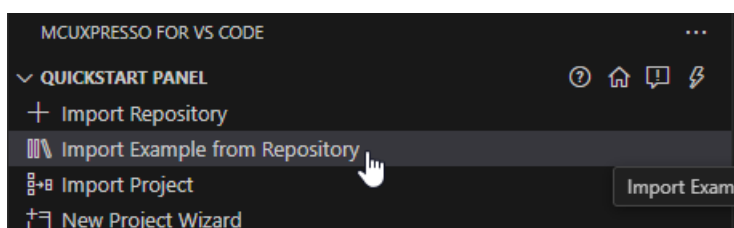
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



Building Example Applications

Import Example Project

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



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

3. Click **Import**


Import Example from Repository X

Import Example from Repository

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

Toolchain: (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009 (C:\NXP\MCUXpressoIDE_24. | v

Board: FRDM-MCXC444 | v



FRDM-MCXC444

Template: demo_apps/hello_world | v

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

App type: Freestanding application | v

Name: frdmmcxc444_hello_world

Location: c:\nxp_examples Browse...

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

Open readme file after project is imported

Import

Application Types **Repository Applications:**

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

Freestanding Applications:

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

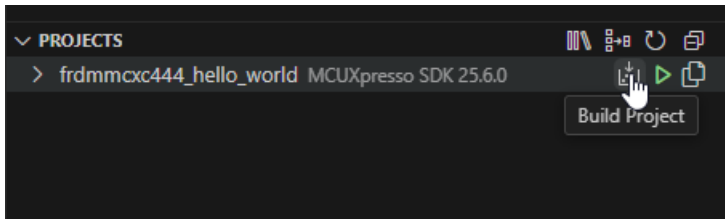
Trust Confirmation VS Code will prompt you to confirm if the imported files are trusted. Click **Yes** to proceed.

Building Projects

Build Process

1. Navigate to **PROJECTS** view

2. Find your project
3. Click the **Build Project** icon

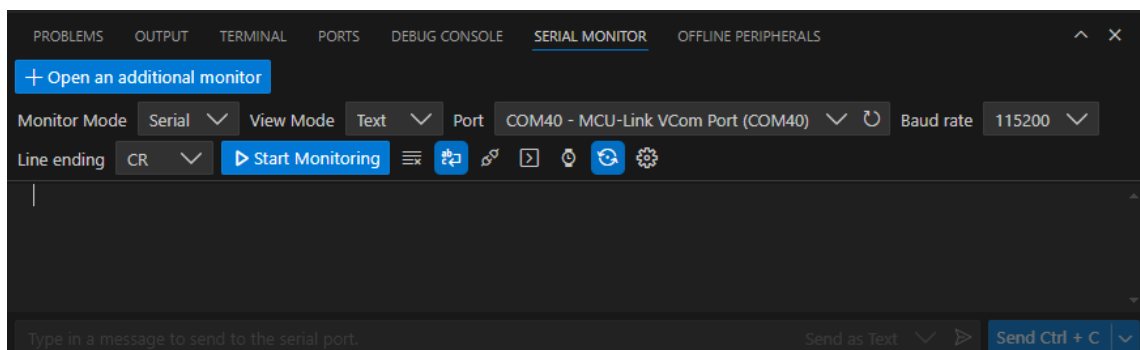


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

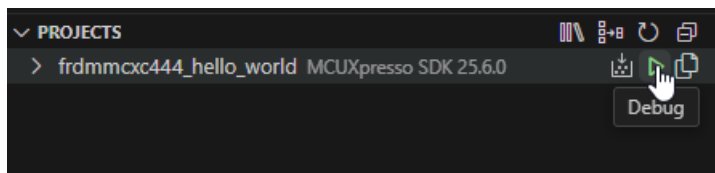
1. Open **Serial Monitor** from VS Code's integrated terminal



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

Debug Session

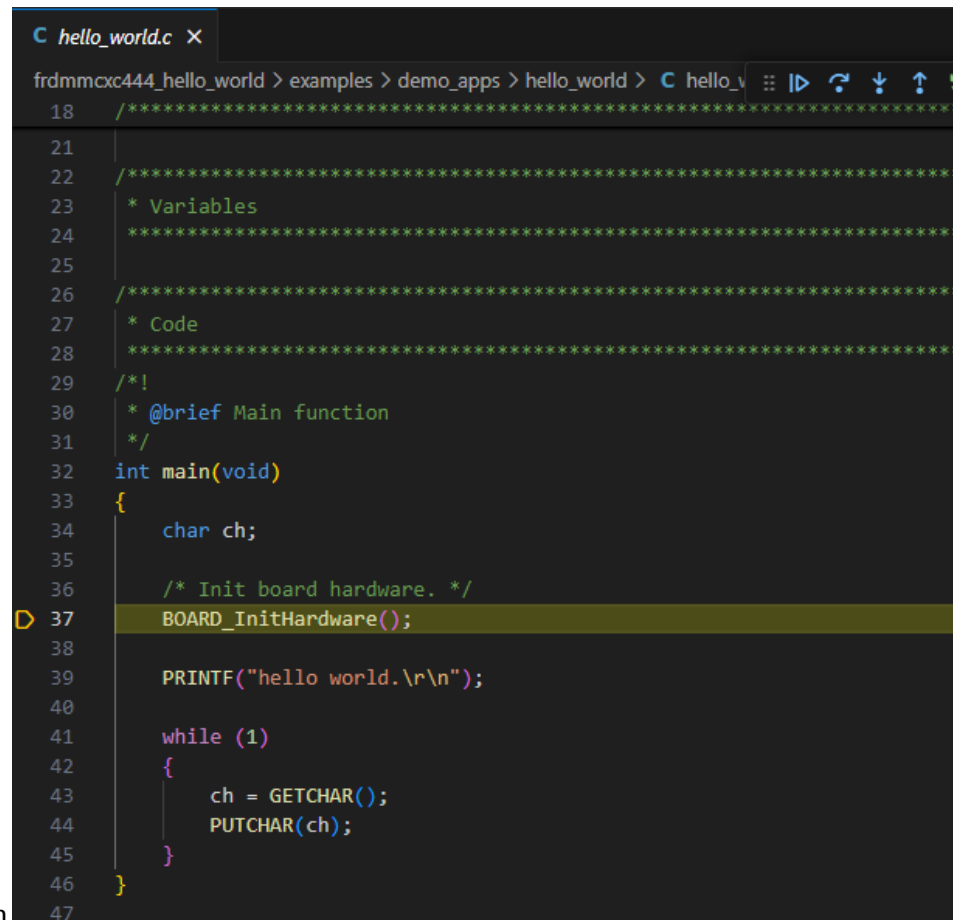
1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



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

Debug Controls Use the debug controls to manage execution:

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



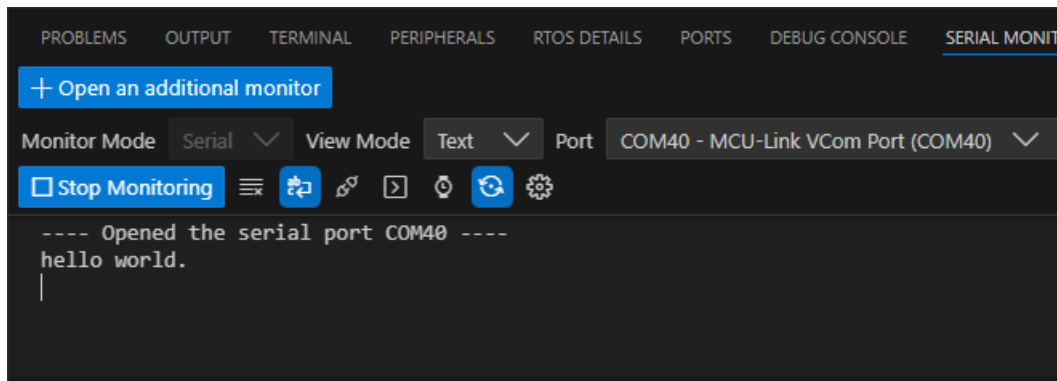
```

18  /*****
21
22  /*****
23  * Variables
24  *****/
25
26  /*****
27  * Code
28  *****/
29  /*!
30  * @brief Main function
31  */
32  int main(void)
33  {
34      char ch;
35
36      /* Init board hardware. */
37      BOARD_InitHardware();
38
39      PRINTF("hello world.\r\n");
40
41      while (1)
42      {
43          ch = GETCHAR();
44          PUTCHAR(ch);
45      }
46  }
47

```

- **Stop:** Terminate debug session

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



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

Project Configuration

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

- Example applications
- Board configurations

- Middleware components
- Build system integration

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

Troubleshooting

Import Issues SDK not detected:

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

Project import failures:

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

Build Problems Build failures:

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

Debug Issues Debug session fails:

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

Serial monitor problems:

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

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

Advanced Features

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

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

Next Steps

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

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

Prerequisites

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

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

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

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

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

Basic Build Commands

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

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

Specifying Build Configuration

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

# Debug build with specific toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar --config debug
```

Multicore Applications For multicore devices, specify the core ID:

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

For multicore projects using sysbuild:

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

Shield Support For boards with shields:

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

Advanced Build Options

Clean Builds Force a complete rebuild:

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

Dry Run See what commands would be executed:

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

Device Variants For boards supporting multiple device variants:

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

Project Configuration

CMake Configuration Only Run configuration without building:

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

Interactive Configuration Launch the configuration GUI:

```
west build -t guiconfig
```

Flashing and Debugging

Flash Application Flash the built application to your board:

```
west flash -r linkserver
```

Debug Session Start a debugging session:

```
west debug -r linkserver
```

IDE Project Generation Generate IDE project files for traditional IDEs:

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

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

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

Troubleshooting

Build Failures Use pristine builds to resolve dependency issues:

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

Toolchain Issues Verify environment variables are set correctly:

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

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

Getting Help Display help information:

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

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

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

Best Practices

Project Organization

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

Build Efficiency

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

Development Workflow

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

Next Steps

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

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

Prerequisites

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

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

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

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

Manual Workflow Use the following steps:

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

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

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

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

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

Changes in the Config tools generated source code modules may require adjustments to the toolchain project to ensure a successful build. These changes may mean, for example, adding the newly generated files, adding include paths, required drivers, or other SDK components. This section describes how to manually resolve the changes needed in the project within the toolchain projects based on the SDK project managed by the West tool.

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

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

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

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

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

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

The MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with Arm Cortex-M-based devices from NXP, including its general purpose, crossover and Bluetooth-enabled MCUs. MCUXpresso SW and Tools for DSC further extends the SDK support to current 32-bit Digital Signal Controllers. The MCUXpresso SDK includes production-grade software with integrated RTOS (optional), integrated enabling software technologies (stacks and middleware), reference software, and more.

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

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

MCUXpresso SDK

As part of the MCUXpresso software and tools, MCUXpresso SDK is the evolution of Kinetis SDK, includes support for LPC, DSC, PN76, and i.MX System-on-Chip (SoC). The same drivers, APIs, and

middleware are still available with support for Kinetis, LPC, DSC, and i.MX silicon. The MCUXpresso SDK adds support for the MCUXpresso IDE, an Eclipse-based toolchain that works with all MCUXpresso SDKs. Easily import your SDK into the new toolchain to access to all of the available components, examples, and demos for your target silicon. In addition to the MCUXpresso IDE, support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

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

Development tools

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

- CodeWarrior Development Studio v11.2 with CodeWarrior for DSC v11.2 SP1 (Service Pack 1)

Supported development systems

This release supports board and devices listed in following table. The board and devices in bold were tested in this release.

De-velopment boards	MCU devices			
MC56F8 EVK	MC56F83663VLH, MC56F83763VLH, MC56F83783AVLH, MC56F83789VLL ,	MC56F83683VLH, MC56F83766VLK, MC56F83783VLH, MC56F83686VLK	MC56F83689VLL, MC56F83769AVLL, MC56F83786VLK,	MC56F83763AVLH, MC56F83769VLL, MC56F83789AVLL,

Release contents

Table 1 provides an overview of the MCUXpresso DSC 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
Documentation	<install_dir>/docs
Driver, SoC header files, extension header files and feature header files	<install_dir>/devices/<device_name>
Peripheral Drivers	<install_dir>/devices/<device_name>/drivers
Utilities such as debug console	<install_dir>/devices/<device_name>/utilities
Middleware	<install_dir>/middleware

MCUXpresso SDK release package

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

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

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

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

Middleware

USB Type-C PD Stack See the *MCUXpresso SDK USB Type-C PD Stack User's Guide* (document MCUXSDKUSBPDUG) for more information

USB Host, Device, OTG Stack See the *MCUXpresso SDK USB Stack User's Guide* (document MCUXSDKUSBSUG) for more information.

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

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

Known issues

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

PRINTF issue for program address space When project is compiled with SDM, print the address in program address space malfunction.

- Failed example when SDM
 - `PRINTF("%p", main);` Root cause: in SDM, %p is treated as 16-bit value, however main in program address space is still considered as 32-bit.
- Workaround(compliant with SDM and LDM)
 - `PRINTF("0x%x", (uint32_t)main);`

OSJTAG USB function failure The JM60 USB port, **J8**, is used for OSJTAG and USB CDC bridge. The JM60 USB port cannot work properly after being replugged when board is powered by another power source.

One example of the failure case:

1. Power the board with DSC USB port, **J21**.
2. Replug/Re-power the JM60 USB port.

Demos not support SDM As ROM API is built in LDM, flash driver demo only support LDM.

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
-

CADC

[2.2.0]

- New Features
- Supported platforms which don't have ANA4 expansion MUX.

[2.1.0]

- Improvements
- Added some APIs to support some devices that equipped expansion mux.

[2.0.1]

- Bug Fixes
- Fixed the bug that channel mode set to wrong value.
- Fixed the bug that independent parallel mode set to wrong value.
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

CLOCK

[2.1.0]

- Accumulated bug fix
- MISRA warning fix

[2.0.0]

- Initial version.
-

CMP

[2.0.1]

- Improvements
- Supported MC56F82xxxx and MC56F84xxxx.
 - Bug Fixes
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

COMMON

[2.6.3]

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

[2.6.2]

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

[2.6.1]

- Improvements
 - Support Cortex M23.

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

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

[2.4.2]

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

[2.4.1]

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

[2.4.0]

- New Features
 - Added `EnableIRQWithPriority`, `IRQ_SetPriority`, and `IRQ_ClearPendingIRQ` for ARM.
 - Added `MSDK_EnableCpuCycleCounter`, `MSDK_GetCpuCycleCount` for ARM.

[2.3.3]

- New Features
 - Added `NETC` into status group.

[2.3.2]

- Improvements
 - Make driver aarch64 compatible

[2.3.1]

- Bug Fixes
 - Fixed MAKE_VERSION overflow on 16-bit platforms.

[2.3.0]

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

[2.2.10]

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

[2.2.9]

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

[2.2.8]

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

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

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

[2.2.0]

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

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

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

[2.1.2]

- Improvements
 - Add SUPPRESS_FALL_THROUGH_WARNING() macro for the usage of suppressing fallthrough warning.

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

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

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

COP

[2.2.3]

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

[2.2.2]

- Bug Fixes
- Added configuration of CWP bits in COP_Init, fixed write protection bEnableWriteProtect cannot be configured as part of cop_config_t.

[2.2.1]

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

[2.2.0]

- Improvements
- Updated cop_config_t member naming.
- Deleted COP_Disable API, added COP_Enable to enable/disable COP.

[2.1.0]

- Improvements
- API interface changes:
 - Renamed “COP_EnableInterrupts/COP_DisableInterrupts” to “COP_EnableInterrupt/COP_DisableInterrupt” and remove unnecessary parameter.
 - New Features
- Added APIs to enable/disable the COP COP Loss of Reference counter.

[2.0.0]

- Initial version.
-

CRC

[2.0.1]

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

[2.0.0]

- Initial version.
-

DAC

[2.0.1]

- Improvements
- Supported MC56F82xxx and MC56F84xxx.
 - Bug Fixes
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

DMAMUX

[2.0.0]

- Initial version.
-

EDMA

[2.0.3]

- Fixed the MISRA-2012 violations.
- Fixed rule 10.3.

[2.0.2]

- Fixed the MISRA-2012 violations.
- Fixed rule 5.8, 9.2, 10.3, 10.4, 11.6.

[2.0.1]

- Code modification for SDM compliance

[2.0.0]

- Initial version.
-

EVTG

[2.0.0]

- Initial version.
-

EWM

[2.0.2]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

FLASH

[3.2.0]

- Change file name to fsl_flash.x

[3.1.0]

- Accumulated bug fix
- MISRA warning fix

[3.0.0]

- Initial version.
-

FLEXCAN

[2.1.2]

- Bug Fixes
- Fixed the violation of MISRA-2012 rule : 10.1, 14.3, 17.7.

[2.1.1]

- Bug Fixes
- Fixed the violation of MISRA-2012 rule.
- Fixed CBT SEG2 potential overflow issue.

[2.1.0]

- Improvement
- Add bEnableTimingCalc member in flexcan_config_t structure to make users can choose to use their desired timing values or use auto calculate timing values to initialize FLEXCAN module.
- Update FLEXCAN_CalculateImprovedTimingValues/FLEXCAN_FDCalculateImprovedTimingValues APIs to improvement the bit timing calculate.
- Add FLEXCAN_CalculateImprovedTimingValuesWithCBT API.
- Move TDC config operations to FLEXCAN_SetFDTimingConfig API.
- Add FLEXCAN_SetBaudRate/FLEXCAN_SetFDBaudRate APIs to enable auto update CAN bit timing configuration base on baud rate parameter.
- Update FLEXCAN_WriteFDTxMb to make it can send both CAN FD frame and classic frame.

[2.0.2]

- Bug Fixes
- Fixed build error by adding macro isolation to all CAN FD codes.

[2.0.1]

- Bug Fixes
- Fixed build warnings.

[2.0.0]

- Initial version.
-

GPIO

[2.0.1]

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

[2.0.0]

- Initial version.
-

I2C

[2.0.1]

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

[2.0.0]

- Initial version.
-

INTC

[2.0.1]

- Improvements
- Added doxygen comments.

[2.0.0]

- Initial version.
-

MCM

[2.0.1]

- Improvements
- Supported MC56F82xxx and MC56F84xxx.

[2.0.0]

- Initial version.
-

PIT

[2.3.1]

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

[2.3.0]

- Improvements
- Filtered Preset input to reset PIT counter.
- Support SYNC_OUT output stretch and toggle mode.
- Added PIT_SetPresetFiltConfig() to set FILT register configurations.
- Added PIT_SetSyncOutConfig() to set SYNC register configurations.

[2.2.1]

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

[2.2.0]

- Improvements
- Updated pit_config_t member naming.
- Removed some APIs for prescaler and clock source selection.

[2.1.0]

- Improvements
- Updated PIT clock source and PIT prescaler with more meaningful comments.
- Updated PIT_SetTimerPeriod() and PIT_GetCurrentTimerCount() with 16-bit parameter.
- Deleted mask parameter for PIT_ClearStatusFlags/PIT_EnableInterrupts/PIT_DisableInterrupts.
- Added PIT_SetTimerClockSource() API to configure clock source.
- Added PIT_EnableSlaveMode() API to configure slave mode.

[2.0.1]

- New Features
- Added PIT_SetTimerPrescaler() API to configure clock prescaler value.

[2.0.0]

- Initial version.
-

PMC

[2.1.0]

- Improvements
- Added PMC_SetVrefTrim() and PMC_SetVcapTrim() APIs to support MC56F80xxx.

[2.0.0]

- Initial version.
-

eFlexPWM

[2.2.0]

- New Features
- Supported capture PWM input filter.
- Supported different PWM deadtime count register width.
 - Bug Fixes
- Fixed wrong pwm_sm_pwm_out_t enum order issue.

[2.1.1]

- Bug Fixes
- Fixed build error when soc not support Capture A/B features.

[2.1.0]

- Improvements
- Supported MC56F80xxx.

[2.0.2]

- Bug Fixes
- Fixed clear status flags API doesn't work issue.

[2.0.1]

- Improvements
- Supported MC56F82xxx and MC56F84xxx .
 - Bug Fixes
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

QSCI

[2.0.5]

- New Feature
 - Added common IRQ handler entry QSCI_DriverIRQHandler.

[2.0.4]

- Bug Fixes
- Fixed DMA transfer blocking issue by enabling tx idle interrupt after DMA transmission finishes, and invoke completion callback after tx idle interrupt occurs.

[2.0.3]

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

[2.0.2]

- Improvements
- Supported QSCI which has 13-bit integer and 3-bit fractional baud rate selection.

[2.0.1]

- Bug Fixes
- Fixed bug that when starting the non-blocking receive, the rx idle interrupt is not enabled, and when receiving is done the rx idle interrupt is not disabled.

[2.0.0]

- Initial version.
-

QTMR

[2.0.1]

- Improvements
- Supported to get TMR capture register address.
 - Bug Fixes
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

Queued SPI

[2.1.2]

- New Feature
 - Added common IRQ handler entry QSPI_DriverIRQHandler.

[2.1.1]

- Bug Fixes
- Fixed wrong baudrate calculation method.

[2.1.0]

- Bug Fixes
- Fixed wrong definitions of interrupt enable/disable masks.
- Fixed wrong usage of QSPI_DisableInterrupts.
- Fixed wrong type casts.
- Fixed bug for master blocking transfer of rx FIFO overflow.

[2.0.0]

- Initial version.
-

SIM

[2.2.0]

- Use dedicated SIM driver for MC56F83xxx

[2.1.0]

- Improvements
- Updated support for MC56F82xxx and MC56F84xxx.
 - Bug Fixes
- Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

DSC_XBARA

[2.0.1]

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

[2.0.0]

- Initial version.
-

Chapter 2

MC56F83789

2.1 CADC: 12-bit Cyclic Analog-to-Digital Converter Driver

`void CADC_Init(ADC_Type *base, const cadc_config_t *psConfig)`

Initializes the CADC module, such as scan mode, DMA trigger source, interrupt mask and so on.

This function is to make the initialization for using CADC module. The operations are:

- Enable the clock for CADC.
- Set power up delay and Idle work mode.
- Set DMA trigger source.
- Enable the interrupts(Including High/Low limit interrupt, zero crossing interrupt interrupt, end of scan interrupt and each sample slot's scan interrupt).
- Set scan mode.
- Set disabled sample slot for the scan.
- Set scan control options.
- Set selected channels' mode.
- Set gain for each channel.
- Config contera and converterB.

Note: The high limit value, low limit value, offset value and zerocrossing mode of each sample slot will not be configured in this function, to set those options, the APIs in “Sample Slot Control Interfaces” function group can be used.

Parameters

- `base` – CADC peripheral base address.
- `psConfig` – Pointer to configuration structure. See `cadc_config_t`.

`void CADC_GetDefaultConfig(cadc_config_t *psConfig)`

Gets an available pre-defined options(such as scan mode, DMA trigger source, interrupt mask and so on) for module's configuration.

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

```

psConfig->eDMATriggerSource = kCADC_DMATrigSrcEndofScan;
psConfig->eIdleWorkMode = kCADC_IdleKeepNormal;
psConfig->u16PowerUpDelay = 26U;
psConfig->u32EnabledInterruptMask = 0U;
psConfig->eScanMode = kCADC_ScanModeTriggeredParallelSimultaneous;
psConfig->uDisabledSampleSlot.u32SampleDisVal = 0xFF0F0UL;
psConfig->uScanControl.u32ScanCtrlVal = 0x0UL;
psConfig->eChannelGain[x] = kCADC_SignalGainX1;
psConfig->sampleSlotScanInterruptEnableMask = kCADC_NonSampleSlotMask;
For the default setting of converter, please see CADC_GetConverterDefaultConfig().

```

Parameters

- psConfig – Pointer to configuration structure. See `cadc_config_t`.

`void CADC_Deinit(ADC_Type *base)`

De-initializes the CADC module, including power down both converter and disable the clock(Optional).

This function is to make the de-initialization for using CADC module. The operations are:

- Power down both converter.
- Disable the clock for CADC.

Parameters

- base – CADC peripheral base address.

`static inline void CADC_SetScanMode(ADC_Type *base, cadc_scan_mode_t eScanMode)`

Sets the scan mode(such as Sequential scan mode, Simultaneous parallel scan mode, Independent parallel scan

mode) of dual converters.

Parameters

- base – CADC peripheral base address.
- eScanMode – Dual converters' scan mode, please see `cadc_scan_mode_t` for details.

`static inline void CADC_SetScanControl(ADC_Type *base, cadc_scan_control_t uScanControl)`

The function provides the ability to pause and await new sync in the conversion sequence.

Parameters

- base – CADC peripheral base address.
- uScanControl – The scan control value, please refer to `cadc_scan_control_t` for details.

`void CADC_SetChannelMode(ADC_Type *base, cadc_channel_mode_t eChannelMode)`

Sets mode for the specific channel(Each channel can be set as single-end, fully differential and unipolar differential(Optional) mode).

Parameters

- base – CADC peripheral base address.
- eChannelMode – The channel mode to be set, please refer to `cadc_channel_mode_t` for details.

```
void CADC_SetChannelGain(ADC_Type *base, cadc_channel_number_t eChannelNumber,
                        cadc_channel_gain_t eChannelGain)
```

Sets the gain(Supports X1, X2, X4) of selected channel.

Parameters

- base – ADC peripheral base address.
- eChannelNumber – The number of channel, please refer to *cadc_channel_number_t*.
- eChannelGain – The gain amplification, please refer to *cadc_channel_gain_t* for details.

```
void CADC_GetSampleSlotDefaultConfig(cadc_sample_slot_config_t *psConfig)
```

Gets sample slot default configuration including zero crossing mode, high limit value, low limit value and offset value.

```
psConfig->eZeroCrossingMode = kCADC_ZeroCrossingDisabled;
psConfig->u16HighLimitValue = 0x7FF8U;
psConfig->u16LowLimitValue = 0x0U;
psConfig->u16OffsetValue = 0x0U;
```

Parameters

- psConfig – Pointer to configuration structure. See *cadc_sample_slot_config_t*.

```
void CADC_SetSampleSlotConfig(ADC_Type *base, cadc_sample_slot_index_t eSampleIndex,
                             const cadc_sample_slot_config_t *psConfig)
```

Configures the options(including zero crossing mode, high limit value, low limit value and offset value) for sample slot.

Note: This function can be used to set high limit value, low limit value, offset value and zerocrossing mode of the sample slot.

Parameters

- base – ADC peripheral base address.
- eSampleIndex – Index of sample slot in conversion sequence. Please refer to *cadc_sample_slot_index_t*.
- psConfig – Pointer to configuration structure. See *cadc_sample_slot_config_t*.

```
void CADC_SetSampleSlotZeroCrossingMode(ADC_Type *base, cadc_sample_slot_index_t
                                         eSampleIndex,
                                         cadc_sample_slot_zero_crossing_mode_t
                                         eZeroCrossingMode)
```

Sets zero-crossing mode for the selected sample slot.

Parameters

- base – ADC peripheral base address.
- eSampleIndex – The index of sample slot. Please refer to *cadc_sample_slot_index_t* for details.
- eZeroCrossingMode – Zero crossing mode, please refer to *cadc_sample_slot_zero_crossing_mode_t* for details.

```
void CADC_RouteChannelToSampleSlot(ADC_Type *base, cadc_sample_slot_index_t
                                   eSampleIndex, cadc_channel_number_t
                                   eChannelNumber)
```

Routes the channel to the sample slot.

Parameters

- *base* – ADC peripheral base address.
- *eSampleIndex* – The index of sample slot, please refer to *cadc_sample_slot_index_t* for details.
- *eChannelNumber* – Sample channel number, please refer to *cadc_channel_number_t* for details.

```
static inline void CADC_SetSampleSlotLowLimitValue(ADC_Type *base,
                                                    cadc_sample_slot_index_t eSampleIndex,
                                                    uint16_t u16LowLimitValue)
```

Sets the low limit value(-32768 ~ 32767 with lower three bits of fixed value 0) for the specific sample slot.

Parameters

- *base* – ADC peripheral base address.
- *eSampleIndex* – The index of sample slot. Please refer to *cadc_sample_slot_index_t* for details.
- *u16LowLimitValue* – Low limit value(-32768 ~ 32767 with lower three bits of fixed value 0). Original value formation as hardware register, with 3-bits left shifted.

```
static inline void CADC_SetSampleSlotHighLimitValue(ADC_Type *base,
                                                    cadc_sample_slot_index_t eSampleIndex,
                                                    uint16_t u16HighLimitValue)
```

Sets the high limit value(-32768 ~ 32767 with lower three bits of fixed value 0) for the specific sample slot.

Parameters

- *base* – ADC peripheral base address.
- *eSampleIndex* – The index of sample slot. Please refer to *cadc_sample_slot_index_t* for details.
- *u16HighLimitValue* – High limit value(-32768 ~ 32767 with lower three bits of fixed value 0). Original value formation as hardware register, with 3-bits left shifted.

```
static inline void CADC_SetSampleSlotOffsetValue(ADC_Type *base, cadc_sample_slot_index_t
                                                  eSampleIndex, uint16_t u16OffsetValue)
```

Sets the offset value(-32768 ~ 32767 with lower three bits of fixed value 0) for the specific sample slot.

Parameters

- *base* – ADC peripheral base address.
- *eSampleIndex* – The index of sample slot. Please refer to *cadc_sample_slot_index_t* for details.
- *u16OffsetValue* – Offset value(-32768 ~ 32767 with lower three bits of fixed value 0). Original value formation as hardware register, with 3-bits left shifted.

```
static inline uint16_t CADC_GetSampleSlotResultValue(ADC_Type *base,
                                                    cadc_sample_slot_index_t eSampleIndex)
```

Gets the sample result value.

This function is to get the sample result value. The returned value keeps its original formation just like in hardware result register. It includes the sign bit as the MSB and 3-bit left shifted value.

Parameters

- `base` – ADC peripheral base address.
- `eSampleIndex` – Index of sample slot. For the counts of sample slots, please refer to `cadc_sample_slot_index_t` for details.

Returns

Sample's conversion value.

```
void CADC_GetConverterDefaultConfig(cadc_converter_config_t *psConfig)
```

Gets available pre-defined settings (such as clock divisor, reference voltage source, and so on) for each converter's configuration.

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

```
psConfig->u16ClockDivisor = 4U; (ADC clock = Peripheral clock / 5)
psConfig->eSpeedMode = kCADC_SpeedMode0; (Chip specific)
psConfig->eHighReferenceVoltageSource = kCADC_ReferenceVoltageVrefPad;
psConfig->eLowReferenceVoltageSource = kCADC_ReferenceVoltageVrefPad;
psConfig->u16SampleWindowCount = 0U; (Chip specific)
psConfig->bEnableDMA = false;
psConfig->bPowerUp = false;
psConfig->bScanInitBySync = true;
```

Parameters

- `psConfig` – Pointer to configuration structure. See `cadc_converter_config_t`.

```
void CADC_SetConverterConfig(ADC_Type *base, cadc_converter_id_t eConverterId, const
                             cadc_converter_config_t *psConfig)
```

Configures the options (such as clock divisor, reference voltage source, and so on) for the converter.

This function can be used to configure the converter. The operations are:

- Set clock divisor;
- Set reference voltage source
- Enable/Disable DMA
- Power-up/power-down converter

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter Id. See `cadc_converter_id_t`.
- `psConfig` – Pointer to configuration structure. See `cadc_converter_config_t`.

```
static inline void CADC_EnableConverter(ADC_Type *base, cadc_converter_id_t eConverterId,
                                       bool bEnable)
```

Changes the converter to stop mode or normal mode.

The conversion should only be launched after the converter is in normal mode. When in stop mode, the current scan is stopped and no further scans can start. All the software trigger and hardware trigger are ignored.

Parameters

- base – ADC peripheral base address.
- eConverterId – The converter Id. See `cadc_converter_id_t`.
- bEnable – Used to change the operation mode.
 - **true** Changed to normal mode.
 - **false** Changed to stop mode

```
static inline void CADC_EnableConverterSyncInput(ADC_Type *base, cadc_converter_id_t
                                               eConverterId, bool bEnable)
```

Enables/Disables the external sync input pulse to initiate a scan.

Note: When in “Once” scan mode, this gate would be off automatically after an available sync is received. User needs to enable the input again manually if another sync signal is wanted.

Parameters

- base – ADC peripheral base address.
- eConverterId – The converter Id. See `cadc_converter_id_t`.
- bEnable – Enable the feature or not.
 - **true** Used a SYNC input pulse or START command to initiate a scan.
 - **false** Only use the START command to initiate a scan.

```
static inline void CADC_DoSoftwareTriggerConverter(ADC_Type *base, cadc_converter_id_t
                                                  eConverterId)
```

Uses software trigger to start a conversion sequence.

This function is to do the software trigger to the converter. The software trigger can used to start a conversion sequence.

Parameters

- base – ADC peripheral base address.
- eConverterId – The ID of the converter to be started. See `cadc_converter_id_t`.

```
static inline void CADC_SetConverterClockDivisor(ADC_Type *base, cadc_converter_id_t
                                                eConverterId, uint16_t u16ClockDivisor)
```

Sets clock divisor(Range from 0 to 63) for converterA and conveter B.

Parameters

- base – ADC peripheral base address.
- eConverterId – The converter Id. See `cadc_converter_id_t`.
- u16ClockDivisor – Converter’s clock divisor for the clock source. Available setting range is 0-63.
 - When the clockDivisor is 0, the divisor is 2.
 - For all other clockDivisor values, the divisor is 1 more than the decimal value of clockDivisor: `clockDivisor + 1`

```
void CADC_SetConverterReferenceVoltageSource(ADC_Type *base, cadc_converter_id_t
                                             eConverterId, cadc_reference_voltage_source_t
                                             eHighReferenceVoltage,
                                             cadc_reference_voltage_source_t
                                             eLowReferenceVoltage)
```

Sets converter's reference voltage source(Including high reference voltage source and low reference voltage source).

Parameters

- base – CADC peripheral base address.
- eConverterId – The converter Id. See *cadc_converter_id_t*.
- eHighReferenceVoltage – High voltage reference source, please refer to *cadc_reference_voltage_source_t*.
- eLowReferenceVoltage – Low voltage reference source, please refer to *cadc_reference_voltage_source_t*.

```
void CADC_EnableConverterPower(ADC_Type *base, cadc_converter_id_t eConverterId, bool
                               bEnable)
```

Powers up/down the specific converter.

This function is to enable the power for the converter. The converter should be powered up before the conversion. Once this API is called to power up the converter, the converter would be powered on after a few moment (so-called power up delay, the function CADC_SetPowerUpDelay() can be used to set the power up delay), so that the power would be stable.

Parameters

- base – CADC peripheral base address.
- eConverterId – The converter to be powered. See *cadc_converter_id_t*.
- bEnable – Powers up/down the converter.
 - **true** Power up the specific converter.
 - **false** Power down the specific converter.

```
static inline void CADC_EnableConverterDMA(ADC_Type *base, cadc_converter_id_t
                                           eConverterId, bool bEnable)
```

Enables/Disables the converter's DMA feature.

Parameters

- base – CADC peripheral base address.
- eConverterId – The converter id. See *cadc_converter_id_t*.
- bEnable – Enables/Disables the DMA.
 - **true** Enable the converter's DMA.
 - **false** Disable the converter's DMA.

```
void CADC_SetConverterMuxAuxConfig(ADC_Type *base, cadc_converter_id_t eConverterId,
                                   const cadc_exp_mux_aux_config_t *psMuxAuxConfig)
```

Configures selected converter's expansion mux and aux settings.

Parameters

- base – ADC peripheral base address.

- `eConverterId` – The converter id, see `cadc_converter_id_t`.
- `psMuxAuxConfig` – Pointer to `cadc_exp_mux_aux_config_t` structure.

```
static inline void CADC_ResetConverterExpMuxScan(ADC_Type *base, cadc_converter_id_t
                                                eConverterId)
```

Resets selected converter's expansion mux scan.

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter id, see `cadc_converter_id_t`.

```
static inline void CADC_SetConverterExpansionMuxOperateMode(ADC_Type *base,
                                                           cadc_converter_id_t
                                                           eConverterId,
                                                           cadc_expansion_mux_operate_mode_t
                                                           eOperateMode)
```

Sets selected converter's expansion mux operate mode.

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter id, see `cadc_converter_id_t`.
- `eOperateMode` – Used to set expansion mux operate mode.

```
static inline void CADC_SetConverterAuxiliaryControl(ADC_Type *base, cadc_converter_id_t
                                                    eConverterId, uint16_t u16AuxControl)
```

Sets selected converter's auxiliary control set.

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter id, see `cadc_converter_id_t`.
- `u16AuxControl` – The mask of auxiliary control, should be the OR'ed value of `cadc_auxiliary_control_t`.

```
static inline void CADC_SetConverterMuxChannels(ADC_Type *base, cadc_converter_id_t
                                                eConverterId, uint32_t
                                                u32MuxChannelMask)
```

Sets selected converter's mux channels.

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter id, see `cadc_converter_id_t`.
- `u32MuxChannelMask` – The mask of mux selection of all mux solts, should be the OR'ed value of `cadc_expansion_mux_selection_t`.

```
static inline void CADC_SetExpansionMuxAuxDisabledSlot(ADC_Type *base, cadc_converter_id_t
                                                       eConverterId,
                                                       cadc_expansion_disabled_mux_slot_t
                                                       eDisabledMuxSlot)
```

Set selected converter's mux and aux disabled slot.

Parameters

- `base` – ADC peripheral base address.
- `eConverterId` – The converter id, see `cadc_converter_id_t`.

- `eDisabledMuxSlot` – The mux slot to disabled, please refer to `cadc_expansion_disabled_mux_slot_t`.

```
static inline void CADC_SetPowerUpDelay(ADC_Type *base, uint16_t u16PowerUpDelay)
```

Sets power up delay(The number of ADC clocks to power up the converters before allowing a scan to start).

Parameters

- `base` – CADC peripheral base address.
- `u16PowerUpDelay` – The number of ADC clocks to power up an ADC converter. Ranges from 0 to 63.

```
static inline void CADC_EnableAutoPowerDownMode(ADC_Type *base, bool bEnable)
```

Enables/Disables auto-powerdown converters when the module is not in use for a scan.

Parameters

- `base` – CADC peripheral base address.
- `bEnable` – Enable/Disable auto-powerdown mode.
 - **true** Enable auto-powerdown mode, so when the module is not in use, it will auto-powerdown.
 - **false** Disable auto-powerdown mode, so when the module is not in use, the power will still on.

```
static inline void CADC_SetDMATriggerSource(ADC_Type *base, cadc_dma_trigger_source_t eDMATriggerSource)
```

Sets DMA trigger source(available selections are “End of scan” and “Sample Ready”).

Parameters

- `base` – CADC peripheral base address.
- `eDMATriggerSource` – DMA trigger source. Please refer to `cadc_dma_trigger_source_t` for details.

```
static inline void CADC_EnableInterrupts(ADC_Type *base, uint32_t u32Mask)
```

Enables the interrupts(such as high/low limit interrupts, end of scan interrupts, and so on).

Parameters

- `base` – CADC peripheral base address.
- `u32Mask` – Mask value for converters interrupt events. Should be the OR'ed value of `_cadc_interrupt_enable`.

```
static inline void CADC_DisableInterrupts(ADC_Type *base, uint32_t u32Mask)
```

Disables the interrupts(such as high/low limit interrupts, end of scan interrupts, and so on).

Parameters

- `base` – CADC peripheral base address.
- `u32Mask` – Mask value for converts interrupt events. Should be the OR'ed value of `_cadc_interrupt_enable`.

```
static inline uint16_t CADC_GetMiscStatusFlags(ADC_Type *base)
```

Gets Miscellaneous status flags, such as end of scan status flag, high/low limit interrupt flags and so on.

Note: This API will return the current status of the ADC module, including high limit interrupt status, low limit status flag, zero crossing interrupt status, End of scan interrupt status, conversion in progress status. But some status flags are not included in this function. To get sample slot ready status flag,

please invoking `CADC_GetSampleSlotReadyStatusFlags()`, to get sample slot limit violations status please invoking `CADC_ClearSampleSlotLowLimitStatusFlags()` and `CADC_GetSampleSlotHighLimitStatusFlags()`, to get zerocrossing status please invoking `CADC_GetSampleSlotZeroCrossingStatusFlags()`. To get converters' power status please invoke `CADC_GetPowerStatusFlag()`.

Parameters

- `base` – ADC peripheral base address.

Returns

Mask value for the event flags. See `_adc_misc_status_flags`.

```
static inline void CADC_ClearMiscStatusFlags(ADC_Type *base, uint16_t u16Flags)
```

Clears Miscellaneous status flags(Only for “end of scan” status flags).

Note: Only `kADC_ConverterAEndOfScanFlag` and `kADC_ConverterBEndOfScanFlag` can be cleared. And sample slot related status flags can not be cleared in this function. To clear the status flags of limit violations, please invoking `CADC_ClearSampleSlotLowLimitStatusFlags()` and `CADC_ClearSampleSlotHighLimitStatusFlags()`, to clear the status flags of zero crossing mode, please invoking `CADC_ClearSampleSlotZeroCrossingStatusFlags()`.

Parameters

- `base` – ADC peripheral base address.
- `u16Flags` – Mask value for the event flags to be cleared. See `_adc_misc_status_flags`. Only the enumeration `kADC_ConverterAEndOfScanFlag` and `kADC_ConverterBEndOfScanFlag` are useful.

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

Gets sample slots ready status flag, those status flags are cleared by reading the corresponding sample slots' result.

Parameters

- `base` – ADC peripheral base address.

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

Gets sample slot low limit status flags(Each bit represents one sample slot).

Parameters

- `base` – ADC peripheral base address.

Returns

The value of all sample slots' low limit status. Each bit represents one sample slot.

```
static inline void CADC_ClearSampleSlotLowLimitStatusFlags(ADC_Type *base, uint32_t u32SampleMask)
```

Clears sample slot's low limit status flags(Each bit represents one sample slot).

Parameters

- `base` – ADC peripheral base address.
- `u32SampleMask` – Mask value of sample slots. This parameter should be the OR'ed value of `adc_sample_slot_mask_t`.

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

Gets sample slot high limit status flags(Each bit represents one sample slot).

Parameters

- base – ADC peripheral base address.

Returns

The value of all sample slots' high limit status. Each bit represents each sample slot.

```
static inline void ADC_ClearSampleSlotHighLimitStatusFlags(ADC_Type *base, uint32_t
                                                         u32SampleMask)
```

Clears sample slot's high limit status flags(Each bit represents one sample slot).

Parameters

- base – ADC peripheral base address.
- u32SampleMask – Mask value of sample slots. This parameter should be the OR'ed value of `cadc_sample_slot_mask_t`.

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

Gets sample slot zero crossing status flags(Each bit represents one sample slot).

Parameters

- base – ADC peripheral base address.

Returns

The value of all sample slots' zero crossing status. Each bit represents each sample slot.

```
static inline void ADC_ClearSampleSlotZeroCrossingStatusFlags(ADC_Type *base, uint32_t
                                                            u32SampleMask)
```

Clears sample slot's zero crossing status flags(Each bit represents one sample slot).

Parameters

- base – ADC peripheral base address.
- u32SampleMask – Mask value of sample slots. This parameter should be the OR'ed value of `cadc_sample_slot_mask_t`.

```
static inline uint16_t ADC_GetPowerStatusFlags(ADC_Type *base)
```

Gets converters power status(Those power status can not be cleared).

Parameters

- base – ADC peripheral base address.

Returns

The mask value of the converters' power status flag, see `_cadc_converter_power_status_flags`.

```
static inline uint16_t ADC_GetConverterExpMuxChannelScanCompStatusFlags(ADC_Type *base)
```

Gets converter's expansion mux channel scan complete status flags.

Parameters

- base – ADC peripheral base address.

Returns

uint16_t The mask value of converters' expansion mux channel scan status flags, see `_cadc_expansion_mux_status_flags`.

```
static inline void CADC_ClearConverterExpMuxChannelScanCompStatusFlags(ADC_Type *base,
                                                                    uint16_t
                                                                    u16FlagMask)
```

Clears converter's expansion mux channel scan complete status flags.

Parameters

- base – ADC peripheral base address.
- u16FlagMask – The mask value of `_cadc_expansion_mux_status_flags`.

```
FSL_CADC_DRIVER_VERSION
```

CADC driver version.

```
enum _cadc_misc_status_flags
```

CADC miscellaneous status flags used to tell peripheral's miscellaneous status, such as zero-crossing, end of scan flags.

Values:

```
enumerator kCADC_ZeroCrossingInterruptFlag
```

Zero crossing encountered. IRQ pending if enabled Zero Crossing Interrupt.

```
enumerator kCADC_HighLimitInterruptFlag
```

High limit exceeded flag. IRQ pending if enabled high limit interrupt.

```
enumerator kCADC_LowLimitInterruptFlag
```

Low limit exceeded flag. IRQ pending if enabled low limit interrupt.

```
enumerator kCADC_ConverterAInProgressFlag
```

Conversion in progress, converter A.

```
enumerator kCADC_ConverterBInProgressFlag
```

Conversion in progress, converter B.

```
enumerator kCADC_ConverterAEndOfScanFlag
```

End of scan, converter A.

```
enumerator kCADC_ConverterBEndOfScanFlag
```

End of scan, converter B.

```
enumerator kCADC_StatusAllFlags
```

```
enum _cadc_converter_power_status_flags
```

The enumeration of converter power status.

Values:

```
enumerator kCADC_ConverterAPowerDownFlag
```

The converterA is powered down.

```
enumerator kCADC_ConverterBPowerDownFlag
```

The converterB is powered down.

```
enum _cadc_expansion_mux_status_flags
```

The enumeration of expansion mux channel scan complete interrupt request status flag.

Values:

```
enumerator kCADC_ANA4ExpMuxAuxScanCompInterruptFlag
```

ANA4 Expansion MUX Channel Scan Complete Interrupt flag.

```
enumerator kCADC_ANB4ExpMuxAuxScanCompInterruptFlag
```

ANB4 Expansion MUX Channel Scan Complete Interrupt flag.

enum _cadc_interrupt_enable

CADC Interrupts enumeration.

Values:

enumerator kCADC_Sample0ScanInterruptEnable

If sample0 is converted, generate the scan interrupt.

enumerator kCADC_Sample1ScanInterruptEnable

If sample1 is converted, generate the scan interrupt.

enumerator kCADC_Sample2ScanInterruptEnable

If sample2 is converted, generate the scan interrupt.

enumerator kCADC_Sample3ScanInterruptEnable

If sample3 is converted, generate the scan interrupt.

enumerator kCADC_Sample4ScanInterruptEnable

If sample4 is converted, generate the scan interrupt.

enumerator kCADC_Sample5ScanInterruptEnable

If sample5 is converted, generate the scan interrupt.

enumerator kCADC_Sample6ScanInterruptEnable

If sample6 is converted, generate the scan interrupt.

enumerator kCADC_Sample7ScanInterruptEnable

If sample7 is converted, generate the scan interrupt.

enumerator kCADC_Sample8ScanInterruptEnable

If sample8 is converted, generate the scan interrupt.

enumerator kCADC_Sample9ScanInterruptEnable

If sample9 is converted, generate the scan interrupt.

enumerator kCADC_Sample10ScanInterruptEnable

If sample10 is converted, generate the scan interrupt.

enumerator kCADC_Sample11ScanInterruptEnable

If sample11 is converted, generate the scan interrupt.

enumerator kCADC_Sample12ScanInterruptEnable

If sample12 is converted, generate the scan interrupt.

enumerator kCADC_Sample13ScanInterruptEnable

If sample13 is converted, generate the scan interrupt.

enumerator kCADC_Sample14ScanInterruptEnable

If sample14 is converted, generate the scan interrupt.

enumerator kCADC_Sample15ScanInterruptEnable

If sample15 is converted, generate the scan interrupt.

enumerator kCADC_ANA4ExpMuxScanCompleteInterruptEnable

If ANA4 expansion MUX channel scan complete, generate the interrupt.

enumerator kCADC_ANB4ExpMuxScanCompleteInterruptEnable

If ANB4 expansion MUX channel scan complete, generate the interrupt.

enumerator kCADC_HighLimitInterruptEnable

If the result value is greater than the high limit value, generate high limit interrupt.

enumerator kCADC_LowLimitInterruptEnable

If the result value is less than the low limit value, generate low limit interrupt.

enumerator kCADC_ZeroCrossingInterruptEnable

If the current value has a sign change from the previous result in the selected zero crossing mode, generate the zero crossing mode

enumerator kCADC_ConversionCompleteInterrupt0Enable

Upon the completion of the scan, generate the end of scan interrupt, when the scan mode is selected as sequential mode or simultaneous parallel mode. For looping scan mode, the interrupt will trigger after the completion of each iteration of loop.

enumerator kCADC_ConversionCompleteInterrupt1Enable

When the scan mode is independent parallel mode, up the completion of the converter scan, generate te end of scan interrupt. For looping scan mode, the interrupt will trigger after the completion of each iteration of loop.

enumerator kCADC_ALLInterruptEnable

enum _cadc_converter_id

CADC Converter identifier.

Values:

enumerator kCADC_ConverterA

Converter A.

enumerator kCADC_ConverterB

Converter B.

enum _cadc_idle_work_mode

The enumeration of work mode when the module is not used.

Values:

enumerator kCADC_IdleKeepNormal

Keep normal.

enumerator kCADC_IdleAutoPowerDown

Fall into power down mode automatically.

enum _cadc_dma_trigger_source

The enumeration of DMA trigger source.

Values:

enumerator kCADC_DMATrigSrcEndofScan

DMA trigger source is end of scan interrupt.

enumerator kCADC_DMATrigSrcSampleReady

DMA trigger source is RDY bits.

enum _cadc_scan_mode

The enumeration of dual converter's scan mode.

Values:

enumerator kCADC_ScanModeOnceSequential

Once (single) sequential.

enumerator kCADC_ScanModeOnceParallelIndependent

Once parallel independently.

enumerator kCADC_ScanModeLoopSequential

Loop sequential.

enumerator kCADC_ScanModeLoopParallelIndependent

Loop parallel independently.

enumerator kCADC_ScanModeTriggeredSequential

Triggered sequential.

enumerator kCADC_ScanModeTriggeredParallelIndependent

Triggered parallel independently.

enumerator kCADC_ScanModeOnceParallelSimultaneous

Once parallel simultaneously.

enumerator kCADC_ScanModeLoopParallelSimultaneous

Loop parallel simultaneously.

enumerator kCADC_ScanModeTriggeredParallelSimultaneous

Triggered parallel simultaneously.

enum _cadc_reference_voltage_source

The enumeration of converter's reference voltage source.

Values:

enumerator kCADC_ReferenceVoltageVrefPad

VREF pin.

enumerator kCADC_ReferenceVoltageChannelPad

ANx2 or ANx3 pin.

enum _cadc_channel_gain

The enumeration of sample slot connected channel gain.

Values:

enumerator kCADC_SignalGainX1

Gain x1.

enumerator kCADC_SignalGainX2

Gain x2.

enumerator kCADC_SignalGainX4

Gain x4.

enum _cadc_channel_mode

The enumeration of all channels' channel mode.

Values:

enumerator kCADC_ANA0_1_SingleEnd

ANA0 and ANA1 both configured as single ended inputs.

enumerator kCADC_ANA0_1_FullyDifferential

ANA0 configured as fully differential positive input, ANA1 configured as fully differential negative input.

enumerator kCADC_ANA0_1_UnipolarDifferential

ANA0 configured as unipolar differential positive input, ANA1 configured as unipolar differential negative input.

enumerator kCADC_ANA2_3_SingleEnd

ANA2 and ANA3 both configured as single ended inputs.

enumerator kCADC_ANA2_3_FullyDifferential

ANA2 configured as fully differential positive input, ANA3 configured as fully differential negative input.

enumerator kCADC_ANA2_3_UnipolarDifferential

ANA2 configured as unipolar differential positive input, ANA3 configured as unipolar differential negative input.

enumerator kCADC_ANB0_1_SingleEnd

ANB0 and ANB1 both configured as single ended inputs.

enumerator kCADC_ANB0_1_FullyDifferential

ANB0 configured as fully differential positive input, ANB1 configured as fully differential negative input.

enumerator kCADC_ANB0_1_UnipolarDifferential

ANB0 configured as unipolar differential positive input, ANB1 configured as unipolar differential negative input.

enumerator kCADC_ANB2_3_SingleEnd

ANB2 and ANB3 both configured as single ended inputs.

enumerator kCADC_ANB2_3_FullyDifferential

ANB2 configured as fully differential positive input, ANB3 configured as fully differential negative input.

enumerator kCADC_ANB2_3_UnipolarDifferential

ANB2 configured as unipolar differential positive input, ANB3 configured as unipolar differential negative input.

enumerator kCADC_ANA4_5_SingleEnd

ANA4 and ANA5 both configured as single ended inputs.

enumerator kCADC_ANA4_5_FullyDifferential

ANA4 configured as fully differential positive input, ANA5 configured as fully differential negative input.

enumerator kCADC_ANA4_5_UnipolarDifferential

ANA4 configured as unipolar differential positive input, ANA5 configured as unipolar differential negative input.

enumerator kCADC_ANA6_7_SingleEnd

ANA6 and ANA7 both configured as single ended inputs.

enumerator kCADC_ANA6_7_FullyDifferential

ANA6 configured as fully differential positive input, ANA7 configured as fully differential negative input.

enumerator kCADC_ANA6_7_UnipolarDifferential

ANA6 configured as unipolar differential positive input, ANA7 configured as unipolar differential negative input.

enumerator kCADC_ANB4_5_SingleEnd

ANB4 and ANB5 both configured as single ended inputs.

enumerator kCADC_ANB4_5_FullyDifferential

ANB4 configured as fully differential positive input, ANB5 configured as fully differential negative input.

enumerator kCADC_ANB4_5_UnipolarDifferential

ANB4 configured as unipolar differential positive input, ANB5 configured as unipolar differential negative input.

enumerator kCADC_ANB6_7_SingleEnd

ANB6 and ANB7 both configured as single ended inputs.

enumerator kCADC_ANB6_7_FullyDifferential

ANB6 configured as fully differential positive input, ANB7 configured as fully differential negative input.

enumerator kCADC_ANB6_7_UnipolarDifferential

ANB6 configured as unipolar differential positive input, ANB7 configured as unipolar differential negative input.

enum _cadc_channel_number

The enumerator of all channels that can be routed to the specific sample slot.

Values:

enumerator kCADC_SingleEndANA0_DiffANA0pANA1n

Single Ended ANA0 Signal Or Differential ANA0+, ANA1- signal.

enumerator kCADC_SingleEndANA1_DiffANA0pANA1n

Single Ended ANA1 Signal Or Differential ANA0+, ANA1- signal.

enumerator kCADC_SingleEndANA2_DiffANA2pANA3n

Single Ended ANA2 Signal Or Differential ANA2+, ANA3- signal.

enumerator kCADC_SingleEndANA3_DiffANA2pANA3n

Single Ended ANA3 Signal Or Differential ANA2+, ANA3- signal.

enumerator kCADC_SingleEndANA4_DiffANA4pANA5n

Single Ended ANA4 Signal Or Differential ANA4+, ANA5- signal.

enumerator kCADC_SingleEndANA5_DiffANA4pANA5n

Single Ended ANA5 Signal Or Differential ANA4+, ANA5- signal.

enumerator kCADC_SingleEndANA6_DiffANA6pANA7n

Single Ended ANA6 Signal Or Differential ANA6+, ANA7- signal.

enumerator kCADC_SingleEndANA7_DiffANA6pANA7n

Single Ended ANA7 Signal Or Differential ANA6+, ANA7- signal.

enumerator kCADC_SingleEndANB0_DiffANB0pANB1n

Single Ended ANB0 Signal Or Differential ANB0+, ANB1- signal.

enumerator kCADC_SingleEndANB1_DiffANB0pANB1n

Single Ended ANB1 Signal Or Differential ANB0+, ANB1- signal.

enumerator kCADC_SingleEndANB2_DiffANB2pANB3n

Single Ended ANB2 Signal Or Differential ANB2+, ANB3- signal.

enumerator kCADC_SingleEndANB3_DiffANB2pANB3n

Single Ended ANB3 Signal Or Differential ANB2+, ANB3- signal.

enumerator kCADC_SingleEndANB4_DiffANB4pANB5n

Single Ended ANB4 Signal Or Differential ANB4+, ANB5- signal.

enumerator kCADC_SingleEndANB5_DiffANB4pANB5n

Single Ended ANB5 Signal Or Differential ANB4+, ANB5- signal.

enumerator kCADC_SingleEndANB6_DiffANB6pANB7n

Single Ended ANB6 Signal Or Differential ANB6+, ANB7- signal.

enumerator kCADC_SingleEndANB7_DiffANB6pANB7n

Single Ended ANB7 Signal Or Differential ANB6+, ANB7- signal.

enum `_cadc_sample_slot_mask`

The enumeration of sample slot mask.

Values:

enumerator `kCADC_NonSampleSlotMask`

enumerator `kCADC_SampleSlot0Mask`

The mask of sample slot0.

enumerator `kCADC_SampleSlot1Mask`

The mask of sample slot1.

enumerator `kCADC_SampleSlot2Mask`

The mask of sample slot2.

enumerator `kCADC_SampleSlot3Mask`

The mask of sample slot3.

enumerator `kCADC_SampleSlot4Mask`

The mask of sample slot4.

enumerator `kCADC_SampleSlot5Mask`

The mask of sample slot5.

enumerator `kCADC_SampleSlot6Mask`

The mask of sample slot6.

enumerator `kCADC_SampleSlot7Mask`

The mask of sample slot7.

enumerator `kCADC_SampleSlot8Mask`

The mask of sample slot8.

enumerator `kCADC_SampleSlot9Mask`

The mask of sample slot9.

enumerator `kCADC_SampleSlot10Mask`

The mask of sample slot10.

enumerator `kCADC_SampleSlot11Mask`

The mask of sample slot11.

enumerator `kCADC_SampleSlot12Mask`

The mask of sample slot12.

enumerator `kCADC_SampleSlot13Mask`

The mask of sample slot13.

enumerator `kCADC_SampleSlot14Mask`

The mask of sample slot14.

enumerator `kCADC_SampleSlot15Mask`

The mask of sample slot15.

enumerator `kCADC_AllSampleSlotsMask`

The mask of all sample slots.

enum `_cadc_sample_slot_index`

The enumeration of sample slot index.

Values:

enumerator kCADC_SampleSlot0Index
The index of sample slot0.

enumerator kCADC_SampleSlot1Index
The index of sample slot1.

enumerator kCADC_SampleSlot2Index
The index of sample slot2.

enumerator kCADC_SampleSlot3Index
The index of sample slot3.

enumerator kCADC_SampleSlot4Index
The index of sample slot4.

enumerator kCADC_SampleSlot5Index
The index of sample slot5.

enumerator kCADC_SampleSlot6Index
The index of sample slot6.

enumerator kCADC_SampleSlot7Index
The index of sample slot7.

enumerator kCADC_SampleSlot8Index
The index of sample slot8.

enumerator kCADC_SampleSlot9Index
The index of sample slot9.

enumerator kCADC_SampleSlot10Index
The index of sample slot10.

enumerator kCADC_SampleSlot11Index
The index of sample slot11.

enumerator kCADC_SampleSlot12Index
The index of sample slot12.

enumerator kCADC_SampleSlot13Index
The index of sample slot13.

enumerator kCADC_SampleSlot14Index
The index of sample slot14.

enumerator kCADC_SampleSlot15Index
The index of sample slot15.

enum _cadc_sample_slot_sequential_mode_disabled
The enumeration for the sample slot to be disabled in sequential mode.

Values:

enumerator kCADC_Sample0Disabled
Disable Sample slot0, the scan will stop at sample slot0 in sequential scan mode

enumerator kCADC_Sample1Disabled
Disable Sample slot1, the scan will stop at sample slot1 in sequential scan mode

enumerator kCADC_Sample2Disabled
Disable Sample slot2, the scan will stop at sample slot2 in sequential scan mode

enumerator kCADC_Sample3Disabled

Disable Sample slot3, the scan will stop at sample slot3 in sequential scan mode

enumerator kCADC_Sample4Disabled

Disable Sample slot4, the scan will stop at sample slot4 in sequential scan mode

enumerator kCADC_Sample5Disabled

Disable Sample slot5, the scan will stop at sample slot5 in sequential scan mode

enumerator kCADC_Sample6Disabled

Disable Sample slot6, the scan will stop at sample slot6 in sequential scan mode

enumerator kCADC_Sample7Disabled

Disable Sample slot7, the scan will stop at sample slot7 in sequential scan mode

enumerator kCADC_Sample8Disabled

Disable Sample slot8, the scan will stop at sample slot8 in sequential scan mode

enumerator kCADC_Sample9Disabled

Disable Sample slot9, the scan will stop at sample slot9 in sequential scan mode

enumerator kCADC_Sample10Disabled

Disable Sample slot10, the scan will stop at sample slot10 in sequential scan mode

enumerator kCADC_Sample11Disabled

Disable Sample slot11, the scan will stop at sample slot11 in sequential scan mode

enumerator kCADC_Sample12Disabled

Disable Sample slot12, the scan will stop at sample slot12 in sequential scan mode

enumerator kCADC_Sample13Disabled

Disable Sample slot13, the scan will stop at sample slot13 in sequential scan mode

enumerator kCADC_Sample14Disabled

Disable Sample slot14, the scan will stop at sample slot14 in sequential scan mode

enumerator kCADC_Sample15Disabled

Disable Sample slot15, the scan will stop at sample slot15 in sequential scan mode

enum _cadc_sample_slot_simultParallel_mode_disabled

The enumeration for the sample slot to be disabled in simultaneous parallel mode.

Values:

enumerator kCADC_Sample0_8Disabled

Disable Sample slot0 and Sample Slot 8, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot0 and Sample slot 8.

enumerator kCADC_Sample1_9Disabled

Disable Sample slot1 and Sample Slot 9, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot1 and Sample slot 9.

enumerator kCADC_Sample2_10Disabled

Disable Sample slot2 and Sample Slot 10, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot2 and Sample slot 10.

enumerator kCADC_Sample3_11Disabled

Disable Sample slot3 and Sample Slot 11, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot3 and Sample slot 11.

enumerator kCADC_Sample4_12Disabled

Disable Sample slot4 and Sample Slot 12, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot4 and Sample slot 12.

enumerator kCADC_Sample5_13Disabled

Disable Sample slot5 and Sample Slot 13, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot5 and Sample slot 13.

enumerator kCADC_Sample6_14Disabled

Disable Sample slot6 and Sample Slot 14, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot7 and Sample slot 14.

enumerator kCADC_Sample7_15Disabled

Disable Sample slot7 and Sample Slot 15, in the simultaneous parallel mode the converter A and converter B will stop at Sample slot7 and Sample slot 15.

enum _cadc_sample_slot_independentParallel_mode_convA_disabled

The enumeration for the sample slot to be disabled for the converter A in independent parallel mode.

Values:

enumerator kCADC_ConvASample0Disabled

Disable Sample slot0, the scan will stop at sample slot0 in sequential scan mode

enumerator kCADC_ConvASample1Disabled

Disable Sample slot1, the scan will stop at sample slot1 in sequential scan mode

enumerator kCADC_ConvASample2Disabled

Disable Sample slot2, the scan will stop at sample slot2 in sequential scan mode

enumerator kCADC_ConvASample3Disabled

Disable Sample slot3, the scan will stop at sample slot3 in sequential scan mode

enumerator kCADC_ConvASample4Disabled

Disable Sample slot4, the scan will stop at sample slot4 in sequential scan mode

enumerator kCADC_ConvASample5Disabled

Disable Sample slot5, the scan will stop at sample slot5 in sequential scan mode

enumerator kCADC_ConvASample6Disabled

Disable Sample slot6, the scan will stop at sample slot6 in sequential scan mode

enumerator kCADC_ConvASample7Disabled

Disable Sample slot7, the scan will stop at sample slot7 in sequential scan mode

enumerator kCADC_ConvASampleReserved

Reserved

enum _cadc_sample_slot_indParallel_mode_convB_disabled

The enumeration for the sample slot to be disabled for the converter B in independent parallel mode.

Values:

enumerator kCADC_ConvBSample8Disabled

Disable Sample slot8, the scan will stop at sample slot8 in sequential scan mode

enumerator kCADC_ConvBSample9Disabled

Disable Sample slot9, the scan will stop at sample slot9 in sequential scan mode

enumerator kCADC_ConvBSample10Disabled

Disable Sample slot10, the scan will stop at sample slot10 in sequential scan mode

enumerator kCADC_ConvBSample11Disabled

Disable Sample slot11, the scan will stop at sample slot11 in sequential scan mode

enumerator kCADC_ConvBSample12Disabled

Disable Sample slot12, the scan will stop at sample slot12 in sequential scan mode

enumerator kCADC_ConvBSample13Disabled

Disable Sample slot13, the scan will stop at sample slot13 in sequential scan mode

enumerator kCADC_ConvBSample14Disabled

Disable Sample slot14, the scan will stop at sample slot14 in sequential scan mode

enumerator kCADC_ConvBSample15Disabled

Disable Sample slot15, the scan will stop at sample slot15 in sequential scan mode

enumerator kCADC_ConvBSampleReserved

Reserved

enum _cadc_sample_slot_zero_crossing_mode

The enumeration for the sample slot's zero crossing event.

Values:

enumerator kCADC_ZeroCrossingDisabled

Zero Crossing disabled.

enumerator kCADC_ZeroCrossingForPtoNSign

Zero Crossing enabled for positive to negative sign change.

enumerator kCADC_ZeroCrossingForNtoPSign

Zero Crossing enabled for negative to positive sign change.

enumerator kCADC_ZeroCrossingForAnySignChanged

Zero Crossing enabled for any sign change.

enum _cadc_expansion_mux_operate_mode

The enumeration for expansion multiplexer.

Values:

enumerator kCADC_ExpMuxManualMode

MUX channel feeding to ANA4/ANB4 is selected as MUXSEL0.

enumerator kCADC_ExpMuxScanMode0

The sample completion of ANA4/ANB4 enables subsequent selected channel.

enumerator kCADC_ExpMuxScanMode1

The sample completion of ANA7/ANB7 enables subsequent selected channel.

enumerator kCADC_ExpMuxScanMode2

The sample completion of ANA4/ANB4 or ANA7/ANB7 enables subsequent selected channel.

enum _cadc_auxiliary_control

The enumeration of converter's auxiliary control.

Values:

enumerator kCADC_AuxSel0_Config0

Auxiliary select 0 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel0_Config1

Auxiliary select 0 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel0_Config2

Auxiliary select 0 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel0_Config3
Auxiliary select 0 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel1_Config0
Auxiliary select 1 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel1_Config1
Auxiliary select 1 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel1_Config2
Auxiliary select 1 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel1_Config3
Auxiliary select 1 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel2_Config0
Auxiliary select 2 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel2_Config1
Auxiliary select 2 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel2_Config2
Auxiliary select 2 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel2_Config3
Auxiliary select 2 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel3_Config0
Auxiliary select 3 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel3_Config1
Auxiliary select 3 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel3_Config2
Auxiliary select 3 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel3_Config3
Auxiliary select 3 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel4_Config0
Auxiliary select 4 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel4_Config1
Auxiliary select 4 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel4_Config2
Auxiliary select 4 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel4_Config3
Auxiliary select 4 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel5_Config0
Auxiliary select 5 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel5_Config1
Auxiliary select 5 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel5_Config2
Auxiliary select 5 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel5_Config3
Auxiliary select 5 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel6_Config0
Auxiliary select 6 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel6_Config1
Auxiliary select 6 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel6_Config2
Auxiliary select 6 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel6_Config3
Auxiliary select 6 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enumerator kCADC_AuxSel7_Config0
Auxiliary select 7 controls AUX_SEL0 = 0, AUX_SEL1 = 0.

enumerator kCADC_AuxSel7_Config1
Auxiliary select 7 controls AUX_SEL0 = 1, AUX_SEL1 = 0.

enumerator kCADC_AuxSel7_Config2
Auxiliary select 7 controls AUX_SEL0 = 0, AUX_SEL1 = 1.

enumerator kCADC_AuxSel7_Config3
Auxiliary select 7 controls AUX_SEL0 = 1, AUX_SEL1 = 1.

enum _cadc_expansion_disabled_mux_slot
The enumeration for the expansion mux slot to be disabled.

Values:

enumerator kCADC_ExpMuxNoDisable
Expansion mux scan not disabled.

enumerator kCADC_ExpMux0Disable
Expansion mux slot 0.

enumerator kCADC_ExpMux1Disable
Expansion mux slot 1.

enumerator kCADC_ExpMux2Disable
Expansion mux slot 2.

enumerator kCADC_ExpMux3Disable
Expansion mux slot 3.

enumerator kCADC_ExpMux4Disable
Expansion mux slot 4.

enumerator kCADC_ExpMux5Disable
Expansion mux slot 5.

enumerator kCADC_ExpMux6Disable
Expansion mux slot 6.

enumerator kCADC_ExpMux7Disable
Expansion mux slot 7.

enum _cadc_expansion_mux_selection
The enumeration of expansion mux selection.

Values:

enumerator kCADC_MuxSel0_Channel0
MUX's channel 0 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel1
MUX's channel 1 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel2
MUX's channel 2 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel3
MUX's channel 3 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel4
MUX's channel 4 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel5
MUX's channel 5 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel6
MUX's channel 6 for MUXSEL0.

enumerator kCADC_MuxSel0_Channel7
MUX's channel 7 for MUXSEL0.

enumerator kCADC_MuxSel1_Channel0
MUX's channel 0 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel1
MUX's channel 1 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel2
MUX's channel 2 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel3
MUX's channel 3 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel4
MUX's channel 4 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel5
MUX's channel 5 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel6
MUX's channel 6 for MUXSEL1.

enumerator kCADC_MuxSel1_Channel7
MUX's channel 7 for MUXSEL1.

enumerator kCADC_MuxSel2_Channel0
MUX's channel 0 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel1
MUX's channel 1 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel2
MUX's channel 2 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel3
MUX's channel 3 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel4
MUX's channel 4 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel5
MUX's channel 5 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel6
MUX's channel 6 for MUXSEL2.

enumerator kCADC_MuxSel2_Channel7
MUX's channel 7 for MUXSEL2.

enumerator kCADC_MuxSel3_Channel0
MUX's channel 0 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel1
MUX's channel 1 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel2
MUX's channel 2 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel3
MUX's channel 3 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel4
MUX's channel 4 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel5
MUX's channel 5 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel6
MUX's channel 6 for MUXSEL3.

enumerator kCADC_MuxSel3_Channel7
MUX's channel 7 for MUXSEL3.

enumerator kCADC_MuxSel4_Channel0
MUX's channel 0 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel1
MUX's channel 1 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel2
MUX's channel 2 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel3
MUX's channel 3 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel4
MUX's channel 4 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel5
MUX's channel 5 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel6
MUX's channel 6 for MUXSEL4.

enumerator kCADC_MuxSel4_Channel7
MUX's channel 7 for MUXSEL4.

enumerator kCADC_MuxSel5_Channel0
MUX's channel 0 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel1
MUX's channel 1 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel2
MUX's channel 2 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel3
MUX's channel 3 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel4
MUX's channel 4 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel5
MUX's channel 5 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel6
MUX's channel 6 for MUXSEL5.

enumerator kCADC_MuxSel5_Channel7
MUX's channel 7 for MUXSEL5.

enumerator kCADC_MuxSel6_Channel0
MUX's channel 0 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel1
MUX's channel 1 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel2
MUX's channel 2 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel3
MUX's channel 3 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel4
MUX's channel 4 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel5
MUX's channel 5 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel6
MUX's channel 6 for MUXSEL6.

enumerator kCADC_MuxSel6_Channel7
MUX's channel 7 for MUXSEL6.

enumerator kCADC_MuxSel7_Channel0
MUX's channel 0 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel1
MUX's channel 1 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel2
MUX's channel 2 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel3
MUX's channel 3 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel4
MUX's channel 4 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel5
MUX's channel 5 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel6
MUX's channel 6 for MUXSEL7.

enumerator kCADC_MuxSel7_Channel7
MUX's channel 7 for MUXSEL7.

`typedef enum _cadc_converter_id cadc_converter_id_t`

CADC Converter identifier.

`typedef enum _cadc_idle_work_mode cadc_idle_work_mode_t`

The enumeration of work mode when the module is not used.

`typedef enum _cadc_dma_trigger_source cadc_dma_trigger_source_t`

The enumeration of DMA trigger source.

`typedef enum _cadc_scan_mode cadc_scan_mode_t`

The enumeration of dual converter's scan mode.

`typedef enum _cadc_reference_voltage_source cadc_reference_voltage_source_t`

The enumeration of converter's reference voltage source.

`typedef enum _cadc_channel_gain cadc_channel_gain_t`

The enumeration of sample slot connected channel gain.

`typedef enum _cadc_channel_mode cadc_channel_mode_t`

The enumeration of all channels' channel mode.

`typedef enum _cadc_channel_number cadc_channel_number_t`

The enumerator of all channels that can be routed to the specific sample slot.

`typedef enum _cadc_sample_slot_mask cadc_sample_slot_mask_t`

The enumeration of sample slot mask.

`typedef enum _cadc_sample_slot_index cadc_sample_slot_index_t`

The enumeration of sample slot index.

`typedef enum _cadc_sample_slot_sequential_mode_disabled`

`cadc_sample_slot_sequential_mode_disabled_t`

The enumeration for the sample slot to be disabled in sequential mode.

`typedef enum _cadc_sample_slot_simultParallel_mode_disabled`

`cadc_sample_slot_simultParallel_mode_disabled_t`

The enumeration for the sample slot to be disabled in simultaneous parallel mode.

`typedef enum _cadc_sample_slot_independentParallel_mode_convA_disabled`

`cadc_sample_slot_independentParallel_mode_convA_disabled_t`

The enumeration for the sample slot to be disabled for the converter A in independent parallel mode.

`typedef enum _cadc_sample_slot_indParallel_mode_convB_disabled`

`cadc_sample_slot_independentParallel_mode_convB_disabled_t`

The enumeration for the sample slot to be disabled for the converter B in independent parallel mode.

`typedef enum _cadc_sample_slot_zero_crossing_mode cadc_sample_slot_zero_crossing_mode_t`

The enumeration for the sample slot's zero crossing event.

`typedef enum _cadc_expansion_mux_operate_mode cadc_expansion_mux_operate_mode_t`

The enumeration for expansion multiplexer.

`typedef struct _cadc_sample_slot_independentParallel_mode_disabled`

`cadc_sample_slot_independentParallel_mode_disabled_t`

The structure of the disabled sample slots in independent parallel mode.

`typedef union _cadc_sample_slot_disabled cadc_sample_slot_disabled_t`

The union of disabled sample slot for each scan mode.

```
typedef struct _cadc_sample_config cadc_sample_slot_config_t
```

The structure for configuring the sample slot.

```
typedef struct _cadc_scan_ctrl_sequential_mode cadc_scan_ctrl_seq_mode_t
```

Cadc scan control for sequential scan mode.

Note: Each member of this structure represent one bit of the word. Asserted the structure's member means delay sample until a new sync input occurs. Cleared the structure's member means perform sample immediately after the completion of the current sample.

```
typedef struct _cadc_scan_ctrl_simultParallel_mode cadc_scan_ctrl_simultParallel_mode_t
```

Cadc scan control for simultaneous parallel scan mode.

Note: Each member of this structure represent one bit of the word. Asserted the structure's member means delay sample until a new sync input occurs. Cleared the structure's member means perform sample immediately after the completion of the current sample.

```
typedef struct _cadc_scan_ctrl_independent_parallel_mode_converterA
```

```
cadc_scan_ctrl_independent_parallel_mode_converterA_t
```

The scan ctrl struture for converterA in independent scan mode.

```
typedef struct _cadc_scan_ctrl_independent_parallel_mode_converterB
```

```
cadc_scan_ctrl_independent_parallel_mode_converterB_t
```

The scan ctrl struture for converterB in independent scan mode.

```
typedef union _cadc_scan_ctrl_independent_parallel_mode
```

```
cadc_scan_ctrl_independent_parallel_mode_t
```

The union for converters in independent parallel mode.

```
typedef union _cadc_scan_control cadc_scan_control_t
```

The union of the scan control for each scan mode.

```
typedef enum _cadc_auxiliary_control cadc_auxiliary_control_t
```

The enumeration of conveter's auxiliary control.

```
typedef enum _cadc_expansion_disabled_mux_slot cadc_expansion_disabled_mux_slot_t
```

The enumeration for the expansion mux slot to be disabled.

```
typedef enum _cadc_expansion_mux_selection cadc_expansion_mux_selection_t
```

The enumeration of expansion mux selection.

```
typedef struct _cadc_exp_mux_aux_config cadc_exp_mux_aux_config_t
```

The structure for configuring Cyclic ADC's expansion setting.

```
typedef struct _cadc_converter_config cadc_converter_config_t
```

The structure for configuring each converter.

```
typedef struct _cadc_config cadc_config_t
```

The structure for configuring the Cyclic ADC's setting.

```
CADC_SAMPLE_SLOTS_COUNT
```

Macro for CADC sample slot count.

```
struct _cadc_sample_slot_independentParallel_mode_disabled
```

#include <fsl_cadc.h> The structure of the disabled sample slots in independent parallel mode.

Public Members

cadc_sample_slot_independentParallel_mode_convA_disabled_t eConverterA

The sample slot to be disabled for the converter A, when the scan mode is set as independent parallel mode.

cadc_sample_slot_independentParallel_mode_convB_disabled_t eConverterB

The sample slot to be disabled for the converter B, when the scan mode is set as independent parallel mode.

union *_cadc_sample_slot_disabled*

#include <fsl_cadc.h> The union of disabled sample slot for each scan mode.

Public Members

uint32_t *u32SampleDisVal*

The 32 bits width of disabled sample slot value. This member used to get the disabled sample slot which sets in different scan modes in word type. This member is not recommended to be used to set the disabled sample slot. This member is designed to be used in driver level only, the application should not use this member.

cadc_sample_slot_sequential_mode_disabled_t eSequentialModeDisSample

If the scan mode is selected as sequential mode, the application must use this member to set the disabled sample slot. This member is used to set disabled sample slot when the scan mode is selected as sequential mode. The scan will stop at the first disabled sample slot in that mode. So for the application, this member should be set as one sample slot index that the scan will stop.

cadc_sample_slot_simultParallel_mode_disabled_t eSimultParallelModeDisSample

In simultaneous parallel scan mode, the application must use this member to set the disabled sample slot. In that scan mode, the scan will stop when either converter encounters a disabled sample.

cadc_sample_slot_independentParallel_mode_disabled_t sIndependentParallelModeDisSample

In independent parallel scan mode, the application must use this member to set the disabled sample slot. In that scan mode, the converter will stop scan when it encounters a disabled sample slot. In this mode, the disabled sample slot for converterA and converterB may different.

struct *_cadc_sample_config*

#include <fsl_cadc.h> The structure for configuring the sample slot.

Public Members

cadc_sample_slot_zero_crossing_mode_t eZeroCrossingMode

Zero crossing mode.

uint16_t *u16HighLimitValue*

High limit value. Original value formation as hardware register, with 3-bits left shifted.

uint16_t *u16LowLimitValue*

Low limit value. Original value formation as hardware register, with 3-bits left shifted.

uint16_t *u16OffsetValue*

Offset value. Original value formation as hardware register, with 3-bits left shifted.

```
struct _cadc_scan_ctrl_sequential_mode
#include <fsl_cadc.h> Cadc scan control for sequential scan mode.
```

Note: Each member of this structure represent one bit of the word. Asserted the structure's member means delay sample until a new sync input occurs. Cleared the structure's member means perform sample immediately after the completion of the current sample.

Public Members

```
uint32_t bitSample0
    Control whether delay sample0 until a new sync input occurs.
uint32_t bitSample1
    Control whether delay sample1 until a new sync input occurs.
uint32_t bitSample2
    Control whether delay sample2 until a new sync input occurs.
uint32_t bitSample3
    Control whether delay sample3 until a new sync input occurs.
uint32_t bitSample4
    Control whether delay sample4 until a new sync input occurs.
uint32_t bitSample5
    Control whether delay sample5 until a new sync input occurs.
uint32_t bitSample6
    Control whether delay sample6 until a new sync input occurs.
uint32_t bitSample7
    Control whether delay sample7 until a new sync input occurs.
uint32_t bitSample8
    Control whether delay sample8 until a new sync input occurs.
uint32_t bitSample9
    Control whether delay sample9 until a new sync input occurs.
uint32_t bitSample10
    Control whether delay sample10 until a new sync input occurs.
uint32_t bitSample11
    Control whether delay sample11 until a new sync input occurs.
uint32_t bitSample12
    Control whether delay sample12 until a new sync input occurs.
uint32_t bitSample13
    Control whether delay sample13 until a new sync input occurs.
uint32_t bitSample14
    Control whether delay sample14 until a new sync input occurs.
uint32_t bitSample15
    Control whether delay sample15 until a new sync input occurs.
uint32_t bitsReserved
    Reserved 16 bits.
```

```
struct _cadc_scan_ctrl_simultParallel_mode
#include <fsl_cadc.h> Cadc scan control for simultaneous parallel scan mode.
```

Note: Each member of this structure represent one bit of the word. Asserted the structure's member means delay sample until a new sync input occurs. Cleared the structure's member means perform sample immediately after the completion of the current sample.

Public Members

uint32_t bitSample0_8
Control whether delay sample0 and sample8 until a new sync input occurs.

uint32_t bitSample1_9
Control whether delay sample1 and sample9 until a new sync input occurs.

uint32_t bitSample2_10
Control whether delay sample2 and sample10 until a new sync input occurs.

uint32_t bitSample3_11
Control whether delay sample3 and sample11 until a new sync input occurs.

uint32_t bitsReserved1
Reserved 4 bits.

uint32_t bitSample4_12
Control whether delay sample4 and sample12 until a new sync input occurs.

uint32_t bitSample5_13
Control whether delay sample5 and sample13 until a new sync input occurs.

uint32_t bitSample6_14
Control whether delay sample6 and sample14 until a new sync input occurs.

uint32_t bitSample7_15
Control whether delay sample7 and sample15 until a new sync input occurs.

uint32_t bitsReserved2
Reserved 4 bits.

uint32_t bitsReserved3
Reserved 16 bits.

```
struct _cadc_scan_ctrl_independent_parallel_mode_converterA
#include <fsl_cadc.h> The scan ctrl struture for converterA in independent scan mode.
```

Public Members

uint32_t bitSample0
Control whether delay converterA's sample0 until a new sync input occurs.

uint32_t bitSample1
Control whether delay converterA's sample1 until a new sync input occurs.

uint32_t bitSample2
Control whether delay converterA's sample2 until a new sync input occurs.

uint32_t bitSample3
Control whether delay converterA's sample3 until a new sync input occurs.

uint32_t bitsReserved1

Reserved 4 bits.

uint32_t bitSample4

Control whether delay converterA's sample4 until a new sync input occurs.

uint32_t bitSample5

Control whether delay converterA's sample5 until a new sync input occurs.

uint32_t bitSample6

Control whether delay converterA's sample6 until a new sync input occurs.

uint32_t bitSample7

Control whether delay converterA's sample7 until a new sync input occurs.

uint32_t bitsReserved2

Reserved 4 bits

uint32_t bitsReserved3

Reserved 16 bits.

struct _cadc_scan_ctrl_independent_parallel_mode_converterB

#include <fsl_cadc.h> The scan ctrl structure for converterB in independent scan mode.

Public Members

uint32_t bitsReserved1

Reserved 4 bits.

uint32_t bitSample8

Control whether delay converterB's sample8 until a new sync input occurs.

uint32_t bitSample9

Control whether delay converterB's sample9 until a new sync input occurs.

uint32_t bitSample10

Control whether delay converterB's sample10 until a new sync input occurs.

uint32_t bitSample11

Control whether delay converterB's sample11 until a new sync input occurs.

uint32_t bitsReserved2

Reserved 4 bits.

uint32_t bitSample12

Control whether delay converterB's sample12 until a new sync input occurs.

uint32_t bitSample13

Control whether delay converterB's sample13 until a new sync input occurs.

uint32_t bitSample14

Control whether delay converterB's sample14 until a new sync input occurs.

uint32_t bitSample15

Control whether delay converterB's sample15 until a new sync input occurs.

uint32_t bitsReserved3

Reserved 16 bits.

union _cadc_scan_ctrl_independent_parallel_mode

#include <fsl_cadc.h> The union for converters in independent parallel mode.

Public Members

cadc_scan_ctrl_independent_parallel_mode_converterA_t sConverterA
Scan control for converterA.

cadc_scan_ctrl_independent_parallel_mode_converterB_t sConverterB
Scan control for converterB.

union *_cadc_scan_control*

#include <fsl_cadc.h> The union of the scan control for each scan mode.

Public Members

uint32_t u32ScanCtrlVal
The 32 bits value of the scan control value.

cadc_scan_ctrl_seq_mode_t sSequential
Scan control for sequential scan mode.

cadc_scan_ctrl_simultParallel_mode_t sSimultParallel
Scan control for simultaneous parallel scan mode.

cadc_scan_ctrl_independent_parallel_mode_t uIndependentParallel
Scan control for independent scan mode.

struct *_cadc_exp_mux_aux_config*

#include <fsl_cadc.h> The structure for configuring Cyclic ADC's expansion setting.

Public Members

uint16_t u16AuxControl
The mask of auxiliary control, should be the OR'ed value of *cadc_auxiliary_control_t*.

uint32_t u32MuxChannelMask
The mask of mux selection of all mux solts, should be the OR'ed value of *cadc_expansion_mux_selection_t*.

cadc_expansion_disabled_mux_slot_t disabledMuxSlot
mux slot to disabled in the scan.

struct *_cadc_converter_config*

#include <fsl_cadc.h> The structure for configuring each converter.

Public Members

uint16_t u16ClockDivisor
Converter's clock divisor for the clock source. Available setting range is 0-63.

- When the clockDivisor is 0, the divisor is 2.
- For all other clockDivisor values, the divisor is 1 more than the decimal value of clockDivisor: clockDivisor + 1

cadc_reference_voltage_source_t eHighReferenceVoltageSource
High voltage reference source.

cadc_reference_voltage_source_t eLowReferenceVoltageSource
Low reference voltage source.

`bool bEnableDMA`
Enable/Disable DMA.

`bool bPowerUp`
Power up or power down the converter.

`bool bScanInitBySync`
The member user to control the initiate method of the scan.

- **true** Use a SYNC input pulse or START command to initiate a scan.
- **false** Scan is initiated by the assertion of START command only.

`cadc_exp_mux_aux_config_t muxAuxConfig`
Configuration of expansion mux and auxiliary control.

`struct _cadc_config`
`#include <fsl_cadc.h>` The structure for configuring the Cyclic ADC's setting.

Public Members

`cadc_idle_work_mode_t eIdleWorkMode`
Idle work mode for the module.

`cadc_dma_trigger_source_t eDMATriggerSource`
Selects the dma trigger source for the module.

`uint16_t u16PowerUpDelay`
The number of ADC clocks to power up the converters (if powered up), before allowing a scan to start. The available range is 0 to 63 .

`uint32_t u32EnabledInterruptMask`
The mask of the interrupts to be enabled, should be the OR'ed value of `_cadc_interrupt_enable`.

`cadc_scan_mode_t eScanMode`
The scan mode of the module.

`cadc_sample_slot_disabled_t uDisabledSampleSlot`
The member used to config the which sample slot is disabled for the scan. The scan will continue until the first disabled sample slot is encountered.

`cadc_scan_control_t uScanControl`
Scan control provides the ability to pause and await a new sync signal while current sample completed.

`uint32_t u32ChannelModeMask`
The mask of each channel's mode, should be the OR'ed value of `cadc_channel_mode_t`. Each channel supports single-end and differential(Fully differentail and Unipolar differential). Some devices also support alternate source mode.

`cadc_channel_gain_t eChannelGain[(ADC_RSLT_COUNT)]`
The gain value for each channel. Each element of the array represents the gain of the channel. E.g. `eChannelGain[0]` means channel gain of channel0, which is ANA0.

`cadc_channel_number_t eSampleSlot[(ADC_RSLT_COUNT)]`
The channel assigned to each sample slot. The index of the array represents sample slot index.

cadc_converter_config_t sConverterA

The configuration for converterA.

cadc_converter_config_t sConverterB

The configuration for converterB.

2.2 The Driver Change Log

2.3 CADC Peripheral and Driver Overview

2.4 Clock Driver

enum *_clock_ip_name*

List of IP clock name.

Values:

enumerator kCLOCK_GPIOG
GPIOG clock

enumerator kCLOCK_GPIOF
GPIOF clock

enumerator kCLOCK_GPIOE
GPIOE clock

enumerator kCLOCK_GPIOD
GPIOD clock

enumerator kCLOCK_GPIOC
GPIOC clock

enumerator kCLOCK_GPIOB
GPIOB clock

enumerator kCLOCK_GPIOA
GPIOA clock

enumerator kCLOCK_TB3
Timer B3 clock

enumerator kCLOCK_TB2
Timer B2 clock

enumerator kCLOCK_TB1
Timer B1 clock

enumerator kCLOCK_TB0
Timer B0 clock

enumerator kCLOCK_TA3
Timer A3 clock

enumerator kCLOCK_TA2
Timer A2 clock

enumerator kCLOCK_TA1
Timer A1 clock

enumerator kCLOCK_TA0
Timer A0 clock

enumerator kCLOCK_FLEXCAN
Flex CAN clock

enumerator kCLOCK_I2C1
I2C1 clock

enumerator kCLOCK_I2C0
I2C0 clock

enumerator kCLOCK_QSPI1
QSPI1 clock

enumerator kCLOCK_QSPI0
QSPI0 clock

enumerator kCLOCK_QSCI2
QSCI2 clock

enumerator kCLOCK_QSCI1
QSCI1 clock

enumerator kCLOCK_QSCI0
QSCI0 clock

enumerator kCLOCK_DACA
DAC A clock

enumerator kCLOCK_DACB
DAC B clock

enumerator kCLOCK_PIT1
PIT 1 clock

enumerator kCLOCK_PIT0
PIT 0 clock

enumerator kCLOCK_CRC
CRC clock

enumerator kCLOCK_CYCADC
Cyclic ADC clock

enumerator kCLOCK_CMPD
Comparator D clock

enumerator kCLOCK_CMPC
Comparator C clock

enumerator kCLOCK_CMPB
Comparator B clock

enumerator kCLOCK_CMPA
Comparator A clock

enumerator kCLOCK_PWMBCH3
Enhanced Flexible PWM B3 clock

enumerator kCLOCK_PWMBCH2
Enhanced Flexible PWM B2 clock

enumerator kCLOCK_PWMBCH1
Enhanced Flexible PWM B1 clock

enumerator kCLOCK_PWMBCH0
Enhanced Flexible PWM B0 clock

enumerator kCLOCK_PWMACH3
Enhanced Flexible PWM A3 clock

enumerator kCLOCK_PWMACH2
Enhanced Flexible PWM A2 clock

enumerator kCLOCK_PWMACH1
Enhanced Flexible PWM A1 clock

enumerator kCLOCK_PWMACH0
Enhanced Flexible PWM A0 clock

enumerator kCLOCK_USB
USB clock

enumerator kCLOCK_ROM
ROM clock

enumerator kCLOCK_NOGATE
No clock gate for the IP

enumerator kCLOCK_EDMA

enumerator kCLOCK_EWM

enumerator kCLOCK_XBARA

enumerator kCLOCK_NUM
Total IP clock number

enum _clock_name

List of system-level clock name.

Values:

enumerator kCLOCK_Mstr2xCk
Master 2x clock which feed to core and peripheral

enumerator kCLOCK_SysClk
MCU system/core clock

enumerator kCLOCK_BusClk
Bus clock

enumerator kCLOCK_Bus2xCk
Bus 2x clock

enumerator kCLOCK_FlashClk
Flash clock

enumerator kCLOCK_FastIrcClk
Fast internal RC oscillator, 48M

enumerator kCLOCK_FastIrcDiv6Clk
ROSC 8M, derived from kCLOCK_FastIrcClk div 6

enumerator kCLOCK_SlowIrcClk
Slow internal RC oscillator, 200K

enumerator kCLOCK_CrystalOscClk
Crystal oscillator

enumerator kCLOCK_ExtClk
The selected external clock, it could be crystal oscillator, clkin0, clkin1

enumerator kCLOCK_MstrOscClk
The selected master oscillator clock

enumerator kCLOCK_PllDiv2Clk
PLL output divide 2

enum _clock_crystal_osc_mode
Crystal oscillator mode.

Values:

enumerator kCLOCK_CrystalOscModeFSP
Full swing pierce, high power mode

enumerator kCLOCK_CrystalOscModeLCP
Loop controlled pierce, low power mode

enum _clock_ext_clk_src
List of external clock source.

Values:

enumerator kCLOCK_ExtClkSrcCrystalOsc
External clock source is crystal oscillator

enumerator kCLOCK_ExtClkSrcClkin
External clock source is clock in

enum _clock_ext_clkin_sel
List of clock-in source.

Values:

enumerator kCLOCK_SelClkIn0
Clock in 0 is selected as CLKIN

enumerator kCLOCK_SelClkIn1
Clock in 1 is selected as CLKIN

enum _clock_mstr_osc_clk_src
List of master oscillator source.

Values:

enumerator kCLOCK_MstrOscClkSrcFireDiv6
8M, fast internal RC oscillator divide 6

enumerator kCLOCK_MstrOscClkSrcExt
External clock

enumerator kCLOCK_MstrOscClkSrcSirc
200K, slow internal RC oscillator

enumerator kCLOCK_MstrOscClkSrcFire
48M, fast internal RC oscillator

enum `_clock_mstr_2x_clk_src`

List of master 2x clock source.

Values:

enumerator `kCLOCK_Mstr2xClkSrcMstrOsc`
Master oscillator clock

enumerator `kCLOCK_Mstr2xClkSrcPllDiv2`
PLL output divide 2

enum `_clock_output_clk_src`

List of output clock source.

Values:

enumerator `kCLOCK_OutputClkSrc_Bus`
Bus clock

enumerator `kCLOCK_OutputClkSrc_Bus2x`
Bus 2x clock

enumerator `kCLOCK_OutputClkSrc_BusDiv4`
Bus clock div 4

enumerator `kCLOCK_OutputClkSrc_MstrOSC`
Master oscillator clock

enumerator `kCLOCK_OutputClkSrc_FircDiv6`
8M clock, FIRC/6

enumerator `kCLOCK_OutputClkSrc_Sirc`
200K clock, SIRC

enum `_clock_output_clk_div`

List of output clock divider.

Values:

enumerator `kCLOCK_OutputDiv1`
output clock = selectedClock/1U

enumerator `kCLOCK_OutputDiv2`
output clock = selectedClock/2U

enumerator `kCLOCK_OutputDiv4`
output clock = selectedClock/4U

enumerator `kCLOCK_OutputDiv8`
output clock = selectedClock/8U

enumerator `kCLOCK_OutputDiv16`
output clock = selectedClock/16U

enumerator `kCLOCK_OutputDiv32`
output clock = selectedClock/32U

enumerator `kCLOCK_OutputDiv64`
output clock = selectedClock/64U

enumerator `kCLOCK_OutputDiv128`
output clock = selectedClock/128U

enum `_clock_protection`

List of clock register protection mode.

Values:

enumerator `kCLOCK_Protection_Off`
No protection, and could be changed any time

enumerator `kCLOCK_Protection_On`
Protected, and could be changed any time

enumerator `kCLOCK_Protection_OffLock`
No protection and get locked until chip reset

enumerator `kCLOCK_Protection_OnLock`
Protected and get locked until chip reset

enum `_clock_ip_clk_src`

List of specific IP's clock source.

Values:

enumerator `kCLOCK_IPClkSrc_BusClk`
Bus clock

enumerator `kCLOCK_IPClkSrc_Bus2xClk`
Bus 2x clock

enum `_clock_postscale`

Mstr 2x clock postscale divider.

Values:

enumerator `kCLOCK_PostscaleDiv1`
Mast 2X clock = $\text{clkSrc} / 1$

enumerator `kCLOCK_PostscaleDiv2`
Mast 2X clock = $\text{clkSrc} / 2$

enumerator `kCLOCK_PostscaleDiv4`
Mast 2X clock = $\text{clkSrc} / 4$

enumerator `kCLOCK_PostscaleDiv8`
Mast 2X clock = $\text{clkSrc} / 8$

enumerator `kCLOCK_PostscaleDiv16`
Mast 2X clock = $\text{clkSrc} / 16$

enumerator `kCLOCK_PostscaleDiv32`
Mast 2X clock = $\text{clkSrc} / 32$

enumerator `kCLOCK_PostscaleDiv64`
Mast 2X clock = $\text{clkSrc} / 64$

enumerator `kCLOCK_PostscaleDiv128`
Mast 2X clock = $\text{clkSrc} / 128$

enumerator `kCLOCK_PostscaleDiv256`
Mast 2X clock = $\text{clkSrc} / 256$

enum `_clock_pll_monitor_type`

PLL monitor type structure.

Values:

enumerator kCLOCK_PllMonitorUnLockCoarse
PLL coarse unlock, due to loss of reference clock, power unstable...etc.

enumerator kCLOCK_PllMonitorUnLockFine
PLL fine unlock, due to loss of reference clock, power unstable...etc.

enumerator kCLOCK_PllMonitorLostofReferClk
PLL lost reference clock.

enumerator kCLOCK_PllMonitorAll
All PLL monitor type.

enum _pit_count_clock_source
Describes PIT clock source.

Values:

enumerator kPIT_CountClockSource0
PIT count clock sourced from IP bus clock

enumerator kPIT_CountClockSource1
PIT count clock sourced from alternate clock 1

enumerator kPIT_CountClockSource2
PIT count clock sourced from alternate clock 2

enumerator kPIT_CountClockSource3
PIT count clock sourced from alternate clock 3

enumerator kPIT_CountBusClock
PIT count clock sourced from IP bus clock

enumerator kPIT_CountCrystalClock
PIT count clock sourced from crystal clock

enumerator kPIT_Count8MHzIRCClock
PIT count clock sourced from 8MHz IRC clock

enumerator kPIT_Count200KHzIRCClock
PIT count clock sourced from 200KHz IRC clock

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]

enumerator kEWM_Lpo8MHzIRCClock
EWM clock sourced from 48 MHz IRC div6 clock

enumerator kEWM_LpoCrystalClock
EWM clock sourced from crystal clock

enumerator `kEWM_LpoBusClock`
EWM clock sourced from IP Bus clock

enumerator `kEWM_Lpo200KHzIRCClock`
EWM clock sourced from 200KHz IRC clock

typedef enum `_clock_ip_name` `clock_ip_name_t`
List of IP clock name.

typedef enum `_clock_name` `clock_name_t`
List of system-level clock name.

typedef enum `_clock_crystal_osc_mode` `clock_crystal_osc_mode_t`
Crystal oscillator mode.

typedef enum `_clock_ext_clk_src` `clock_ext_clk_src_t`
List of external clock source.

typedef enum `_clock_ext_clkin_sel` `clock_ext_clkin_sel_t`
List of clock-in source.

typedef enum `_clock_mstr_osc_clk_src` `clock_mstr_osc_clk_src_t`
List of master oscillator source.

typedef enum `_clock_mstr_2x_clk_src` `clock_mstr_2x_clk_src_t`
List of master 2x clock source.

typedef enum `_clock_output_clk_src` `clock_output_clk_src_t`
List of output clock source.

typedef enum `_clock_output_clk_div` `clock_output_clk_div_t`
List of output clock divider.

typedef enum `_clock_protection` `clock_protection_t`
List of clock register protection mode.

typedef enum `_clock_ip_clk_src` `clock_ip_clk_src_t`
List of specific IP's clock source.

typedef enum `_clock_postscale` `clock_postscale_t`
Mstr 2x clock postscale divider.

typedef struct `_clock_protection_config` `clock_protection_config_t`
Clock register protection configuration.

typedef struct `_clock_output_config` `clock_output_config_t`
Clock output configuration.

typedef struct `_clock_config` `clock_config_t`
mcu clock configuration structure.

This is the key configuration structure of clock driver, which define the system clock behavior. The function `CLOCK_SetClkConfig` deploy this configuration structure onto SOC.

typedef enum `_clock_pll_monitor_type` `clock_pll_monitor_type_t`
PLL monitor type structure.

typedef enum `_pit_count_clock_source` `pit_count_clock_source_t`
Describes PIT clock source.

typedef enum `_ewm_lpo_clock_source` `ewm_lpo_clock_source_t`
Describes EWM clock source.

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

Enable IPs clock.

Parameters

- eIpClkName – IP clock name.

```
static inline void CLOCK_DisableClock(clock_ip_name_t eIpClkName)
```

Disable IPs clock.

Parameters

- eIpClkName – IP clock name.

```
static inline void CLOCK_EnableClockInStopMode(clock_ip_name_t eIpClkName)
```

Enable IPs clock in STOP mode.

Parameters

- eIpClkName – IP clock name.

```
static inline void CLOCK_DisableClockInStopMode(clock_ip_name_t eIpClkName)
```

Disable IPs clock in STOP mode.

Parameters

- eIpClkName – IP clock name.

```
static inline void CLOCK_ConfigQsciClockSrc(clock_ip_name_t eQsciClkName, clock_ip_clk_src_t  
eClkSrc)
```

Configure QSCI clock source.

QSCI clock could be bus or bus_2x clock. Default is bus clock.

Parameters

- eQsciClkName – IP(only QSCI is valid) clock name.
- eClkSrc – Clock source.

```
static inline void CLOCK_ConfigI2cFilterClockSrc(clock_ip_name_t eI2cClkName,  
clock_ip_clk_src_t eClkSrc)
```

Configure I2C filter clock source.

I2C filter clock could be bus or bus_2x clock. Default is bus clock.

Parameters

- eI2cClkName – IP(only I2C is valid) clock name.
- eClkSrc – Clock source.

```
static inline void CLOCK_SetSlowIrcTrim(uint16_t u16Trim)
```

Set trim value to 200K slow internal RC oscillator.

The factory trim value is loaded during reset. User may call this function to fine tune the 200K IRC oscillator.

Parameters

- u16Trim – Slow internal RC oscillator trim value.

```
static inline bool CLOCK_GetCrystalOscFailureStatus(void)
```

Get crystal oscillator failure status.

Note: This function should be called only when crystal osc is on and its monitor(MON_ENABLE in OSCTL2 register) is enabled.

Returns

Crystal oscillator status. true: Crystal oscillator frequency is below 680KHz(typical). false: No clock failure or crystal oscillator is off.

```
static inline void CLOCK_SetPllLossOfRefererntTripPoint(uint8_t u8Trip)
```

Set PLL loss of reference trip point.

The trip point default value is 2.

Parameters

- u8Trip – Trip point for loss of reference.

```
static inline void CLOCK_ClearPLLMonitorFlag(clock_pll_monitor_type_t eType)
```

Clear PLL monitor flag.

Parameters

- eType – PLL monitor type.

```
uint32_t CLOCK_GetFreq(clock_name_t eClkName)
```

Get system-level clock frequency.

Parameters

- eClkName – System-level clock name.

Returns

The required clock's frequency in Hz.

```
uint32_t CLOCK_GetIpClkSrcFreq(clock_ip_name_t eIpClkName)
```

Get IP clock frequency.

Parameters

- eIpClkName – IP clock name.

Returns

The required IP clock's frequency in Hz.

```
void CLOCK_SetClkin0Freq(uint32_t u32Freq)
```

Set Clock IN 0 frequency.

It is a must to call this function in advance if system is operated by clkin0.

Parameters

- u32Freq – Clock IN 0 frequency in Hz.

```
void CLOCK_SetClkin1Freq(uint32_t u32Freq)
```

Set Clock IN 1 frequency.

It is a must to call this function in advance if system is operated by clkin1.

Parameters

- u32Freq – Clock IN 1 frequency in Hz.

```
void CLOCK_SetXtalFreq(uint32_t u32Freq)
```

Set crystal oscillator frequency.

It is a must to call this function in advance if system is operated by crystal oscillator.

Parameters

- u32Freq – Crystal oscillator frequency in Hz.

void CLOCK_SetProtectionConfig(*clock_protection_config_t* *psConfig)
Config clock register access protection mode.

Parameters

- psConfig – Pointer for protection configuration.

void CLOCK_SetOutputClockConfig(*clock_output_config_t* *psConfig)
Config output clock.

Parameters

- psConfig – Pointer for clock output configuration.

void CLOCK_SetClkConfig(*clock_config_t* *psConfig)
Config mcu operation clock.

Parameters

- psConfig – Pointer for clock configuration.

void CLOCK_EnableUsbfs0Clock(void)
Enable USB FS clock.

uint32_t CLOCK_EvaluateExtClkFreq(void)
Evaluate external clock frequency and return its frequency in Hz.

This function should be called only when internal FIRC(48M) is on. The evaluated result accuracy depends on:

- a. FIRC accuracy, now it is +/-1%.
- b. Truncation error, because the external clock and FIRC is not synchronised.
- c. External clock frequency, low accuracy for lower external clock frequency.
- d. MCU mstr 2x clock.

For example, for namely 8M external clock, evaluated result may be range in 8M+/-5%.

Returns

Evaluated external frequency in Hz.

void CLOCK_EnablePLLMonitorInterrupt(*clock_pll_monitor_type_t* eType, bool bEnable)
Enable/Disable PLL monitor interrupt.

This function should be called only when PLL is on and its reference clock is external clock. This function is for safety purpose when external clock is lost due to HW failure. The normal flow to call this function:

- a. Call CLOCK_SetClkConfig to enable PLL and external clock to feed the PLL.
- b. Call CLOCK_ClearPLLMonitorFlag.
- c. Call CLOCK_SetPllLossofRefererntTripPoint (optional, setting value is for kCLOCK_PllMonitorLostofReferClk type).
- d. Call this function.
- e. Enable OCCS interrupt with highest priority 3.
- f. When OCCS interrupt occurs, recover clock from the disaster in OCCS_DriveISRHandler function. Such kind of clock recovery is application dependent, and a demo OCCS_DriveISRHandler has been shown in fsl_clock.c

Parameters

- eType – PLL monitor type.
- bEnable – Enable or disable.

FSL_CLOCK_DRIVER_VERSION

CLOCK driver version 2.1.0.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

Definition for delay API in clock driver, users can redefine it to the real application.

GPIO_CLOCKS

Clock ip name array for GPIO.

TMR_CLOCKS

Clock ip name array for quad timer.

FLEXCAN_CLOCKS

Clock ip name array for FLEXCAN.

I2C_CLOCKS

Clock ip name array for I2C.

QSPI_CLOCKS

Clock ip name array for queued SPI.

QSCI_CLOCKS

Clock ip name array for queued SCI.

DAC_CLOCKS

Clock ip name array for DAC.

PIT_CLOCKS

Clock ip name array for PIT.

CRC_CLOCKS

Clock ip name array for CRC.

CADC_CLOCKS

Clock ip name array for cyclic ADC.

CMP_CLOCKS

Clock ip name array for CMP.

PWM_CLOCKS

Clock ip name array for PWM.

USB_CLOCKS

Clock ip name array for USB.

ROM_CLOCKS

Clock ip name array for ROM.

EDMA_CLOCKS

Clock ip name array for EDMA.

EWM_CLOCKS

Clock ip name array for EWM.

XBARA_CLOCKS

Clock ip name array for XBARA.

CLK_GATE_GET_REG_INDEX(x)

CLK_GATE_GET_BIT_INDEX(x)

clock_protection_t eFrqEP

FRQEP bit field in OCCS PROT register, protect COD & ZSRC.

clock_protection_t eOscEP

OSCEP bit field in OCCS PROT register, protect OSCTL1, OSCTL2, PRECS.

clock_protection_t ePllEP

PLLEP bit field in OCCS PROT register, protect PLLDP, LOCIE, LORTP, PLLDB bitfield.

bool bClkOut0En

Clock output 0 enable, CLKDIS0 bit field in SIM CLKOUT register

bool bClkOut1En

Clock output 1 enable, CLKDIS1 bit field in SIM CLKOUT register

clock_output_clk_src_t eClkOut0Src

Clock output 0 clock source, CLKOSSEL0 bit field in SIM CLKOUT register

clock_output_clk_src_t eClkOut1Src

Clock output 1 clock source, CLKOSSEL1 bit field in SIM CLKOUT register

clock_output_clk_div_t eClkDiv

Clock output divider, CLKODIV bit field in SIM CLKOUT register ,it apply to clkout0 & clkout1

bool bCrystalOscEnable

Crystal oscillator enable, COPD bit field in OCCS OSCTL2 register

bool bFircEnable

Fast internal RC oscillator enable, IRC48M_EN bit field in SIM MISC0 register

bool bSircEnable

Slow internal RC oscillator enable, ROPD200K bit field in OCCS OSCTL2 register

bool bPllEnable

PLL enable, PLLPD bit field in OCCS CTRL register

bool bCrystalOscMonitorEnable

Crystal oscillator monitor enable, MON_ENABLE bit field in OCCS OSCTL2 register

clock_crystal_osc_mode_t eCrystalOscMode

Crystal oscillator mode, COHL bit field in OCCS OSCTL1 register

clock_ext_clk_src_t eExtClkSrc

External clock source, EXT_SEL bit field in OCCS OSCTL1 register

clock_ext_clkin_sel_t eClkInSel

Clock IN selection(0 or 1), CLKINSEL bit field in SIM MISC0 register

clock_mstr_osc_clk_src_t eMstrOscClkSrc

Master oscillator selection, PRECS bit field in OCCS CTRL register. When selected kCLOCK_MstrOscClkSrcExt, make sure corresponding pins(crystal osc or clkin pin) has been configured.

clock_mstr_2x_clk_src_t eMstr2xClkSrc

Master 2x clock selection, ZSRC bit field in OCCS CTRL register

clock_postscale_t eMstr2xClkPostScale

Master 2x clock post scale, COD bit field in OCCS DIVBY register

uint32_t u32PllClkFreq

Required PLL output frequency before divide 2

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 _clock_protection_config

#include <fsl_clock.h> Clock register protection configuration.

struct _clock_output_config

#include <fsl_clock.h> Clock output configuration.

struct _clock_config

#include <fsl_clock.h> mcu clock configuration structure.

This is the key configuration structure of clock driver, which define the system clock behavior. The function CLOCK_SetClkConfig deploy this configuration structure onto SOC.

2.5 Driver Change Log

2.6 CMP: Comparator Driver

void CMP_GetDefaultConfig(*cmp_config_t* *psConfig)

Initializes the CMP user configuration structure.

This function initializes the user configuration structure to the default values. It is corresponding to the continuous mode configurations.

Parameters

- psConfig – pointer of *cmp_config_t*.

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

Initializes the CMP.

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

- Enable the clock for CMP module.
- Configure the comparator according to the CMP configuration structure.

Parameters

- base – CMP peripheral base address.
- psConfig – 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.

Parameters

- `base` – CMP peripheral base address.

```
static inline void CMP_Enable(CMP_Type *base, bool bEnable)
```

Enables/disables the CMP module.

Parameters

- `base` – CMP peripheral base address.
- `bEnable` – Enables or disables the module.

```
static inline void CMP_SetInputChannel(CMP_Type *base, cmp_input_mux_t ePlusChannel,  
                                     cmp_input_mux_t eMinusChannel)
```

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.
- `ePlusChannel` – Plus side input channel number.
- `eMinusChannel` – Minus side input channel number.

```
static inline void CMP_SelectOutputSource(CMP_Type *base, cmp_output_source_t  
                                         eOutputSource)
```

Select comparator output source.

Parameters

- `base` – CMP peripheral base address.
- `eOutputSource` – The output signal to be set, please reference `cmp_output_source_t` for details.

```
static inline void CMP_EnableOutputPin(CMP_Type *base, bool bEnable)
```

Enable/Disable Comparator output pin.

Parameters

- `base` – CMP peripheral base address.
- `bEnable` – Enable/Disable comparator output pin. `true` — CMPO is available on the associate CMPO output pin. `false` — CMPO is not available on the associate CMPO output pin.

```
static inline uint8_t CMP_GetComparatorOutput(CMP_Type *base)
```

Get Comparator output.

Parameters

- `base` – CMP peripheral base address.

Return values

`current` – analog comparator output 0 or 1

```
static inline void CMP_SetHysteresisLevel(CMP_Type *base, cmp_hysteresis_level_t  
                                         eHysteresisLevel)
```

Sets hysteresis level.

Parameters

- `base` – CMP peripheral base address.
- `eHysteresisLevel` – The programmable hysteresis level to be set, please refer to `cmp_hysteresis_level_t` for details.

```
static inline void CMP_SetComparasionSpeedMode(CMP_Type *base,
                                              cmp_comparasion_speed_mode_t
                                              eComparatorSpeedMode)
```

Sets comparison speed mode.

Parameters

- base – CMP peripheral base address.
- eComparatorSpeedMode – The comparison speed mode, please reference `cmp_comparasion_speed_mode_t` for details.

```
static inline void CMP_EnableInvertOutput(CMP_Type *base, bool bEnable)
    Enable/Disable comparator invert feature.
```

Parameters

- base – CMP peripheral base address.
- bEnable – Enable/Disable comparator invert feature. true — Inverts the comparator output. false — Does not invert the comparator output.

```
static inline void CMP_EnableWindow(CMP_Type *base, bool bEnable)
    Enable the window function.
```

Parameters

- base – CMP peripheral base address.
- bEnable – true is enable, false is disable.

```
static inline void CMP_SetWindowOutputMode(CMP_Type *base, cmp_window_output_mode_t
                                           eWindowOutputMode)
```

Set the window output mode.

Parameters

- base – CMP peripheral base address.
- eWindowOutputMode – `cmp_window_output_mode_t`.

```
static inline void CMP_EnableExternalSampleMode(CMP_Type *base, bool bEnable)
    Enable/Disable external Sample mode.
```

Parameters

- base – CMP peripheral base address.
- bEnable – true is using external sample mode, false is using interface sample mode.

```
static inline void CMP_SetExternalSampleCount(CMP_Type *base, cmp_external_sample_count_t
                                             eSampleCount)
```

Sets external sample count.

Parameters

- base – CMP peripheral base address.
- eSampleCount – The number of consecutive samples that must agree prior to the comparator output filter accepting a new output state, `cmp_external_sample_count_t`.

```
static inline void CMP_SetInternalFilterCount(CMP_Type *base, cmp_filter_count_t eFilterCount)
    Sets internal filter count.
```

Parameters

- `base` – CMP peripheral base address.
- `eFilterCount` – The number of consecutive samples that must agree prior to the comparator output filter accepting a new output state, `cmp_filter_count_t`.

```
static inline void CMP_SetInternalFilterPeriod(CMP_Type *base, uint8_t u8FilterPeriod)
```

Sets the internal filter period. It is used as the divider to bus clock.

Parameters

- `base` – CMP peripheral base address.
- `u8FilterPeriod` – Filter Period. The divider to the bus clock. Available range is 0-255.

```
void CMP_SetDACConfig(CMP_Type *base, const cmp_dac_config_t *psConfig)
```

Configures the internal DAC.

Parameters

- `base` – CMP peripheral base address.
- `psConfig` – Pointer to the configuration structure.

```
static inline void CMP_SetDACOutputVoltage(CMP_Type *base, uint8_t  
u8OutputVoltageDivider)
```

Sets DAC output voltage.

Parameters

- `base` – CMP peripheral base address.
- `u8OutputVoltageDivider` – The digital value which is related to the desired DAC output voltage,

```
static inline void CMP_EnableInternalDAC(CMP_Type *base, bool bEnable)
```

Enable/Disable internal DAC.

Parameters

- `base` – CMP peripheral base address.
- `bEnable` – Enable/Disable internal DAC. `true` Enable internal DAC. `false` Disable internal DAC.

```
static inline void CMP_SetDACReferenceVoltageSource(CMP_Type *base, cmp_dac_vref_source_t  
eDACVrefSource)
```

Sets internal DAC's reference voltage source.

Parameters

- `base` – CMP peripheral base address.
- `eDACVrefSource` – reference voltage source, please `cmp_dac_vref_source_t`

```
static inline void CMP_EnableInterrupt(CMP_Type *base, cmp_interrupt_request_t  
eInterruptRequest)
```

Interrupt request to enable.

Parameters

- `base` – CMP peripheral base address.
- `eInterruptRequest` – Mask value for interrupts. See `cmp_interrupt_request_t`.

```
static inline cmp_output_flag_t CMP_GetStatusFlags(CMP_Type *base)
```

Gets the status flags.

Parameters

- base – CMP peripheral base address.

Return values

Mask – value for the asserted flags. *cmp_output_flag_t*.

```
static inline void CMP_ClearStatusFlags(CMP_Type *base, cmp_output_flag_t eOutputFlag)
```

Clears the status flags.

Parameters

- base – CMP peripheral base address.
- eOutputFlag – Mask value for the output flags, *cmp_output_flag_t*

```
static inline void CMP_EnableDMA(CMP_Type *base, cmp_dma_request_t eDMARequestType)
```

Enables CMP DMA request.

Parameters

- base – CMP peripheral base address.
- eDMARequestType – eDMA request type, *cmp_dma_request_t*

```
static inline uint32_t CMP_GetComparatorResultRegisterAddress(CMP_Type *base)
```

Get CMP result register address for DMA access.

Parameters

- base – CMP peripheral base address.

Returns

The CMP result register address.

```
FSL_CMP_DRIVER_VERSION
```

CMP driver version.

```
enum _cmp_interrupt_request
```

CMP Interrupt request type definition.

Values:

```
enumerator kCMP_InterruptRequestDisabled
interrupt disabled
```

```
enumerator kCMP_InterruptRequestEnableOutputRisingEdge
Comparator interrupt request enable rising edge.
```

```
enumerator kCMP_InterruptRequestEnableOutputFallingEdge
Comparator interrupt request enable falling edge.
```

```
enumerator kCMP_InterruptRequestEnableAll
comparator interrupt request enable on rising edge or falling edge
```

```
enum _cmp_dma_request
```

CMP DMA request type definition.

Values:

```
enumerator kCMP_DMARequestDisabled
DMA disabled
```

```
enumerator kCMP_DMARequestEnableOutputRisingEdge
Comparator dma request enable on rising edge.
```

enumerator kCMP_DMARequestEnableOutputFallingEdge
Comparator dma request enable on falling edge.

enumerator kCMP_DMARequestEnableAll
comparator dma request enable on rising edge or falling edge

enum _cmp_output_flag
CMP output flags' mask.

Values:

enumerator kCMP_OutputFlagRisingEdge
Rising-edge on the comparison output has occurred.

enumerator kCMP_OutputFlagFallingEdge
Falling-edge on the comparison output has occurred.

enumerator kCMP_OutputFlagBothEdge
Rising-edge and Falling-edge on the comparison output has occurred.

enum _cmp_hysteresis_level
CMP Hysteresis level.

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_comparasion_speed_mode
CMP compassion speed mode enumerator.

Values:

enumerator kCMP_ComparsionModeLowSpeed
Low-Speed Comparison mode has lower current consumption

enumerator kCMP_ComparsionModeHighSpeed
High-Speed Comparison mode has higher current consumption.

enum _cmp_dac_vref_source
CMP DAC Voltage Reference source.

Values:

enumerator kCMP_DACVrefSourceVin1
Vin1 is selected as a resistor ladder network supply reference Vin.

enumerator kCMP_DACVrefSourceVin2
Vin2 is selected as a resistor ladder network supply reference Vin.

enum _cmp_window_output_mode
CMP output value of window.

Values:

enumerator kCMP_WindowOutputLastLatchedValue

When WINDOW signal changes from 1 to 0, COUTA output holds the last latched value before WINDOW signal falls to 0

enumerator kCMP_WindowOutputZeroValue

When WINDOW signal changes from 1 to 0, COUTA output is forced to 0

enum _cmp_filter_count

CMP filter count.

Values:

enumerator kCMP_FilterCountDisable

filter is disabled

enumerator kCMP_FilterCount1

1 sample must agrees, the comparator output is simply sampled

enumerator kCMP_FilterCount2

2 consecutive samples must agrees

enumerator kCMP_FilterCount3

3 consecutive samples must agrees

enumerator kCMP_FilterCount4

4 consecutive samples must agrees

enumerator kCMP_FilterCount5

5 consecutive samples must agrees

enumerator kCMP_FilterCount6

6 consecutive samples must agrees

enumerator kCMP_FilterCount7

7 consecutive samples must agrees

enum _cmp_external_sample_count

CMP external sample count.

Values:

enumerator kCMP_ExternalSampleCount1

1 sample must agrees, the comparator output is simply sampled

enumerator kCMP_ExternalSampleCount2

2 consecutive samples must agrees

enumerator kCMP_ExternalSampleCount3

3 consecutive samples must agrees

enumerator kCMP_ExternalSampleCount4

4 consecutive samples must agrees

enumerator kCMP_ExternalSampleCount5

5 consecutive samples must agrees

enumerator kCMP_ExternalSampleCount6

6 consecutive samples must agrees

enumerator kCMP_ExternalSampleCount7

7 consecutive samples must agrees

enum `_cmp_output_source`

CMP output source enumerator.

Values:

enumerator `kCMP_OutputSourceFromFilterCOUT`

Set the filtered comparator output to equal COUT.

enumerator `kCMP_OutputSourceFromUnfilteredCOUTA`

Set the unfiltered comparator output to equal COUTA.

enum `_cmp_work_mode`

CMP work mode definition.

Values:

enumerator `kCMP_WorkModeWindowBypassAndNoExternalSample`
window block bypassed, external sampling mode disabled

enumerator `kCMP_WorkModeWindowBypassAndExternalSample`
window block bypassed, external SAMPLE mode enable

enumerator `kCMP_WorkModeWindowEnabledAndNoExternalSample`
window block enabled, external sampling mode disabled

typedef enum `_cmp_interrupt_request` `cmp_interrupt_request_t`

CMP Interrupt request type definition.

typedef enum `_cmp_dma_request` `cmp_dma_request_t`

CMP DMA request type definition.

typedef enum `_cmp_output_flag` `cmp_output_flag_t`

CMP output flags' mask.

typedef enum `_cmp_hysteresis_level` `cmp_hysteresis_level_t`

CMP Hysteresis level.

typedef enum `_cmp_comparasion_speed_mode` `cmp_comparasion_speed_mode_t`

CMP compassion speed mode enumerator.

typedef enum `_cmp_dac_vref_source` `cmp_dac_vref_source_t`

CMP DAC Voltage Reference source.

typedef enum `_cmp_window_output_mode` `cmp_window_output_mode_t`

CMP output value of window.

typedef enum `_cmp_filter_count` `cmp_filter_count_t`

CMP filter count.

typedef enum `_cmp_external_sample_count` `cmp_external_sample_count_t`

CMP external sample count.

typedef enum `_cmp_output_source` `cmp_output_source_t`

CMP output source enumerator.

typedef enum `_cmp_work_mode` `cmp_work_mode_t`

CMP work mode definition.

typedef struct `_cmp_dac_config` `cmp_dac_config_t`

CMP internal DAC configuration structure.

```
typedef union _cmp_dma_interrupt_config cmp_dma_interrupt_config_t
    CMP dma/interrupt configure union.
```

Note: , the interrupt request and dma request cannot be used at the same time, that is to say When DMA support is enabled by setting SCR[DMAEN] and the interrupt is enabled by setting SCR[IER], SCR[IEF], or both, the corresponding change on COUT forces a DMA transfer request rather than a CPU interrupt instead

```
typedef struct _cmp_config cmp_config_t
    CMP configuration structure.
```

```
struct _cmp_dac_config
    #include <fsl_cmp.h> CMP internal DAC configuration structure.
```

Public Members

```
cmp_dac_vref_source_t eDACVrefSource
    DAC reference voltage source.
```

```
uint8_t u8DACOutputVoltageDivider
    divider Value for the DAC Output Voltage, DAC output voltage = (VREF / 256) *
    (u8DACOutputVoltageDivider + 1).
```

```
bool bEnableInternalDAC
    flag to control if the internal DAC need to be enabled
```

```
union _cmp_dma_interrupt_config
    #include <fsl_cmp.h> CMP dma/interrupt configure union.
```

Note: , the interrupt request and dma request cannot be used at the same time, that is to say When DMA support is enabled by setting SCR[DMAEN] and the interrupt is enabled by setting SCR[IER], SCR[IEF], or both, the corresponding change on COUT forces a DMA transfer request rather than a CPU interrupt instead

Public Members

```
cmp_dma_request_t eDMARequest
    dma request type
```

```
cmp_interrupt_request_t eInterruptRequest
    interrupt request type
```

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

Public Members

```
cmp_hysteresis_level_t eHysteresisLevel
    CMP hysteresis levelL
```

```
cmp_comparasion_speed_mode_t eComparasionSpeedMode
    CMP comparison speed mode
```

cmp_work_mode_t eWorkMode

CMP work mode

cmp_input_mux_t ePlusInput

CMP plus input mux, the definition of this enum is in soc header file

cmp_input_mux_t eMinusInput

CMP minus input mux, the definition of this enum is in soc header file

cmp_dac_config_t sDacConfig

CMP internal DAC configuration structure *cmp_dac_config_t*

bool bInvertComparatorOutputPolarity

Inverted comparator output polarity.

cmp_window_output_mode_t eWindowOutputMode

only works when cmp work mode is kCMP_WorkModeWindowEnabledAndNoExternalSample

cmp_filter_count_t eFilterCount

Filter Count. Available range is 0-7, 0 is disable internal filter can be used in internal sampling mode only.

uint8_t u8FilterPeriod

Filter Period. The divider to the bus clock. Available range is 0-255, can be used in internal sampling mode. When the filter clock from internal divided bus clock, setting the sample period to 0 will disable the filter

cmp_external_sample_count_t eExternalSampleCount

Available range is 1 - 7, used in external sampling mode only

cmp_output_source_t eOutputSource

cmp output source

bool bEnableOutputPin

the comparator output(CMPO) is driven out on the associated CMPO output pin

cmp_dma_interrupt_config_t uDmaInterruptConfig

CMP interrupt/dma configuration

bool bCMPEnable

flag to control if CMP module start immediately when the configuration is done

2.7 The Driver Change Log

2.8 CMP Peripheral and Driver Overview

2.9 COP: Computer Operating Properly(Watchdog) Driver

void COP_Init(COP_Type *base, const *cop_config_t* *psConfig)

Initializes the COP module with input configuration.

Call this function to do initialization configuration for COP module. The configurations are:

- COP configuration write protect enablement
- Clock source selection for COP module
- Prescaler configuration to the input clock source

- Counter timeout value
- Window value
- WAIT/STOP work mode enablement
- Interrupt enable/disable and interrupt timing value
- Loss of reference counter enablement
- COP enable/disable

note: Once set `bEnableWriteProtect=true`, the `CTRL`, `INTVAL`, `WINDOW` and `TOUT` registers are read-only.

Parameters

- `base` – COP peripheral base address.
- `psConfig` – The pointer to COP configuration structure, `cop_config_t`.

```
void COP_GetDefaultConfig(cop_config_t *psConfig)
```

Prepares an available pre-defined setting for module's configuration.

This function initializes the COP configuration structure to default values.

```
psConfig->bEnableWriteProtect = false;
psConfig->bEnableWait = false;
psConfig->bEnableStop = false;
psConfig->bEnableLossOfReference = false;
psConfig->bEnableInterrupt = false;
psConfig->bEnableCOP = false;
psConfig->ePrescaler = kCOP_ClockPrescalerDivide1;
psConfig->u16TimeoutCount = 0xFFFFU;
psConfig->u16WindowCount = 0xFFFFU;
psConfig->u16InterruptCount = 0xFFU;
psConfig->eClockSource = kCOP_RoscClockSource;
```

Parameters

- `psConfig` – Pointer to the COP configuration structure, `cop_config_t`.

```
static inline void COP_Enable(COP_Type *base, bool bEnable)
```

Enable/Disable the COP module.

This function disables the COP Watchdog. To disable the COP Watchdog, call `COP_Enable(base, false)`.

Parameters

- `base` – COP peripheral base address.
- `bEnable` – Enable the feature or not.

```
static inline void COP_EnableLossOfReferenceCounter(COP_Type *base, bool bEnable)
```

Enables or disables the COP Loss of Reference counter.

This function writes a value into the `COP_CTRL` register to enable or disable the COP Loss of Reference counter.

Parameters

- `base` – COP peripheral base address.
- `bEnable` – Enable the feature or not.

```
static inline void COP_SetTimeoutCount(COP_Type *base, uint16_t u16TimeoutCount)
```

Sets the COP timeout value.

This function sets the COP timeout value, if `psConfig->bEnableWriteProtect` is set to true for calling `WDOG_Init`, the set does not take effect. It should be ensured that the time-out value for the COP is always greater than interrupt time + 40 bus clock cycles. This function writes a value into `COP_TOUT` register, when COP count down to zero from the timeout count value, `COP_RST_B` signal will be asserted. There are some considerations for setting the timeout count after COP is enabled:

- The recommended procedure for changing `TIMEOUT` is to disable the COP by invoking `COP_Enable()`, then call the function `COP_SetTimeoutCount`, and then re-enable the by invoking `COP_Enable()` again.
- Alternatively, call the function `COP_SetTimeoutCount`, then feed the COP by invoking `COP_Refresh()` to reload the timeout.

Parameters

- `base` – COP peripheral base address
- `u16TimeoutCount` – COP timeout value, count of COP clock tick. Use macro definition `MSEC_TO_COUNT` to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

```
static inline void COP_SetInterruptCount(COP_Type *base, uint16_t u16InterruptCount)
```

Sets the COP interrupt value.

This function sets the COP interrupt value, if `psConfig->bEnableWriteProtect` is set to true for calling `WDOG_Init`, the set does not take effect. This function writes a value into `COP_INTVAL` register, if COP interrupt is enabled and COP count down to the interrupt value configured, an interrupt will be triggered. Ensure the COP counter is disabled while the function is called.

Parameters

- `base` – COP peripheral base address
- `u16InterruptCount` – COP interrupt value, count of COP clock tick. Use macro definition `MSEC_TO_COUNT` to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

```
static inline void COP_SetWindowCount(COP_Type *base, uint16_t u16WindowCount)
```

Sets the COP window value.

This function sets the COP window value, if `psConfig->bEnableWriteProtect` is set to true for calling `WDOG_Init`, the set does not take effect. This function writes a value into `COP_WINDOW` register. Ensure the COP counter is disabled while the function is called.

Parameters

- `base` – COP peripheral base address
- `u16WindowCount` – COP window value, count of COP clock tick. Use macro definition `MSEC_TO_COUNT` to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

```
void COP_Refresh(COP_Type *base)
```

Refreshes the COP timer.

This function feeds/services the COP.

Parameters

- `base` – COP peripheral base address.

```
static inline void COP_EnableInterrupt(COP_Type *base)
```

Enables the COP interrupt, if psConfig->bEnableWriteProtect is set to true for calling WDOG_Init, the operation does not take effect.

This function writes a value into the COP_CTRL register to enable the COP interrupt.

Parameters

- base – COP peripheral base address

```
static inline void COP_DisableInterrupt(COP_Type *base)
```

Disables the COP interrupt, if psConfig->bEnableWriteProtect is set to true for calling WDOG_Init, the operation does not take effect.

This function writes a value into the COP_CTRL register to disable the COP interrupt.

Parameters

- base – COP peripheral base address

```
FSL_COP_DRIVER_VERSION
```

COP driver version.

```
COP_FIRST_WORD_OF_REFRESH
```

COP refresh key word.

First word of refresh sequence

```
COP_SECOND_WORD_OF_REFRESH
```

Second word of refresh sequence

```
enum _cop_clock_source
```

enumeration for COP clock source selection.

Values:

```
enumerator kCOP_RoscClockSource
```

COP clock sourced from Relaxation oscillator (ROSC)

```
enumerator kCOP_CoscClockSource
```

COP clock sourced from Crystal oscillator (COSCs)

```
enumerator kCOP_BusClockSource
```

COP clock sourced from IP Bus clock

```
enumerator kCOP_LpoClockSource
```

COP clock sourced from Low speed oscillator

```
enum _cop_clock_prescaler
```

enumeration for COP clock prescaler to input source clock.

Values:

```
enumerator kCOP_ClockPrescalerDivide1
```

Divided by 1

```
enumerator kCOP_ClockPrescalerDivide16
```

Divided by 16

```
enumerator kCOP_ClockPrescalerDivide256
```

Divided by 256

```
enumerator kCOP_ClockPrescalerDivide1024
```

Divided by 1024

`typedef enum _cop_clock_source cop_clock_source_t`
enumeration for COP clock source selection.

`typedef enum _cop_clock_prescaler cop_clock_prescaler_t`
enumeration for COP clock prescaler to input source clock.

`typedef struct _cop_config cop_config_t`
structure for COP module initialization configuration.

`struct _cop_config`
`#include <fsl_cop.h>` structure for COP module initialization configuration.

Public Members

`bool bEnableWriteProtect`
Set COP Write protected

`bool bEnableStop`
Enable or disable COP in STOP mode

`bool bEnableWait`
Enable or disable COP in WAIT mode

`bool bEnableLossOfReference`
Enable or disable COP loss of reference counter

`bool bEnableInterrupt`
Enables or disables COP interrupt

`bool bEnableCOP`
Enables or disables COP module

`cop_clock_source_t eClockSource`
Set COP clock source

`cop_clock_prescaler_t ePrescaler`
Set COP clock prescaler

`uint16_t u16TimeoutCount`
Timeout count in clock cycles, Use macro definition MSEC_TO_COUNT to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

`uint16_t u16WindowCount`
Window count in clock cycles, Use macro definition MSEC_TO_COUNT to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

`uint16_t u16InterruptCount`
Interrupt count in clock cycles, Use macro definition MSEC_TO_COUNT to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

2.10 The Driver Change Log

2.11 COP Peripheral and Driver Overview

2.12 CRC: Cyclic Redundancy Check Driver

```
void CRC_Init(CRC_Type *base, const crc_config_t *psConfig)
```

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.
- psConfig – CRC module configuration structure.

```
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.

```
static inline void CRC_GetDefaultConfig(crc_config_t *psConfig, crc_protocol_type_t eCrcProtocol)
```

Provide default CRC protocol configuration.

The purpose of this API is to initialize the configuration structure to default value for `CRC_Init` to use. Provides the configuration of commonly used CRC protocols. refer to `crc_protocol_type_t`.

This is an example:

```

crc_config_t sConfig;
//LoadCRC-16/MAXIM protocol configuration.
CRC_GetDefaultConfig(&sConfig, kCRC_Crc16);
CRC_Init(CRC, &sConfig);

```

Parameters

- psConfig – CRC protocol configuration structure.
- eCrcProtocol – CRC protocol type. refer to `crc_protocol_type_t`

```
static inline void CRC_SetSeedValue(CRC_Type *base, uint32_t u32CrcSeedValue)
```

Set the CRC seed value.

This function is help to write a 16/32 bit CRC seed value.

Parameters

- base – CRC peripheral address.
- u32CrcSeedValue – The value of seed.

```
static inline void CRC_SetPolynomial(CRC_Type *base, uint32_t u32CrcPolynomial)
```

Set the value of the polynomial for the CRC calculation.

Write a 16-bit or 32-bit polynomial to CRC Polynomial register for the CRC calculation.

Parameters

- base – CRC peripheral address.
- u32CrcPolynomial – The CRC polynomial.

```
static inline void CRC_SetWriteTransposeType(CRC_Type *base, crc_transpose_type_t eTransposeIn)
```

Set CRC type of transpose of write data.

This function help to configure CRC type of transpose of write data.

Parameters

- base – CRC peripheral address.
- eTransposeIn – Type Of transpose for input. See `crc_transpose_type_t`

```
static inline void CRC_SetReadTransposeType(CRC_Type *base, crc_transpose_type_t  
eTransposeOut)
```

Set CRC type of transpose of read data.

This function help to configure CRC type of transpose of read data.

Parameters

- base – CRC peripheral address.
- eTransposeOut – Type Of transpose for output. See `crc_transpose_type_t`

```
static inline void CRC_EnableComplementChecksum(CRC_Type *base, bool bEnable)
```

Enable/Disable complement of read CRC checksum.

Set complement of read CRC checksum. Some CRC protocols require the final checksum to be XORed with 0xFFFFFFFF or 0xFFFF.

Parameters

- base – CRC peripheral address.
- bEnable – True or false. True if the result shall be complement of the actual checksum.

```
static inline void CRC_SetProtocolWidth(CRC_Type *base, crc_bits_t eCrcBits)
```

Set bit width of CRC protocol.

Selects 16-bit or 32-bit CRC protocol.

Parameters

- base – CRC peripheral address.
- eCrcBits – 16 or 32 bit CRC protocol. See `crc_bits_t`

```
void CRC_WriteData(CRC_Type *base, const uint8_t *pu8Data, uint32_t u32DataSize)
```

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.
- pu8Data – Input data stream, MSByte in data[0].
- u32DataSize – Size in bytes of the input data buffer.

```
static inline 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 transpose and complement operations configured.

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 transpose and complement operations configured.

FSL_CRC_DRIVER_VERSION

CRC driver version. Version.

enum _crc_protocol_type

CRC protocol type.

Values:

enumerator kCRC_Crc16

CRC-16/MAXIM protocol.

enumerator kCRC_Crc16CCITT

CRC-16-CCITT protocol.

enumerator kCRC_Crc16Kermit

CRC-16/KERMIT protocol.

enumerator kCRC_Crc32

CRC-32 protocol.

enumerator kCRC_Crc32Posix

CRC-32/POSIX protocol.

enum _crc_bits

CRC protocol bit width.

Values:

enumerator kCRC_Bits16

Generate 16-bit CRC code.

enumerator kCRC_Bits32

Generate 32-bit CRC code.

enum _crc_transpose_type

CRC type of transpose of read/write data.

Values:

enumerator kCRC_TransposeNone

No transpose.

enumerator kCRC_TransposeBits

Transpose bits in bytes.

enumerator kCRC_TransposeBitsAndBytes

Transpose bytes and bits in bytes.

enumerator kCRC_TransposeBytes

Transpose bytes.

typedef enum *_crc_protocol_type* crc_protocol_type_t
CRC protocol type.

typedef enum *_crc_bits* crc_bits_t
CRC protocol bit width.

typedef enum *_crc_transpose_type* crc_transpose_type_t
CRC type of transpose of read/write data.

typedef struct *_crc_config* crc_config_t
CRC protocol configuration.
This structure holds the configuration for the CRC protocol.

struct *_crc_config*
#include <fsl_crc.h> CRC protocol configuration.
This structure holds the configuration for the CRC protocol.

Public Members

uint32_t u32CrcPolynomial
CRC Polynomial, MSBit first. Example polynomial: $0x1021 = 1_0000_0010_0001 = x^{12} + x^5 + 1$

uint32_t u32CrcSeedValue
Starting checksum value

bool bEnableComplementChecksum
Enable/Disable complement of read CRC checksum.

crc_transpose_type_t eTransposeIn
Select type of transpose of input data.

crc_transpose_type_t eTransposeOut
Select type of transpose of output data.

crc_bits_t eCrcBits
Select 16-bit or 32-bit CRC protocol.

2.13 The Driver Change Log

2.14 CRC Peripheral and Driver Overview

2.15 DAC: 12-bit Digital-to-Analog Converter Driver

void DAC_Init(DAC_Type *base, const *dac_config_t* *psConfig)
Initializes the DAC resource, including data format, sync signal, operation mode, etc.

Parameters

- base – DAC peripheral base address.
- psConfig – The pointer to *dac_config_t*.

```
void DAC_Deinit(DAC_Type *base)
```

De-initializes the DAC resource, the clock and power will be gated off.

Invoking this function to power down the analog portion of DAC and disable the DAC clock.

Parameters

- base – DAC peripheral base address.

```
void DAC_GetDefaultConfig(dac_config_t *psConfig)
```

Gets the default DAC configs, such as operation mode, watermark level, sync signal, etc.

```
psConfig->eOperationMode = kDAC_NormalOperationMode;
psConfig->uOperationConfig.sNormalModeConfig.u16DataFIFO = 0U;
psConfig->bEnableDMA = false;
psConfig->eWatermarkLevel = kDAC_WatermarkValue2;
psConfig->eSyncInputEdge = kDAC_RisingEdge;
psConfig->eSpeedMode = kDAC_HighSpeedMode;
psConfig->eDataFormat = kDAC_DataWordRightJustified;
psConfig->eSyncSignal = kDAC_InternalClockSignal;
psConfig->bEnableAnalogPortion = false;
psConfig->bEnableGlitchFilter = true;
psConfig->u8GlitchFilterCount = 29U;
```

Parameters

- psConfig – The pointer to `dac_config_t`.

```
static inline void DAC_SetSyncEdge(DAC_Type *base, dac_sync_input_edge_t eSyncEdge)
```

Selects which SYNC input edge is used for updates, available selections are “no active edge”, “falling edge”, “rising edge”, “both edge”.

Parameters

- base – DAC peripheral base address.
- eSyncEdge – The input edge to be set, please refer to `dac_sync_input_edge_t` for details.

```
static inline void DAC_SetLDOK(DAC_Type *base)
```

Updates the buffered value of `stepSize`, `minValue`, and `maxValue` at the active edge of the SYNC_IN signal.

Note: Allows new values of minimum, maximum, and step value to be updated by active edge of SYNC_IN. This function should be invoked once new values of these buffered registers have been written by software. The LDOK bit will be cleared by an active edge of SYNC_IN.

Note: This function is only useful when the operation mode is selected as Automatic operation mode.

Parameters

- base – DAC peripheral base address.

```
static inline bool DAC_GetLDOKValue(DAC_Type *base)
```

Gets the value of load Okay bit field.

Note: When the SYNC signal is selected as external SYNC_IN signal, the load okay bit will be cleared by an active edge of the SYNC_IN signal. This function can be used to check whether the active edge of the SYNC_IN signal has reached.

Note: This function is only useful when the operation mode is selected as Automatic operation mode.

Parameters

- `base` – DAC peripheral base address.

Returns

- **true** The active edge of SYNC_IN signal has not reached when the SYNC signal is selected as external SYNC_IN signal.
- **false** The active edge of SYNC_IN signal has reached when the SYNC signal is selected as external SYNC_IN signal

```
static inline void DAC_EnableOneShot(DAC_Type *base, bool bEnable)
```

Enables/Disables Oneshot feature, oneshot feature used to determines whether automatic waveform generation creates one waveform or a repeated waveform within the period defined by the active SYNC edges.

Note: This function only useful when the operation mode is selected as automatic operation mode.

Parameters

- `base` – DAC peripheral base address.
- `bEnable` – Enable/Disable oneshot feature.
 - **true** Automatic waveform generation logic will create a single pattern and stop at the final value.
 - **false** Automatic waveform generation logic will create a repeated (continuous) waveform upon receiving an active SYNC edge.

```
static inline void DAC_WriteDataFIFO(DAC_Type *base, uint16_t u16Data)
```

Writes DAC buffered data value based on the data format when the DAC is in normal operation mode.

Parameters

- `base` – DAC peripheral base address.
- `u16Data` – The DAC data to be converted to analog. If the data format is set as `kDAC_DataWordRightJustified` then `u16Data` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16Data` should range from 16 to 65520, which means the lower 4 bits is useless.

```
static inline void DAC_WriteStepSize(DAC_Type *base, uint16_t u16StepSize)
```

Writes Step size based on the data format when the DAC is in automatic operation mode.

Note: This function only useful when the operation mode is selected as automatic operation mode.

Parameters

- `base` – DAC peripheral base address.
- `u16StepSize` – The step value to be added to or subtracted from the current value. If the data format is set as `kDAC_DataWordRightJustified` then `u16StepSize` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16StepSize` should range from 16 to 65520, which means the lower 4 bits is useless.

```
static inline void DAC_WriteMinValue(DAC_Type *base, uint16_t u16MinValue)
```

Writes the minimum value based on the data format when the DAC is in automatic operation mode.

Note: This function only useful when the operation mode is selected as automatic operation mode. If DAC input data is less than the minimum value, output is limited to the minimum value during automatic waveform generation.

Parameters

- `base` – DAC peripheral base address.
- `u16MinValue` – The lower range limit during automatic waveform generation. If the data format is set as `kDAC_DataWordRightJustified` then `u16MinValue` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16MinValue` should range from 16 to 65520, which means the lower 4 bits is useless.

```
static inline void DAC_WriteMaxValue(DAC_Type *base, uint16_t u16MaxValue)
```

Writes the maximum value based on the data format when the DAC is in automatic operation mode.

Note: This function only useful when the operation mode is selected as automatic operation mode. If DAC input data is greater than maximum value, output is limited to maximum value during automatic waveform generation.

Parameters

- `base` – DAC peripheral base address.
- `u16MaxValue` – The upper range limit during automatic waveform generation. If the data format is set as `kDAC_DataWordRightJustified` then `u16MaxValue` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16MaxValue` should range from 16 to 65520, which means the lower 4 bits is useless.

```
static inline void DAC_ConfigRefreshFrequency(DAC_Type *base, uint16_t u16CompareValue)
```

Sets refresh frequency that used to decide when the automatically generated waveform value is updated.

Note: This function only useful when the operation mode is selected as automatic operation mode.

Parameters

- base – DAC peripheral base address.
- u16CompareValue – The compare value(0~65535).
 - **u16CompareValue=0** The generated waveform will be updated every clock cycle.
 - **u16CompareValue=N(N!=0)** The generated waveform will be updated every N+1 clock cycles.

static inline void DAC_EnableDMA(DAC_Type *base, bool bEnable)

Enables/Disables DMA request that to be generated when the FIFO is below the watermark level.

Note: This function is only useful when the operation mode is selected as Normal mode.

Parameters

- base – DAC peripheral base address.
- bEnable – Enable/Disable DMA support.
 - **true** Enable DMA support.
 - **false** Disable DMA support.

static inline void DAC_SetWatermarkLevel(DAC_Type *base, dac_watermark_level_t eWatermarkLevel)

Sets watermark level which is used for asserting a DMA request.

Note: When the level of FIFO is less than or equal to the Watermark level, a DMA request will be sent. This function is only useful when the operation mode is selected as Normal mode.

Parameters

- base – DAC peripheral base address.
- eWatermarkLevel – The watermark level of FIFO, please refer to dac_watermark_level_t for details.

static inline void DAC_EnableGlitchFilter(DAC_Type *base, bool bEnable)

Enables/Disables Glitch filter.

Parameters

- base – DAC peripheral base address.
- bEnable – Enable/Disable Glitch filter.
 - **true** Enable glitch filter.
 - **false** Disable glitch filter.

static inline void DAC_SetGlitchFilterCount(DAC_Type *base, uint8_t u8FilterCount)

Sets glitch filter count value(ranges from 0 to 63) that represents the number of clock cycles for which the DAC output is held unchanged after new data is presented to the analog DAC's inputs.

Parameters

- base – DAC peripheral base address.

- `u8FilterCount` – The count of glitch filter. This count represents the number of clock cycles for which the DAC output is held unchanged after new data is presented to the analog DAC's inputs.

```
static inline void DAC_SetSpeedMode(DAC_Type *base, dac_speed_mode_t eSpeedMode)
```

Selects speed mode, high speed mode uses more power and low speed mode saves power.

Parameters

- `base` – DAC peripheral base address.
- `eSpeedMode` – The speedMode to be set, please refer to `dac_speed_mode_t` for details.

```
static inline void DAC_EnableAnalogPortion(DAC_Type *base, bool bEnable)
```

Enables/Disables the operation of the analog portion of the DAC.

The function controls the power-up of the analog portion of the DAC. If powered up the analog portion, the DAC module will output the value currently presented to the Data register. The analog portion should be powered up when the DAC is in use. If power down the analog portion, the output of the DAC module will be pulled low. The analog portion should be powered down when the DAC is not in use.

Parameters

- `base` – DAC peripheral base address.
- `bEnable` – Power up/down the analog portion of the DAC.
 - **true** Power up the analog portion of the DAC, and the DAC will output the value currently presented to its inputs.
 - **false** Power down the analog portion of the DAC, and its output will be pulled down.

```
static inline uint16_t DAC_GetFIFOStatusFlags(DAC_Type *base)
```

Gets the fifo status flag of selected DAC instance.

Parameters

- `base` – DAC peripheral base address.

Returns

The status flags of DAC module, should be the OR'ed value of `_dac_fifo_status_flags`.

```
FSL_DAC_DRIVER_VERSION
```

DAC driver version.

```
enum _dac_fifo_status_flags
```

The enumeration of DAC status flags, including FIFO full status flag and FIFO empty status flag.

Values:

```
enumerator kDAC_FIFOFullStatusFlag
```

Indicate the FIFO is full.

```
enumerator kDAC_FIFOEmptyStatusFlag
```

Indicate the FIFO is empty.

```
enum _dac_operation_mode
```

The enumeration of DAC operation mode, including normal operation mode and automatic operation mode.

Values:

enumerator kDAC_NormalOperationMode

Normal Mode to generate an analog representation of digital words.

enumerator kDAC_AutomaticOperationMode

Automatic Mode to generate waveform without requiring CPU or core assistance.

enum _dac_sync_signal_selection

The enumeration of DAC sync signal mode, including internal clock signal and external SYNC_IN signal.

Values:

enumerator kDAC_InternalClockSignal

Internal Clock signal is selected as SYNC signal, data written to the buffered registers is used on the next clock cycle

enumerator kDAC_ExternalSyncInSignal

Peripheral external signal is selected as SYNC signal, the update occurs on the active edge of SYNC_IN signal.

enum _dac_waveform_type

The enumeration of waveform type, such as square waveform, triangle waveform, etc.

Values:

enumerator kDAC_RepeatSawtoothWaveform0

DAC generates repeated sawtooth waveform0. The automatic waveform generation logic will create a repeated sawtooth waveform0 upon receiving an active SYNC edge, and the waveform repeats when it reaches its minimum and maximum value. The waveform increases from starting value to max value firstly. Like this following shown:



enumerator kDAC_RepeatSawtoothWaveform1

DAC generates sawtooth waveform1. The automatic waveform generation logic will create a repeated sawtooth waveform1 upon receiving an active SYNC edge, and the waveform repeats when it reaches its minimum and maximum value. The waveform decreases from starting value to min value firstly. Like this following shown:



enumerator kDAC_RepeatTriangleWaveform0

The automatic waveform generation logic will create a repeated triangle waveform0 upon receiving an active SYNC edge, and the waveform repeats when it reaches its minimum and maximum value. In this type the generated triangle waveform rises from the starting value. Like this following shown:



enumerator kDAC_RepeatTriangleWaveform1

The automatic waveform generation logic will create a repeated triangle waveform1 upon receiving an active SYNC edge, and the waveform repeats when it reaches its

minimum and maximum value. In this type the generated triangle waveform drops from the starting value. Like this following shown:



enumerator `kDAC_OneShotSawtoothWaveform0`

Automatic waveform generation logic will create a single pattern and stop at the final value. It will remain at the final value until a new active edge occurs on the SYNC input, and then the waveform will be repeated. Like this following shown:



enumerator `kDAC_OneShotSawtoothWaveform1`

Automatic waveform generation logic will create a single pattern and stop at the final value. It will remain at the final value until a new active edge occurs on the SYNC input, and then the waveform will be repeated. Like this following shown:



enum `_dac_sync_input_edge`

The enumeration of sync input edge that used for updates buffered registers, such as Falling edge, etc.

Values:

enumerator `kDAC_NoActiveEdge`

No active edge is selected, it means the SYNC input is ignored.

enumerator `kDAC_FallingEdge`

Updates occur on the falling edge of the SYNC input.

enumerator `kDAC_RisingEdge`

Updates occur on the rising edge of the SYNC input.

enumerator `kDAC_BothEdges`

Updates occur on both edges of the SYNC input.

enum `_dac_speed_mode`

The enumeration of DAC speed mode, including high speed mode and low speed mode.

Values:

enumerator `kDAC_HighSpeedMode`

In High Speed Mode, the setting time of the DAC module is 1us, but the DAC module uses more power.

enumerator `kDAC_LowSpeedMode`

In Low Speed Mode, the DAC module uses less power but takes more time to settle.

enum `_dac_watermark_level`

The enumeration of FIFO watermark level.

Values:

enumerator `kDAC_WatermarkValue0`

Watermark value is 0.

enumerator `kDAC_WatermarkValue2`

Watermark value is 2.

enumerator `kDAC_WatermarkValue4`

Watermark value is 4.

enumerator `kDAC_WatermarkValue6`

Watermark value is 6

enum `_dac_data_format`

The enumeration of DAC data format, including right right-justified and left-justified.

Values:

enumerator `kDAC_DataWordRightJustified`

The 12 bits data is right-justified.

enumerator `kDAC_DataWordLeftJustified`

The 12 bits data is left-justified.

typedef enum `_dac_operation_mode` `dac_operation_mode_t`

The enumeration of DAC operation mode, including normal operation mode and automatic operation mode.

typedef enum `_dac_sync_signal_selection` `dac_sync_signal_selection_t`

The enumeration of DAC sync signal mode, including internal clock signal and external SYNC_IN signal.

typedef enum `_dac_waveform_type` `dac_waveform_type_t`

The enumeration of waveform type, such as square waveform, triangle waveform, etc.

typedef enum `_dac_sync_input_edge` `dac_sync_input_edge_t`

The enumeration of sync input edge that used for updates buffered registers, such as Falling edge, etc.

typedef enum `_dac_speed_mode` `dac_speed_mode_t`

The enumeration of DAC speed mode, including high speed mode and low speed mode.

typedef enum `_dac_watermark_level` `dac_watermark_level_t`

The enumeration of FIFO watermark level.

typedef enum `_dac_data_format` `dac_data_format_t`

The enumeration of DAC data format, including right right-justified and left-justified.

typedef struct `_dac_normal_mode_config` `dac_normal_mode_config_t`

The structure of configuration when the operation mode is selected as normal operation mode.

typedef struct `_dac_automatic_mode_config` `dac_automatic_mode_config_t`

The structure of configuration when the operation mode is selected as automatic operation mode.

typedef union `_dac_operation_config` `dac_operation_config_u`

The union of operation modes' configuration.

typedef struct `_dac_config` `dac_config_t`

The structure for configuring the DAC.

This structure is used to config the DAC module, to initialize the DAC module, user must set the member of this structure. This structure will cost 20 Byte memory space.

`struct _dac_normal_mode_config`

#include <fsl_dac.h> The structure of configuration when the operation mode is selected are normal operation mode.

Public Members

`bool bEnableDMA`

Enable/Disable DMA support.

- **true** Enable DMA support.
- **false** Disable DMA support.

`uint16_t u16DataFIFO`

The FIFO watermark level, if the level of FIFO is less than or equal to the watermark level field, a DMA request should be sent. The DAC data to be converted to analog. If the data format is set as `kDAC_DataWordRightJustified` then `u16DataFIFO` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16DataFIFO` should range from 16 to 65520, which means the lower 4 bits is useless.

`struct _dac_automatic_mode_config`

#include <fsl_dac.h> The structure of configuration when the operation mode is selected as automatic operation mode.

Public Members

`dac_waveform_type_t eWaveformType`

The type of waveform to be generated.

`uint16_t u16MinValue`

The step size to be added to or subtracted from the current value. If the data format is set as `kDAC_DataWordRightJustified` then `u16StepSize` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16StepSize` should range from 16 to 65520, which means the lower 4 bits is useless.

The minimum value is the lower range limit during automatic waveform generation. If the data format is set as `kDAC_DataWordRightJustified` then `u16MinValue` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16MinValue` should range from 16 to 65520, which means the lower 4 bits is useless.

`uint16_t u16MaxValue`

The maximum value is the upper range limit during automatic waveform generation. If the data format is set as `kDAC_DataWordRightJustified` then `u16MaxValue` should range from 0 to 4095, which means the higher 4 bits is useless. If the data format is set as `kDAC_DataWordLeftJustified`, then the `u16MaxValue` should range from 16 to 65520, which means the lower 4 bits is useless.

`uint16_t u16StartValue`

The start value of the waveform, should larger than the minimum value and smaller than the maximum value.

`uint16_t u16CompareValue`

The compare value that used to decide the frequency of REFRESH signal. The available range is 0 ~ 65535.

- **u16CompareValue=0** The REFRESH signal's frequency is equal to the clock's frequency so that the generated waveform will be updated every clock cycle.

- **u16CompareValue=N(N!=0)** The REFRESH signal's frequency is equal to clock's frequency divided N+1 so that the generated waveform will be updated every N+1 clock cycles

union `_dac_operation_config`

`#include <fsl_dac.h>` The union of operation modes' configuration.

Public Members

`dac_normal_mode_config_t` `sNormalModeConfig`

The configuration of normal operation mode, such as buffered data, watermark level, etc.

`dac_automatic_mode_config_t` `sAutomaticModeConfig`

The configuration of automatic operation mode, such as step size, minimum value, maximum value, etc.

struct `_dac_config`

`#include <fsl_dac.h>` The structure for configuring the DAC.

This structure is used to config the DAC module, to initialize the DAC module, user must set the member of this structure. This structure will cost 20 Byte memory space.

Public Members

`dac_operation_mode_t` `eOperationMode`

The operation mode. The available selections are `kDAC_NormalOperationMode` and `kDAC_AutomaticOperationMode`.

`dac_sync_signal_selection_t` `eSyncSignal`

The selected sync signal that used to update buffered data, the available selections are `kDAC_InternalClockSignal` and `kDAC_ExternalSyncInSignal`

`dac_sync_input_edge_t` `eSyncInputEdge`

The SYNC input edge used to update buffered registers. The buffered value will be updated at the selected active edge of SYNC_IN signal.

`dac_data_format_t` `eDataFormat`

The data format of DAC instance. The available selections are `kDAC_DataWordRightJustified` and `kDAC_DataWordLeftJustified`

`dac_operation_config_u` `uOperationConfig`

The configuration of operation mode.

bool `bEnableGlitchFilter`

Enable/Disable glitch suppression filter.

- **true** Enable glitch filter.
- **false** Disable glitch filter.

uint8_t `u8GlitchFilterCount`

The count(ranges from 0 to 63) represents the number of clock cycles for which the DAC output is held unchanged after new data is presented to the analog DAC's inputs.

`dac_speed_mode_t` `eSpeedMode`

The speed mode of DAC instance. The available selections are `kDAC_HighSpeedMode` and `kDAC_LowSpeedMode`.

bool bEnableAnalogPortion

Power up/down the analog portion.

- **true** Power up the analog portion of the DAC, and the DAC will output the value currently presented to its inputs.
- **false** Power down the analog portion of the DAC, and its output will be pulled down.

2.16 The Driver Change Log

2.17 DAC Peripheral and Driver Overview

2.18 DMAMUX: DMA Channel Multiplexer Driver

```
static inline void DMAMUX_ConnectChannelToTriggerSource(DMAMUX_Type *base,
                                                       dmamux_dma_channel_t eChannel,
                                                       dma_request_source_t eSource)
```

Connect the DMAMUX channel to trigger source.

This function will connect a source to the specify dma channel and enable that channel

Parameters

- base – DMAMUX peripheral base address.
- eChannel – DMAMUX channel index, dmamux_dma_channel_t.
- eSource – DMA request source.

```
static inline void DMAMUX_DisconnectChannelFromTriggerSource(DMAMUX_Type *base,
                                                            dmamux_dma_channel_t
                                                            eChannel)
```

Disconnect the DMAMUX channel.

This function will disable the specified channel and reset the channel source.

Parameters

- base – DMAMUX peripheral base address.
- eChannel – DMAMUX channel index, dmamux_dma_channel_t.

FSL_DMAMUX_DRIVER_VERSION

DMAMUX driver version.

enum _dmamux_dma_channel

List of Dmamux dma channels.

Values:

enumerator kDMAMUX_DMACHannel0

Dmamux dma channel 0

enumerator kDMAMUX_DMACHannel1

Dmamux dma channel 1

enumerator kDMAMUX_DMACHannel2

Dmamux dma channel 2

```
enumerator kDMAMUX_DMACHannel3
    Dmamux dma channel 3
typedef enum _dmamux_dma_channel dmamux_dma_channel_t
    List of Dmamux dma channels.
```

2.19 The Driver Change Log

2.20 DMAMUX Peripheral and Driver Overview

2.21 EDMA: Enhanced Direct Memory Access Driver

```
void EDMA_GetDefaultConfig(edma_config_t *psConfig)
    Get default edma peripheral configuration.
```

Note: This function will reset all of the configuration structure members to zero firstly, then apply default configurations to the structure.

Parameters

- *psConfig* – pointer to user's eDMA config structure, see *edma_config_t* for detail.

```
void EDMA_Init(DMA_Type *base, edma_config_t *psConfig)
    EDMA initialization.
```

Parameters

- *base* – eDMA peripheral base address.
- *psConfig* – pointer to user's eDMA config structure, see *edma_transfer_config_t* for detail.

```
void EDMA_Deinit(DMA_Type *base)
    EDMA De-initialization.
```

Parameters

- *base* – eDMA peripheral base address.

```
static inline void EDMA_EnableContinuousChannelLinkMode(DMA_Type *base, bool bEnable)
    Enable/Disable arbitration before the channel been activate by minor loop link trigger from itself.
```

A minor loop channel link made to itself does not go through channel arbitration before being activated again. Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself. This effectively applies the minor loop offsets and restarts the next minor loop.

Note: Do not use continuous link mode with a channel linking to itself if there is only one minor loop iteration per service request, for example, if the channel's NBYTES value is the same as either the source or destination size. The same data transfer profile can be achieved by simply increasing the NBYTES value, which provides more efficient, faster processing.

Parameters

- base – EDMA peripheral base address.
- bEnable – true is channel link to itself without arbitration false is channel link to itself with arbitration

```
static inline void EDMA_EnableMinorLoopMapping(DMA_Type *base, bool bEnable)
```

Enable/Disable redefine the minor loop bytes register.

The TCDn.word2 is redefined to include individual enable fields, an offset field and the NBYTES field, the offset will be applied to source/destination address after minor loop complete

Parameters

- base – EDMA peripheral base address.
- bEnable – true is minor loop bytes register redefined to individual enable/offset/minor loop bytes fields. false is minor loop bytes register defined as minor loop bytes fields only.

```
static inline void EDMA_EnableHaltOnError(DMA_Type *base, bool bEnable)
```

Enable/Disable the eDMA halt when error occur feature.

Any error causes the HALT bit to set will cause the EDMA halt. Subsequently, all service requests are ignored until the HALT bit is cleared

Parameters

- base – EDMA peripheral base address.
- bEnable – true is Stall the start of any new channels when error occur. false is eDMA service request operation normal when error occur.

```
static inline void EDMA_SetArbitration(DMA_Type *base, edma_arbitration_type_t eArbitration)
```

set EDMA arbitration type to fixed priority or round robin.

Parameters

- base – EDMA peripheral base address.
- eArbitration – Arbitration by priority or round robin, edma_arbitration_type_t.

```
void EDMA_GetChannelDefaultTransferConfig(edma_channel_transfer_config_t *psTransfer,
uint32_t u32SrcAddr, uint32_t u32DstAddr,
uint32_t u32BytesEachRequest, uint32_t
u32TotalBytes, edma_channel_transfer_width_t
eTransferWidth, edma_channel_transfer_type_t
eTransferType)
```

Get channel default transfer configuration.

Note: 1. This function will reset all of the configuration structure members to zero firstly, then apply default configurations to the structure.

- a. No interrupt enabled by this function by default, if application would like to use DMA interrupt please enable it manually by psTransfer->u16EnabledInterruptMask = _edma_channel_interrupt
-

Parameters

- psTransfer – pointer to user's eDMA channel configure structure, see edma_channel_transfer_config_t for detail.
- u32SrcAddr – source address, must be byte address.

- u32DstAddr – destination address, must be byte address.
- u32BytesEachRequest – bytes to be transferred in each request (namely, in each minor loop).
- u32TotalBytes – total bytes to be transferred.
- eTransferWidth – it represents how many bits are transferred in each read/write.
- eTransferType – eDMA channel transfer type.

```
void EDMA_SetChannelTransferConfig(DMA_Type *base, edma_channel_t eChannel,  
                                  edma_channel_transfer_config_t *psTransfer)
```

EDMA set channel transfer configurations.

Note: 1.This function must not be called while the channel transfer is ongoing or it causes unpredictable results. 2.The psLinkTCD must be configured before invoke this API if scatter/gather function is needed 3.The edma channel request may be enabled after the channel transfer configure done according to the transfer configurations.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- psTransfer – pointer to user's eDMA channel configure structure, see edma_channel_transfer_config_t for detail.

```
void EDMA_SetChannelMinorLoopOffset(DMA_Type *base, edma_channel_t eChannel, bool  
                                     bEnableSrcMinorLoopOffset, bool  
                                     bEnableDestMinorLoopOffset, int32_t  
                                     i32MinorLoopOffset)
```

Configures the eDMA channel minor loop offset value.

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- bEnableSrcMinorLoopOffset – True is enable source address minor offset, otherwise is disable
- bEnableDestMinorLoopOffset – True is enable source address minor offset, otherwise is disable
- i32MinorLoopOffset – Minor loop offset value.

```
void EDMA_SetChannelPreemption(DMA_Type *base, edma_channel_t eChannel, bool  
                                bSuspendedByHighPriorityChannel, bool  
                                bSuspendLowPriorityChannel, uint8_t u8Priority)
```

Configures the eDMA channel preemption configurations.

This function configures the channel preemption attribute and the priority of the channel.

Note: , this function is used only in fixed-priority channel arbitration mode.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number
- bSuspendedByHighPriorityChannel – True is the channel can be suspended by high priority channel, otherwise cannot.
- bSuspendLowPriorityChannel – True is the channel can suspend low priority channel, otherwise cannot.
- u8Priority – Channel priority.

```
void EDMA__EnableMinorLoopChannelLink(DMA_Type *base, edma_channel_t eChannel,  
                                       edma_channel_t eLinkChannel)
```

Enable the minor loop channel link and configure the linked channel number.

This function configures the minor link mode. The minor link means that the channel link is triggered every time that the minor loop bytes transferred complete.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- eLinkChannel – The linked channel number.

```
static inline void EDMA__DisableMinorLoopChannelLink(DMA_Type *base, edma_channel_t  
                                                    eChannel)
```

Disable the minor loop channel link for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.

```
void EDMA__EnableMajorLoopChannelLink(DMA_Type *base, edma_channel_t eChannel,  
                                       edma_channel_t eLinkChannel)
```

Enable the major loop channel link and configure the linked channel number.

This function configures the major link mode. The major link means that the channel link is triggered when the CITER is exhausted.

Note: Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- eLinkChannel – The linked channel number.

```
static inline void EDMA__DisableMajorLoopChannelLink(DMA_Type *base, edma_channel_t  
                                                    eChannel)
```

Disable the major loop channel link for the eDMA transfer.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.

```
void EDMA_SetChannelBandWidth(DMA_Type *base, edma_channel_t eChannel,  
                             edma_channel_bandwidth_t eBandWidth)
```

Sets the edma channel stall cycles after each R/W.

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Note: : 1.If the source and destination sizes are equal, this field is ignored between the first and second transfers and after the last write of each minor loop. This behavior is a side effect of reducing start-up latency. 2.When executing a large, zero wait-stated memory-to-memory transfer, insert bandwidth control using the TCD_CSR[BWC] bits to avoid: •* Starvation of another master accessing the memory. •* Any delay in writing a TCD duloop the transfer.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- eBandWidth – A bandwidth setting, which can be one of the *edma_channel_bandwidth_t*

```
void EDMA_SetChannelModulo(DMA_Type *base, edma_channel_t eChannel,  
                           edma_channel_modulo_t eSrcModulo, edma_channel_modulo_t  
                           eDestModulo)
```

Sets the source address range and the destination address range for the eDMA transfer.

This function defines a specific address range of source/destination address, after the source/destination address hits the range boundary, source/destination address will wrap to origin value.

Setting this field provides the ability to implement a circular data queue easily. For data queues require loop power-of-2 size bytes, the queue should start at a 0-modulo-size address and the SMOD field should be set to the appropriate value for the queue, freezing the desired number of upper address bits. The value programmed into this field specifies the number of lower address bits allowed to change

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- eSrcModulo – A source modulo value.
- eDestModulo – A destination modulo value.

```
static inline void EDMA_EnableChannelAsyncRequestInStopMode(DMA_Type *base,  
                                                           edma_channel_t eChannel,  
                                                           bool bEnable)
```

Enables the edma channel async request in stop mode.

The EARS register is used to enable or disable the DMA requests in Enable Request Register (ERQ) by AND'ing the bits of these two registers in stop mode only.

Parameters

- base – eDMA peripheral base address.

- `eChannel` – eDMA channel number.
- `bEnable` – The command to enable (true) or disable (false).

```
static inline void EDMA_EnableChannelAutoStopRequest(DMA_Type *base, edma_channel_t
                                                    eChannel, bool bEnable)
```

Enables the edma channel auto disable request after major loop complete.

The eDMA hardware automatically clears the corresponding ERQ bit when the current major iteration count reaches zero.

Parameters

- `base` – eDMA peripheral base address.
- `eChannel` – eDMA channel number.
- `bEnable` – The command to enable (true) or disable (false).

```
void EDMA_SetChannelMajorLoopOffset(DMA_Type *base, edma_channel_t eChannel, int32_t
                                     i32SourceOffset, int32_t i32DestOffset)
```

Configures the eDMA channel major loop offset feature.

Adjustment value added to the source/destination address at the completion of the major iteration count

Parameters

- `base` – eDMA peripheral base address.
- `eChannel` – edma channel number.
- `i32SourceOffset` – source address offset.
- `i32DestOffset` – destination address offset.

```
static inline void EDMA_EnableChannelRequest(DMA_Type *base, edma_channel_t eChannel,
                                             bool bEnable)
```

Enable/disable the eDMA hardware channel request.

This function enables the hardware channel request.

Parameters

- `base` – eDMA peripheral base address.
- `eChannel` – eDMA channel number.
- `bEnable` – true is start, false is stop.

```
static inline void EDMA_SoftwareTriggerChannelStart(DMA_Type *base, edma_channel_t
                                                    eChannel)
```

Starts the eDMA transfer by using the software trigger.

This function starts a minor loop transfer only, the channel will halt when minor loop complete, so application should re-call the function to start the transfer again.

Parameters

- `base` – eDMA peripheral base address.
- `eChannel` – eDMA channel number.

```
uint32_t EDMA_GetChannelRemainingMajorLoopCount(DMA_Type *base, edma_channel_t
                                                  eChannel)
```

Gets the remaining major loop count from the eDMA current channel TCD.

This function checks the TCD (Transfer Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Note: 1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccurate.

- a. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: $\text{RemainingBytes} = \text{RemainingMajorLoopCount} * \text{NBYTES}(\text{initially configured})$
-

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.

Return values

Major – loop count which has not been transferred yet for the current TCD.

```
void EDMA_EnableChannelInterrupts(DMA_Type *base, edma_channel_t eChannel, uint16_t u16InterruptsMask, bool bEnable)
```

Enables the edma channel interrupts according to a provided mask, the mask is a logical OR of enumerator members `_edma_channel_interrupt_enable`.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.
- u16InterruptsMask – the mask is a logical OR of enumerator members `_edma_channel_interrupt_enable`.
- bEnable – true is enable, false is disable.

```
uint16_t EDMA_GetChannelStatusFlags(DMA_Type *base, edma_channel_t eChannel)
```

Gets the eDMA channel status flags.

Note: if the function return error status, application can call `EDMA_GetErrorStatusFlags` for the detail error status.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.

Return values

The – mask of channel status flags. Users need to use the `_edma_channel_status_flags` type to decode the return variables.

```
void EDMA_ClearChannelStatusFlags(DMA_Type *base, edma_channel_t eChannel, uint16_t u16StatusFlags)
```

Clears the eDMA channel status flags.

Parameters

- base – eDMA peripheral base address.
- eChannel – eDMA channel number.

- `u16StatusFlags` – The mask of channel status to be cleared. Users need to use the defined `_edma_channel_status_flags` type.

```
static inline uint32_t EDMA_GetErrorStatusFlags(DMA_Type *base)
```

Gets the eDMA channel error status flags.

Parameters

- `base` – eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the `_edma_error_status_flags` type to decode the return variables.

```
void EDMA_ConfigChannelSoftwareTCD(edma_channel_tcd_t *psTcd,  
                                   edma_channel_transfer_config_t *psTransfer)
```

Sets TCD fields according to the user's channel transfer configuration structure, `edma_channel_transfer_config_t`.

Application should be careful about the TCD pool buffer storage class,

- For the platform has cache, the software TCD should be put in non cache section
- The TCD pool buffer should have a consistent storage class.

Note: Application should be careful when using the minor loop offset feature with this function, please make sure the EMLM bit is asserted, although `EDMA_InitChannel` will set this bit by default, if the bit is cleared, application can use `EDMA_EnableMinorLoopMapping` to enable the feature.

Note: This function enables the auto stop request feature.

Parameters

- `psTcd` – Pointer to the TCD structure.
- `psTransfer` – channel transfer configuration pointer.

```
void EDMA_InstallChannelSoftwareTCD(DMA_Type *base, edma_channel_t eChannel,  
                                    edma_channel_tcd_t *psTcd)
```

Push content of software TCD structure into hardware TCD register.

Parameters

- `base` – EDMA peripheral base address.
- `eChannel` – EDMA channel number.
- `psTcd` – Point to TCD structure.

```
void EDMA_TransferCreateHandle(DMA_Type *base, edma_handle_t *psHandle, edma_channel_t  
                              eChannel, edma_channel_tcd_t *psTcdPool, uint32_t  
                              u32TcdCount, edma_transfer_callback_t pfCallback, void  
                              *pUserData)
```

Creates the eDMA channel handle.

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

Parameters

- `base` – eDMA peripheral base address.

- psHandle – eDMA handle pointer. The eDMA handle stores callback function and parameters.
- eChannel – eDMA channel number.
- psTcdPool – A memory pool to store TCDs. It must be 32 bytes aligned.
- u32TcdCount – The number of TCD slots.
- pfCallback – eDMA callback function pointer.
- pUserData – A parameter for the callback function.

status_t EDMA_TransferSubmitSingleTransfer(*edma_handle_t* *psHandle,
edma_channel_transfer_config_t *psTransfer)

Submits the eDMA single transfer configuration.

Application can submit single transfer when

- a. channel is idle, the transfer request will be submitted to eDMA channel TCD register directly
- b. channel is idle, a previous transfer request is pending, the new transfer request will be submitted to the installed TCD pool and linked to the pending one.
- c. channel is active, the transfer request will be submitted to the installed TCD pool and linked to previous one.

It is suggest that application should check the return value of this function to make sure that the transfer request is submitted successfully.

Note: , 1.Please be aware of that tcd pool maintain is unprotect by default, that is to say, the behavior of multiple task trying to access the same channel is undefine, application can protect the channel by itself or overwrite EDMA_ENTER_CRITICAL_SECTION/EDMA_LEAVE_CRITICAL_SECTION to have edma driver protect the TCD pool maintain. 2.Since the destination major loop offset feature register is reused as scatter gather tcd address, so the two features cannot be used together, if the destination major loop offset feature is used, then the transfer request will be submit hardware TCD directly.

Parameters

- psHandle – eDMA handle pointer.
- psTransfer – pointer to user's eDMA channel configure structure, see *edma_channel_transfer_config_t* for detail.

Return values

- kStatus_Success – It means submit transfer request succeed.
- kStatus_EDMA_ChannelQueueFull – It means TCD queue is full. Submit transfer request is not allowed.

status_t EDMA_TransferSubmitLoopTransfer(*edma_handle_t* *psHandle,
edma_channel_transfer_config_t *psTransfer,
uint32_t transferLoopCount)

Submits the eDMA scatter gather transfer configurations.

The function is target for submit loop transfer request, the ring transfer request means that the transfer request TAIL is link to HEAD, such as, A->B->C->D->A, or A->A

To use the ring transfer feature, the application should allocate several transfer object, such as

```
edma_channel_transfer_config_t transfer[2];
EDMA_TransferSubmitLoopTransfer(psHandle, &transfer, 2U);
```

Then eDMA driver will link transfer[0] and transfer[1] to each other

Note: Application should check the return value of this function to avoid transfer request submit failed

Parameters

- psHandle – eDMA handle pointer
- psTransfer – pointer to user's eDMA channel configure structure, see edma_channel_transfer_config_t for detail
- transferLoopCount – the count of the transfer ring, if loop count is 1, that means that the one will link to itself.

Return values

- kStatus_Success – It means submit transfer request succeed
- kStatus_EDMA_ChannelBusy – channel is in busy status
- kStatus_EDMA_ChannelQueueFull – It means TCD pool is not len enough for the ring transfer request

```
void EDMA_TransferStart(edma_handle_t *psHandle)
```

eDMA starts transfer.

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

- psHandle – eDMA handle pointer.

```
void EDMA_TransferStop(edma_handle_t *psHandle)
```

eDMA stops transfer.

This function disables the channel request to pause the transfer. Users can call EDMA_StartTransfer() again to resume the transfer.

Parameters

- psHandle – eDMA handle pointer.

```
void EDMA_TransferAbort(edma_handle_t *psHandle)
```

eDMA aborts transfer.

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

- psHandle – DMA handle pointer.

```
void EDMA_TransferHandleIRQ(edma_handle_t *psHandle)
```

eDMA IRQ handler for the current major loop transfer completion.

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

Note: For the case using TCD queue, when the major iteration count is exhausted, additional Interfaces are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this

time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As `sga` and `sga_index` are calculated based on the `DLAST_SGA` bit field lies in the `TCD_CSR` register, the `sga_index` in this case should be 2 (`DLAST_SGA` of TCD[1] stores the address of TCD[2]). Thus, the “`tcdUsed`” updated should be (`tcdUsed - 2U`) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and `tcdUsed` updated are identical for them. `tcdUsed` are both 0 in this case as no TCD to be loaded.

See the “eDMA basic data flow” in the eDMA Functional description section of the Reference Manual for further details.

Parameters

- `psHandle` – eDMA handle pointer.

`FSL_EDMA_DRIVER_VERSION`

EDMA driver version.

`_edma_transfer_status` eDMA transfer status The enumerator used for transactional interface only.

Values:

enumerator `kStatus_EDMA_ChannelQueueFull`

TCD queue is full.

enumerator `kStatus_EDMA_ChannelBusy`

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

enum `_edma_channel_transfer_type`

eDMA transfer type

Values:

enumerator `kEDMA_ChannelTransferMemoryToMemory`

Transfer type from memory to memory assume that the both source and destination address are incremental

enumerator `kEDMA_ChannelTransferPeripheralToMemory`

Transfer type peripher to memory assume that the source address is fixed

enumerator `kEDMA_ChannelTransferMemoryToPeripheral`

Transfer type from memory to peripheral assume that the destination address is fixed

enumerator `kEDMA_ChannelTransferPeripheralToPeripheral`

Transfer type from Peripheral to peripheral assume that both source and destination address are fixed

enum `_edma_channel_interrupt_enable`

eDMA interrupt source

The eDMA peripheral support generate interrupt when half of the total request bytes transferred or all of the request bytes transferred.

Values:

enumerator kEDMA_ChannelErrorInterruptEnable
Enable error interrupt

enumerator kEDMA_ChannelMajorLoopCompleteInterruptEnable
Enable interrupt while major count exhausted.

enumerator kEDMA_ChannelMajorLoopHalfCompleteInterruptEnable
Enable interrupt while major count to half value.

enumerator kEDMA_ChannelAllInterruptEnable
Enable all the interrupt.

enum _edma_channel_status_flags

_edma_channel_status_flags eDMA channel status flags.

Values:

enumerator kEDMA_ChannelStatusErrorFlag
eDMA error flag, an error occurred in a transfer

enumerator kEDMA_ChannelStatusMajorLoopCompleteFlag
Major loop complete flag, set while transfer finished, CITER value exhausted

enumerator kEDMA_ChannelStatusMajorLoopHalfCompleteFlag
Major loop half complete flag

enum _edma_error_status_flags

_edma_error_status_flags eDMA channel detail error status flags.

Values:

enumerator kEDMA_ChannelDestinationBusErrorFlag
Bus error on destination address

enumerator kEDMA_ChannelSourceBusErrorFlag
Bus error on the source address

enumerator kEDMA_ChannelScatterGatherErrorFlag
Error on the Scatter/Gather address, not 32byte aligned.

enumerator kEDMA_ChannelNbytesErrorFlag
NBYTES/CITER configuration error

enumerator kEDMA_ChannelDestinationOffsetErrorFlag
Destination offset not aligned with destination size

enumerator kEDMA_ChannelDestinationAddressErrorFlag
Destination address not aligned with destination size

enumerator kEDMA_ChannelSourceOffsetErrorFlag
Source offset not aligned with source size

enumerator kEDMA_ChannelSourceAddressErrorFlag
Source address not aligned with source size

enumerator kEDMA_ChannelErrorChannelFlag
Error channel number of the canceled channel number

enumerator kEDMA_ChannelPriorityErrorFlag
Channel priority is not unique.

enumerator kEDMA_ChannelTransferCanceledFlag
Transfer canceled

enumerator kEDMA_ChannelValidFlag

No error occurred, this bit is 0. Otherwise, it is 1.

enum _edma_arbitration_type

eDMA arbitration type

Values:

enumerator kEDMA_ArbitrationFixedPriority

channel arbitration by fixed priority

enumerator kEDMA_ArbitrationRoundRobin

Channel arbitration by round robin

enum _edma_channel

edma channel index

Values:

enumerator kEDMA_Channel0

EDMA channel 0

enumerator kEDMA_Channel1

EDMA channel 1

enumerator kEDMA_Channel2

EDMA channel 2

enumerator kEDMA_Channel3

EDMA channel 3

enum _edma_channel_transfer_width

eDMA transfer width configuration

Values:

enumerator kEDMA_ChannelTransferWidth8Bits

Source/Destination data transfer width is 1 byte every time

enumerator kEDMA_ChannelTransferWidth16Bits

Source/Destination data transfer width is 2 bytes every time

enumerator kEDMA_ChannelTransferWidth32Bits

Source/Destination data transfer width is 4 bytes every time

enumerator kEDMA_ChannelTransferWidth128Bits

Source/Destination data transfer size is 16 bytes every time

enum _edma_channel_modulo

eDMA channel modulo configuration

The eDMA modulo feature can be used to specify the address range of the source/destination address, it is useful to implement a circular data queue.

Values:

enumerator kEDMA_ChannelModuloDisable

Disable modulo

enumerator kEDMA_ChannelModulo2bytes

Circular buffer size is 2 bytes.

enumerator kEDMA_ChannelModulo4bytes

Circular buffer size is 4 bytes.

enumerator kEDMA_ChannelModulo8bytes
Circular buffer size is 8 bytes.

enumerator kEDMA_ChannelModulo16bytes
Circular buffer size is 16 bytes.

enumerator kEDMA_ChannelModulo32bytes
Circular buffer size is 32 bytes.

enumerator kEDMA_ChannelModulo64bytes
Circular buffer size is 64 bytes.

enumerator kEDMA_ChannelModulo128bytes
Circular buffer size is 128 bytes.

enumerator kEDMA_ChannelModulo256bytes
Circular buffer size is 256 bytes.

enumerator kEDMA_ChannelModulo512bytes
Circular buffer size is 512 bytes.

enumerator kEDMA_ChannelModulo1Kbytes
Circular buffer size is 1 K bytes.

enumerator kEDMA_ChannelModulo2Kbytes
Circular buffer size is 2 K bytes.

enumerator kEDMA_ChannelModulo4Kbytes
Circular buffer size is 4 K bytes.

enumerator kEDMA_ChannelModulo8Kbytes
Circular buffer size is 8 K bytes.

enumerator kEDMA_ChannelModulo16Kbytes
Circular buffer size is 16 K bytes.

enumerator kEDMA_ChannelModulo32Kbytes
Circular buffer size is 32 K bytes.

enumerator kEDMA_ChannelModulo64Kbytes
Circular buffer size is 64 K bytes.

enumerator kEDMA_ChannelModulo128Kbytes
Circular buffer size is 128 K bytes.

enumerator kEDMA_ChannelModulo256Kbytes
Circular buffer size is 256 K bytes.

enumerator kEDMA_ChannelModulo512Kbytes
Circular buffer size is 512 K bytes.

enumerator kEDMA_ChannelModulo1Mbytes
Circular buffer size is 1 M bytes.

enumerator kEDMA_ChannelModulo2Mbytes
Circular buffer size is 2 M bytes.

enumerator kEDMA_ChannelModulo4Mbytes
Circular buffer size is 4 M bytes.

enumerator kEDMA_ChannelModulo8Mbytes
Circular buffer size is 8 M bytes.

enumerator kEDMA_ChannelModulo16Mbytes
Circular buffer size is 16 M bytes.

enumerator kEDMA_ChannelModulo32Mbytes
Circular buffer size is 32 M bytes.

enumerator kEDMA_ChannelModulo64Mbytes
Circular buffer size is 64 M bytes.

enumerator kEDMA_ChannelModulo128Mbytes
Circular buffer size is 128 M bytes.

enumerator kEDMA_ChannelModulo256Mbytes
Circular buffer size is 256 M bytes.

enumerator kEDMA_ChannelModulo512Mbytes
Circular buffer size is 512 M bytes.

enumerator kEDMA_ChannelModulo1Gbytes
Circular buffer size is 1 G bytes.

enumerator kEDMA_ChannelModulo2Gbytes
Circular buffer size is 2 G bytes.

enum _edma_channel_bandwidth
edma channel Bandwidth control

Generally, as the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. This bandwidth field forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth.

The default configuration is kEDMA_BandwidthStallNone.

Values:

enumerator kEDMA_ChannelBandwidthStallNone
No eDMA engine stalls.

enumerator kEDMA_ChannelBandwidthStall4Cycle
eDMA engine stalls for 4 cycles after each read/write.

enumerator kEDMA_ChannelBandwidthStall8Cycle
eDMA engine stalls for 8 cycles after each read/write.

typedef enum _edma_channel_transfer_type edma_channel_transfer_type_t
eDMA transfer type

typedef enum _edma_arbitration_type edma_arbitration_type_t
eDMA arbitration type

typedef enum _edma_channel edma_channel_t
edma channel index

typedef enum _edma_channel_transfer_width edma_channel_transfer_width_t
eDMA transfer width configuration

typedef enum _edma_channel_modulo edma_channel_modulo_t
eDMA channel modulo configuration

The eDMA modulo feature can be used to specify the address range of the source/destination address, it is useful to implement a circular data queue.

```
typedef enum _edma_channel_bandwidth edma_channel_bandwidth_t
    edma channel Bandwidth control
```

Generally, as the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. This bandwidth field forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth.

The default configuration is kEDMA_BandwidthStallNone.

```
typedef struct _edma_channel_Preemption_config edma_channel_Preemption_config_t
    eDMA channel priority configuration, useful to the fixed priority arbitration type
```

```
typedef struct _edma_channel_tcd edma_channel_tcd_t
    edma channel software tcd definition
```

```
typedef struct _edma_channel_transfer_config edma_channel_transfer_config_t
    edma channel transfer configuration
```

The transfer configuration structure support full feature configuration of the transfer control descriptor.

1.To perform a simple transfer, below members should be initialized at least .u32SrcAddr - source address .u32DstAddr - destination address .eSrcWidthOfEachTransfer - data width of source address .eDstWidthOfEachTransfer - data width of destination address, normally it should be as same as eSrcWidthOfEachTransfer .u32BytesEachRequest - bytes to be transferred in each DMA request .u32TotalBytes - total bytes to be transferred .i16SrcOffsetOfEachTransfer - offset value in bytes unit to be applied to source address as each source read is completed .i16DstOffsetOfEachTransfer - offset value in bytes unit to be applied to destination address as each destination write is completed bEnableChannelRequest - channel request can be enabled together with transfer configure submission

2.The transfer configuration structure also support advance feature: Programmable source/destination address range(MODULO) Programmable minor loop offset Programmable major loop offset Programmable channel chain feature Programmable channel transfer control descriptor link feature

Note: User should pay attention to the transfer size alignment limitation

- a. the u32BytesEachRequest should align with the eSrcWidthOfEachTransfer and the eDstWidthOfEachTransfer that is to say $u32BytesEachRequest \% eSrcWidthOfEachTransfer$ should be 0
 - b. the i16SrcOffsetOfEachTransfer and i16DstOffsetOfEachTransfer must be aligne with transfer width
 - c. the u32TotalBytes should align with the u32BytesEachRequest
 - d. the u32SrcAddr should align with the eSrcWidthOfEachTransfer
 - e. the u32DstAddr should align with the eDstWidthOfEachTransfer
 - f. the u32SrcAddr should align with eSrcAddrModulo if modulo feature is enabled
 - g. the u32DstAddr should align with eDstAddrModulo if modulo feature is enabled If any-one of above condition can not be satisfied, the edma interfaces will generate assert error.
-

```
typedef struct _edma_config edma_config_t
    edma configuration structure
```

This structure target for whole edma module configurations.

```
typedef struct _edma_handle edma_handle_t  
    handler for eDMA
```

```
typedef void (*edma_transfer_callback_t)(edma_handle_t *psHandle, void *pUserData, bool  
bTransferDone, uint32_t u32Tcds)
```

Define callback function for eDMA.

This callback function is called in the EDMA interrupt handler function. In normal mode, running into callback function means the transfer users need is done. In scatter gather mode, run into callback function means a transfer control block (tcd) is finished. Not all transfer finished, users can get the finished tcd numbers using interface `EDMA_GetUnusedTCDNumber`.

Param handle

EDMA handle pointer, users shall not touch the values inside.

Param userData

The callback user parameter pointer. Users can use this parameter to involve things users need to change in EDMA callback function.

Param transferDone

If the current loaded transfer done. In normal mode it means if all transfer done. In scatter gather mode, this parameter shows is the current transfer block in EDMA register is done. As the load of core is different, it will be different if the new tcd loaded into EDMA registers while this callback called. If true, it always means new tcd still not loaded into registers, while false means new tcd already loaded into registers.

Param tcds

How many tcds are done from the last callback. This parameter only used in scatter gather mode. It tells user how many tcds are finished between the last callback and this.

```
EDMA_ENTER_CRITICAL_SECTION()
```

edma transactional tcd pool resource protection lock definition Application should overwrite below two macros if multi task trying to access the same channel.

```
EDMA_LEAVE_CRITICAL_SECTION()
```

```
struct _edma_channel_Preemption_config
```

```
#include <fsl_edma.h> eDMA channel priority configuration, useful to the fixed priority arbitration type
```

Public Members

```
bool bSuspendedByHighPriorityChannel
```

a channel can be suspended by other channel with higher priority

```
bool bSuspendLowPriorityChannel
```

a channel can suspend other channel with low priority

```
uint8_t u8ChannelPriority
```

Channel priority

```
struct _edma_channel_transfer_config
```

```
#include <fsl_edma.h> edma channel transfer configuration
```

The transfer configuration structure support full feature configuration of the transfer control descriptor.

1.To perform a simple transfer, below members should be initialized at least .u32SrcAddr - source address .u32DstAddr - destination address .eSrcWidthOfEachTransfer - data width of source address .eDstWidthOfEachTransfer - data width of destination address, normally it should be as same as eSrcWidthOfEachTransfer .u32BytesEachRequest - bytes to be transferred in each DMA request .u32TotalBytes - total bytes to be transferred .i16SrcOffsetOfEachTransfer - offset value in bytes unit to be applied to source address as each source read is completed .i16DstOffsetOfEachTransfer - offset value in bytes unit to be applied to destination address as each destination write is completed bEnableChannelRequest - channel request can be enabled together with transfer configure submission

2.The transfer configuration structure also support advance feature: Programmable source/destination address range(MODULO) Programmable minor loop offset Programmable major loop offset Programmable channel chain feature Programmable channel transfer control descriptor link feature

Note: User should pay attention to the transfer size alignment limitation

- a. the u32BytesEachRequest should align with the eSrcWidthOfEachTransfer and the eDstWidthOfEachTransfer that is to say $u32BytesEachRequest \% eSrcWidthOfEachTransfer$ should be 0
 - b. the i16SrcOffsetOfEachTransfer and i16DstOffsetOfEachTransfer must be aligne with transfer width
 - c. the u32TotalBytes should align with the u32BytesEachRequest
 - d. the u32SrcAddr should align with the eSrcWidthOfEachTransfer
 - e. the u32DstAddr should align with the eDstWidthOfEachTransfer
 - f. the u32SrcAddr should align with eSrcAddrModulo if modulo feature is enabled
 - g. the u32DstAddr should align with eDstAddrModulo if modulo feature is enabled If any-one of above condition can not be satisfied, the edma interfaces will generate assert error.
-

Public Members

uint32_t u32SrcAddr
source address

uint32_t u32DstAddr
destination address

edma_channel_transfer_width_t eSrcWidthOfEachTransfer
source width of each transfer

edma_channel_transfer_width_t eDstWidthOfEachTransfer
destination width of each transfer

uint32_t u32BytesEachMinorLoop
bytes in each minor loop or each request range: $1 - (2^{30} - 1)$ when minor loop mapping is enabled range: $1 - (2^{10} - 1)$ when minor loop mapping is enabled and source or dest minor loop offset is enabled range: $1 - (2^{32} - 1)$ when minor loop mapping is disabled

uint16_t u16MinorLoopCountsEachMajorLoop
minor loop counts in each major loop, should be 1 at least for each transfer range: $(0 - (2^{15} - 1))$ when minor loop channel link is disabled range: $(0 - (2^9 - 1))$ when minor loop channel link is enabled total bytes in a transfer = $u16MinorLoopCountsEachMajorLoop * u32BytesEachMinorLoop$

`uint16_t u16EnabledInterruptMask`
channel interrupt to enable, can be OR'ed value of `_edma_channel_interrupt_enable`

`int16_t i16SrcOffsetOfEachTransfer`
Sign-extended offset value in byte unit applied to the current source address to form the next-state value as each source read is completed

`edma_channel_modulo_t eSrcAddrModulo`
source circular data queue range

`int32_t i32SrcMajorLoopOffset`
source major loop offset

`int16_t i16DstOffsetOfEachTransfer`
Sign-extended offset value in byte unit applied to the current destination address to form the next-state value as each destination write is completed.

`edma_channel_modulo_t eDstAddrModulo`
destination circular data queue range

`int32_t i32DstMajorLoopOffset`
destination major loop offset

`bool bEnableSrcMinorLoopOffset`
enable source minor loop offset

`bool bEnableDstMinorLoopOffset`
enable dest minor loop offset

`int32_t i32MinorLoopOffset`
burst offset, the offset will be applied after minor loop update

`bool bEnableChannelMajorLoopLink`
channel link when major loop complete

`edma_channel_t eMajorLoopLinkChannel`
major loop link channel number

`bool bEnableChannelMinorLoopLink`
channel link when minor loop complete

`edma_channel_t eMinorLoopLinkChannel`
minor loop link channel number

`edma_channel_bandwidth_t eChannelBandWidth`
channel bandwidth

`bool bDisableRequestAfterMajorLoopComplete`
the channel's ERQ bit can be cleared after the major loop complete automatically

`bool bEnableChannelRequest`
enable the channel request signal

`edma_channel_tcd_t *psLinkTCD`
pointer to the link transfer control descriptor

`struct _edma_channel_tcd`

`#include <fsl_edma.h>` eDMA software Transfer control descriptor structure.

This structure is same as eDMA hardware channel TCD registers, user doesn't need to understand the structures, since eDMA driver will responsible for configure it.

The software TCD is useful to configure a software TCD which is linked by the channel hardware TCD to have scatter/gather feature without using transactional interface.

Public Members

- `__IO uint32_t u32SADDR`
SADDR register, used to save source address
- `__IO uint16_t u16SOFF`
SOFF register, offset bytes added to source address every transfer
- `__IO uint16_t u16ATTR`
ATTR register, source/destination transfer size and modulo
- `__IO uint32_t u32NBYTES`
Nbytes register, minor loop length in bytes
- `__IO uint32_t u32SLAST`
SLAST register, adjustment value added to the source address at the completion of the major loop
- `__IO uint32_t u32DADDR`
DADDR register, used for destination address
- `__IO uint16_t u16DOFF`
DOFF register, offset bytes added to destination address every transfer
- `__IO uint16_t u16CITER`
CITER register, current minor loop numbers, for unfinished minor loop.
- `__IO uint32_t u32DLAST_SGA`
DLASTSGA register, next tcd address used in scatter-gather mode
- `__IO uint16_t u16CSR`
CSR register, for TCD control status
- `__IO uint16_t u16BITER`
BITER register, begin minor loop count.

`struct _edma_config`

`#include <fsl_edma.h>` edma configuration structure

This structure target for whole edma module configurations.

Public Members

- `bool bEnableContinuousLinkMode`
Enable (true) continuous link mode. Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.
- `bool bEnableHaltOnError`
Enable (true) transfer halt on error. Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.
- `bool bEnableDmaInDebugMode`
Enable(true) eDMA debug mode. When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.
- `bool bEnableMinorLoopMapping`
TCDn.word2 is redefined to include individual enable fields, an offset field, and the NBYTES field. The individual enable fields allow the minor loop offset to be applied to the source address, the destination address, or both. The NBYTES field is reduced when either offset is enabled

edma_arbitration_type_t eArbitrationType

Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection

edma_channel_Preemption_config_t sChannelPreemptionConfig[1]

channel preemption configuration

edma_channel_transfer_config_t *psChannelTransferConfig[1]

channel transfer configuration pointer

struct *_edma_handle*

#include <fsl_edma.h> eDMA transfer handle structure

Public Members

edma_transfer_callback_t pfCallback

Callback function for major count exhausted.

void *pUserData

Callback function parameter.

DMA_Type *psBase

eDMA peripheral base address.

edma_channel_tcd_t *psTcdPool

Pointer to memory stored TCDs.

edma_channel_t eChannel

eDMA channel number.

volatile uint8_t u8Header

The first TCD index. Should point to the next TCD to be loaded into the eDMA engine.

volatile uint8_t u8Tail

The last TCD index. Should point to the next TCD to be stored into the memory pool.

volatile uint8_t u8TcdUsed

The number of used TCD slots. Should reflect the number of TCDs can be used/loaded in the memory.

volatile uint8_t u8TcdSize

The total number of TCD slots in the queue.

2.22 The Driver Change Log

2.23 EDMA Peripheral and Driver Overview

2.24 EVTG: Event Generator Driver

void EVTG_Init(EVTG_Type *base, *evtg_index_t* eEvtgIndex, *evtg_config_t* *psConfig)

Initialize EVTG with a user configuration structure.

Parameters

- base – EVTG base address.
- eEvtgIndex – EVTG instance index.

- psConfig – EVTG initial configuration structure pointer.

```
static inline void EVTG_GetDefaultConfig(evtg_config_t *psConfig, evtg_flipflop_mode_t
                                         eFlipflopMode)
```

Loads default values to the EVTG configuration structure.

The purpose of this API is to initialize the configuration structure to default value for EVTG_Init() to use. The Flip-Flop can be configured as Bypass mode, RS trigger mode, T-FF mode, D-FF mode, JK-FF mode, Latch mode. Please check RM INTC chapter for more details.

Parameters

- psConfig – EVTG initial configuration structure pointer.
- eFlipflopMode – EVTG flip flop mode. see @ ref _evtg_flipflop_mode

```
static inline void EVTG_ForceFlipflopInitOutput(EVTG_Type *base, evtg_index_t eEvtgIndex,
                                                evtg_flipflop_init_output_t
                                                eFlipflopInitOutputValue)
```

Force Flip-flop initial output value to be presented on flip-flop positive output.

Parameters

- base – EVTG base address.
- eEvtgIndex – EVTG instance index.
- eFlipflopInitOutputValue – EVTG flip-flop initial output control. see *evtg_flipflop_init_output_t*

```
static inline void EVTG_SetProductTermInput(EVTG_Type *base, evtg_index_t eEvtgIndex,
                                           evtg_aoi_index_t eAOIIndex,
                                           evtg_aoi_product_term_t eProductTerm,
                                           evtg_input_index_t eInputIndex,
                                           evtg_aoi_input_config_t eInput)
```

Configure each input value of AOI product term. Each selected input term in each product term can be configured to produce a logical 0 or 1 or pass the true or complement of the selected event input. Adapt to some simple aoi expressions.

Parameters

- base – EVTG base address.
- eEvtgIndex – EVTG instance index.
- eAOIIndex – EVTG AOI index. see enum ref *evtg_aoi_index_t*
- eProductTerm – EVTG product term index.
- eInputIndex – EVTG input index.
- eInput – EVTG input configuration with enum *evtg_aoi_input_config_t*.

```
void EVTG_ConfigAOIProductTerm(EVTG_Type *base, evtg_index_t eEvtgIndex, evtg_aoi_index_t
                               eAOIIndex, evtg_aoi_product_term_t eProductTerm,
                               evtg_aoi_product_term_config_t *psProductTermConfig)
```

Configure AOI product term by initializing the product term configuration structure.

Parameters

- base – EVTG base address.
- eEvtgIndex – EVTG instance index.
- eAOIIndex – EVTG AOI index. see enum *evtg_aoi_index_t*
- eProductTerm – EVTG AOI product term index.

- psProductTermConfig – Pointer to EVTG product term configuration structure. see ref_evtg_aoi_product_term_config

FSL_EVTG_DRIVER_VERSION

EVTG driver version.

enum _evtg_index

EVTG instance index.

Values:

enumerator kEVTG_Index0

EVTG instance index 0.

enumerator kEVTG_Index1

EVTG instance index 1.

enumerator kEVTG_Index2

EVTG instance index 2.

enumerator kEVTG_Index3

EVTG instance index 3.

enum _evtg_input_index

EVTG input index.

Values:

enumerator kEVTG_InputA

EVTG input A.

enumerator kEVTG_InputB

EVTG input B.

enumerator kEVTG_InputC

EVTG input C.

enumerator kEVTG_InputD

EVTG input D.

enum _evtg_aoi_index

EVTG AOI index.

Values:

enumerator kEVTG_AOI0

EVTG AOI index 0.

enumerator kEVTG_AOI1

EVTG AOI index 1.

enum _evtg_aoi_product_term

EVTG AOI product term index.

Values:

enumerator kEVTG_ProductTerm0

EVTG AOI product term index 0.

enumerator kEVTG_ProductTerm1

EVTG AOI product term index 1.

enumerator kEVTG_ProductTerm2

EVTG AOI product term index 2.

enumerator kEVTG_ProductTerm3
EVTG AOI product term index 3.

enum _evtg_aoi_input_config
EVTG input configuration.

Values:

enumerator kEVTG_Input_LogicZero
Force input in product term to a logical zero.

enumerator kEVTG_Input_DirectPass
Pass input in product term.

enumerator kEVTG_Input_Complement
Complement input in product term.

enumerator kEVTG_Input_LogicOne
Force input in product term to a logical one.

enum _evtg_aoi_outfilter_count
EVTG AOI Output Filter Sample Count.

Values:

enumerator kEVTG_AOIOutFilter_SampleCount3
EVTG AOI output filter sample count is 3.

enumerator kEVTG_AOIOutFilter_SampleCount4
EVTG AOI output filter sample count is 4.

enumerator kEVTG_AOIOutFilter_SampleCount5
EVTG AOI output filter sample count is 5.

enumerator kEVTG_AOIOutFilter_SampleCount6
EVTG AOI output filter sample count is 6.

enumerator kEVTG_AOIOutFilter_SampleCount7
EVTG AOI output filter sample count is 7.

enumerator kEVTG_AOIOutFilter_SampleCount8
EVTG AOI output filter sample count is 8.

enumerator kEVTG_AOIOutFilter_SampleCount9
EVTG AOI output filter sample count is 9.

enumerator kEVTG_AOIOutFilter_SampleCount10
EVTG AOI output filter sample count is 10.

enum _evtg_outfdbk_override_input
EVTG output feedback override control mode. When FF is configured as JK-FF mode, need EVTG_OUTA feedback to EVTG input and replace one of the four inputs.

Values:

enumerator kEVTG_Output_OverrideInputA
Replace input A.

enumerator kEVTG_Output_OverrideInputB
Replace input B.

enumerator kEVTG_Output_OverrideInputC
Replace input C.

enumerator kEVTG_Output_OverrideInputD
Replace input D.

enum _evtg_flipflop_mode
EVTG flip flop mode configuration.

Values:

enumerator kEVTG_FFMode_Bypass
Bypass mode (default). In this mode, user can choose to enable or disable input sync logic and filter function.

enumerator kEVTG_FFMode_RSTTrigger
RS trigger mode. In this mode, user can choose to enable or disable input sync logic and filter function.

enumerator kEVTG_FFMode_TFF
T-FF mode. In this mode, input sync or filter has to be enabled to remove the possible glitch.

enumerator kEVTG_FFMode_DFF
D-FF mode. In this mode, input sync or filter has to be enabled to remove the possible glitch.

enumerator kEVTG_FFMode_JKFF
JK-FF mode. In this mode, input sync or filter has to be enabled to remove the possible glitch.

enumerator kEVTG_FFMode_Latch
Latch mode. In this mode, input sync or filter has to be enabled to remove the possible glitch.

enum _evtg_flipflop_initoutput
EVTG flip-flop initial value.

Values:

enumerator kEVTG_FF_InitOut0
Configure the positive output of flip-flop as 0.

enumerator kEVTG_FF_InitOut1
Configure the positive output of flip-flop as 1.

typedef enum _evtg_index evtg_index_t
EVTG instance index.

typedef enum _evtg_input_index evtg_input_index_t
EVTG input index.

typedef enum _evtg_aoi_index evtg_aoi_index_t
EVTG AOI index.

typedef enum _evtg_aoi_product_term evtg_aoi_product_term_t
EVTG AOI product term index.

typedef enum _evtg_aoi_input_config evtg_aoi_input_config_t
EVTG input configuration.

typedef enum _evtg_aoi_outfilter_count evtg_aoi_outfilter_count_t
EVTG AOI Output Filter Sample Count.

typedef enum *_evtg_outfdbk_override_input* evtg_outfdbk_override_input_t
 EVTG output feedback override control mode. When FF is configured as JK-FF mode, need EVTG_OUTA feedback to EVTG input and replace one of the four inputs.

typedef enum *_evtg_flipflop_mode* evtg_flipflop_mode_t
 EVTG flip flop mode configuration.

typedef enum *_evtg_flipflop_initoutput* evtg_flipflop_init_output_t
 EVTG flip-flop initial value.

typedef struct *_evtg_aoi_outfilter_config* evtg_aoi_outfilter_config_t
 The structure for configuring an AOI output filter sample.

AOI output filter sample count represent the number of consecutive samples that must agree prior to the AOI output filter accepting an transition. AOI output filter sample period represent the sampling period (in IP bus clock cycles) of the AOI output signals. Each AOI output is sampled multiple times at the rate specified by this period.

For the modes with Filter function enabled, filter delay is “(FILT_CNT + 3) x FILT_PER + 2”.

typedef struct *_evtg_aoi_product_term_config* evtg_aoi_product_term_config_t
 The structure for configuring an AOI product term.

typedef struct *_evtg_aoi_config* evtg_aoi_config_t
 EVTG AOI configuration structure.

typedef struct *_evtg_config* evtg_config_t
 EVTG configuration covering all configurable fields.

struct *_evtg_aoi_outfilter_config*
#include <fsl_evtg.h> The structure for configuring an AOI output filter sample.

AOI output filter sample count represent the number of consecutive samples that must agree prior to the AOI output filter accepting an transition. AOI output filter sample period represent the sampling period (in IP bus clock cycles) of the AOI output signals. Each AOI output is sampled multiple times at the rate specified by this period.

For the modes with Filter function enabled, filter delay is “(FILT_CNT + 3) x FILT_PER + 2”.

Public Members

evtg_aoi_outfilter_count_t eSampleCount
 EVTG AOI output filter sample count. refer to *evtg_aoi_outfilter_count_t*.

uint8_t u8SamplePeriod
 EVTG AOI output filter sample period, within 0~255. If sample period value is 0x00 (default), then the input filter is bypassed.

struct *_evtg_aoi_product_term_config*
#include <fsl_evtg.h> The structure for configuring an AOI product term.

Public Members

evtg_aoi_input_config_t eAInput
 Input A configuration.

evtg_aoi_input_config_t eBInput
 Input B configuration.

evtg_aoi_input_config_t eCInput
 Input C configuration.

evtg_aoi_input_config_t eDInput
Input D configuration.

struct *_evtg_aoi_config*
#include <fsl_evtg.h> EVTG AOI configuration structure.

Public Members

evtg_aoi_outfilter_config_t sAOIOutFilterConfig
EVTG AOI output filter sample configuration structure.

evtg_aoi_product_term_config_t sProductTerm0
Configure AOI product term0.

evtg_aoi_product_term_config_t sProductTerm1
Configure AOI product term1.

evtg_aoi_product_term_config_t sProductTerm2
Configure AOI product term2.

evtg_aoi_product_term_config_t sProductTerm3
Configure AOI product term3.

struct *_evtg_config*
#include <fsl_evtg.h> EVTG configuration covering all configurable fields.

Public Members

bool bEnableInputASync
Enable/Disable EVTG A input synchronous with bus clk.

bool bEnableInputBSync
Enable/Disable EVTG B input synchronous with bus clk.

bool bEnableInputCSync
Enable/Disable EVTG C input synchronous with bus clk.

bool bEnableInputDSync
Enable/Disable EVTG D input synchronous with bus clk.

evtg_outfdbk_override_input_t eOutfdbkOverrideinput
EVTG output feedback to EVTG input and replace one of the four inputs.

evtg_flipflop_mode_t eFlipflopMode
Flip-Flop can be configured as one of Bypass mode, RS trigger mode, T-FF mode, D-FF mode, JK-FF mode, Latch mode.

bool bEnableFlipflopInitOutput
Flip-flop initial output value enable/disable.

evtg_flipflop_init_output_t eFlipflopInitOutputValue
Flip-flop initial output value configuration.

evtg_aoi_config_t sAOI0Config
Configure EVTG AOI0.

evtg_aoi_config_t sAOI1Config
Configure EVTG AOI1.

2.25 The Driver Change Log

2.26 EVTG Peripheral and Driver Overview

2.27 EWM: External Watchdog Monitor Driver

`void EWM_Init(EWM_Type *base, const ewm_config_t *psConfig)`

Initializes the EWM peripheral.

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration.

This is an example.

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

Note: 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.

Parameters

- `base` – EWM peripheral base address
- `psConfig` – 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 *psConfig)`

Initializes the EWM configuration structure.

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

```
ewmConfig->bEnableEWM = true;
ewmConfig->bEnableEWMInput = false;
ewmConfig->eInputAssertState = kEWM_EwmInZeroAssert;
ewmConfig->bEnableInterrupt = false;
ewmConfig->eClockSource = kEWM_LpoClockSource0;
ewmConfig->u8ClockDivder = 0;
ewmConfig->u8CompareLowValue = 0;
ewmConfig->u8CompareHighValue = 0xFEU;
```

See also:

`ewm_config_t`

Parameters

- psConfig – Pointer to the EWM configuration structure.

static inline void EWM_EnableInterrupt(EWM_Type *base)

Enables the EWM interrupt.

This function enables the EWM interrupt.

Parameters

- base – EWM peripheral base address

static inline void EWM_DisableInterrupt(EWM_Type *base)

Disables the EWM interrupt.

This function disables the EWM interrupt.

Parameters

- base – EWM peripheral base address

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.

enum _ewm_input_assert_state

Assert pin voltage configuration.

Values:

enumerator kEWM_EwmInZeroAssert

EWM-in assert with low-voltage logic

enumerator kEWM_EwmInOneAssert

EWM-in assert with high-voltage logic

typedef enum _ewm_input_assert_state ewm_input_assert_state_t

Assert pin voltage configuration.

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

uint8_t bEnableEWM

Enable EWM module

uint8_t bEnableEWMInput

Enable EWM_in input

uint8_t bEnableInterrupt

Enable EWM interrupt

ewm_input_assert_state_t eInputAssertState
EWM_in signal assertion state select

ewm_lpo_clock_source_t eClockSource
Clock source select

uint8_t u8ClockDivder
EWM counter clock is clockSource/(clockDivder+1)

uint8_t u8CompareLowValue
Compare low-register value

uint8_t u8CompareHighValue
Compare high-register value, maximum setting is 0xFE

2.28 The Driver Change Log

2.29 EWM Peripheral and Driver Overview

2.30 Flash Driver

enum _flash_driver_version_constants
Flash driver version for ROM.
Values:

enumerator kFLASH_DriverVersionName
Flash driver version name.

enumerator kFLASH_DriverVersionMajor
Major flash driver version.

enumerator kFLASH_DriverVersionMinor
Minor flash driver version.

enumerator kFLASH_DriverVersionBugfix
Bugfix for flash driver version.

enum _flash_driver_api_keys
Enumeration for FLASH driver API keys.

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

Values:

enumerator kFLASH_ApiEraseKey

enum _flash_read_resource_opt
Enumeration for the two possible options of flash read resource command.

Values:

enumerator kFLASH_ResourceOptionFlashIfr
Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR

enumerator kFLASH_ResourceOptionVersionId

Select code for the version ID

enum _flash_margin_value

Enumeration for supported FTFx margin levels.

Values:

enumerator kFLASH_MarginValueNormal

Use the 'normal' read level for 1s.

enumerator kFLASH_MarginValueUser

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

enumerator kFLASH_MarginValueFactory

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

enumerator kFLASH_MarginValueInvalid

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

enum _flash_security_state

Enumeration for the three possible FTFx security states.

Values:

enumerator kFLASH_SecurityStateNotSecure

Flash is not secure.

enumerator kFLASH_SecurityStateBackdoorEnabled

Flash backdoor is enabled.

enumerator kFLASH_SecurityStateBackdoorDisabled

Flash backdoor is disabled.

enum _flash_swap_control_opt

Enumeration for the possible options of Swap control commands.

Values:

enumerator kFLASH_SwapControlOptionIntializeSystem

An option used to initialize the Swap system

enumerator kFLASH_SwapControlOptionSetInUpdateState

An option used to set the Swap in an update state

enumerator kFLASH_SwapControlOptionSetInCompleteState

An option used to set the Swap in a complete state

enumerator kFLASH_SwapControlOptionReportStatus

An option used to report the Swap status

enumerator kFLASH_SwapControlOptionDisableSystem

An option used to disable the Swap status

enum _flash_swap_state

Enumeration for the possible flash Swap status.

Values:

enumerator kFLASH_SwapStateUninitialized

Flash Swap system is in an uninitialized state.

enumerator kFLASH_SwapStateReady

Flash Swap system is in a ready state.

enumerator kFLASH_SwapStateUpdate
Flash Swap system is in an update state.

enumerator kFLASH_SwapStateUpdateErased
Flash Swap system is in an updateErased state.

enumerator kFLASH_SwapStateComplete
Flash Swap system is in a complete state.

enumerator kFLASH_SwapStateDisabled
Flash Swap system is in a disabled state.

enum _flash_swap_block_status
Enumeration for the possible flash Swap block status.

Values:

enumerator kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero
Swap block status is that lower half program block at zero.

enumerator kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero
Swap block status is that upper half program block at zero.

enum _flash_memory_type
Enumeration for FTFx memory type.

Values:

enumerator kFLASH_MemTypePflash

enumerator kFLASH_MemTypeFlexnvm

enum _flash_property_tag
Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflashSectorSize
Pflash sector size property.

enumerator kFLASH_PropertyPflashTotalSize
Pflash total size property.

enumerator kFLASH_PropertyPflashBlockSize
Pflash block size property.

enumerator kFLASH_PropertyPflashBlockCount
Pflash block count property.

enumerator kFLASH_PropertyPflashBlockBaseAddr
Pflash block base address property.

enumerator kFLASH_PropertyVersion

enum _flash_status
FTFx driver status codes.

Values:

enumerator kStatus_FLASH_Success
API is executed successfully

enumerator kStatus_FLASH_InvalidArgument
Invalid argument

enumerator `kStatus_FLASH_SizeError`
Error size

enumerator `kStatus_FLASH_AlignmentError`
Parameter is not aligned with the specified baseline

enumerator `kStatus_FLASH_AddressError`
Address is out of range

enumerator `kStatus_FLASH_AccessError`
Invalid instruction codes and out-of bound addresses

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

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

enumerator `kStatus_FLASH_UnknownProperty`
Unknown property.

enumerator `kStatus_FLASH_EraseKeyError`
API erase key is invalid.

enumerator `kStatus_FLASH_CommandNotSupported`
Flash API is not supported.

enumerator `kStatus_FLASH_SwapSystemNotInUninitialized`
Swap system is not in an uninitialized state.

enumerator `kStatus_FLASH_SwapIndicatorAddressError`
The swap indicator address is invalid.

enumerator `kStatus_FLASH_ReadOnlyProperty`
The flash property is read-only.

enumerator `kStatus_FLASH_InvalidPropertyValue`
The flash property value is out of range.

enumerator `kStatus_FLASH_InvalidSpeculationOption`
The option of flash prefetch speculation is invalid.

typedef union `_standard_version` `standard_version_t`

typedef enum `_flash_read_resource_opt` `flash_read_resource_opt_t`
Enumeration for the two possible options of flash read resource command.

typedef enum `_flash_margin_value` `flash_margin_value_t`
Enumeration for supported FTFx margin levels.

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

typedef enum `_flash_swap_control_opt` `flash_swap_control_opt_t`
Enumeration for the possible options of Swap control commands.

typedef enum `_flash_swap_state` `flash_swap_state_t`
Enumeration for the possible flash Swap status.

typedef enum `_flash_swap_block_status` `flash_swap_block_status_t`
Enumeration for the possible flash Swap block status.

```
typedef struct _flash_swap_state_config flash_swap_state_config_t
```

Flash Swap information.

```
typedef struct _flash_special_mem flash_spec_mem_t
```

ftfx special memory access information.

```
typedef struct _flash_mem_desc_t flash_mem_desc_t
```

Flash memory descriptor.

```
typedef struct _flash_ops_config flash_ops_config_t
```

Active FTFx information for the current operation.

```
typedef struct _flash_ifr_desc flash_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.

```
typedef struct _flash_config flash_config_t
```

```
typedef enum _flash_property_tag flash_property_tag_t
```

Enumeration for various flash properties.

```
FSL_FLASH_DRIVER_VERSION
```

Flash driver version for SDK.

```
FSL_FLASH_DRIVER_VERSION_ROM
```

Flash driver version for ROM.

```
MAKE_STATUS(group, code)
```

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

```
FTFX_FLASH_COUNT
```

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.

```
flash_status_t FLASH_Init(flash_config_t *config)
```

Initializes the global flash properties structure members.

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

Parameters

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

Return values

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

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

Returns the desired flash property.

Parameters

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

- `value` – A pointer to the value returned for the desired flash property.

Return values

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

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

Erases the flash sectors encompassed by parameters passed into function.

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

Parameters

- `config` – The pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be phrase-aligned (64 bits).
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be erased. Must be phrase-aligned (64 bits).
- `key` – The value used to validate all flash erase APIs. Use `kFLASH_ApiEraseKey` here.

Return values

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

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

Erases entire flash, after done flash is on security status.

Parameters

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

Return values

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

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

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

Erases the entire flash, after done flash is on unsecurity status.

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_EraseKeyError` – API erase key is invalid.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during command execution.

`flash_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 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 phrase-aligned (64 bits).
- `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 phrase-aligned (64 bits).

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.

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

flash_status_t FLASH_ProgramOnce(*ftfx_config_t* *config, uint32_t index, 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. Index is 0~7.
- `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 phrase-aligned (64 bits). This value is ignored, it's always 8 bytes that are written into specified IFR area.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

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

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

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

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `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 phrase-aligned (64 bits).
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be phrase-aligned (64 bits).
- `margin` – Read margin choice. Choose from the members in `flash_margin_value_t`.

Return values

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

- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

flash_status_t FLASH_VerifyEraseAll(*flash_config_t* *config, *flash_margin_value_t* margin)

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

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

Parameters

- config – A pointer to the storage for the driver runtime state.
- margin – Read margin choice. Choose from the members in `flash_margin_value_t`.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

flash_status_t FLASH_VerifyProgram(*flash_config_t* *config, *uint32_t* start, *uint32_t* lengthInBytes, *const uint8_t* *expectedData, *flash_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 longword-aligned (32 bits).
- lengthInBytes – The length, given in bytes (not words or long-words), to be verified. Must be longword-aligned (32 bits).
- expectedData – A pointer to the expected data that is to be verified against.
- margin – Read margin choice. Choose from the members in `flash_margin_value_t`. “`kFLASH_MarginValueNormal`” is not for this command.
- failedAddress – A pointer to the returned failing address, which is used by PGMCHK command. So it’s always longword-aligned.

- `failedData` – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

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

`flash_status_t` FLASH_GetSecurityState(`flash_config_t` *config, `flash_security_state_t` *state)

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

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

Parameters

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

Return values

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

`flash_status_t` FLASH_SecurityBypass(`flash_config_t` *config, `const uint8_t` *backdoorKey)

Allows users to bypass security with a backdoor key.

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

Parameters

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

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

flash_status_t FLASH_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. Index is 0~7.
- 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. This value is ignored, it's always 8 bytes that are read out from the specified IFR area.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FLASH_ProtectionViolation – The program/erase operation is requested to execute on protected areas.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.

flash_status_t FLASH_ReadResource(*flash_config_t* *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, *flash_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 resource to be read. Must be phrase-aligned (64 bits).
- 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 phrase-aligned (64 bits). When option is “kFLASH_ResourceOptionFlashIfr”, lengthInBytes can't be larger than 64. When option is “kFLASH_ResourceOptionVersionId”, lengthInBytes should be set to 8.
- option – The resource option which indicates which area should be read back. Choose from “kFLASH_ResourceOptionFlashIfr” and “kFLASH_ResourceOptionVersionId”.

Return values

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

- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.

flash_status_t FLASH_Swap(*flash_config_t* *config, uint32_t address, uint8_t 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_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_SwapIndicatorAddressError` – Swap indicator address is invalid.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_ProtectionViolation` – The program/erase operation is requested to execute on protected areas.
- `kStatus_FLASH_CommandFailure` – Run-time error during command execution.
- `kStatus_FLASH_SwapSystemNotInUninitialized` – Swap system is not in an uninitialized state.

```
typedef int flash_status_t
```

```
uint8_t bugfix
```

```
uint8_t minor
```

```
uint8_t major
```

```
char name
```

```
struct _standard_version B
```

```
uint32_t version
```

```
flash_swap_state_t flashSwapState
```

The current Swap system status.

```
flash_swap_block_status_t currentSwapBlockStatus
```

The current Swap block status.

flash_swap_block_status_t nextSwapBlockStatus

The next Swap block status.

uint32_t base

Base address of flash special memory.

uint32_t size

size of flash special memory.

uint32_t count

flash special memory count.

uint8_t type

Type of flash block.

uint8_t index

Index of flash block.

uint8_t reserved[2]

uint32_t isIndBlock

uint32_t hasIndPfsizereg

uint32_t hasProtControl

uint32_t hasIndProtReg

uint32_t hasXaccControl

uint32_t hasIndXaccReg

uint32_t __pad0__

uint32_t ProtRegBits

struct *_flash_mem_desc_t* feature

uint32_t blockBase

A base address of the 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.

flash_spec_mem_t accessSegmentMem

flash_spec_mem_t protectRegionMem

uint32_t convertedAddress

A converted address for the current flash type.

uint8_t sectorCmd

uint8_t sectionCmd

uint8_t resourceCmd

uint8_t checkCmd

```
uint8_t swapCtrlCmd
uint8_t blockWriteUnitSize
uint8_t reserved[2]
struct _flash_ops_config addrAligment
uint32_t has4ByteIdxSupport
uint32_t has8ByteIdxSupport
uint32_t __pad0__
struct _flash_ifr_desc feature
uint8_t versionIdStart
uint8_t versionIdSize
uint16_t ifrMemSize
uint32_t pflashIfrStart
uint32_t dflashIfrStart
uint32_t pflashSwapIfrStart
struct _flash_ifr_desc resRange
uint16_t mix8byteIdxStart
uint16_t mix8byteIdxEnd
struct _flash_ifr_desc idxInfo
flash_mem_desc_t flashDesc
flash_ops_config_t opsConfig
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
uint16_t reserved
uint32_t *runCmdFuncAddr
    An buffer point to the flash execute-in-RAM function.
flash_ifr_desc_t ifrDesc
ftfx_config_t ftfxConfig[1]
uint32_t dummy[1]
uint8_t byte[4]
union _standard_version
    #include <fsl_flash.h>
```

```

struct _flash_swap_state_config
    #include <fsl_flash.h> Flash Swap information.
struct _flash_special_mem
    #include <fsl_flash.h> ftfx special memory access information.
struct _flash_mem_desc_t
    #include <fsl_flash.h> Flash memory descriptor.
struct _flash_ops_config
    #include <fsl_flash.h> Active FTFx information for the current operation.
struct _flash_ifr_desc
    #include <fsl_flash.h> Flash IFR memory descriptor.
struct _ftfx_config
    #include <fsl_flash.h> Flash driver state information.

    An instance of this structure is allocated by the user of the flash driver and passed into each
    of the driver APIs.
struct _flash_config
    #include <fsl_flash.h>
union UNION_Type_aligned_for_longword
    #include <fsl_flash.h>
struct B
struct feature
struct addrAligment
struct feature
struct resRange
struct idxInfo

```

2.31 Driver Change Log

2.32 FlexCAN Driver

```
void FLEXCAN_Init(CAN_Type *base, const flexcan_config_t *psConfig)
```

Initializes a FlexCAN instance with classic / FD mode supported.

This function initializes the FlexCAN module with user-defined settings. User can call the FLEXCAN_GetDefaultConfig function as a starting point for the needed init configuration structure.

Note: If user wants to initialize the peripheral in CAN FD mode, user can call the FLEXCAN_GetFDDefaultConfig to get the flexcan_fd_config_t and assign the address of the data structure to the member psFDConfig of flexcan_config_t

Parameters

- base – FlexCAN peripheral base address.
- psConfig – Pointer to the user-defined configuration structure.

```
void FLEXCAN_Deinit(CAN_Type *base)
```

De-initializes a FlexCAN instance.

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

- base – FlexCAN peripheral base address.

```
void FLEXCAN_GetDefaultConfig(flexcan_config_t *psConfig, uint32_t u32ClkFreqHz)
```

Gets the default configuration structure.

This API does not provide specific classic CAN bit timing configuration values (sTiming-Config), but set bEnableTimingCalc to true by default, and it will make FLEXCAN_Init() calculate and configure bit timing parameters based on the frequency of the peripheral (u32ClkFreqHz) and classic CAN bit rate (u32BaudRateBps).

```
psConfig->bEnableTimingCalc      = true;
psConfig->u32ClkFreqHz          = u32ClkFreqHz;
psConfig->eClkSrc                = kFLEXCAN_ClkSrc0;
psConfig->u32BaudRateBps        = 500000U;
psConfig->u8MaxMsgBufNum        = 16;
psConfig->bEnableLoopBack        = false;
psConfig->bEnableTimerSync       = true;
psConfig->bEnableSelfWakeup      = false;
psConfig->eWakeupSrc             = kFLEXCAN_WakeupSrcUnfiltered;
psConfig->bEnableIndividMask     = false;
psConfig->bDisableSelfReception  = false;
psConfig->bEnableListenOnlyMode  = false;
psConfig->bEnableDoze            = false;
psConfig->psFDConfig = NULL;
```

Note: By default, CAN FD mode is disabled and not configured.

Parameters

- psConfig – Pointer to the FlexCAN configuration structure.
- u32ClkFreqHz – Flexcan peripheral clock

```
void FLEXCAN_GetFDDefaultConfig(flexcan_config_t *psConfig, flexcan_fd_config_t *psFDConfig)
```

Gets the default configuration structure for Flexcan FD mode.

The function provided the default value of the data structure as code below

```
psConfig->u32BaudRateBps        = 2000000U;
psConfig->eMsgSize               = kFLEXCAN_64BperMB;
psConfig->bEnableTxDelayCompensation = true;
psConfig->bEnableBitRateSwitch   = true;
```

Parameters

- psConfig – Pointer to the FlexCAN configuration structure.
- psFDConfig – Pointer to FD configuration structure.

```
void FLEXCAN_EnterFreezeMode(CAN_Type *base)
```

Enter FlexCAN Freeze Mode.

This function makes the FlexCAN work under Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

```
void FLEXCAN_ExitFreezeMode(CAN_Type *base)
```

Exit FlexCAN Freeze Mode.

This function makes the FlexCAN leave Freeze Mode.

Parameters

- base – FlexCAN peripheral base address.

```
bool FLEXCAN_CalculateImprovedTimingValuesWithCBT(flexcan_timing_config_t
                                                    *psTimingConfig, uint32_t u32IdealSp,
                                                    uint32_t u32BaudRateBps, uint32_t
                                                    u32SrcClkHz)
```

Calculates the improved timing values by specific baud rates for classical CAN (use CBT register)

Note: The CBT register support 8 ~ 129 time quanta

Parameters

- psTimingConfig – Pointer to the FlexCAN timing configuration structure.
- u32IdealSp – The desired sample point, unit is one thousandth. (like 875, 800, 750).
- u32BaudRateBps – The classical CAN speed in bps defined by user.
- u32SrcClkHz – The Source clock data speed in bps.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

```
bool FLEXCAN_FDCalculateImprovedTimingValues(flexcan_timing_config_t *psTimingConfig,
                                              flexcan_timing_config_t *psFDTimingConfig,
                                              uint32_t u32BaudRateBps, uint32_t
                                              u32FDBaudRateBps, uint32_t u32SrcClkHz)
```

Calculates the improved timing values by specific baudrates for CANFD.

Parameters

- psTimingConfig – Pointer to the FlexCAN timing configuration structure.
- psFDTimingConfig – Pointer to the FlexCAN FD timing configuration structure.
- u32BaudRateBps – The CANFD bus control speed in bps defined by user.
- u32FDBaudRateBps – The CANFD bus data speed in bps defined by user.
- u32SrcClkHz – The Source clock data speed in bps.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

```
bool FLEXCAN_CalculateImprovedTimingValues(flexcan_timing_config_t *psTimingConfig,
                                              uint32_t u32BaudRateBps, uint32_t
                                              u32SrcClkHz)
```

Calculates the improved timing values by specific baud rates for classical CAN (use CTRL1 register)

Note: The CTRL1 register support 8 ~ 25 time quanta

Parameters

- psTimingConfig – Pointer to the FlexCAN timing configuration structure.
- u32BaudRateBps – The classical CAN speed in bps defined by user.
- u32SrcClkHz – The Source clock data speed in bps.

Returns

TRUE if timing configuration found, FALSE if failed to find configuration

status_t FLEXCAN_SetFDBaudRate(CAN_Type *base, uint32_t u32SrcClkHz, uint32_t u32BaudRateBps, uint32_t u32FDBaudRateBps)

Set Baud Rate of FlexCAN FD mode.

This function set the baud rate of FlexCAN FD frame.

Parameters

- base – FlexCAN peripheral base address.
- u32SrcClkHz – Source Clock in Hz.
- u32BaudRateBps – Frame Baud Rate in Bps.
- u32FDBaudRateBps – FD frame Baud Rate in Bps.

Return values

- kStatus_Success – Set baud rate success.
- kStatus_Fail – Set baud rate fail.

void FLEXCAN_SetFDTimingConfig(CAN_Type *base, const flexcan_timing_config_t *psConfig)

Sets the FlexCAN FD protocol timing characteristic.

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_Init() and fill the baud rate field with a desired value. This provides the default timing characteristics to the module.

Note that calling FLEXCAN_SetFDTimingConfig() overrides the baud rate set in FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- psConfig – Pointer to the timing configuration structure.

status_t FLEXCAN_SetBaudRate(CAN_Type *base, uint32_t u32SrcClkHz, uint32_t u32BaudRateBps)

Set Baud Rate of FlexCAN classic mode.

This function calculates the improved timing values base on specific baud rates and sets these values in timing register (CTRL1/CBT).

Parameters

- base – FlexCAN peripheral base address.
- u32SrcClkHz – Source Clock in Hz.
- u32BaudRateBps – Desired classic CAN Baud Rate in Bps.

Return values

- kStatus_Success – Set baud rate success.

- kStatus_Fail – Set baud rate fail.

void FLEXCAN_SetTimingConfig(CAN_Type *base, const flexcan_timing_config_t *psConfig)
Sets the FlexCAN protocol timing characteristic.

This function gives user settings to classic CAN timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_GetDefaultConfig and FLEXCAN_Init() to fill the baud rate field with auto-calculates values (base on the default baud rate). Users also can call FLEXCAN_SetBaudRate() to set auto-calculates timing values base on the desired baud rate.

Note: that calling FLEXCAN_SetTimingConfig() overrides the baud rate set in FLEXCAN_Init() or FLEXCAN_SetBaudRate().

Parameters

- base – FlexCAN peripheral base address.
- psConfig – Pointer to the timing configuration structure.

void FLEXCAN_SetRxMbGlobalMask(CAN_Type *base, uint32_t u32RecMsgBufs)
Sets the FlexCAN receive message buffer global mask.

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

- base – FlexCAN peripheral base address.
- u32RecMsgBufs – Rx Message Buffer Global Mask value.

void FLEXCAN_SetRxFifoGlobalMask(CAN_Type *base, uint32_t u32RecFifos)
Sets the FlexCAN receive FIFO global mask.

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

- base – FlexCAN peripheral base address.
- u32RecFifos – Rx Fifo Global Mask value.

void FLEXCAN_SetRxIndividualMask(CAN_Type *base, uint8_t u8MaskIdx, uint32_t u32Mask)
Sets the FlexCAN receive individual mask.

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the FLEXCAN_Init(). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

- base – FlexCAN peripheral base address.
- u8MaskIdx – The Index of individual Mask.
- u32Mask – Rx Individual Mask value.

void FLEXCAN_SetTxMbConfig(CAN_Type *base, uint8_t u8MsgBufIdx, bool bEnable)
Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The Message Buffer index.
- bEnable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

```
void FLEXCAN_SetFDTxMbConfig(CAN_Type *base, uint8_t u8MsgBufIdx, bool bEnable)
```

Configures a FlexCAN transmit message buffer.

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The Message Buffer index.
- bEnable – Enable/disable Tx Message Buffer.
 - true: Enable Tx Message Buffer.
 - false: Disable Tx Message Buffer.

```
void FLEXCAN_SetRxMbConfig(CAN_Type *base, uint8_t u8MsgBufIdx, const  
flexcan_rx_mb_config_t *psRxMsgBufConfig, bool bEnable)
```

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The Message Buffer index.
- psRxMsgBufConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- bEnable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

```
void FLEXCAN_SetFDRxMbConfig(CAN_Type *base, uint8_t u8MsgBufIdx, const  
flexcan_rx_mb_config_t *psRxMsgBufConfig, bool bEnable)
```

Configures a FlexCAN Receive Message Buffer.

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The Message Buffer index.
- psRxMsgBufConfig – Pointer to the FlexCAN Message Buffer configuration structure.
- bEnable – Enable/disable Rx Message Buffer.
 - true: Enable Rx Message Buffer.
 - false: Disable Rx Message Buffer.

```
void FLEXCAN_SetRxFifoConfig(CAN_Type *base, const flexcan_rx_fifo_config_t
                             *psRxFifoConfig, bool bEnable)
```

Configures the FlexCAN Rx FIFO.

This function configures the Rx FIFO with given Rx FIFO configuration.

Parameters

- base – FlexCAN peripheral base address.
- psRxFifoConfig – Pointer to the FlexCAN Rx FIFO configuration structure.
- bEnable – Enable/disable Rx FIFO.
 - true: Enable Rx FIFO.
 - false: Disable Rx FIFO.

```
static inline uint32_t FLEXCAN_GetStatusFlags(CAN_Type *base)
```

Gets the FlexCAN module interrupt flags.

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators `_flexcan_status_flags`. To check the specific status, compare the return value with enumerators in `_flexcan_status_flags`.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the `_flexcan_status_flags`.

```
static inline void FLEXCAN_ClearStatusFlags(CAN_Type *base, uint32_t u32StatusFlags)
```

Clears status flags with the provided mask.

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

- base – FlexCAN peripheral base address.
- u32StatusFlags – The status flags to be cleared, it is logical OR value of `_flexcan_status_flags`.

```
static inline void FLEXCAN_GetBusErrCount(CAN_Type *base, uint8_t *pu8TxErrBuf, uint8_t
                                           *pu8RxErrBuf)
```

Gets the FlexCAN Bus Error Counter value.

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

- base – FlexCAN peripheral base address.
- pu8TxErrBuf – Buffer to store Tx Error Counter value.
- pu8RxErrBuf – Buffer to store Rx Error Counter value.

```
static inline uint64_t FLEXCAN_GetMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Gets the FlexCAN Message Buffer interrupt flags.

This function gets the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

```
static inline void FLEXCAN_ClearMbStatusFlags(CAN_Type *base, uint64_t mask)
```

Clears the FlexCAN Message Buffer interrupt flags.

This function clears the interrupt flags of a given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_EnableInterrupts(CAN_Type *base, uint32_t u32InterruptFlags)
```

Enables FlexCAN interrupts according to the provided mask.

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.
- u32InterruptFlags – The interrupts to enable. Logical OR of `_flexcan_interrupt_enable`.

```
static inline void FLEXCAN_DisableInterrupts(CAN_Type *base, uint32_t u32InterruptFlags)
```

Disables FlexCAN interrupts according to the provided mask.

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see `_flexcan_interrupt_enable`.

Parameters

- base – FlexCAN peripheral base address.
- u32InterruptFlags – The interrupts to disable. Logical OR of `_flexcan_interrupt_enable`.

```
static inline void FLEXCAN_EnableMbInterrupts(CAN_Type *base, uint64_t mask)
```

Enables FlexCAN Message Buffer interrupts.

This function enables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
static inline void FLEXCAN_DisableMbInterrupts(CAN_Type *base, uint64_t mask)
```

Disables FlexCAN Message Buffer interrupts.

This function disables the interrupts of given Message Buffers.

Parameters

- base – FlexCAN peripheral base address.
- mask – The ORed FlexCAN Message Buffer mask.

```
void FLEXCAN_EnableRxFifoDMA(CAN_Type *base, bool bEnable)
```

Enables or disables the FlexCAN Rx FIFO DMA request.

This function enables or disables the DMA feature of FlexCAN build-in Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline uint32_t FLEXCAN_GetRxFifoHeadAddr(CAN_Type *base)
```

Gets the Rx FIFO Head address.

This function returns the FlexCAN Rx FIFO Head address, which is mainly used for the DMA/eDMA use case.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN Rx FIFO Head address.

```
static inline void FLEXCAN_Enable(CAN_Type *base, bool bEnable)
```

Enables or disables the FlexCAN module operation.

This function enables or disables the FlexCAN module.

Parameters

- base – FlexCAN base pointer.
- bEnable – true to enable, false to disable.

```
status_t FLEXCAN_WriteTxMb(CAN_Type *base, uint8_t u8MsgBufIdx, const flexcan_frame_t *psTxFrame)
```

Writes a classic CAN frame to the Transmit Message Buffer.

This function writes a classic CAN frame to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The FlexCAN Message Buffer index.
- psTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – Write Tx Message Buffer Successfully.
- kStatus_Fail – Tx Message Buffer is currently in use.

```
status_t FLEXCAN_ReadRxMb(CAN_Type *base, uint8_t u8MsgBufIdx, flexcan_frame_t *psRxFrame)
```

Reads a classic CAN frame from Receive Message Buffer.

This function reads a classic CAN frame from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The FlexCAN Message Buffer index.
- psRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_ReadRxFifo(CAN_Type *base, *flexcan_frame_t* *psRxFrame)

Reads a FlexCAN Message from Rx FIFO.

This function reads a CAN message from the FlexCAN build-in Rx FIFO.

Parameters

- base – FlexCAN peripheral base address.
- psRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.

status_t FLEXCAN_WriteFDTxMb(CAN_Type *base, uint8_t u8MsgBufIdx, const *flexcan_fd_frame_t* *psTxFrame)

Writes a CAN frame to the Transmit Message Buffer.

This function writes a CAN FD frame or classic CAN frame to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN FD Message transmit. After that the function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.
- psTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – Write Tx Message Buffer Successfully.
- kStatus_Fail – Tx Message Buffer is currently in use.

status_t FLEXCAN_ReadFDRxMb(CAN_Type *base, uint8_t u8MsgBufIdx, *flexcan_fd_frame_t* *psRxFrame)

Reads a CAN frame from Receive Message Buffer.

This function reads a CAN FD frame or classic CAN frame from a specified Receive Message Buffer. The function fills a receive CAN FD message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

- base – FlexCAN peripheral base address.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.
- psRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_TransferSendBlocking(CAN_Type *base, uint8_t u8MsgBufIdx, *flexcan_frame_t* *psTxFrame)

Performs a polling send transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- u8MsgBufIdx – The FlexCAN Message Buffer index.
- psTxFrame – Pointer to CAN message frame to be sent.

Return values

- kStatus_Success – Write Tx Message Buffer Successfully.
- kStatus_Fail – Tx Message Buffer is currently in use.

status_t FLEXCAN_TransferReceiveBlocking(CAN_Type *base, uint8_t u8MsgBufIdx, flexcan_frame_t *psRxFrame)

Performs a polling receive transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- u8MsgBufIdx – The FlexCAN Message Buffer index.
- psRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

status_t FLEXCAN_TransferReceiveFifoBlocking(CAN_Type *base, flexcan_frame_t *psRxFrame)

Performs a polling receive transaction from Rx FIFO on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- psRxFrame – Pointer to CAN message frame structure for reception.

Return values

- kStatus_Success – Read Message from Rx FIFO successfully.
- kStatus_Fail – Rx FIFO is not enabled.

status_t FLEXCAN_TransferFDSEndBlocking(CAN_Type *base, uint8_t u8MsgBufIdx, flexcan_fd_frame_t *psTxFrame)

Performs a polling send transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.
- psTxFrame – Pointer to CAN FD message frame to be sent.

Return values

- kStatus_Success – Write Tx Message Buffer Successfully.
- kStatus_Fail – Tx Message Buffer is currently in use.

status_t FLEXCAN_TransferFDReceiveBlocking(CAN_Type *base, uint8_t u8MsgBufIdx, *flexcan_fd_frame_t* *psRxFrame)

Performs a polling receive transaction on the CAN bus.

Note that a transfer handle does not need to be created before calling this API.

Parameters

- base – FlexCAN peripheral base pointer.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.
- psRxFrame – Pointer to CAN FD message frame structure for reception.

Return values

- kStatus_Success – Rx Message Buffer is full and has been read successfully.
- kStatus_FLEXCAN_RxOverflow – Rx Message Buffer is already overflowed and has been read successfully.
- kStatus_Fail – Rx Message Buffer is empty.

void FLEXCAN_TransferCreateHandle(CAN_Type *base, *flexcan_handle_t* *psHandle, *flexcan_transfer_callback_t* pfCallback, void *pUserData)

Initializes the FlexCAN handle.

This function initializes the FlexCAN handle, which can be used for other FlexCAN transactional APIs. Usually, for a specified FlexCAN instance, call this API once to get the initialized handle.

Parameters

- base – FlexCAN peripheral base address.
- psHandle – FlexCAN handle pointer.
- pfCallback – The callback function.
- pUserData – The parameter of the callback function.

status_t FLEXCAN_TransferSendNonBlocking(*flexcan_handle_t* *psHandle, *flexcan_mb_transfer_t* *psMsgBufXfer)

Sends a message using IRQ.

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- psHandle – FlexCAN handle pointer.
- psMsgBufXfer – FlexCAN Message Buffer transfer structure. See the *flexcan_mb_transfer_t*.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

status_t FLEXCAN_TransferReceiveNonBlocking(*flexcan_handle_t* *psHandle, *flexcan_mb_transfer_t* *psMsgBufXfer)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- psHandle – FlexCAN handle pointer.
- psMsgBufXfer – FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

- kStatus_Success – Start Rx Message Buffer receiving process successfully.
- kStatus_FLEXCAN_RxBusy – Rx Message Buffer is in use.

`status_t FLEXCAN_TransferReceiveFifoNonBlocking(flexcan_handle_t *psHandle, flexcan_fifo_transfer_t *psFifoXfer)`

Receives a message from Rx FIFO using IRQ.

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

- psHandle – FlexCAN handle pointer.
- psFifoXfer – FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t.

Return values

- kStatus_Success – Start Rx FIFO receiving process successfully.
- kStatus_FLEXCAN_RxFifoBusy – Rx FIFO is currently in use.

`uint32_t FLEXCAN_GetTimeStamp(flexcan_handle_t *psHandle, uint8_t u8MsgBufIdx)`

Gets the detail index of Mailbox's Timestamp by handle.

Then function can only be used when calling non-blocking Data transfer (TX/RX) API, After TX/RX data transfer done (User can get the status by handler's callback function), we can get the detail index of Mailbox's timestamp by handle, Detail non-blocking data transfer API (TX/RX) contain. -FLEXCAN_TransferSendNonBlocking -FLEXCAN_TransferFDSendNonBlocking -FLEXCAN_TransferReceiveNonBlocking -FLEXCAN_TransferFDReceiveNonBlocking -FLEXCAN_TransferReceiveFifoNonBlocking

Parameters

- psHandle – FlexCAN handle pointer.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.

Return values

the – index of mailbox 's timestamp stored in the handle.

`void FLEXCAN_TransferAbortSend(flexcan_handle_t *psHandle, uint8_t u8MsgBufIdx)`

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- psHandle – FlexCAN handle pointer.
- u8MsgBufIdx – The FlexCAN Message Buffer index.

`void FLEXCAN_TransferAbortReceive(flexcan_handle_t *psHandle, uint8_t u8MsgBufIdx)`

Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- psHandle – FlexCAN handle pointer.
- u8MsgBufIdx – The FlexCAN Message Buffer index.

void FLEXCAN_TransferAbortReceiveFifo(*flexcan_handle_t* *psHandle)

Aborts the interrupt driven message receive from Rx FIFO process.

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

- psHandle – FlexCAN handle pointer.

void FLEXCAN_TransferHandleIRQ(*flexcan_handle_t* *psHandle)

FlexCAN IRQ handle function.

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

- psHandle – FlexCAN handle pointer.

status_t FLEXCAN_TransferFDSendNonBlocking(*flexcan_handle_t* *psHandle,
flexcan_mb_transfer_t *psMsgBufXfer)

Sends a message using IRQ.

This function sends a message (classic CAN frame or CAN FD frame) using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

- psHandle – FlexCAN handle pointer.
- psMsgBufXfer – FlexCAN FD Message Buffer transfer structure. See the *flexcan_mb_transfer_t*.

Return values

- kStatus_Success – Start Tx Message Buffer sending process successfully.
- kStatus_Fail – Write Tx Message Buffer failed.
- kStatus_FLEXCAN_TxBusy – Tx Message Buffer is in use.

status_t FLEXCAN_TransferFDReceiveNonBlocking(*flexcan_handle_t* *psHandle,
flexcan_mb_transfer_t *psMsgBufXfer)

Receives a message using IRQ.

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

- psHandle – FlexCAN handle pointer.
- psMsgBufXfer – FlexCAN FD Message Buffer transfer structure. See the *flexcan_mb_transfer_t*.

Return values

- kStatus_Success – Start Rx Message Buffer receiving process successfully.
- kStatus_FLEXCAN_RxBusy – Rx Message Buffer is in use.

void FLEXCAN_TransferFDAbortSend(*flexcan_handle_t* *psHandle, uint8_t u8MsgBufIdx)

Aborts the interrupt driven message send process.

This function aborts the interrupt driven message send process.

Parameters

- psHandle – FlexCAN handle pointer.
- u8MsgBufIdx – The FlexCAN FD Message Buffer index.

`void FLEXCAN_TransferFDAbortReceive(flexcan_handle_t *psHandle, uint8_t u8MsgBufIdx)`
 Aborts the interrupt driven message receive process.

This function aborts the interrupt driven message receive process.

Parameters

- `psHandle` – FlexCAN handle pointer.
- `u8MsgBufIdx` – The FlexCAN FD Message Buffer index.

`FSL_FLEXCAN_DRIVER_VERSION`

FlexCAN driver version.

FlexCAN transfer status codes, used by bus operation APIs and transactional APIs as return value to indicate the current status as the API's execution result, or used in the callback to indicate transfer results.

Values:

enumerator `kStatus_FLEXCAN_TxBusy`

Tx Message Buffer is Busy.

enumerator `kStatus_FLEXCAN_TxIdle`

Tx Message Buffer is Idle.

enumerator `kStatus_FLEXCAN_TxSwitchToRx`

Remote Message is send out and Message buffer changed to Receive one.

enumerator `kStatus_FLEXCAN_RxBusy`

Rx Message Buffer is Busy.

enumerator `kStatus_FLEXCAN_RxIdle`

Rx Message Buffer is Idle.

enumerator `kStatus_FLEXCAN_RxOverflow`

Rx Message Buffer is Overflowed.

enumerator `kStatus_FLEXCAN_RxFifoBusy`

Rx Message FIFO is Busy.

enumerator `kStatus_FLEXCAN_RxFifoIdle`

Rx Message FIFO is Idle.

enumerator `kStatus_FLEXCAN_RxFifoOverflow`

Rx Message FIFO is overflowed.

enumerator `kStatus_FLEXCAN_RxFifoWarning`

Rx Message FIFO is almost overflowed.

enumerator `kStatus_FLEXCAN_ErrorStatus`

FlexCAN Module Error and Status.

enumerator `kStatus_FLEXCAN_WakeUp`

FlexCAN is waken up from STOP mode.

enumerator `kStatus_FLEXCAN_UnHandled`

UnHandled Interrupt asserted.

enumerator `kStatus_FLEXCAN_RxRemote`

Rx Remote Message Received in Mail box.

enum `_flexcan_frame_format`
FlexCAN frame format.

Values:

enumerator `kFLEXCAN_FrameFormatStandard`
Standard frame format attribute.

enumerator `kFLEXCAN_FrameFormatExtend`
Extend frame format attribute.

enum `_flexcan_frame_type`
FlexCAN frame type.

Values:

enumerator `kFLEXCAN_FrameTypeData`
Data frame type attribute.

enumerator `kFLEXCAN_FrameTypeRemote`
Remote frame type attribute.

enum `_flexcan_clock_source`
FlexCAN clock source.

Note: The user must ensure the protocol engine clock tolerance according to the CAN Protocol standard (ISO 11898-1).

Values:

enumerator `kFLEXCAN_ClkSrc0`
FlexCAN Protocol Engine clock selected by user as SRC == 0.

enumerator `kFLEXCAN_ClkSrc1`
FlexCAN Protocol Engine clock selected by user as SRC == 1.

enum `_flexcan_wake_up_source`
FlexCAN wake up source.

Values:

enumerator `kFLEXCAN_WakeupSrcUnfiltered`
FlexCAN uses unfiltered Rx input to detect edge.

enumerator `kFLEXCAN_WakeupSrcFiltered`
FlexCAN uses filtered Rx input to detect edge.

enum `_flexcan_rx_fifo_filter_type`
FlexCAN Rx Fifo Filter type.

Values:

enumerator `kFLEXCAN_RxFifoFilterTypeA`
One full ID (standard and extended) per ID Filter element.

enumerator `kFLEXCAN_RxFifoFilterTypeB`
Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

enumerator `kFLEXCAN_RxFifoFilterTypeC`
Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

enumerator `kFLEXCAN_RxFifoFilterTypeD`
All frames rejected.

enum `_flexcan_mb_size`

FlexCAN Message Buffer Data Size.

Values:

enumerator `kFLEXCAN_8BperMB`

Selects 8 bytes per Message Buffer.

enumerator `kFLEXCAN_16BperMB`

Selects 16 bytes per Message Buffer.

enumerator `kFLEXCAN_32BperMB`

Selects 32 bytes per Message Buffer.

enumerator `kFLEXCAN_64BperMB`

Selects 64 bytes per Message Buffer.

enum `_flexcan_rx_fifo_priority`

FlexCAN Rx FIFO priority.

The matching process starts from the Rx MB(or Rx FIFO) with higher priority. If no MB(or Rx FIFO filter) is satisfied, the matching process goes on with the Rx FIFO(or Rx MB) with lower priority.

Values:

enumerator `kFLEXCAN_RxFifoPriorityLow`

Matching process start from Rx Message Buffer first.

enumerator `kFLEXCAN_RxFifoPriorityHigh`

Matching process start from Rx FIFO first.

enum `_flexcan_interrupt_enable`

FlexCAN interrupt configuration structure, default settings all disabled.

This structure contains the settings for all of the FlexCAN Module interrupt configurations. Note: FlexCAN Message Buffers and Rx FIFO have their own interrupts.

Values:

enumerator `kFLEXCAN_BusOffInterruptEnable`

Bus Off interrupt.

enumerator `kFLEXCAN_ErrorInterruptEnable`

Error interrupt.

enumerator `kFLEXCAN_RxWarningInterruptEnable`

Rx Warning interrupt.

enumerator `kFLEXCAN_TxWarningInterruptEnable`

Tx Warning interrupt.

enumerator `kFLEXCAN_WakeUpInterruptEnable`

Wake Up interrupt.

enumerator `kFLEXCAN_AllInterruptEnable`

enum `_flexcan_status_flags`

FlexCAN status flags.

This provides constants for the FlexCAN status flags for use in the FlexCAN functions. Note: The CPU read action will clear all error flag in `kFLEXCAN_ErrorFlag`.

Values:

enumerator kFLEXCAN_FDStuffingError
Stuffing Error.

enumerator kFLEXCAN_FDFormError
Form Error.

enumerator kFLEXCAN_FDCrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_FDBit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_FDBit1Error
Unable to send recessive bit.

enumerator kFLEXCAN_StuffingError
Stuffing Error.

enumerator kFLEXCAN_FormError
Form Error.

enumerator kFLEXCAN_CrcError
Cyclic Redundancy Check Error.

enumerator kFLEXCAN_AckError
Received no ACK on transmission.

enumerator kFLEXCAN_Bit0Error
Unable to send dominant bit.

enumerator kFLEXCAN_Bit1Error
Unable to send recessive bit.

enumerator kFLEXCAN_ErrorFlag
All FlexCAN read clear Error Status.

enumerator kFLEXCAN_FDErrorIntFlag
FD frames Error Interrupt Flag.

enumerator kFLEXCAN_BusoffDoneIntFlag
Bus Off Done Interrupt Flag.

enumerator kFLEXCAN_OverrunError
Error Overrun Status.

enumerator kFLEXCAN_TxWarningIntFlag
Tx Warning Interrupt Flag.

enumerator kFLEXCAN_RxWarningIntFlag
Rx Warning Interrupt Flag.

enumerator kFLEXCAN_BusOffIntFlag
Bus Off Interrupt Flag.

enumerator kFLEXCAN_ErrorIntFlag
Error Interrupt Flag.

enumerator kFLEXCAN_WakeUpIntFlag
Wake-Up Interrupt Flag.

enumerator kFLEXCAN_SynchFlag
CAN Synchronization Status Flag.

enumerator `kFLEXCAN_TxErrorWarningFlag`
Tx Error Warning Status Flag.

enumerator `kFLEXCAN_RxErrorWarningFlag`
Rx Error Warning Status Flag.

enumerator `kFLEXCAN_IdleFlag`
CAN IDLE Status Flag.

enumerator `kFLEXCAN_FaultConfinementFlag`
Fault Confinement Status Flag.

enumerator `kFLEXCAN_TransmittingFlag`
FlexCAN In Transmission Status Flag.

enumerator `kFLEXCAN_ReceivingFlag`
FlexCAN In Reception Status Flag.

enumerator `kFLEXCAN_StatusAllFlags`
All status/interrupt flags which are write clearable.

enum `_flexcan_mb_flags`

FlexCAN Rx FIFO status flags.

The FlexCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Values:

enumerator `kFLEXCAN_RxFifoOverflowFlag`
Rx FIFO overflow flag.

enumerator `kFLEXCAN_RxFifoWarningFlag`
Rx FIFO almost full flag.

enumerator `kFLEXCAN_RxFifoFrameAvlFlag`
Frames available in Rx FIFO flag.

typedef enum `_flexcan_frame_format` `flexcan_frame_format_t`
FlexCAN frame format.

typedef enum `_flexcan_frame_type` `flexcan_frame_type_t`
FlexCAN frame type.

typedef enum `_flexcan_clock_source` `flexcan_clock_source_t`
FlexCAN clock source.

Note: The user must ensure the protocol engine clock tolerance according to the CAN Protocol standard (ISO 11898-1).

typedef enum `_flexcan_wake_up_source` `flexcan_wake_up_source_t`
FlexCAN wake up source.

typedef enum `_flexcan_rx_fifo_filter_type` `flexcan_rx_fifo_filter_type_t`
FlexCAN Rx Fifo Filter type.

typedef enum `_flexcan_mb_size` `flexcan_mb_size_t`
FlexCAN Message Buffer Data Size.

typedef enum *_flexcan_rx_fifo_priority* flexcan_rx_fifo_priority_t
FlexCAN Rx FIFO priority.

The matching process starts from the Rx MB(or Rx FIFO) with higher priority. If no MB(or Rx FIFO filter) is satisfied, the matching process goes on with the Rx FIFO(or Rx MB) with lower priority.

typedef struct *_flexcan_frame* flexcan_frame_t
FlexCAN message frame structure.

typedef struct *_flexcan_fd_frame* flexcan_fd_frame_t
CAN FD message frame structure.

Note: This structure can contain a CAN FD frame (bitEdl equal 1) or classic CAN frame (bitEdl equal 0).

typedef struct *_flexcan_timing_config* flexcan_timing_config_t
FlexCAN protocol timing characteristic configuration structure.

typedef struct *_flexcan_fd_config* flexcan_fd_config_t
Configuration for the Flexcan CAN FD mode. It only cover FD related parts.

It is used as member of flexcan_config_t to do the CAN FD specific init on init this peripheral.

typedef struct *_flexcan_config* flexcan_config_t
FlexCAN module configuration structure.

typedef struct *_flexcan_rx_mb_config* flexcan_rx_mb_config_t
FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

typedef struct *_flexcan_rx_fifo_config* flexcan_rx_fifo_config_t
FlexCAN Rx FIFO configuration structure.

typedef struct *_flexcan_mb_transfer* flexcan_mb_transfer_t
FlexCAN Message Buffer transfer.

typedef struct *_flexcan_fifo_transfer* flexcan_fifo_transfer_t
FlexCAN Rx FIFO transfer.

typedef struct *_flexcan_handle* flexcan_handle_t
FlexCAN handle structure definition. .

typedef void (*flexcan_transfer_callback_t)(flexcan_handle_t *psHandle, status_t status, uint32_t result, void *pUserData)

FlexCAN transfer callback function.

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_FLEXCAN_ErrorStatus, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

uint32_t FLEXCAN_GetInstance(CAN_Type *base)
Get the FlexCAN instance from peripheral base address.

Parameters

- base – FlexCAN peripheral base address.

Returns

FlexCAN instance.

FLEXCAN_WAIT_TIMEOUT

DLC_LENGTH_DECODE(dlc)

FlexCAN frame length helper macro.

FLEXCAN_ID_STD(id)

FlexCAN Frame ID helper macro.

Standard Frame ID helper macro.

FLEXCAN_ID_EXT(id)

Extend Frame ID helper macro.

FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)

FlexCAN Rx Message Buffer Mask helper macro.

Standard Rx Message Buffer Mask helper macro.

FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)

Extend Rx Message Buffer Mask helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)

FlexCAN Rx FIFO Mask helper macro.

Standard Rx FIFO Mask helper macro Type A helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)

Standard Rx FIFO Mask helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)

Standard Rx FIFO Mask helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id)

Standard Rx FIFO Mask helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id)

Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id)

Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.

FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)

Standard Rx FIFO Mask helper macro Type C lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type A helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type B upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)

Extend Rx FIFO Mask helper macro Type B lower part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)

Extend Rx FIFO Mask helper macro Type C upper part helper macro.

FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)

Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)`
Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)`
Extend Rx FIFO Mask helper macro Type C lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide)`
FlexCAN Rx FIFO Filter helper macro.
Standard Rx FIFO Filter helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)`
Standard Rx FIFO Filter helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)`
Standard Rx FIFO Filter helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)`
Standard Rx FIFO Filter helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)`
Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)`
Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id)`
Standard Rx FIFO Filter helper macro Type C lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide)`
Extend Rx FIFO Filter helper macro Type A helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)`
Extend Rx FIFO Filter helper macro Type B upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)`
Extend Rx FIFO Filter helper macro Type B lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id)`
Extend Rx FIFO Filter helper macro Type C upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)`
Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)`
Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.

`FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id)`
Extend Rx FIFO Filter helper macro Type C lower part helper macro.

`struct _flexcan_frame`
`#include <fsl_flexcan.h>` FlexCAN message frame structure.

`struct _flexcan_fd_frame`
`#include <fsl_flexcan.h>` CAN FD message frame structure.

Note: This structure can contain a CAN FD frame (bitEdl equal 1) or classic CAN frame (bitEdl equal 0).

`struct _flexcan_timing_config`
`#include <fsl_flexcan.h>` FlexCAN protocol timing characteristic configuration structure.

Public Members

uint16_t u16PreDivider
Clock Pre-scaler Division Factor.

uint8_t u8RJumpWidth
Re-sync Jump Width.

uint8_t u8PhaseSeg1
Phase Segment 1.

uint8_t u8PhaseSeg2
Phase Segment 2.

uint8_t u8PropSeg
Propagation Segment.

struct *_flexcan_fd_config*

#include <fsl_flexcan.h> Configuration for the Flexcan CAN FD mode. It only cover FD related parts.

It is used as member of *flexcan_config_t* to do the CAN FD specific init on init this peripheral.

struct *_flexcan_config*

#include <fsl_flexcan.h> FlexCAN module configuration structure.

Public Members

bool bEnableTimingCalc
Enable or Disable the bit timing calculation in FLEXCAN_Init() API.

uint32_t u32BaudRateBps
FlexCAN baud rate in bps (for classical CAN or CANFD nominal phase). Only meaningful when the bEnableTimingCalc is true.

uint32_t u32ClkFreqHz
FlexCAN peripheral clock frequency in HZ. Only meaningful when the bEnableTimingCalc is true.

flexcan_timing_config_t sTimingConfig
Protocol timing configuration. Only meaningful when the bEnableTimingCalc is false.

flexcan_clock_source_t eClkSrc
Clock source for FlexCAN Protocol Engine.

uint8_t u8MaxMsgBufNum
The maximum number of Message Buffers used by user.

bool bEnableIndividMask
Enable or Disable Rx Individual Mask.

flexcan_wake_up_source_t eWakeupSrc
Wake up source selection.

bool bEnableSelfWakeup
Enable or Disable Self Wakeup Mode.

bool bEnableLoopBack
Enable or Disable Loop Back Self Test Mode.

bool bEnableTimerSync
Enable or Disable Timer Synchronization.

`bool bDisableSelfReception`
Enable or Disable Self Reflection.

`bool bEnableListenOnlyMode`
Enable or Disable Listen Only Mode.

`bool bEnableDoze`
Enable or Disable Doze Mode.

`struct _flexcan_rx_mb_config`

#include <fsl_flexcan.h> FlexCAN Receive Message Buffer configuration structure.

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Public Members

`uint32_t u32Id`

CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

`flexcan_frame_format_t eFormat`

CAN Frame Identifier format(Standard of Extend).

`flexcan_frame_type_t eType`

CAN Frame Type(Data or Remote).

`struct _flexcan_rx_fifo_config`

#include <fsl_flexcan.h> FlexCAN Rx FIFO configuration structure.

Public Members

`uint32_t *pu32IdFilterTable`

Pointer to the FlexCAN Rx FIFO identifier filter table.

`uint8_t u8IdFilterNum`

The quantity of filter elements.

`flexcan_rx_fifo_filter_type_t eIdFilterType`

The FlexCAN Rx FIFO Filter type.

`flexcan_rx_fifo_priority_t ePriority`

The FlexCAN Rx FIFO receive priority.

`struct _flexcan_mb_transfer`

#include <fsl_flexcan.h> FlexCAN Message Buffer transfer.

Public Members

`flexcan_frame_t *psFrame`

The buffer of CAN Message to be transfer.

`uint8_t u8MsgBufIdx`

The index of Message buffer used to transfer Message.

`struct _flexcan_fifo_transfer`

#include <fsl_flexcan.h> FlexCAN Rx FIFO transfer.

Public Members

flexcan_frame_t *psFrame

The buffer of CAN Message to be received from Rx FIFO.

struct *_flexcan_handle*

#include <fsl_flexcan.h> FlexCAN handle structure.

Public Members

flexcan_transfer_callback_t pfCallback

Callback function.

void *pUserData

FlexCAN callback function parameter.

flexcan_fd_frame_t *volatile psFDMbFrameBufs[CAN_WORD1_COUNT]

The buffer for received data from Message Buffers.

flexcan_frame_t *volatile psRxFifoFrameBuf

The buffer for received data from Rx FIFO.

volatile uint8_t u8MsgBufStates[CAN_WORD1_COUNT]

Message Buffer transfer state.

volatile uint8_t u8RxFifoState

Rx FIFO transfer state.

volatile uint32_t u32TimeStamps[CAN_WORD1_COUNT]

Mailbox transfer timestamp.

struct *__unnamed10__*

Public Members

uint32_t bitsTimestamp

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t bitsLength

CAN frame payload length in bytes(Range: 0~8).

uint32_t bitType

CAN Frame Type(DATA or REMOTE).

uint32_t bitFormat

CAN Frame Identifier(STD or EXT format).

uint32_t *__pad0__*

Reserved.

uint32_t bitsIdHit

CAN Rx FIFO filter hit id(This value is only used in Rx FIFO receive mode).

struct *__unnamed12__*

Public Members

uint32_t bitsId

CAN Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.

uint32_t __pad0__

Reserved.

union __unnamed14__

Public Members

struct __flexcan__frame

struct __flexcan__frame

struct __unnamed16__

Public Members

uint32_t u32DataWord0

CAN Frame payload word0.

uint32_t u32DataWord1

CAN Frame payload word1.

struct __unnamed18__

Public Members

uint8_t u8DataByte3

CAN Frame payload byte3.

uint8_t u8DataByte2

CAN Frame payload byte2.

uint8_t u8DataByte1

CAN Frame payload byte1.

uint8_t u8DataByte0

CAN Frame payload byte0.

uint8_t u8DataByte7

CAN Frame payload byte7.

uint8_t u8DataByte6

CAN Frame payload byte6.

uint8_t u8DataByte5

CAN Frame payload byte5.

uint8_t u8DataByte4

CAN Frame payload byte4.

struct __unnamed20__

Public Members

uint32_t bitsTimestamp

FlexCAN internal Free-Running Counter Time Stamp.

uint32_t bitsLength

CAN FD frame data length code (DLC), range see `_flexcan_fd_frame_length`, When the length ≤ 8 , it equals to the data length in bytes, otherwise the number of valid frame data is not equal to the length value. User can use `DLC_LENGTH_DECODE(length)` macro to get the number of valid data bytes.

uint32_t bitType

CAN Frame Type(DATA or REMOTE).

uint32_t bitFormat

CAN Frame Identifier(STD or EXT format).

uint32_t bitSrr

Substitute Remote request.

uint32_t bitsCode

Message Buffer Code.

uint32_t bitEsi

Error State Indicator.

uint32_t bitBrs

Bit Rate Switch.

uint32_t bitEdl

Extended Data Length.

struct `__unnamed22__`

Public Members

uint32_t bitsId

CAN Frame Identifier, should be set using `FLEXCAN_ID_EXT()` or `FLEXCAN_ID_STD()` macro.

uint32_t `__pad0__`

Reserved.

union `__unnamed24__`

Public Members

struct `_flexcan_fd_frame`

struct `_flexcan_fd_frame`

struct `__unnamed26__`

Public Members

uint32_t u32DataWord[16]

CAN FD Frame payload, 16 double word maximum.

struct `__unnamed28__`

Public Members

`uint8_t` `u8DataByte3`
CAN Frame payload byte3.

`uint8_t` `u8DataByte2`
CAN Frame payload byte2.

`uint8_t` `u8DataByte1`
CAN Frame payload byte1.

`uint8_t` `u8DataByte0`
CAN Frame payload byte0.

`uint8_t` `u8DataByte7`
CAN Frame payload byte7.

`uint8_t` `u8DataByte6`
CAN Frame payload byte6.

`uint8_t` `u8DataByte5`
CAN Frame payload byte5.

`uint8_t` `u8DataByte4`
CAN Frame payload byte4.

2.33 The Driver Change Log

2.34 FlexCAN Peripheral and Driver Overview

2.35 GPIO: General-Purpose Input/Output Driver

`void` `GPIO_PinInit`(`GPIO_Type` *`base`, `gpio_pin_t` `ePin`, `const` `gpio_config_t` *`psConfig`)

Initializes a GPIO pin with provided structure `gpio_config_t` covering all configuration fields.

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 function to configure all GPIO Pin configurable fields.

```
gpio_config_t sConfig = {
    .eDirection = kGPIO_DigitalOutput,
    .eMode      = kGPIO_ModeGpio,
    .eOutMode   = kGPIO_OutputOpenDrain,
    .eSlewRate  = kGPIO_SlewRateFast,
    .eOutLevel  = kGPIO_OutputLow,
    .eDriveStrength = kGPIO_DriveStrengthLow,
    .ePull      = kGPIO_PullDisable,
    .eInterruptMode = kGPIO_InterruptDisable,
}
GPIO_PinInit(GPIOA, kGPIO_Pin1, &psConfig);
```

Note: If GPIO glitch is critical for your application, do not use this API instead using the API in GPIO pin configuration interfaces to do with the glitch during GPIO mode transition in accordance to your board design.

Parameters

- `base` – GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- `psConfig` – GPIO pin configuration pointer

```
static inline void GPIO_PinSetPeripheralMode(GPIO_Type *base, gpio_pin_t ePin,
                                           gpio_peripheral_mode_t eMode)
```

Configure the GPIO Pin as Peripheral mode or GPIO mode for one pin.

Configure GPIO can be configured as Peripheral mode or GPIO mode for one pin.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- `eMode` – Peripheral mode or GPIO mode for this pin. See `gpio_peripheral_mode_t`

```
static inline void GPIO_PinSetPeripheralMux(gpio_peripheral_mux_t eMux)
```

Configure the multiplexing of GPIO pins to different peripheral.

Configure the MUX of GPIO pins to different peripheral functionality.

Note: User still need to call the `GPIO_PinSetPeripheralMode`.

Parameters

- `eMux` – GPIO peripheral MUX when configured as peripheral mode.

```
static inline void GPIO_PinSetDirection(GPIO_Type *base, gpio_pin_t ePin, gpio_direction_t
                                       eDirection)
```

Configure the GPIO pin as Input or Output for one pin.

Configure the GPIO pin as Input or Output for one pin.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- `eDirection` – Direction of GPIO pin. `gpio_direction_t`

```
static inline void GPIO_PinSetOutputMode(GPIO_Type *base, gpio_pin_t ePin,
                                         gpio_output_mode_t eOutMode)
```

Configure GPIO pin output as Push-Pull or Open-Drain for one pin.

Configure GPIO pin output as Push-Pull or Open-Drain. This function applies while pin is configured as output. See `gpio_direction_t` and API `GPIO_PinSetDirection`.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- eOutMode – Push-Pull/Open-Drain output mode. See `gpio_output_mode_t`.

```
static inline void GPIO_PinSetDriveStrength(GPIO_Type *base, gpio_pin_t ePin,  
                                           gpio_output_drive_strength_t eDriveStrength)
```

Configure High/Low drive strength when Pin is configured as output for one pin.

Configure High/Low drive strength when Pin is configured as output. See `gpio_direction_t` and API `GPIO_PinSetDirection`.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- eDriveStrength – High/Low driver strength. See `gpio_output_drive_strength_t`

```
static inline void GPIO_PinSetSlewRate(GPIO_Type *base, gpio_pin_t ePin,  
                                       gpio_output_slew_rate_t eSlewRate)
```

Configure GPIO pin Fast/Slow slew rate when pin is configured as output.

Configure GPIO pin Fast/Slow slew rate when pin is configured as output. See `gpio_direction_t` and API `GPIO_PinSetDirection`.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- eSlewRate – Fast/Slow slewrate. See `gpio_output_slew_rate_t`

```
static inline void GPIO_PinSetPullResistorMode(GPIO_Type *base, gpio_pin_t ePin,  
                                              gpio_pull_mode_t ePullMode)
```

Configure Pull resistor for GPIO pin to Disable/Pull-Up/Pull-Down.

Configure Pull resistor for GPIO pin to Disable/Pull-Up/Pull-Down.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- ePullMode – Pull Mode as Disable/Pull-Up/Pull-Down. See `gpio_pull_mode_t`

```
static inline void GPIO_PinWrite(GPIO_Type *base, gpio_pin_t ePin, gpio_output_level_t  
                                eOutput)
```

Set GPIO Pin as High/Low voltage level on Output.

Set GPIO Pin as High/Low voltage level on Output.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- eOutput – Output as High level or Low Level. See `gpio_output_level_t`.

```
static inline void GPIO_PortSet(GPIO_Type *base, uint16_t u16Pins)
```

Set GPIO multiple pins output High voltage level without impact pins.

Set GPIO multiple pins output High voltage level without impact other pins. Multiple pins are configured by OR enumerator from `gpio_pin_t`

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

```
static inline void GPIO_PinSet(GPIO_Type *base, gpio_pin_t ePin)
```

Output High voltage level for GPIO Pin when configured as Output.

Output High voltage level for GPIO Pin when configured as Output.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

```
static inline void GPIO_PortClear(GPIO_Type *base, uint16_t u16Pins)
```

Set GPIO multiple pins belong to same PORT output Low voltage level when these pins are configured as output.

Set GPIO multiple pins belong to same PORT output Low voltage level when these pins are configured as output. Multiple pins are configured by ORing enumerators from `gpio_pin_t`

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

```
static inline void GPIO_PinClear(GPIO_Type *base, gpio_pin_t ePin)
```

Output Low voltage level for GPIO Pin when configured as Output.

Output Low voltage level for GPIO Pin when configured as Output.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

```
static inline void GPIO_PortToggle(GPIO_Type *base, uint16_t u16Pins)
```

Toggle GPIO multiple pins belong to same PORT when these pins are configured as output.

Toggle GPIO multiple pins belong to same PORT when these pins are configured as output. Multiple pins are configured by ORing enumerators from `gpio_pin_t`

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

```
static inline void GPIO_PinToggle(GPIO_Type *base, gpio_pin_t ePin)
```

Toggle the GPIO output voltage level when configured as Output.

Toggle the GPIO output voltage level when configured as Output.

Note: GPIO peripheral register do not get register to toggle directly. It is implemented by read back the GPIO output level and write to the register with reverted level.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

```
static inline uint16_t GPIO_PortRead(GPIO_Type *base)
```

Read High/Low voltage level for multiple GPIO pins from the pin or the data bus.

Read High/Low voltage level for multiple GPIO pins from the pin if the pin is configured as input or the data bus. When the device comes out of reset, GPIO pins are configured as inputs with internal pull disabled. As a result, the reset value of this pin is undefined. For different PORT, the available pins number is different. User need use the return value to OR with the `gpio_pin_t` to decide whether that pin is logic high or low.

```
if (GPIO_PortRead(GPIOA) & (uint16_t)kGPIO_Pin0)
{
    //GPIOA Pin 0 is High
}
else
{
    //GPIOA Pin 0 is Low
}
```

Parameters

- `base` – GPIO peripheral base pointer

Returns

Voltage level for multiple GPIO pins from the pin or the data bus.

```
static inline uint8_t GPIO_PinRead(GPIO_Type *base, gpio_pin_t ePin)
```

Read High/Low voltage level for one GPIO pin from the pin or the data bus.

Read High/Low voltage level from the pin if the pin is configured as input or the data bus.

Note: When the device comes out of reset, GPIO pins are configured as inputs with internal pull disabled. As a result, the reset value of this pin is undefined.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

Return values

- 1 – Voltage level for one GPIO pin from the pin or the data bus is high.
- 0 – Voltage level for one GPIO pin from the pin or the data bus is low.

```
static inline uint16_t GPIO_PortReadRawData(GPIO_Type *base)
```

Read Raw voltage high/low level data from the pins or peripheral bus for multiple pins belong to same PORT.

Read Raw voltage high/low level data from the pins or peripheral bus. Values are not clocked and are subject to change at any time. Read several times to ensure a stable value.

The reset value of this register depends on the default PIN state. User need use the return value to OR with the `gpio_pin_t` to decide whether that pin is logic high or low.

```
if (GPIO_PortReadRawData(GPIOA) & (uint16_t)kGPIO_Pin0)
{
    //GPIOA Pin 0 is High
}
else
{
    //GPIOA Pin 0 is Low
}
```

Parameters

- `base` – GPIO peripheral base pointer

Returns

Voltage high/low level data from the pins or peripheral bus for multiple pins belong to same PORT.

```
static inline uint8_t GPIO_PinReadRawData(GPIO_Type *base, gpio_pin_t ePin)
```

Read Raw logic level data from the pins or peripheral bus for one pin.

Read Raw voltage high/low level data from the pins or peripheral bus. Values are not clocked and are subject to change at any time. Read several times to ensure a stable value. The reset value of this register depends on the default PIN state.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

Return values

- 1 – Raw voltage level data from the pins or peripheral bus is high.
- 0 – Raw voltage level data from the pins or peripheral bus is low.

```
static inline void GPIO_PinSetInterruptConfig(GPIO_Type *base, gpio_pin_t ePin,
                                             gpio_interrupt_mode_t eIntConfig)
```

Configure GPIO Pin interrupt detection condition.

Configure GPIO Pin interrupt detection condition

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.
- `eIntConfig` – Interrupt detection condition for rising edge/down edge or no detection. See `gpio_interrupt_mode_t`

```
static inline void GPIO_PortAssertSWInterrupts(GPIO_Type *base, uint16_t u16Pins)
```

Assert software interrupt for multiple pins belong to same port which will generate interrupt.

This API is only for software testing of a software interrupt capability. When the software interrupt is asserted, an interrupt is generated. The interrupt is generated continually until this software interrupt is de-asserted.

Parameters

- `base` – GPIO peripheral base pointer

- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

`static inline void GPIO_PortDeassertSWInterrupts(GPIO_Type *base, uint16_t u16Pins)`

De-Assert software interrupt for multiple pins belong to same port which will stop generating interrupt.

This API is only for software testing of a software interrupt capability.

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

`static inline void GPIO_PinAssertSWInterrupt(GPIO_Type *base, gpio_pin_t ePin)`

Assert software interrupt for one pin which will generate interrupt.

This API is only for software testing of a software interrupt capability. When the software interrupt is asserted, an interrupt is generated. The interrupt is generated continually until this software interrupt is de-asserted.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

`static inline void GPIO_PinDeassertSWInterrupt(GPIO_Type *base, gpio_pin_t ePin)`

De-Assert software interrupt for one pin which will stop generating interrupt.

This API is only for software testing of a software interrupt capability.

Parameters

- `base` – GPIO peripheral base pointer
- `ePin` – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

`static inline void GPIO_PortEnableInterrupts(GPIO_Type *base, uint16_t u16Pins)`

Enable interrupt detection for multiple pins belong to same port.

This API is to enable interrupt detection on rising edge or falling edge.

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

`static inline void GPIO_PortDisableInterrupts(GPIO_Type *base, uint16_t u16Pins)`

Disable interrupt detection for multiple pins belong to same port.

This API is to disable interrupt detection on rising edge or falling edge.

Parameters

- `base` – GPIO peripheral base pointer
- `u16Pins` – GPIO pins which is ORed by `gpio_pin_t`. # `kGPIO_Pin0` | `kGPIO_Pin3` means Pin 0 and Pin 3.

`static inline void GPIO_PinEnableInterrupt(GPIO_Type *base, gpio_pin_t ePin)`

Enable interrupt detection for one pin.

This API is to enable interrupt detection on rising edge or falling edge.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

```
static inline void GPIO_PinDisableInterrupt(GPIO_Type *base, gpio_pin_t ePin)
```

Disable interrupt detection for one pin.

This API is to disable interrupt detection on rising edge or falling edge.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

```
static inline uint16_t GPIO_PortGetInterruptPendingStatusFlags(GPIO_Type *base)
```

Get interrupt pending status flags all pins belong to same port.

Get interrupt pending status flags for all pins belong to same port. User need to use the `gpio_pin_t` to OR with the return value, if the result is not 0, this flag is set. otherwise, this flag is not set.

Note: this flags can only be cleared by calling `GPIO_PortClearEdgeDetectedStatusFlag` if it is caused by edge detected or by calling `GPIO_PortEnableSWInterrupt` if it is caused by SW interrupt.

```
if (GPIO_PortGetInterruptPendingStatusFlags(GPIOA) & (uint16_t)kGPIO_Pin0)
{
    //Interrupt occurred on GPIOA Pin 0.
}
else
{
    //No Interrupt on GPIOA Pin 0.
}
```

Parameters

- base – GPIO peripheral base pointer

Returns

Interrupt pending status flags all pins belong to same port.

```
static inline uint16_t GPIO_PinGetInterruptPendingStatusFlags(GPIO_Type *base, gpio_pin_t
ePin)
```

Get interrupt pending status flags for one pin.

Get interrupt pending status flags for one pin.

Note: this flags can only be cleared by calling `GPIO_PortClearEdgeDetectedStatusFlag` if it is caused by edge detected or by calling `GPIO_PortEnableSWInterrupt` if it is caused by SW interrupt.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by `gpio_pin_t`. Note that not all Pins existed in SoC and user need to check the data sheet.

Return values

- 1 – Interrupt occurred.
- 0 – No Interrupt.

static inline uint16_t GPIO_PortGetEdgeDetectedStatusFlags(GPIO_Type *base)

Get Edge detected status flags for all pins belong to same port.

Get edge detected status flags for all pins in the PORT. This status flag can only be detected when interrupt detection is enabled by GPIO_PortEnableInterrupt or GPIO_PinEnableInterrupt.

```
if (GPIO_PortGetEdgeDetectedStatusFlags(GPIOA) & (uint16_t)kGPIO_Pin0)
{
    //An edge detected on GPIOA Pin 0.
}
else
{
    //No edge detected on GPIOA Pin 0.
}
```

Parameters

- base – GPIO peripheral base pointer

Returns

Detected edge status flags for all pins belong to same port.

static inline uint8_t GPIO_PinGetEdgeDetectedStatusFlag(GPIO_Type *base, gpio_pin_t ePin)

Get Edge detected status flags for one pin.

Get edge detected status flags for one pin. This status flag can only be detected when interrupt detection is enabled by GPIO_PortEnableInterrupt or GPIO_PinEnableInterrupt.

Parameters

- base – GPIO peripheral base pointer
- ePin – GPIO pin identifier. User enumerator provided by gpio_pin_t. Note that not all Pins existed in SoC and user need to check the data sheet.

Return values

- 1 – An edge detected.
- 0 – No edge detected.

static inline void GPIO_PortClearEdgeDetectedStatusFlags(GPIO_Type *base, uint16_t u16Pins)

Clear Edge detected status flags for multiple pins belong to same port.

Clear Edge Detected status flags for multiple pins belong to same port.

Parameters

- base – GPIO peripheral base pointer
- u16Pins – GPIO pins which is ORed by gpio_pin_t. # kGPIO_Pin0 | kGPIO_Pin3 means Pin 0 and Pin 3.

static inline void GPIO_PinClearEdgeDetectedStatusFlags(GPIO_Type *base, gpio_pin_t ePin)

Clear Edge detected status flags for one pin.

Clear Edge Detected status flags for one pin.

Parameters

- base – GPIO peripheral base pointer

- ePin – GPIO pin identifier. User enumerator provided by gpio_pin_t. Note that not all Pins existed in SoC and user need to check the data sheet.

FSL_GPIO_DRIVER_VERSION

GPIO driver version.

enum _gpio_pin

GPIO Pin identifier with each pin get a unique bit thus they can be ORed.

Values:

enumerator kGPIO_Pin0

GPIO PORT Pin 0.

enumerator kGPIO_Pin1

GPIO PORT Pin 1.

enumerator kGPIO_Pin2

GPIO PORT Pin 2.

enumerator kGPIO_Pin3

GPIO PORT Pin 3.

enumerator kGPIO_Pin4

GPIO PORT Pin 4.

enumerator kGPIO_Pin5

GPIO PORT Pin 5.

enumerator kGPIO_Pin6

GPIO PORT Pin 6.

enumerator kGPIO_Pin7

GPIO PORT Pin 7.

enumerator kGPIO_Pin8

GPIO PORT Pin 8.

enumerator kGPIO_Pin9

GPIO PORT Pin 9.

enumerator kGPIO_Pin10

GPIO PORT Pin 10.

enumerator kGPIO_Pin11

GPIO PORT Pin 11.

enumerator kGPIO_Pin12

GPIO PORT Pin 12.

enumerator kGPIO_Pin13

GPIO PORT Pin 13.

enumerator kGPIO_Pin14

GPIO PORT Pin 14.

enumerator kGPIO_Pin15

GPIO PORT Pin 15.

enum _gpio_peripheral_mode

GPIO Pin peripheral/gpio mode option.

Values:

enumerator kGPIO_ModeGpio
Set GPIO pin as GPIO Mode.

enumerator kGPIO_ModePeripheral
Set GPIO pin as Peripheral Mode.

enum _gpio_direction
GPIO Pin input/output direction option.

Values:

enumerator kGPIO_DigitalInput
Set GPIO pin as digital input.

enumerator kGPIO_DigitalOutput
Set GPIO pin as digital output.

enum _gpio_pull_mode
GPIO Pin pull resistor mode option.

Values:

enumerator kGPIO_PullDown
Internal pull-down resistor is enabled.

enumerator kGPIO_PullUp
Internal pull-up resistor is enabled.

enumerator kGPIO_PullDisable
Internal pull-up/down resistor is disabled.

enum _gpio_output_mode
GPIO Pin output mode option.

Values:

enumerator kGPIO_OutputOpenDrain
Open drain output mode.

enumerator kGPIO_OutputPushPull
Push pull output mode.

enum _gpio_output_level
GPIO Pin output High/Low level option.

Values:

enumerator kGPIO_OutputLow
Set GPIO pin output low voltage level.

enumerator kGPIO_OutputHigh
Set GPIO pin output high voltage level.

enum _gpio_output_slew_rate
GPIO Pin output Fast/Slow slew rate option.

Values:

enumerator kGPIO_SlewRateFast
Set GPIO pin output Fast slew rate.

enumerator kGPIO_SlewRateSlow
Set GPIO pin output Slow slew rate.

enum `_gpio_output_drive_strength`
 GPIO Pin output High/Low drive strength option.
Values:
 enumerator `kGPIO_DriveStrengthLow`
 Set GPIO pin output Low-drive strength.
 enumerator `kGPIO_DriveStrengthHigh`
 Set GPIO pin output High-drive strength.

enum `_gpio_interrupt_mode`
 GPIO Pin interrupt detect option.
Values:
 enumerator `kGPIO_InterruptRisingEdge`
 Interrupt on rising edge.
 enumerator `kGPIO_InterruptFallingEdge`
 Interrupt on falling edge.
 enumerator `kGPIO_InterruptDisable`
 Interrupt is disabled.

typedef enum `_gpio_pin` `gpio_pin_t`
 GPIO Pin identifier with each pin get a unique bit thus they can be ORed.

typedef enum `_gpio_peripheral_mode` `gpio_peripheral_mode_t`
 GPIO Pin peripheral/gpio mode option.

typedef enum `_gpio_direction` `gpio_direction_t`
 GPIO Pin input/output direction option.

typedef enum `_gpio_pull_mode` `gpio_pull_mode_t`
 GPIO Pin pull resistor mode option.

typedef enum `_gpio_output_mode` `gpio_output_mode_t`
 GPIO Pin output mode option.

typedef enum `_gpio_output_level` `gpio_output_level_t`
 GPIO Pin output High/Low level option.

typedef enum `_gpio_output_slew_rate` `gpio_output_slew_rate_t`
 GPIO Pin output Fast/Slow slew rate option.

typedef enum `_gpio_output_drive_strength` `gpio_output_drive_strength_t`
 GPIO Pin output High/Low drive strength option.

typedef enum `_gpio_interrupt_mode` `gpio_interrupt_mode_t`
 GPIO Pin interrupt detect option.

typedef struct `_gpio_config` `gpio_config_t`
 GPIO Pin configuration covering all configurable fields when GPIO is configured in GPIO mode.

`GPIO_MUX_ENUM_TO_PORT_INDEX(emux)`
 Helper MACRO function to extract Port Index. (GPIOA, GPIOB, GPIOC, and so on.) The fields located in bit 8 - bit 11.

`GPIO_MUX_ENUM_TO_PIN_INDEX(emux)`
 Helper MACRO function to extract Pin Index. The fields located in bit 4 - bit 7.

GPIO_MUX_ENUM_TO_REG_VALUE(emux)

Helper MACRO function to extract Pin mux config register value. The fields located in bit 0 - bit 1.

GPIO_MUX_ENUM_TO_PIN_MASK(emux)

Helper MACRO function to extract Pin mux config mask.

GPIO_MUX_ENUM_TO_PIN_VALUE(emux)

Helper MACRO function to extract Pin mux config register value on a GPIO Pin.

struct `_gpio_config`

#include <fsl_gpio.h> GPIO Pin configuration covering all configurable fields when GPIO is configured in GPIO mode.

Public Members

gpio_direction_t eDirection

GPIO direction, input or output

gpio_peripheral_mode_t eMode

GPIO mode as peripheral or GPIO

gpio_peripheral_mux_t eMux

Set the peripheral type if GPIO is configured as peripheral

gpio_output_mode_t eOutMode

GPIO Open-Drain/Push-Pull output mode.

gpio_output_slew_rate_t eSlewRate

GPIO Fast/Slow slew rate output mode.

gpio_output_level_t eOutLevel

GPIO Output High/Low level.

gpio_output_drive_strength_t eDriveStrength

GPIO output Drive strength High/Low.

gpio_pull_mode_t ePull

GPIO Pull resistor mode configuration.

gpio_interrupt_mode_t eInterruptMode

GPIO interrupt detection condition configuration.

2.36 The Driver Change Log

2.37 GPIO Peripheral and Driver Overview

2.38 I2C: Inter-Integrated Circuit Driver

void I2C_MasterGetDefaultConfig(*i2c_master_config_t* *psMasterConfig, uint32_t u32SrcClockHz)

Sets the I2C master configuration structure to default values.

The purpose of this API is to initialize the configuration structure to default value for I2C_MasterInit to use. Use the unchanged structure in I2C_MasterInit or modify the structure before calling I2C_MasterInit. This is an example:

```
i2c_master_config_t config;
I2C_MasterGetDefaultConfig(&config, 12000000U);
config.u32BaudRateBps = 100000;
I2C_MasterInit(I2C0, &config);
```

Parameters

- psMasterConfig – Pointer to the master configuration structure.
- u32SrcClockHz – The clock source frequency for I2C module.

void I2C_MasterInit(I2C_Type *base, const *i2c_master_config_t* *psMasterConfig)

Initializes the I2C peripheral to operate as master.

This API initialize the I2C module for master operation, including the feature configuration of high drive capacity, double buffer, glitch filter, SCL timeout value, stop hold off enable and transfer baudrate. User can also configure whether to enable the module in the function.

The configuration structure can be filled manually or be set with default values by calling I2C_MasterGetDefaultConfig. This is an example.

```
I2C_MasterGetDefaultConfig(&config, 12000000U);
I2C_MasterInit(I2C0, &config);
```

Note: If FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL is enabled by user, the init function will not ungate I2C clock source before initialization, to avoid hardfault, user has to manually enable ungate the clock source before calling the API

Parameters

- base – I2C base pointer
- psMasterConfig – Pointer to the master configuration structure

void I2C_MasterDeinit(I2C_Type *base)

De-initializes the I2C peripheral. Call this API to disable the I2C module.

Parameters

- base – I2C base pointer

void I2C_SlaveGetDefaultConfig(*i2c_slave_config_t* *psSlaveConfig, uint16_t u16PrimaryAddress, uint32_t u32SrcClockHz)

Sets the I2C slave configuration structure to default values.

The purpose of this API is to initialize the configuration structure for I2C_SlaveInit to use. Use the unchanged initialized structure in I2C_SlaveInit or modify the structure before calling I2C_SlaveInit. This is an example.

```
i2c_slave_config_t config;
I2C_SlaveGetDefaultConfig(&config, 0x23U, 12000000U);
```

Parameters

- psSlaveConfig – Pointer to the slave configuration structure.
- u16PrimaryAddress – For 7-bit address low 7-bit is used, for 10-bit address low 10-bit is used.
- u32SrcClockHz – The clock source frequency for I2C module.

```
void I2C_SlaveInit(I2C_Type *base, const i2c_slave_config_t *psSlaveConfig)
```

Initializes the I2C peripheral to operate as slave.

This API initialize the I2C module for slave operation, including the feature configuration of high drive capacity, double buffer, glitch filter, SCL timeout value, stop hold off enable, addressing mode, alert response/general call monitoring, auto baudrate control, low power mode wake up, SCL/SDA setup and hold time. User can also configure whether to enable the module in the function.

The configuration structure can be filled manually or be set with default values by calling `I2C_SlaveGetDefaultConfig`. This is an example.

```
i2c_slave_config_t sConfig;
I2C_SlaveGetDefaultConfig(&sConfig, 12000000U);
sConfig.u8PrimaryAddress = 0x2AU;
I2C_SlaveInit(I2C0, &sConfig);
```

Note: If `FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL` is enabled by user, the init function will not ungate I2C clock source before initialization, to avoid hardfault, user has to manually enable ungate the clock source before calling the API

Parameters

- base – I2C base pointer
- psSlaveConfig – Pointer to the slave configuration structure

```
static inline void I2C_SlaveDeinit(I2C_Type *base)
```

De-initializes the I2C peripheral. Call this API to disable the I2C module.

Parameters

- base – I2C base pointer

```
void I2C_GetDefaultConfig(i2c_config_t *psConfig, uint16_t u16PrimaryAddress, uint32_t
                        u32SrcClockHz)
```

Sets the I2C configuration structure to default values.

The purpose of this API is to initialize the configuration structure for `I2C_Init` to use. Use the unchanged initialized structure in `I2C_Init` or modify the structure before calling `I2C_Init`. This is an example.

```
i2c_config_t config;
I2C_GetDefaultConfig(&config, 0x23U, 12000000U);
```

Parameters

- psConfig – Pointer to the slave configuration structure.
- u16PrimaryAddress – For 7-bit address low 7-bit is used, for 10-bit address low 10-bit is used.
- u32SrcClockHz – The clock source frequency for I2C module.

```
void I2C_Init(I2C_Type *base, const i2c_config_t *psConfig)
```

Initializes the I2C peripheral.

This API initialize the I2C module, including the feature configuration of high drive capacity, double buffer, glitch filter, SCL timeout value, stop hold off enable, addressing mode, alert response/general call monitoring, auto baudrate control, low power mode wake up and baudrate. User can also configure whether to enable the module in the function.

The configuration structure can be filled manually or be set with default values by calling `I2C_GetDefaultConfig`. This is an example.

```
i2c_config_t sConfig;
I2C_GetDefaultConfig(&sConfig, 0x23U, 12000000U);
I2C_Init(I2C0, &sConfig);
```

Note: If `FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL` is enabled by user, the init function will not ungate I2C clock source before initialization, to avoid hardfault, user has to manually enable ungate the clock source before calling the API

Parameters

- `base` – I2C base pointer
- `psConfig` – Pointer to the I2C configuration structure

```
static inline void I2C_Deinit(I2C_Type *base)
```

De-initializes the I2C peripheral. Call this API to disable the I2C module.

Parameters

- `base` – I2C base pointer

```
uint16_t I2C_MasterGetStatusFlags(I2C_Type *base)
```

Gets the I2C master hardware status flags.

Parameters

- `base` – I2C base pointer

Returns

the mask of status flags, can be a single flag or several flags in `_i2c_status_flags` ORed together.

```
static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint16_t u16StatusFlags)
```

Clears the I2C master status flags.

Parameters

- `base` – I2C base pointer
- `u16StatusFlags` – The status flag mask, can be a single flag or several flags in `_i2c_status_flags` ORed together. These flags among `_i2c_status_flags` can be cleared:
 - `kI2C_StartDetectInterruptFlag` (only supported on certain SoCs)
 - `kI2C_StopDetectInterruptFlag` (only supported on certain SoCs)
 - `kI2C_ArbitrationLostInterruptFlag`
 - `kI2C_InterruptPendingFlag`
 - `kI2C_RangeAddressMatchInterruptFlag`
 - `kI2C_AddressAsSlaveInterruptFlag`
 - `kI2C_SclLowTimeoutFlag`
 - `kI2C_SdaLowTimeoutInterruptFlag`

```
static inline uint16_t I2C_SlaveGetStatusFlags(I2C_Type *base)
```

Gets the I2C slave hardware status flags.

Parameters

- base – I2C base pointer

Returns

the mask of status flags, can be a single flag or several flags in `_i2c_status_flags` ORed together.

```
static inline void I2C_SlaveClearStatusFlags(I2C_Type *base, uint16_t u16StatusFlags)
```

Clears the I2C slave status flags.

Parameters

- base – I2C base pointer
- u16StatusFlags – The status flag mask, can be a single flag or several flags in `_i2c_status_flags` ORed together. These flags among `_i2c_status_flags` can be cleared:
 - kI2C_StartDetectInterruptFlag (only supported on certain SoCs)
 - kI2C_StopDetectInterruptFlag (only supported on certain SoCs)
 - kI2C_ArbitrationLostInterruptFlag
 - kI2C_InterruptPendingFlag
 - kI2C_RangeAddressMatchInterruptFlag
 - kI2C_AddressAsSlaveInterruptFlag
 - kI2C_SclLowTimeoutFlag
 - kI2C_SdaLowTimeoutInterruptFlag

```
void I2C_EnableInterrupts(I2C_Type *base, uint8_t u8Interrupts)
```

Enables I2C interrupt source.

Note: Before enabling `kI2C_GlobalInterruptEnable`, check `kI2C_RangeAddressMatchInterruptFlag` and `kI2C_AddressAsSlaveInterruptFlag` first, because any write operation on C1 will clear these 2 bits.

Parameters

- base – I2C base pointer
- u8Interrupts – The interrupt source mask, can be a single source or several sources in `_i2c_interrupt_enable` ORed together.

```
void I2C_DisableInterrupts(I2C_Type *base, uint8_t u8Interrupts)
```

Disables I2C interrupt source.

Note: Before disabling `kI2C_GlobalInterruptEnable`, check `kI2C_RangeAddressMatchInterruptFlag` and `kI2C_AddressAsSlaveInterruptFlag` first, because any write operation on C1 will clear these 2 bits.

Parameters

- base – I2C base pointer
- u8Interrupts – The interrupt source mask, can be a single source or several sources in `_i2c_interrupt_enable` ORed together.

```
uint8_t I2C_GetEnabledInterrupts(I2C_Type *base)
```

Get all the enabled interrupt sources.

Parameters

- base – I2C base pointer

Returns

The interrupt source mask, can be a single source or several sources in `_i2c_interrupt_enable` ORed together.

```
static inline bool I2C_IsMaster(I2C_Type *base)
```

Returns whether the I2C module is in master mode.

Parameters

- base – I2C base pointer

Returns

True for master mode, false for slave mode.

```
static inline void I2C_Reset(I2C_Type *base)
```

Sets the I2C register value to reset value.

Parameters

- base – I2C base pointer

```
static inline void I2C_Enable(I2C_Type *base, bool bEnable)
```

Enables or disables the I2C peripheral operation.

Parameters

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

```
static inline void I2C_EnableFastAck(I2C_Type *base, bool bEnable)
```

Enables/Disables fast NACK/ACK feature.

When enabled, writing 0/1 to TXAK generates an ACK/NACK after receiving a data byte, when disabled, writing 0/1 to TXAK generates an ACK/NACK on the following receiving data byte.

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

```
static inline void I2C_EnableHighDrive(I2C_Type *base, bool bEnable)
```

Enables/Disables high drive.

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

```
static inline void I2C_EnableStopHold(I2C_Type *base, bool bEnable)
```

Enables/Disables double buffer.

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

static inline void I2C_SetTransferDirection(I2C_Type *base, *i2c_data_direction_t* eDataDirection)
Sets I2C module data direction.

Parameters

- base – I2C base pointer
- eDataDirection – kI2C_Write to write data to bus, kI2C_Read to read data from bus

void I2C_SetSclTimeoutValue(I2C_Type *base, uint16_t u16SclTimeout_Ms, uint32_t u32SrcClockHz)

brief Sets the I2C SCL timeout value.

After the I2C module is initialized, user can call this function to change the timeout value.

param base I2C base pointer. param u16SclTimeout_Ms The SCL timeout value in ms. param u32SrcClockHz I2C peripheral clock frequency in Hz

void I2C_SetGlitchFilter(I2C_Type *base, uint16_t u16GlitchFilter_Ns, uint32_t u32SrcClockHz)
brief Sets the I2C master glitch filter width.

After the I2C module is initialized as master, user can call this function to change the glitch filter width.

param base I2C base pointer. param u16GlitchFilter_Ns The GLitch filter length in nano seconds. param u32SrcClockHz I2C peripheral clock frequency in Hz

static inline void I2C_EnableDMA(I2C_Type *base, bool bEnable)
Enables/disables the I2C DMA request.

Parameters

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

static inline uint32_t I2C_GetDataRegAddr(I2C_Type *base)

Gets the I2C data register address. This API is used to provide the transfer address for I2C DMA transfer.

Parameters

- base – I2C base pointer

Returns

data register address

static inline void I2C_SlaveEnableAlertResponse(I2C_Type *base, bool bEnable)
Enables/Disables alert response.

When enabled, I2C slave will monitor the bus line, and when alert response address is received address match status flag will be set.

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

static inline void I2C_SlaveEnableSecondaryAddress(I2C_Type *base, bool bEnable)
Enables/Disables secondary address.

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

```
static inline void I2C_SlaveEnableGeneralCall(I2C_Type *base, bool bEnable)
    Enables/Disables general call.
```

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

```
static inline void I2C_SlaveEnableWakeUp(I2C_Type *base, bool bEnable)
    Enables/Disables slave low power wakeup.
```

Parameters

- base – I2C base pointer
- bEnable – True to enable, false to disable.

```
void I2C_SlaveSetAddressingMode(I2C_Type *base, i2c_slave_address_mode_t eAddressingMode,
    uint16_t u16Address, uint8_t u8MaxAddress)
```

Configure the slave addressing mode.

After the I2C module is initialized as slave, user can call this function to change the configuration of slave addressing mode.

Parameters

- base – I2C base pointer.
- eAddressingMode – The slave addressing mode, single address or range address.
- u16Address – I2C slave address. For 7-bit address low 7-bit is used, for 10-bit address low 10-bit is used.
- u8MaxAddress – The maximum boundary of slave address used in a range address match.

```
void I2C_MasterSetBaudRate(I2C_Type *base, uint32_t u32BaudRateBps, uint32_t
    u32SrcClockHz)
```

Sets the I2C master transfer baud rate.

After the I2C module is initialized as master, user can call this function to change the transfer baud rate.

Parameters

- base – I2C base pointer.
- u32BaudRateBps – the baud rate value in bits-per-second.
- u32SrcClockHz – I2C peripheral clock frequency in Hz

```
static inline uint8_t I2C_ReadByte(I2C_Type *base)
    Reads one byte from data register directly.
```

Parameters

- base – I2C base pointer

Returns

The data read from data register.

```
static inline void I2C_WriteByte(I2C_Type *base, uint8_t u8Data)
    Writes one byte to the data register directly.
```

Parameters

- base – I2C base pointer
- u8Data – The byte to write.

```
static inline void I2C_SendAck(I2C_Type *base, bool bIsAck)
```

Sends an acknowledge/non-acknowledge signal to the bus on the following receiving byte if SMB[FAACK] is cleared, or on the current receiving byte if SMB[FAACK] is set.

Parameters

- base – I2C base pointer
- bIsAck – True to send ACK signal, false to send NACK signal

```
status_t I2C_MasterStart(I2C_Type *base, uint8_t u8Address, i2c_master_transfer_direction_t eDirection)
```

Sends a START signal on the I2C bus then the 7-bit slave address with transmit/receive bit.

This function is used to initiate a new transfer in master mode, by sending a START signal, then the slave address with transmit/receive bit to I2C bus.

Note: The return value of this API only indicates whether the start signal is sent to bus, user has to check `kI2C_ArbitrationLostInterruptFlag` and `kI2C_ReceiveNakFlag` using `I2C_MasterClearStatusFlags` to see if valid slave device is available or the arbitration is lost.

Parameters

- base – I2C base pointer
- u8Address – 7-bit slave device address.
- eDirection – Master transfer directions(transmit/receive).

Return values

- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flags to change.
- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy.

```
status_t I2C_MasterRepeatedStart(I2C_Type *base, uint8_t u8Address, i2c_master_transfer_direction_t eDirection)
```

Sends a repeated START signal, then the 7-bit slave address with transmit/receive bit to I2C bus.

Parameters

- base – I2C peripheral base pointer
- u8Address – 7-bit slave device address.
- eDirection – Master transfer directions(transmit/receive).

Return values

- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flags to change.
- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy but not occupied by current I2C master.

```
status_t I2C_MasterStop(I2C_Type *base)
```

Sends a STOP signal on the I2C bus.

Parameters

- base – I2C base pointer

Return values

- `kStatus_Success` – Successfully send the stop signal.
- `kStatus_I2C_Timeout` – Send stop signal failed, timeout.

`status_t I2C_MasterWriteBlocking(I2C_Type *base, const uint8_t *pu8Data, uint16_t txSize, bool bSendStop)`

Sends a piece of data to I2C bus in master mode in blocking way.

Call this function when using I2C as master to send certain bytes of data to bus when start signal is already sent. User can specify whether to send a stop signal after the data. This function uses the blocking way, which means it does not return until all the data is sent to bus and stop signal is successfully issued (if user configures the stop signal).

Parameters

- `base` – I2C base pointer.
- `pu8Data` – The pointer to the data to be transmitted.
- `txSize` – The length in bytes of the data to be transmitted.
- `bSendStop` – Whether to send stop signal after the data transfer.

Return values

- `kStatus_Success` – Successfully complete the data transmission.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.
- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flags to change.

`status_t I2C_MasterReadBlocking(I2C_Type *base, uint8_t *pu8RxBuff, uint16_t rxSize, bool bSendStop)`

Receives a piece of data from I2C bus in master mode in blocking way.

Call this function when using I2C as master to receive certain bytes of data from bus when start signal is already sent. User can specify whether to send a stop signal after the data. This function uses the blocking way, which means it does not return until all the data has been received and stop signal is successfully issued (if user configures the stop signal).

Note: If user configures to send stop signal after the data, this function stops the bus before reading the final byte from data register. Without stopping the bus prior to the final read, the bus will issue another read, resulting in garbage data being read into the data register.

Parameters

- `base` – I2C base pointer.
- `pu8RxBuff` – The pointer to the data to store the received data.
- `rxSize` – The length in bytes of the data to be received.
- `bSendStop` – Whether to send stop signal after the data transfer.

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 *pu8TxBuff, uint16_t txSize)`

Sends a piece of data to I2C bus in slave mode in blocking way.

Call this function to let I2C slave poll register status until it is addressed, then slave sends `txSize` of data to bus until all the data has been sent to bus or until it is nacked.

Parameters

- `base` – I2C base pointer.
- `pu8TxBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Return values

- `kStatus_Success` – Successfully complete the data transmission.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.
- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flags to change.

`status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *pu8RxBuff, uint16_t rxSize)`

Receives a piece of data from I2C bus in slave mode in blocking way.

Call this function to let I2C slave poll register status until it is addressed, then slave receives `rxSize` of data until all the data has been received.

Parameters

- `base` – I2C base pointer.
- `pu8RxBuff` – 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 *psTransferConfig)`

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.
- `psTransferConfig` – Pointer to the transfer configuration structure.

Return values

- `kStatus_Success` – Successfully complete the data transmission.
- `kStatus_I2C_Busy` – Previous transmission still not finished.
- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flagsto change.

- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.

```
void I2C_MasterTransferCreateHandle(I2C_Type *base, i2c_master_transfer_handle_t *psHandle,
                                   i2c_master_transfer_callback_t pfCallback, void
                                   *pUserData)
```

Initializes the I2C master transfer in interrupt way.

This function is responsible for initializig master transfer `psHandle`, installing user call-back, registering master IRQ handling function and opening global interrupt.

Parameters

- `base` – I2C base pointer.
- `psHandle` – pointer to `i2c_master_transfer_handle_t` structure to store the transfer state.
- `pfCallback` – pointer to user callback function.
- `pUserData` – User configurable pointer to any data, function, structure etc that user wish to use in the callback

```
status_t I2C_MasterTransferNonBlocking(i2c_master_transfer_handle_t *psHandle,
                                       i2c_master_transfer_t *psTransferConfig)
```

Initiates a master transfer on the I2C bus in interrupt way.

Note: Transfer in interrupt way is non-blocking which means this API returns immediately after transfer initiates. User can call `I2C_MasterTransferGetCount` to get the count of data that master has transmitted/received and check transfer status. If the return status is `kStatus_NoTransferInProgress`, the transfer is finished. Also if user installs a user callback when calling `I2C_MasterTransferCreateHandle` before, the callback will be invoked when transfer finishes.

Parameters

- `psHandle` – pointer to `i2c_master_transfer_handle_t` structure which stores the transfer state.
- `psTransferConfig` – Pointer to the transfer configuration structure.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_I2C_Busy` – Previous transmission still not finished.
- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flagsto change.

```
status_t I2C_MasterTransferGetCount(i2c_master_transfer_handle_t *psHandle, uint16_t *count)
```

Gets the master transfer count and status during a interrupt transfer.

Parameters

- `psHandle` – pointer to `i2c_master_transfer_handle_t` structure which stores the transfer state.
- `count` – Pointer to number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_NoTransferInProgress` – Curent no transfer is in progress.

- `kStatus_Success` – Successfully obtained the count.

`status_t I2C_MasterTransferAbort(i2c_master_transfer_handle_t *psHandle)`

Aborts an in-process transfer in interrupt way.

Note: This API can be called at any time after a transfer of interrupt way initiates and before it finishes to abort the transfer early.

Parameters

- `psHandle` – pointer to `i2c_master_transfer_handle_t` structure which stores the transfer state

Return values

- `kStatus_I2C_Busy` – Master lost arbitration, bus is in use by other master.
- `kStatus_I2C_Timeout` – Transfer error, timeout happens when waiting for status flags to change.
- `kStatus_Success` – Successfully abort the transfer.

`void I2C_SlaveTransferCreateHandle(I2C_Type *base, i2c_slave_transfer_handle_t *psHandle, i2c_slave_transfer_callback_t pfCallback, void *pUserData)`

Initializes the I2C slave transfer in interrupt way.

This function is responsible for initializing slave transfer handle, installing user callback, registering slave IRQ handling function and opening global interrupt.

Parameters

- `base` – I2C base pointer.
- `psHandle` – pointer to `i2c_slave_transfer_handle_t` structure to store the transfer state.
- `pfCallback` – pointer to user callback function.
- `pUserData` – User configurable pointer to any data, function, structure etc that user wish to use in the callback

`status_t I2C_SlaveTransferNonBlocking(i2c_slave_transfer_handle_t *psHandle, i2c_slave_transfer_t *psTransferConfig)`

Sets I2C slave ready to process bus events.

Call this API to let I2C start monitoring bus events driven by I2C master on bus. When user specified event occurs, callback will be invoked passes event information to the callback.

Note: When `kI2C_SlaveOutOfTransmitDataEvent` and `kI2C_SlaveOutOfReceiveSpaceEvent` occurred, slave callback will always be revoked regardless which events user choose to enable. This means user need not configure them in the `psTransferConfig`. If user wants to enable all the events, use `kI2C_SlaveAllEvents` for convenience.

Parameters

- `psHandle` – Pointer to `i2c_slave_transfer_handle_t` structure which stores the transfer state.
- `psTransferConfig` – I2C transfer configuration.

Return values

- `kStatus_Success` – I2C slave set to standby state successfully and ready to process events.

- `kStatus_I2C_Busy` – I2C slave has already been started on this handle.

`status_t I2C_SlaveTransferGetCount(i2c_slave_transfer_handle_t *psHandle, uint16_t *count)`

Gets how many bytes slave have transferred in current data buffer.

Parameters

- `psHandle` – pointer to `i2c_slave_transfer_handle_t` structure.
- `count` – Number of bytes slave have transferred after the last start/repeated start.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_NoTransferInProgress` – Current no transfer is in progress.
- `kStatus_Success` – Successfully obtained the count.

`void I2C_SlaveTransferAbort(i2c_slave_transfer_handle_t *psHandle)`

Aborts the slave transfer.

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

Parameters

- `psHandle` – pointer to `i2c_slave_transfer_handle_t` structure which stores the transfer state.

`FSL_I2C_DRIVER_VERSION`

I2C driver version.

I2C API status codes, used by bus operation APIs and transactional APIs as return value to indicate the bus's current status as the API's execution result, or used in the callback to indicate transfer results.

Values:

enumerator `kStatus_I2C_Busy`

I2C bus is busy.

enumerator `kStatus_I2C_Idle`

I2C Bus is idle.

enumerator `kStatus_I2C_Nak`

I2C detected NACK on bus. When in SMBus mode, this means the receiver nacks transmitter before PEC byte.

enumerator `kStatus_I2C_ArbitrationLost`

I2C lost arbitration during addressing.

enumerator `kStatus_I2C_Timeout`

Timeout happens when waiting for status flags to change.

enumerator `kStatus_I2C_Addr_Nak`

NACK was detected during the address probe.

enumerator `kStatus_I2C_Pec_Error`

Detected NACK for the PEC byte in transmit, or the received PEC does not match with the calculated CRC.

enum `_i2c_status_flags`

I2C hardware status flags.

These enumerations can be ORed together to form bit masks. The masks can be used as parameter by `I2C_MasterClearStatusFlags` and `I2C_SlaveClearStatusFlags`, or as return value by `I2C_MasterGetStatusFlags` and `I2C_SlaveGetStatusFlags`.

Values:

enumerator `kI2C_ReceiveNakFlag`

I2C received none ack flag

enumerator `kI2C_InterruptPendingFlag`

I2C interrupt pending flag. Byte transfer complete, address match, arbitration lost, start/stop detection and SMBus timeout can all cause this status bit to set. This flag can be cleared.

enumerator `kI2C_SlaveTransmitFlag`

I2C slave write/read status flag

enumerator `kI2C_RangeAddressMatchInterruptFlag`

Received address is within address range. This flag can be cleared.

enumerator `kI2C_ArbitrationLostInterruptFlag`

This flag can be cleared.

enumerator `kI2C_BusBusyFlag`

I2C bus busy flag.

enumerator `kI2C_AddressAsSlaveInterruptFlag`

Addressed as slave, including general call, alert response, primary/secondary address and range address match. This flag can be cleared.

enumerator `kI2C_ByteTransferCompleteInterruptFlag`

I2C transfer complete flag.

enumerator `kI2C_StopDetectInterruptFlag`

This flag can be cleared.

enumerator `kI2C_StartDetectInterruptFlag`

This flag can be cleared.

enumerator `kI2C_SclLowTimeoutFlag`

I2C SCL signal low timeout flag. This flag can be cleared.

enumerator `kI2C_BusIdleFlag`

I2C SCL and SDA both high timeout flag indicating bus idle.

enumerator `kI2C_SdaLowTimeoutInterruptFlag`

I2C SDA signal low timeout flag. This flag can be cleared.

enumerator `kI2C_StatusAllFlags`

All flags which are clearable.

enum `_i2c_interrupt_enable`

I2C interrupt enable/disable source.

These enumerations can be ORed together to form bit masks. The masks can be used as parameter by `I2C_EnableInterrupts`, `I2C_DisableInterrupts`, or as return value by `I2C_GetEnabledInterrupts`.

Values:

enumerator `ki2c_GlobalInterruptEnable`
I2C global interrupt.

enumerator `ki2c_StartStopDetectInterruptEnable`
I2C start&stop detect interrupt.

enumerator `ki2c_SdaLowTimeoutInterruptEnable`
I2C SDA low timeout interrupt.

enumerator `ki2c_AllInterruptEnable`

enum `_i2c_slave_address_mode`
Slave addressing mode, address match or range address match.

Values:

enumerator `ki2c_AddressMatch`
7-bit addressing mode.

enumerator `ki2c_AddressRangeMatch`
Range address match addressing mode.

enumerator `ki2c_AddressMatch10bit`
10-bit addressing mode.

enum `_i2c_data_direction`
Peripheral data direction.

Values:

enumerator `ki2c_Read`
I2C read data from bus.

enumerator `ki2c_Write`
I2C write data to bus.

enum `_i2c_master_transfer_control_flags`
I2C master transfer control flags.

These flags can be ORed together to form bit mask. The mask is used to configure master transfer's start/stop condition in `i2c_master_transfer_t::u8ControlFlagMask`.

Values:

enumerator `ki2c_TransferStartStopFlag`
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_master_transfer_direction`
Master transfer direction.

Values:

enumerator `ki2c_MasterTransmit`
Master transmits data to slave.

enumerator `kI2C_MasterReceive`

Master receives data from slave.

enum `_i2c_slave_transfer_event`

Set of slave transfer events.

This enumeration lists all the protocol level events that may happen during slave transfer. They can be used for two related purposes:

- a. User can select certain events and combined them by OR operation to form a mask, and use the mask to configure slave transfer configuration structure `i2c_slave_transfer_t::u8EventMask`. If any of these selected events happens, driver will alert user by invoking callback.
- b. When slave callback is invoked, user has to know which specific event occurred. Callback uses slave transfer configuration structure `i2c_slave_transfer_t` as 2nd parameter, its member `i2c_slave_transfer_t::eEvent` shows which event just happened.

Values:

enumerator `kI2C_SlaveAddressMatchEvent`

Slave detects general call or alert response address, or primary/secondary/range address is matched after a start or repeated start.

enumerator `kI2C_SlaveOutOfTransmitDataEvent`

Slave runs out of data to transmit, request a new data buffer.

enumerator `kI2C_SlaveOutOfReceiveSpaceEvent`

Slave runs out of space to store received data, request a new data buffer.

enumerator `kI2C_SlaveStartEvent`

A start/repeated start was detected.

enumerator `kI2C_SlaveCompletionEvent`

Slave detects a stop signal, or slave is nacked by master during master-receive, or slave has finished transmit/receive previously configured amount of data.

enumerator `kI2C_SlaveGeneralCallEvent`

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

enumerator `kI2C_SlaveAllEvents`

A bit mask of all available events.

typedef struct `_i2c_master_config` `i2c_master_config_t`

I2C master configuration structure.

This structure includes all the master operation needed features, user can configure these features one by one manually, or call `I2C_MasterGetDefaultConfig` to set the structure to default value. Then, call `I2C_MasterInit` to initialize I2C module. After initialization, the I2C module can only operate as master. To deinitialize I2C, call `I2C_MasterDeinit`.

typedef enum `_i2c_slave_address_mode` `i2c_slave_address_mode_t`

Slave addressing mode, address match or range address match.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`

I2C slave configuration structure.

This structure includes all the slave operation needed features, user can configure these features one by one manually, or call `I2C_SlaveGetDefaultConfig` to set the structure to default value. Then, call `I2C_SlaveInit` to initialize I2C module. After initialization, the I2C module can only operate as slave. To deinitialize I2C, call `I2C_SlaveDeinit`.

```
typedef struct i2c_config i2c_config_t
```

I2C configuration structure.

This structure includes all I2C features, user can configure user can configure these features one by one manually, or call `I2C_GetDefaultConfig` to set the structure to default value. Then, call `I2C_Init` to initialize I2C module. To deinitialize I2C, call `I2C_Deinit`.

```
typedef enum i2c_data_direction i2c_data_direction_t
```

Peripheral data direction.

```
typedef enum i2c_master_transfer_direction i2c_master_transfer_direction_t
```

Master transfer direction.

```
typedef struct i2c_master_transfer i2c_master_transfer_t
```

I2C master transfer configuration structure.

This structure definition includes all the user configurable features, that are used to control single I2C transfer of master mode, in polling way or in interrupt way.

```
typedef struct i2c_master_transfer_handle i2c_master_transfer_handle_t
```

Forward declaration of the I2C master transfer handle structure. .

```
typedef void (*i2c_master_transfer_callback_t)(i2c_master_transfer_handle_t *psHandle)
```

I2C master transfer callback function definition.

Defines the interface of user callback function used in master interrupt transfer. The callback function shall be defined and declared in application level by user. Before starting master transfer by calling `I2C_MasterTransferNonBlocking`, call `I2C_MasterTransferCreateHandle` to install the user callback. When master transfer ends successfully or failed due to any event like arbitration lost or nacked by slave, user callback will be invoked by driver. And then user can decide what to do next in the callback according to its parameter `completionStatus` which indicates how the transfer ends.

Param `psHandle`

I2C transfer handle, which contains the information of base pointer, `completionStatus` and user data.

```
typedef enum i2c_slave_transfer_event i2c_slave_transfer_event_t
```

Set of slave transfer events.

This enumeration lists all the protocol level events that may happen during slave transfer. They can be used for two related purposes:

- a. User can select certain events and combined them by OR operation to form a mask, and use the mask to configure slave transfer configuration structure `i2c_slave_transfer_t::u8EventMask`. If any of these selected events happens, driver will alert user by invoking callback.
- b. When slave callback is invoked, user has to know which specific event occurred. Callback uses slave transfer configuration structure `i2c_slave_transfer_t` as 2nd parameter, its member `i2c_slave_transfer_t::eEvent` shows which event just happened.

```
typedef struct i2c_slave_transfer i2c_slave_transfer_t
```

I2C slave transfer configuration structure.

Covers slave transfer data buffer pointer, data size and the events user want driver to alert.

Note: Unlike master who controls the transfer flow, slave has to monitor any bus event and change its configuration accordingly. So this slave transfer configuration structure is also used as second parameter of callback, for user to change the transfer configuration in the callback. The read-only member `eEvent` shows which event occurred that causes the callback being invoked.

```
typedef struct _i2c_slave_transfer_handle i2c_slave_transfer_handle_t
```

Forward declaration of the I2C slave transfer handle structure. .

```
typedef void (*i2c_slave_transfer_callback_t)(i2c_slave_transfer_handle_t *psHandle)
```

I2C slave transfer callback function definition.

Defines the interface of slave user callback function. The callback function shall be defined and declared in application level by user. Before calling I2C_SlaveTransferNonBlocking to let I2C slave ready to process bus events, call I2C_SlaveTransferCreateHandle first to install the user callback to slave handle. When I2C slave meets user selected events, callback will be invoked and user can decide the following steps in the callback. All the events that can trigger callback are listed in *i2c_slave_transfer_event_t*.

Param psHandle

I2C transfer handle, which contains the information of base pointer, completionStatus transfer data, data length and user data.

```
uint32_t I2C_GetInstance(I2C_Type *base)
```

Gets instance number for I2C module.

Parameters

- base – I2C base pointer.

Return values

The – number of the instance.

```
I2C_HAS_STOP_DETECT
```

```
I2C_RETRY_TIMES
```

Retry times when checking status flags.

```
I2C_SMBUS_ENABLE
```

Control whether to use SMBus features.

```
I2C_MASTER_FACK_CONTROL
```

Control whether to enable FACK for master read operation in non SMBus transfer. This is used to lower transfer speed by clock stretch for MCU with FSL_FEATURE_I2C_HAS_DOUBLE_BUFFERING supported and enabled.

```
I2C_GET_TRANSFER_COMPLETION_STATUS(psHandle)
```

```
I2C_GET_TRANSFER_USER_DATA(psHandle)
```

```
I2C_GET_SLAVE_TRANSFER_EVENT(psHandle)
```

```
I2C_GET_SLAVE_TRANSFER_DATA_POINTER(psHandle)
```

```
I2C_GET_SLAVE_TRANSFER_DATASIZE(psHandle)
```

```
I2C_GET_SLAVE_TRANSFERRED_COUNT(psHandle)
```

```
struct _i2c_master_config
```

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

This structure includes all the master operation needed features, user can configure these features one by one manually, or call I2C_MasterGetDefaultConfig to set the structure to default value. Then, call I2C_MasterInit to initialize I2C module. After initialization, the I2C module can only operate as master. To deinitialize I2C, call I2C_MasterDeinit.

Public Members

bool bEnableModule

Enable the I2C peripheral during initialization.

bool bEnableStopHold

Control the stop hold enable. I2C_FLT_SHEN

uint8_t u8GlitchFilterWidth

Control the width of the glitch filter. I2C_FLT_FLT

uint8_t u8Interrupts

Mask of the interrupts to be enabled in the init function.

uint32_t u32BaudRateBps

Baud rate value in bits-per-second. I2C_F_MULT, I2C_F_ICR

uint32_t u32SrcClockHz

The clock source frequency for I2C module.

struct `_i2c_slave_config`

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

This structure includes all the slave operation needed features, user can configure these features one by one manually, or call `I2C_SlaveGetDefaultConfig` to set the structure to default value. Then, call `I2C_SlaveInit` to initialize I2C module. After initialization, the I2C module can only operate as slave. To deinitialize I2C, call `I2C_SlaveDeinit`.

Public Members

bool bEnableModule

Enable the I2C peripheral during initialization.

bool bEnableStopHold

Control the stop hold enable. I2C_FLT_SHEN

uint8_t u8GlitchFilterWidth

Control the width of the glitch filter. I2C_FLT_FLT

bool bEnableWakeUp

Enable/disable waking up MCU from low-power mode. I2C_C1_WUEN

bool bEnableGeneralCall

Enable the general call addressing mode, not affected by address length. General call address is 0x00. I2C_C2_GCAEN

i2c_slave_address_mode_t eAddressingMode

Addressing mode chosen from *i2c_slave_address_mode_t*. I2C_C2_ADEXT
I2C_C2_RMEN

uint16_t bitsPrimaryAddress

Primary Slave address. I2C_A1_AD[7] I2C_C2_AD[3]

uint16_t bitsMaxAddress

The maximum boundary of slave address used in a range address match addressing. In range address match, address greater than primary address and less than or equal to upper address is marked as matched. I2C_RA_RAD[7]

uint8_t u8Interrupts

Mask of the interrupts to be enabled in the init function.

uint32_t u32SclStopHoldTime_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.

uint32_t u32SrcClockHz

The clock source frequency for I2C module.

struct _i2c_config

#include <fsl_i2c.h> I2C configuration structure.

This structure includes all I2C features, user can configure user can configure these features one by one manually, or call I2C_GetDefaultConfig to set the structure to default value. Then, call I2C_Init to initialize I2C module. To deinitialize I2C, call I2C_Deinit.

Public Members

bool bEnableModule

Enable the I2C peripheral during initialization.

bool bEnableStopHold

Control the stop hold enable. I2C_FLT_SHEN

uint8_t u8GlitchFilterWidth

Control the width of the glitch filter. I2C_FLT_FLT

bool bEnableWakeUp

Enable/disable waking up MCU from low-power mode. I2C_C1_WUEN

bool bEnableGeneralCall

Enable the general call addressing mode, not affected by address length. General call address is 0x00. I2C_C2_GCAEN

i2c_slave_address_mode_t eAddressingMode

Addressing mode chosen from *i2c_slave_address_mode_t*. I2C_C2_ADEXT
I2C_C2_RMEN

uint16_t bitsPrimaryAddress

Primary Slave address. I2C_A1_AD[7] I2C_C2_AD[3]

uint16_t bitsMaxAddress

The maximum boundary of slave address used in a range address match addressing. In range address match, address greater than primary address and less than or equal to upper address is marked as matched. I2C_RA_RAD[7]

uint8_t u8Interrupts

Mask of the interrupts to be enabled in the init function.

uint32_t u32BaudRateBps

Baud rate value in bits-per-second. I2C_F_MULT, I2C_F_ICR

uint32_t u32SrcClockHz

The clock source frequency for I2C module.

struct _i2c_master_transfer

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

This structure definition includes all the user configurable features, that are used to control single I2C transfer of master mode, in polling way or in interrupt way.

Public Members

uint8_t u8ControlFlagMask

The transfer flag which controls the transfer start/stop signal, refer `_i2c_master_transfer_flags`.

uint8_t u8SlaveAddress

7-bit slave address.

i2c_master_transfer_direction_t eDirection

Transfer direction, `kI2C_MasterTransmit` or `kI2C_MasterReceive`.

uint8_t *pu8Command

Pointer to command code.

uint8_t u8CommandSize

Size of the command code, max value 4.

uint8_t *volatile pu8Data

Pointer to the send/receive data buffer.

volatile uint16_t u16DataSize

Transfer size.

struct `_i2c_master_transfer_handle`

`#include <fsl_i2c.h>` I2C master transfer handle.

Note: If user wants to use the transactional API to transfer data in interrupt way in master mode, one I2C instance should and can only be allocated one master handle.

Note: The handle is maintained by I2C driver internally, which means the transfer state is retained and user shall not modify its state `u8State` in application level. If user only wish to use transactional APIs without understanding its mechanism, it is not necessary to understand these members.

Public Members

I2C_Type *base

I2C base pointer to the I2C instance that assigned to this handle.

i2c_master_transfer_t sTransfer

I2C master transfer structure.

uint16_t u16TransferSize

Total bytes to be transferred.

uint8_t u8State

A transfer state maintained during transfer.

i2c_master_transfer_callback_t pfCompletionCallback

Callback function invoked when the transfer is finished.

status_t completionStatus

I2C master transfer complete status, indicating how the transfer ends.

void *pUserData

User configurable pointer to any data, function, structure etc that user wish to use in the callback

struct `_i2c_slave_transfer`

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

Covers slave transfer data buffer pointer, data size and the events user want driver to alert.

Note: Unlike master who controls the transfer flow, slave has to monitor any bus event and change its configuration accordingly. So this slave transfer configuration structure is also used as second parameter of callback, for user to change the transfer configuration in the callback. The read-only member `eEvent` shows which event occurred that causes the callback being invoked.

Public Members

`uint8_t` `u8EventMask`

Mask of the events. When these events occur during transfer driver will alert user using callback.

`uint8_t *volatile` `pu8Data`

Pointer to the buffer of data to send, or to store received data.

`volatile uint16_t` `u16DataSize`

Transfer size.

`i2c_slave_transfer_event_t` `eEvent`

The event that caused the callback being invoked. Read-only.

struct `_i2c_slave_transfer_handle`

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

Note: If user wants to use the transactional API to transfer data in slave mode, one I2C instance should and can only be allocated one handle.

Note: The handle is maintained by I2C driver internally, which means the transfer state is retained and user shall not modify its state `u8State` in application level. If user only wish to use transactional APIs without understanding its mechanism, it is not necessary to understand these members.

Public Members

`I2C_Type *``base`

I2C base pointer to the I2C instance that assigned to this handle.

`i2c_slave_transfer_t` `sTransfer`

I2C slave transfer structure.

`uint16_t` `u16TransferredCount`

The number of bytes actually transferred for current data buffer.

`uint8_t` `u8State`

A transfer state maintained during transfer.

`i2c_slave_transfer_callback_t` `pfCallback`

Callback function invoked at the transfer event.

status_t completionStatus

I2C slave transfer complete status, indicating how the transfer ends, such as `kStatus_I2C_Nak` indicates the slave was nacked by master before all the data was sent. This parameter is only useful when `eEvent` is `kI2C_SlaveCompletionEvent`.

`void *pUserData`

User configurable pointer to any data, function, structure etc that user wish to use in the callback.

2.39 The Driver Change Log

2.40 I2C_EDMA: EDMA based I2C Driver

```
void I2C_MasterCreateEDMAHandle(I2C_Type *base, i2c_master_edma_transfer_handle_t
                                *psHandle, i2c_master_edma_transfer_callback_t
                                pfCallback, void *pUserData, DMA_Type *edmaBase,
                                edma_channel_t eEdmaChannel)
```

Initializes the I2C handle which is used in transactional functions.

Parameters

- `base` – I2C peripheral base address.
- `psHandle` – A pointer to the `i2c_master_edma_transfer_handle_t` structure.
- `pfCallback` – A pointer to the user callback function.
- `pUserData` – A user parameter passed to the callback function.
- `edmaBase` – Edma base address.
- `eEdmaChannel` – eDMA channel for master transfer.

```
status_t I2C_MasterTransferEDMA(i2c_master_edma_transfer_handle_t *psHandle,
                                i2c_master_transfer_t *psTransfer)
```

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

Parameters

- `psHandle` – A pointer to the `i2c_master_edma_transfer_handle_t` structure.
- `psTransfer` – A pointer to the transfer structure of `i2c_master_transfer_t`.

Return values

- `kStatus_Success` – Successfully completed the data transmission.
- `kStatus_I2C_Busy` – A previous transmission is still not finished.
- `kStatus_I2C_Timeout` – Transfer error, waits for a signal timeout.
- `kStatus_I2C_ArbitrationLost` – Transfer error, arbitration lost.
- `kStatus_I2C_Nak` – Transfer error, receive NAK during transfer.

```
status_t I2C_MasterTransferGetCountEDMA(i2c_master_edma_transfer_handle_t *psHandle,
                                         uint16_t *pu16Count)
```

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

Parameters

- `psHandle` – A pointer to the `i2c_master_edma_transfer_handle_t` structure.
- `pu16Count` – A number of bytes transferred by the non-blocking transaction.

void I2C_MasterTransferAbortEDMA(*i2c_master_edma_transfer_handle_t* *psHandle)

Aborts a master eDMA non-blocking transfer early.

Parameters

- psHandle – A pointer to the *i2c_master_edma_transfer_handle_t* structure.

FSL_I2C_EDMA_DRIVER_VERSION

I2C EDMA driver version.

typedef struct *i2c_master_edma_transfer_handle* *i2c_master_edma_transfer_handle_t*

Retry times for waiting flag.

I2C master eDMA handle typedef.

typedef void (**i2c_master_edma_transfer_callback_t*)(*i2c_master_edma_transfer_handle_t* *handle)

I2C master eDMA transfer callback typedef.

struct *i2c_master_edma_transfer_handle*

#include <fsl_i2c_edma.h> I2C master eDMA transfer structure. This struct address should be sizeof(*edma_channel_tcd_t*) aligned.

Public Members

edma_channel_tcd_t sTcd

TCD for EDMA transfer.

I2C_Type *base

I2C base pointer to the I2C instance that assigned to this handle.

i2c_master_transfer_t sTransfer

I2C master transfer structure.

uint16_t u16TransferSize

Total bytes to be transferred.

uint8_t u8State

I2C master transfer status.

edma_handle_t sEdmaHandle

The eDMA handler used.

status_t completionStatus

I2C master transfer complete status, indicating how the transfer ends.

i2c_master_edma_transfer_callback_t pfCallback

A callback function called after the eDMA transfer is finished.

void *pUserData

User configurable pointer to any data, function, structure etc that user wish to use in the callback

2.41 I2C Peripheral and Driver Overview

2.42 INTC: Interrupt Controller Driver

```
static inline void INTC_SetIRQPriorityNum(IRQn_Type eIrq, uint8_t u8PriorityNum)
```

Disable IRQ or Enable IRQ with priority.

There are similar function in `fsl_common`:

- `EnableIRQWithPriority`,
- `DisableIRQ`,
- `EnableIRQ`,
- `IRQ_SetPriority`.

This function is faster and simpler than those in `fsl_common`. Generally, this function and IRQ functions in `fsl_common` are either-or, don't use them together for same IRQn type, but feasible that different IRQn type use them simultaneously, for example: It is OK OCCS_IRQn use `INTC_SetIRQPriorityLevel`, and ADC12_CC1_IRQn use `EnableIRQWithPriority`. It is NOT OK that OCCS_IRQn use `INTC_SetIRQPriorityLevel` and `EnableIRQWithPriority` simultaneously.

Note: Please note a none-zero priority number does directly map to priority level, simple summary is as below, you could check RM INTC chapter for more details.

- Some IPs have priority level 1~3, maps priority number 1 to priority 1, 2 to priority 2, 3 to priority 3.
 - Some IPs have priority level 0~2, maps priority number 1 to priority 0, 2 to priority 1, 3 to priority 2.
-

Parameters

- `eIrq` – The IRQ number.
- `u8PriorityNum` – IRQ interrupt priority number.
 - 0: disable IRQ.
 - 1-3: enable IRQ and set its priority, 3 is the highest priority for this IRQ and 1 is the lowest priority.

```
static inline void INTC_SetVectorBaseAddress(uint32_t u32VectorBaseAddr)
```

Set the base address vector table. The value in `INTC_VBA` is used as the upper 13 bits of the interrupt vector `VAB[20:0]`.

Parameters

- `u32VectorBaseAddr` – Vector table base address. The address requires 256 words (512 bytes) aligned. Take the vector table in `MC56F83xxx_Vectors.c` as example for how to implement this table.

```
static inline void INTC_SetFastIRQVectorHandler0(vector_type_t eVector, fast_irq_handler
                                                pfHandler)
```

Set the IRQ handler for fast IRQ0. The INTC takes the vector address from the appropriate `FIVAL0` and `FIVAH0` registers, instead of generating an address that is an offset from the vector base address (VBA).

Parameters

- `eVector` – The vector number.
- `pfHandler` – Pointer to the fast IRQ handler function, see `fast_irq_handler` definition for more info.

```
static inline void INTC_SetFastIRQVectorHandler1(vector_type_t eVector, fast_irq_handler  
                                             pfHandler)
```

Set the IRQ handler for fast IRQ1. The INTC takes the vector address from the appropriate FIVAL1 and FIVAH1 registers, instead of generating an address that is an offset from the vector base address (VBA).

Parameters

- eVector – The eVector number.
- pfHandler – Pointer to the fast IRQ handler function, see @ ref fast_irq_handler definition for more info.

```
static inline uint8_t INTC_GetIRQPermittedPriorityLevel(void)
```

Get IRQ permitted priority levels. Interrupt exceptions may be nested to allow the servicing of an IRQ with higher priority than the current exception.

The return value indicate the priority level needed for a new IRQ to interrupt the current interrupt being sent to the Core.

Return values

- 0 – Required nested exception priority levels are 0, 1, 2, or 3.
- 1 – Required nested exception priority levels are 1, 2, or 3.
- 2 – Required nested exception priority levels are 2 or 3.
- 3 – Required nested exception priority level is 3.

```
static inline bool INTC_GetPendingIRQ(vector_type_t eVector)
```

Check if IRQ is pending for execution. Before the ISR is entered, IRQ is pending. After the ISR is entered, the IRQ is not pending.

Parameters

- eVector – The IRQ vector number.

Return values

True – if interrupt is pending, otherwise return false.

```
static inline uint16_t INTC_GetLatestRespondedVectorNumber(void)
```

Get the latest responded IRQ's vector number. It shows the Vector Address Bus used at the time the last IRQ was taken.

Note: Return value of the function call could be different according to where the function call is invoked.

- when called in normal ISR handler, it returns current ISR's vector number defined in vector_type_t.
 - when called in fast IRQ handler, it returns the lower address bits of the jump address.
 - when called in none ISR handler code, it returns previous responded IRQ vector number defined in vector_type_t or fast IRQ low address bits.
-

Returns

The latest vector number.

```
FSL_INTC_DRIVER_VERSION
```

INTC driver version.

```
typedef void (*fast_irq_handler)(void)
```

The handle of the fast irq handler function.

Normally this function should be guarded by: `#pragma interrupt fast` and `#pragma interrupt off`.

```
INTC_DisableIRQ(x)
```

Macro to disable the IRQ.

```
INTC_PEND_REG_INDEX(x)
```

Helper Macro function to extract IRQ pending register index comparing to `INTC_IRQP0`.

```
INTC_PEND_BIT_INDEX(x)
```

Helper Macro function to extract pending IRQs bit index.

```
INTC_TYPE_REG_INDEX(x)
```

Helper Macro function to extract IRQ priority register index comparing to `INTC_IRP0`.

```
INTC_TYPE_BIT_INDEX(x)
```

Helper Macro function to extract IRQs priority bit index.

2.43 The Driver Change Log

2.44 INTC Peripheral and Driver Overview

2.45 Common Driver

```
status_t EnableIRQWithPriority(IRQn_Type irq, uint8_t priNum)
```

Enable the IRQ, and also set the interrupt priority.

- Some IPs maps 1 to priority 1, 2 to priority 2, 3 to priority 3
- Some IPs maps 1 to priority 0, 2 to priority 1, 3 to priority 2

User should check chip's RM to get its corresponding interrupt priority.

When `priNum` set as 0, then `SDK_DSC_DEFAULT_INT_PRIO` is set instead. When `priNum` set as number larger than 3, then only the 2 LSB take effect, for example, setting `priNum` to 5 is the same with setting it to 1.

This function configures INTC module, application could call the INTC driver directly for the same purpose.

Note: The parameter `priNum` is range in 1~3, and its value is **NOT** directly map to interrupt priority.

Parameters

- `irq` – The IRQ to enable.
- `priNum` – Priority number set to interrupt controller register. Larger number means higher priority. The allowed range is 1~3, and its value is **NOT** directly map to interrupt priority. In other words, the same priority number means different interrupt priority levels for different IRQ, please check reference manual for the relationship. When pass in 0, then `SDK_DSC_DEFAULT_INT_PRIO` is set to priority register.

Returns

Currently only returns `kStatus_Success`, will enhance in the future.

status_t DisableIRQ(IRQn_Type irq)

Disable specific interrupt.

This function configures INTC module, application could call the INTC driver directly for the same purpose.

Parameters

- `irq` – The IRQ to disable.

Returns

Currently only returns `kStatus_Success`, will enhance in the future.

status_t EnableIRQ(IRQn_Type irq)

Enable specific interrupt.

The recommended workflow is calling `IRQ_SetPriority` first, then call `EnableIRQ`. If `IRQ_SetPriority` is not called first, then the interrupt is enabled with default priority value `SDK_DSC_DEFAULT_INT_PRIO`.

Another recommended workflow is calling `EnableIRQWithPriority` directly, it is the same with calling `IRQ_SetPriority + EnableIRQ`.

This function configures INTC module, application could call the INTC driver directly for the same purpose.

Parameters

- `irq` – The IRQ to enable.

Returns

Currently only returns `kStatus_Success`, will enhance in the future.

status_t IRQ_SetPriority(IRQn_Type irq, uint8_t priNum)

Set the IRQ priority.

- Some IPs maps 1 to priority 1, 2 to priority 2, 3 to priority 3
- Some IPs maps 1 to priority 0, 2 to priority 1, 3 to priority 2

User should check chip's RM to get its corresponding interrupt priority

When `priNum` set as 0, then `SDK_DSC_DEFAULT_INT_PRIO` is set instead. When `priNum` set as number larger than 3, then only the 2 LSB take effect, for example, setting `priNum` to 5 is the same with setting it to 1.

This function configures INTC module, application could call the INTC driver directly for the same purpose.

Note: The parameter `priNum` is range in 1~3, and its value is **NOT** directly map to interrupt priority.

Parameters

- `irq` – The IRQ to set.
- `priNum` – Priority number set to interrupt controller register. Larger number means higher priority, 0 means disable the interrupt. The allowed range is 0~3, and its value is **NOT** directly map to interrupt priority. In other words, the same priority number means different interrupt priority levels for different IRQ, please check reference manual for the relationship.

Returns

Currently only returns `kStatus_Success`, will enhance in the future.

`FSL_COMMON_DRIVER_VERSION`

common driver version.

`DEBUG_CONSOLE_DEVICE_TYPE_NONE`

No debug console.

`DEBUG_CONSOLE_DEVICE_TYPE_UART`

Debug console based on UART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPUART`

Debug console based on LPUART.

`DEBUG_CONSOLE_DEVICE_TYPE_LPSCI`

Debug console based on LPSCI.

`DEBUG_CONSOLE_DEVICE_TYPE_USBCDC`

Debug console based on USBCDC.

`DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM`

Debug console based on FLEXCOMM.

`DEBUG_CONSOLE_DEVICE_TYPE_IUART`

Debug console based on i.MX UART.

`DEBUG_CONSOLE_DEVICE_TYPE_VUSART`

Debug console based on LPC_VUSART.

`DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART`

Debug console based on LPC_USART.

`DEBUG_CONSOLE_DEVICE_TYPE_SWO`

Debug console based on SWO.

`DEBUG_CONSOLE_DEVICE_TYPE_QSCI`

Debug console based on QSCI.

`MIN(a, b)`

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

`MAX(a, b)`

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

`UINT16_MAX`

Max value of `uint16_t` type.

`UINT32_MAX`

Max value of `uint32_t` type.

`MCUX_MASK_INVERT_8(mask)`

8-bit mask inversion.

`MCUX_MASK_INVERT_16(mask)`

16-bit mask inversion.

`MCUX_MASK_INVERT_32(mask)`

32-bit mask inversion for completeness.

`MCUX_REG_WRITE8(reg, value)`

8-bit register write macro

MCUX_REG_WRITE16(reg, value)
16-bit register write macro

MCUX_REG_WRITE32(reg, value)
32-bit register write macro

MCUX_REG_READ8(reg)
8-bit register read macro

MCUX_REG_READ16(reg)
16-bit register read macro

MCUX_REG_READ32(reg)
32-bit register read macro

MCUX_REG_BIT_SET8(reg, mask)
8-bit register bit set macro

MCUX_REG_BIT_SET16(reg, mask)
16-bit register bit set macro

MCUX_REG_BIT_SET32(reg, mask)
32-bit register bit set macro

MCUX_REG_BIT_CLEAR8(reg, mask)
8-bit register bit clear macro

MCUX_REG_BIT_CLEAR16(reg, mask)
16-bit register bit clear macro

MCUX_REG_BIT_CLEAR32(reg, mask)
32-bit register bit clear macro

MCUX_REG_BIT_GET8(reg, mask)
8-bit register bit get macro

MCUX_REG_BIT_GET16(reg, mask)
16-bit register bit get macro

MCUX_REG_BIT_GET32(reg, mask)
32-bit register bit get macro

MCUX_REG_MODIFY8(reg, mask, value)
32-bit register read-modify-write macro

MCUX_REG_MODIFY16(reg, mask, value)
16-bit register read-modify-write macro

MCUX_REG_MODIFY32(reg, mask, value)
32-bit register read-modify-write macro

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_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

AT_NONCACHEABLE_SECTION(var)

AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)

AT_NONCACHEABLE_SECTION_INIT(var)

AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, 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 I2C status codes.

enumerator kStatusGroup_LPSCI

Group number for LPSCI status codes.

enumerator kStatusGroup_LPUART

Group number for LPUART status codes.

enumerator kStatusGroup_SPI

Group number for SPI status code.

enumerator kStatusGroup_XRDC

Group number for XRDC status code.

enumerator kStatusGroup_SEMA42

Group number for SEMA42 status code.

enumerator kStatusGroup_SDHC

Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

- enumerator kStatusGroup_DMA
Group number for DMA status codes.
- enumerator kStatusGroup_EDMA
Group number for EDMA status codes.
- enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.
- enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.
- enumerator kStatusGroup_LTC
Group number for LTC status codes.
- enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.
- enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.
- enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.
- enumerator kStatusGroup_DMIC
Group number for DMIC status codes.
- enumerator kStatusGroup_SDIF
Group number for SDIF status codes.
- enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.
- enumerator kStatusGroup_OTP
Group number for OTP status codes.
- enumerator kStatusGroup_MCAN
Group number for MCAN status codes.
- enumerator kStatusGroup_CAAM
Group number for CAAM status codes.
- enumerator kStatusGroup_ECSPi
Group number for ECSPi status codes.
- enumerator kStatusGroup_USDHC
Group number for USDHC status codes.
- enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.
- enumerator kStatusGroup_DCP
Group number for DCP status codes.
- enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.
- enumerator kStatusGroup_ESAI
Group number for ESAI status codes.
- enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

- enumerator `kStatusGroup_MMDC`
Group number for MMDC status codes.
- enumerator `kStatusGroup_PDM`
Group number for MIC status codes.
- enumerator `kStatusGroup_SDMA`
Group number for SDMA status codes.
- enumerator `kStatusGroup_ICS`
Group number for ICS status codes.
- enumerator `kStatusGroup_SPDIF`
Group number for SPDIF status codes.
- enumerator `kStatusGroup_LPC_MINISPI`
Group number for LPC_MINISPI status codes.
- enumerator `kStatusGroup_HASHCRYPT`
Group number for Hashcrypt status codes
- enumerator `kStatusGroup_LPC_SPI_SSP`
Group number for LPC_SPI_SSP status codes.
- enumerator `kStatusGroup_I3C`
Group number for I3C status codes
- enumerator `kStatusGroup_LPC_I2C_1`
Group number for LPC_I2C_1 status codes.
- enumerator `kStatusGroup_NOTIFIER`
Group number for NOTIFIER status codes.
- enumerator `kStatusGroup_DebugConsole`
Group number for debug console status codes.
- enumerator `kStatusGroup_SEMC`
Group number for SEMC status codes.
- enumerator `kStatusGroup_ApplicationRangeStart`
Starting number for application groups.
- enumerator `kStatusGroup_IAP`
Group number for IAP status codes
- enumerator `kStatusGroup_SFA`
Group number for SFA status codes
- enumerator `kStatusGroup_SPC`
Group number for SPC status codes.
- enumerator `kStatusGroup_PUF`
Group number for PUF status codes.
- enumerator `kStatusGroup_TOUCH_PANEL`
Group number for touch panel status codes
- enumerator `kStatusGroup_VBAT`
Group number for VBAT status codes
- enumerator `kStatusGroup_XSPI`
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_AUDMIX
Group number for AUDMIX status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

- enumerator kStatusGroup_OSA
Group number for OSA status codes.
- enumerator kStatusGroup_COMMON_TASK
Group number for Common task status codes.
- enumerator kStatusGroup_MSG
Group number for messaging status codes.
- enumerator kStatusGroup_SDK_OCOTP
Group number for OCOTP status codes.
- enumerator kStatusGroup_SDK_FLEXSPINOR
Group number for FLEXSPINOR status codes.
- enumerator kStatusGroup_CODEC
Group number for codec status codes.
- enumerator kStatusGroup_ASRC
Group number for codec status ASRC.
- enumerator kStatusGroup_OTFAD
Group number for codec status codes.
- enumerator kStatusGroup_SDIOSLV
Group number for SDIOSLV status codes.
- enumerator kStatusGroup_MECC
Group number for MECC status codes.
- enumerator kStatusGroup_ENET_QOS
Group number for ENET_QOS status codes.
- enumerator kStatusGroup_LOG
Group number for LOG status codes.
- enumerator kStatusGroup_I3CBUS
Group number for I3CBUS status codes.
- enumerator kStatusGroup_QSCI
Group number for QSCI status codes.
- enumerator kStatusGroup_ELEMU
Group number for ELEMU status codes.
- enumerator kStatusGroup_QUEUEDSPI
Group number for QSPI status codes.
- enumerator kStatusGroup_POWER_MANAGER
Group number for POWER_MANAGER status codes.
- enumerator kStatusGroup_IPED
Group number for IPED status codes.
- enumerator kStatusGroup_ELS_PKC
Group number for ELS PKC status codes.
- enumerator kStatusGroup_CSS_PKC
Group number for CSS PKC status codes.
- enumerator kStatusGroup_HOSTIF
Group number for HOSTIF status codes.

enumerator `kStatusGroup_CLIF`
Group number for CLIF status codes.

enumerator `kStatusGroup_BMA`
Group number for BMA status codes.

enumerator `kStatusGroup_NETC`
Group number for NETC status codes.

enumerator `kStatusGroup_ELE`
Group number for ELE status codes.

enumerator `kStatusGroup_GLIKEY`
Group number for GLIKEY status codes.

enumerator `kStatusGroup_AON_POWER`
Group number for AON_POWER status codes.

enumerator `kStatusGroup_AON_COMMON`
Group number for AON_COMMON status codes.

enumerator `kStatusGroup_ENDAT3`
Group number for ENDAT3 status codes.

enumerator `kStatusGroup_HIPERFACE`
Group number for HIPERFACE status codes.

enumerator `kStatusGroup_NPX`
Group number for NPX status codes.

enumerator `kStatusGroup_ELA_CSEC`
Group number for ELA_CSEC status codes.

enumerator `kStatusGroup_FLEXIO_T_FORMAT`
Group number for T-format status codes.

enumerator `kStatusGroup_FLEXIO_A_FORMAT`
Group number for A-format status codes.

enumerator `kStatusGroup_LPC_QSPI`
Group number for LPC QSPI status codes.

Generic status return codes.

Values:

enumerator `kStatus_Success`
Generic status for Success.

enumerator `kStatus_Fail`
Generic status for Fail.

enumerator `kStatus_ReadOnly`
Generic status for read only failure.

enumerator `kStatus_OutOfRange`
Generic status for out of range access.

enumerator `kStatus_InvalidArgument`
Generic status for invalid argument check.

enumerator kStatus_Timeout

Generic status for timeout.

enumerator kStatus_NoTransferInProgress

Generic status for no transfer in progress.

enumerator kStatus_Busy

Generic status for module is busy.

enumerator kStatus_NoData

Generic status for no data is found for the operation.

typedef int32_t status_t

Type used for all status and error return values.

uint8_t bool

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 uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

static inline void EnableGlobalIRQ(uint32_t irqSts)

Enable the global IRQ.

static inline bool isIRQAllowed(void)

Check if currently core is able to response IRQ.

void SDK_DelayCoreCycles(uint32_t u32Num)

Delay core cycles. Please note that, this API uses software loop for delay, the actual delayed time depends on core clock frequency, where the function is located (ram or flash), flash clock, possible interrupt.

Parameters

- u32Num – Number of core clock cycle which needs to be delayed.

uint32_t SDK_CovertUsToCount(uint32_t u32Us, uint32_t u32Hz)

Covert us to count with fixed-point calculation.

Note: u32Us must not be greater than 4294

Parameters

- u32Us – Time in us
- u32Hz – Clock frequency in Hz

Returns

The count value

uint32_t SDK_CovertCountToUs(uint32_t u32Count, uint32_t u32Hz)

Covert count to us with fixed-point calculation.

Note: u32Hz must not be greater than 429496729UL(0xFFFFFFFFUL/10UL)

Parameters

- u32Count – Count value
- u32Hz – Clock frequency in Hz

Returns

The us value

uint32_t SDK_CovertMsToCount(uint32_t u32Ms, uint32_t u32Hz)

Covert ms to count with fixed-point calculation.

Note: u32Ms must not be greater than 42949UL @ u32Hz = 100M

Parameters

- u32Ms – Time in us
- u32Hz – Clock frequency in Hz

Returns

The count value

uint32_t SDK_CovertCountToMs(uint32_t u32Count, uint32_t u32Hz)

Covert count to ms with fixed-point calculation.

Note: u32Hz must not be greater than 429496729UL(0xFFFFFFFFUL/10UL)

Parameters

- u32Count – Count value
- u32Hz – Clock frequency in Hz

Returns

The us value

void SDK_DelayAtLeastMs(uint32_t delayTime_ms, uint32_t coreClock_Hz)

Delay at least for some time in millisecond unit. 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_ms – Delay time in unit of millisecond.
- coreClock_Hz – Core clock frequency with Hz.

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

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

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version	Minor Version	Bug Fix
31 25 24	17 16	9 8	0

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING(;)” need to be added at the end of each case section which misses “break;”statement.

true

false

SDK_ISR_EXIT_BARRIER

SDK_DSC_DEFAULT_INT_PRIO

Default DSC interrupt priority number.

SetIRQBasePriority(x)

Set base core IRQ priority, that core will response the interrupt request with priority >= base IRQ priority.

PeriphReadReg(reg)

Read register value.

Example: val = PeriphReadReg(OCCS->OSCTL2);

Parameters

- reg – Register name.

Returns

The value of register.

PeriphWriteReg(reg, data)

Write data to register.

Example: PeriphWriteReg(OCCS->OSCTL2, 0x278U);

Parameters

- reg – Register name.
- data – Data wrote to register.

PeriphSetBits(reg, bitMask)

Set specified bits in register.

Example: PeriphSetBits(OCCS->OSCTL2, 0x12U);

Parameters

- reg – Register name.
- bitMask – Bits mask, set bits will be set in the register.

PeriphClearBits(reg, bitMask)

Clear specified bits in register.

Example: PeriphClearBits(OCCS->OSCTL2, 0x12U);

Parameters

- reg – Register name.
- bitMask – Bits mask, set bits will be cleared in the register.

PeriphInvertBits(reg, bitMask)

Invert specified bits in register.

Example: PeriphInvertBits(OCCS->OSCTL2, 0x12U);

Parameters

- reg – Register name.
- bitMask – Bits mask, set bits will be inverted in the register.

PeriphGetBits(reg, bitMask)

Get specified bits in register.

Example: val = PeriphGetBits(OCCS->OSCTL2, 0x23U);

Parameters

- reg – Register name.
- bitMask – Bits mask, specify the getting bits.

Returns

The value of specified bits.

PeriphWriteBitGroup(reg, bitMask, bitValue)

Write group of bits to register.

Example: PeriphWriteBitGroup(OCCS->DIVBY, OCCS_DIVBY_COD_MASK, OCCS_DIVBY_COD(23U)); PeriphWriteBitGroup(OCCS->DIVBY, OCCS_DIVBY_COD_MASK | OCCS_DIVBY_PLLDB_MASK, \ OCCS_DIVBY_COD(23U) | OCCS_DIVBY_PLLDB(49U));

Parameters

- reg – Register name.
- bitMask – Bits mask, mask of the group of bits.

- bitValue – This value will be written into the bit group specified by parameter bitMask.

PeriphSafeClearFlags(reg, allFlagsMask, flagMask)

Clear (acknowledge) flags which are active-high and are cleared-by-write-one.

This macro is useful when a register is comprised by normal read-write bits and cleared-by-write-one bits. Example: `PeriphSafeClearFlags(PWMA->FAULT[0].FSTS, PWM_FSTS_FFLAG_MASK, PWM_FSTS_FFLAG(2));`

Parameters

- reg – Register name.
- allFlagsMask – Mask for all flags which are active-high and are cleared-by-write-one.
- flagMask – The selected flags(cleared-by-write-one) which are supposed to be cleared.

PeriphSafeClearBits(reg, allFlagsMask, bitMask)

Clear selected bits without modifying (acknowledge) bit flags which are active-high and are cleared-by-write-one.

This macro is useful when a register is comprised by normal read-write bits and cleared-by-write-one bits. Example: `PeriphSafeClearBits(PWMA->FAULT[0].FSTS, PWM_FSTS_FFLAG_MASK, PWM_FSTS_FHALF(2));`

Parameters

- reg – Register name.
- allFlagsMask – Mask for all flags which are active-high and are cleared-by-write-one.
- bitMask – The selected bits which are supposed to be cleared.

PeriphSafeSetBits(reg, allFlagsMask, bitMask)

Set selected bits without modifying (acknowledge) bit flags which are active-high and are cleared-by-write-one.

This macro is useful when a register is comprised by normal read-write bits and cleared-by-write-one bits. Example: `PeriphSafeSetBits(PWMA->FAULT[0].FSTS, PWM_FSTS_FFLAG_MASK, PWM_FSTS_FHALF(2));`

Parameters

- reg – Register name.
- allFlagsMask – Mask for all flags which are active-high and are cleared-by-write-one.
- bitMask – The selected bits which are supposed to be set.

PeriphSafeWriteBitGroup(reg, allFlagsMask, bitMask, bitValue)

Write group of bits without modifying (acknowledge) bit flags which are active-high and are cleared-by-write-one.

This macro is useful when a register is comprised by normal read-write bits and cleared-by-write-one bits. Example: `PeriphSafeWriteBitGroup(PWMA->FAULT[0].FSTS, PWM_FSTS_FFLAG_MASK, PWM_FSTS_FHALF_MASK, PWM_FSTS_FHALF(3U));`
`PeriphSafeWriteBitGroup(PWMA->FAULT[0].FSTS, PWM_FSTS_FFLAG_MASK, PWM_FSTS_FHALF_MASK | PWM_FSTS_FFULL_MASK, \ PWM_FSTS_FHALF(3U) | PWM_FSTS_FFULL(2U));`

Parameters

- reg – Register name.

- `allFlagsMask` – Mask for all flags which are active-high and are cleared-by-write-one.
- `bitMask` – Bits mask, mask of the group of bits.
- `bitValue` – This value will be written into the bit group specified by parameter `bitMask`.

`SDK_GET_REGISTER_BYTE_ADDR(ipType, ipBase, regName)`

Get IP register byte address with `uint32_t` type.

This macro is useful when a register byte address is required, especially in SDM mode.
Example: `SDK_GET_REGISTER_BYTE_ADDR(ADC_Type, ADC, RSLT[0]);`

Parameters

- `ipType` – IP register mapping struct type.
- `ipBase` – IP instance base pointer, WORD address.
- `regName` – Member register name of IP register mapping struct.

`MSDK_REG_SECURE_ADDR(x)`

`MSDK_REG_NONSECURE_ADDR(x)`

2.46 MCM: Miscellaneous Control Module Driver

static inline `mcm_datapath_width_t` `MCM_GetDataPathWidth(MCM_Type *base)`

Indicates if the datapath is 32 or 64 bits wide.

Parameters

- `base` – MCM base address.

Returns

The device's datapath width, please refer to `mcm_datapath_width_t`.

static inline `uint16_t` `MCM_GetCrossbarSwitchSlaveConfig(MCM_Type *base)`

Gets crossbar switch (AXBS) slave configuration that indicates the presence/absence of bus slave connections to the device's crossbar switch.

Parameters

- `base` – MCM base address.

Returns

Crossbar switch (AXBS) slave configuration, each bit in the return value indicates if there is a corresponding connection to the AXBS slave input port. For example if the result is `0x1`, it means a bus slave connection to AXBS input port 0 is present.

static inline `uint16_t` `MCM_GetCrossbarSwitchMasterConfig(MCM_Type *base)`

Gets crossbar switch (AXBS) master configuration that indicates the presence/absence of bus master connections to the device's crossbar switch.

Parameters

- `base` – MCM base address.

Returns

Crossbar switch (AXBS) master configuration, each bit in the return value indicates if there is a corresponding connection to the AXBS master input port. For example if the result is `0x1`, it means a bus master connection to AXBS input port 0 is present.

static inline void MCM_ClearFlashControllerCache(MCM_Type *base)

Clears Flash Controller Cache, 1 cycle active.

Parameters

- base – MCM base address.

static inline void MCM_DisableFlashControllerDataCaching(MCM_Type *base, bool bDisable)

Disables/Enables flash controller data caching.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable flash controller data caching.
 - **true** Disable flash controller data caching.
 - **false** Enable flash controller data caching.

static inline void MCM_DisableFlashControllerInstructionCaching(MCM_Type *base, bool bDisable)

Disables/Enables flash controller instruction caching.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable flash controller instruction caching.
 - **true** Disable flash controller instruction caching.
 - **false** Enable flash controller instruction caching.

static inline void MCM_DisableFlashControllerCache(MCM_Type *base, bool bDisable)

Disables/Enables flash controller cache.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable flash controller cache.
 - **true** Disable flash controller cache.
 - **false** Enable flash controller cache.

static inline void MCM_DisableFlashControllerDataSpeculation(MCM_Type *base, bool bDisable)

Disables/Enables flash controller data speculation.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable flash controller data speculation.
 - **true** Disable flash controller data speculation.
 - **false** Enable flash controller data speculation.

static inline void MCM_DisableFlashControllerSpeculation(MCM_Type *base, bool bDisable)

Disables/Enables flash controller speculation.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable flash controller speculation.
 - **true** Disable flash controller speculation.
 - **false** Enable flash controller speculation.

```
static inline void MCM_DisableDSP56800EXCoreInstructions(MCM_Type *base, bool bDisable)
```

Disables/Enables the instruction support only by DSP56800EX core, the instructions supported only by the DSP56800EX core are the BPSC and 32-bit multiply and MAC instructions.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable 32-bit multiply and MAC instructions.
 - **true** BFSC and 32-bit multiply and MAC instructions disabled.
 - **false** BFSC and 32-bit multiply and MAC instructions enabled.

```
static inline void MCM_DisableCoreReverseCarry(MCM_Type *base, bool bDisable)
```

Disables/Enables core reverse carry.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to enable/disable reverse carry.
 - **true** Disable bit-reverse addressing mode.
 - **false** Enable bit-reverse addressing mode.

```
static inline void MCM_DisableDSP56800EXNewShadowRegion(MCM_Type *base, bool bDisable)
```

Disables/Enables the additional AGU shadow registers on the DSP56800EX core.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to disable/enable the additional AGU shadow register on the DPS core.
 - **true** Only the AGU shadow registers supported by the DSP56800E core are enabled.
 - **false** The additional AGU shadow registers on the DSP56800EX core are also enabled.

```
static inline void MCM_DisableCoreInstructionBuffer(MCM_Type *base, bool bDisable)
```

Disables/Enables core instruction buffer.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to disable/enable core longword instruction buffer.
 - **true** Disable core longword instruction buffer.
 - **false** Enable core longword instruction buffer.

```
static inline void MCM_DisableFlashMemoryControllerStall(MCM_Type *base, bool bDisable)
```

Disables/Enables the flash memory controller's ability to allow flash memory access to initiate when a flash memory command is executing.

Parameters

- base – MCM peripheral base address.
- bDisable – Used to disable/enable stall logic.
 - **true** Stall logic is disabled. While a flash memory command is executing, an attempted flash memory access causes a bus error.

- **false** Stall logic is disabled. While a flash memory command is executing, a flash memory access can occur without causing a bus error. The flash memory command completes execution, and then the flash memory access occurs.

```
static inline void MCM_SetAxsDMAControllerPriority(MCM_Type *base,  
                                                  mcm_axbs_dma_core_priority_t  
                                                  ePriority)
```

Sets the priority of the DMA controller in the AXBS crossbar switch arbitration scheme.

Parameters

- base – MCM base address.
- ePriority – The selected DMA controller priority in Crossbar switch arbitration scheme, please refer to `mcm_axbs_dma_core_priority_t`.

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

Gets the address of the last core access terminated with an error response.

Parameters

- base – MCM base address.

Returns

address of the last core access terminated with an error response.

```
void MCM_GetCoreFaultAttribute(MCM_Type *base, mcm_core_fault_attribute_t *psAttribute)
```

Gets the processor's attributes of the last faulted core access to the system bus.

Parameters

- base – MCM peripheral base address.
- psAttribute – The pointer of structure `mcm_core_fault_attribute_t`.

```
static inline mcm_last_fault_access_location_t MCM_GetCoreFaultLocation(MCM_Type *base)
```

Gets the location of the last captured fault.

Parameters

- base – MCM peripheral base address.

Returns

The location of the last captured fault, please refer to `mcm_last_fault_access_location_t`.

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

Gets the data associated with the last faulted processor write data access from the device's internal bus.

Parameters

- base – MCM base address.

Returns

The data associated with the last faulted processor write data access.

```
static inline void MCM_EnableCoreFaultInterrupt(MCM_Type *base, bool bEnable)
```

Enables/Disables core fault error interrupt.

Parameters

- base – MCM peripheral base address.
- bEnable – Used to enable/disable the core fault error interrupt.
 - **true** Enables core fault error interrupt, so an error interrupt will be generated to the interrupt controller on a faulted system bus cycle.

- **false** Disables core fault error interrupt, so an error interrupt will not be generated to the interrupt controller on a faulted system bus cycle.

```
static inline uint8_t MCM_GetCoreFaultStatusFlags(MCM_Type *base)
```

Gets the core fault error status flags, including core fault error interrupt flag and core fault error data lost flag.

Parameters

- base – MCM peripheral base address.

Returns

The current status flags, should be the OR'ed value of `_mcm_status_flags`.

```
static inline void MCM_ClearCoreFaultStatusFlags(MCM_Type *base, uint8_t u8StatusFlags)
```

Clears the core fault error status flags, including core fault error interrupt flag and core fault error data lost flag.

Parameters

- base – MCM peripheral base address.
- u8StatusFlags – The status flags to be cleared, should be the OR'ed value of `_mcm_status_flags`.

```
static inline void MCM_EnableResourceProtection(MCM_Type *base, bool bEnable)
```

Enables/Disables resource protection.

Parameters

- base – MCM peripheral base address.
- bEnable – Used to enable/disable memory resource protection.
 - **true** Enable memory resource protection.
 - **false** Disable memory resource protection.

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

Locks the value of the resource protection related registers, after locked the registers' value can not be changed until a system reset.

Parameters

- base – MCM peripheral base address.

```
status_t MCM_SetResourceProtectionConfig(MCM_Type *base, const
                                         mcm_resource_protection_config_t *psConfig)
```

Sets the configuration of resource protection, including flash base address, ram base address, etc.

Parameters

- base – MCM peripheral base address.
- psConfig – The pointer of structure `mcm_resource_protection_config_t`.

Return values

- kStatus_Success – Succeed to setting resource protection related options.
- kStatus_Fail – Fail to set resource protection related options.

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

Gets the 21-bit illegal faulting PC that only for a resource protection fault.

Parameters

- base – MCM peripheral base address.

Returns

The resource protection illegal faulting PC.

```
static inline bool MCM_IsResourceProtectionIllegalFaultValid(MCM_Type *base)
```

Indicates whether an resource protection illegal PC fault has occurred.

Parameters

- base – MCM peripheral base address.

Return values

- true – The resource protection illegal PC fault has occurred.
- false – The resource protection illegal PC fault has not occurred.

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

Clears the resource protection illegal fault bit.

Parameters

- base – MCM peripheral base address.

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

Gets the 21-bit misaligned faulting PC that only for a resource protection fault.

Parameters

- base – MCM peripheral base address.

Returns

The resource protection misaligned faulting PC.

```
static inline bool MCM_IsResourceProtectionMisalignedFaultValid(MCM_Type *base)
```

Indicates whether an resource protection misaligned PC fault has occurred.

Parameters

- base – MCM peripheral base address.

Return values

- true – The resource protection misaligned PC fault has occurred.
- false – The resource protection misaligned PC fault has not occurred.

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

Clears the resource protection misaligned fault bit.

Parameters

- base – MCM peripheral base address.

```
FSL_MCM_DRIVER_VERSION
```

MCM driver version.

```
enum _mcm_status_flags
```

The enumeration of status flags, including core fault error interrupt flag and core fault error data lost flag.

Values:

```
enumerator kMCM_CoreFaultErrorInterruptFlag
```

A bus error has occurred.

```
enumerator kMCM_CoreFaultErrorDataLostFlag
```

A bus error has occurred before the previous error condition was cleared.

enum `_mcm_datapath_width`

The enumeration of datapath width, including 32 bits and 64 bits.

Values:

enumerator `kMCM_Datapath32b`

Datapath width is 32 bits.

enumerator `kMCM_Datapath64b`

Datapath width is 64 bits.

enum `_mcm_axbs_dma_core_priority`

The enumeration of DMA controller priority in the Crossbar switch arbitration scheme.

Values:

enumerator `kMCM_AxbsPriorityCoreHigherThanDMA`

Fixed-priority arbitration is selected: DSC core has a higher priority than the DMA Controller's priority.

enumerator `kMCM_AxbsPriorityCoreDMARoundRobin`

Round-robin priority arbitration is selected: DMA Controller and DSC core have equal priority.

enum `_mcm_last_fault_access_dir`

The enumeration of last faulted core access direction.

Values:

enumerator `kMCM_CoreRead`

Core read access.

enumerator `kMCM_CoreWrite`

Core write access.

enum `_mcm_last_fault_access_size`

The enumeration of last faulted core access size.

Values:

enumerator `kMCM_Access8b`

Last faulted core access size is 8-bit.

enumerator `kMCM_Access16b`

Last faulted core access size is 16-bit.

enumerator `kMCM_Access32b`

Last faulted core access size is 32-bit.

enum `_mcm_last_fault_access_type`

The enumeration of last faulted core access type.

Values:

enumerator `kMCM_AccessInstruction`

Last faulted core access is instruction.

enumerator `kMCM_AccessData`

Last faulted core access is data.

enum `_mcm_last_fault_access_location`

The enumeration of last captured fault Location.

Values:

enumerator `kMCM_ErrOnInstructionBus`
Error occurred on M0 (instruction bus).

enumerator `kMCM_ErrOnOperandABus`
Error occurred on M1 (operand A bus).

enumerator `kMCM_ErrOnOperandBBus`
Error occurred on M2 (operand B bus).

typedef enum `_mcm_datapath_width` `mcm_datapath_width_t`
The enumeration of datapath width, including 32 bits and 64 bits.

typedef enum `_mcm_axbs_dma_core_priority` `mcm_axbs_dma_core_priority_t`
The enumeration of DMA controller priority in the Crossbar switch arbitration scheme.

typedef enum `_mcm_last_fault_access_dir` `mcm_last_fault_access_dir_t`
The enumeration of last faulted core access direction.

typedef enum `_mcm_last_fault_access_size` `mcm_last_fault_access_size_t`
The enumeration of last faulted core access size.

typedef enum `_mcm_last_fault_access_type` `mcm_last_fault_access_type_t`
The enumeration of last faulted core access type.

typedef enum `_mcm_last_fault_access_location` `mcm_last_fault_access_location_t`
The enumeration of last captured fault Location.

typedef struct `_mcm_core_fault_attribute` `mcm_core_fault_attribute_t`
The structure of core fault attributes, contains access type, access size, access direction, etc.

typedef struct `_mcm_resource_protection_config` `mcm_resource_protection_config_t`
The structure of the resource protection config, the set value can be used only when the resource protection is enabled, and this value can be changed only when the resource protection is disabled.

struct `_mcm_core_fault_attribute`
#include <fsl_mcm.h> The structure of core fault attributes, contains access type, access size, access direction, etc.

Public Members

`mcm_last_fault_access_type_t` eType
Indicates the last faulted core access type, please refer to `mcm_last_fault_access_type_t`.

`uint8_t` bitReserved1
Reserved 1 bit.

`bool` bBufferable
Indicates if last faulted core access was bufferable.

- **true** Last faulted core access is bufferable.
- **false** Last faulted core access is non-bufferable.

`uint8_t` bitReserved2
Reserved 1 bit.

`mcm_last_fault_access_size_t` eSize
Indicates last faulted core access size.

`mcm_last_fault_access_dir_t` eDirection
Indicates the last faulted core access direction.

```
struct _mcm_resource_protection_config
```

#include <fsl_mcm.h> The structure of the resource protection config, the set value can be used only when the resource protection is enabled, and this value can be changed only when the resource protection is disabled.

Public Members

```
bool bEnableResourceProtection
```

Enable/Disable resource protection.

- **true** Enable Resource protection.
- **false** Disable Resource protection.

```
uint8_t u8FlashBaseAddress
```

Flash base address for user region, supports 4 KB granularity.

```
uint8_t u8RamBaseAddress
```

Program RAM base address for user region, support 256 byte granularity.

```
uint32_t u32BootRomBaseAddress
```

Boot ROM base address for user region

```
uint32_t u32ResourceProtectionOtherSP
```

Resource protection other stack pointer.

2.47 The Driver Change Log

2.48 MCM Peripheral and Driver Overview

2.49 PIT: Periodic Interrupt Timer (PIT) Driver

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

Ungates the PIT clock, configures the PIT features. The configurations are:

- Clock source selection for PIT module
- Prescaler configuration to the input clock source
- PIT period interval
- PIT slave mode enable/disable
- Interrupt enable/disable
- PIT timer enable/disable
- Preset Polarity positive edge/negative edge

Note: This API should be called at the beginning of the application using the PIT driver and call PIT_StartTimer() API to start PIT timer.

Parameters

- *base* – PIT peripheral base address

- psConfig – 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

void PIT_GetDefaultConfig(*pit_config_t* *psConfig)

Fill in the PIT config structure with the default settings.

This function initializes the PIT configuration structure to default values.

```
psConfig->eClockSource = kPIT_CountClockSource0;
psConfig->bEnableTimer = false;
psConfig->bEnableSlaveMode = false;
psConfig->ePrescaler = kPIT_PrescalerDivBy1;
psConfig->bEnableInterrupt = false;
psConfig->u32PeriodCount = 0xFFFFFFFFU;
psConfig->bEnableNegativeEdge = false;
psConfig->sPresetFilter.u16FilterSamplePeriod = 0x0U;
psConfig->sPresetFilter.u16FilterSampleCount = 0x0U;
psConfig->sPresetFilter.bFilterClock = true;
psConfig->sPresetFilter.eFilterPrescalerPeripheral = kPIT_PrescalerDivBy1;
psConfig->sSyncSource.u8StretchCount = 0x0U;
psConfig->sSyncSource.eSyncOutSel = kPIT_Syncout_Default;
```

Parameters

- psConfig – Pointer to user’s PIT config structure.

static inline void PIT_EnableSlaveMode(PIT_Type *base, bool bEnable)

Enable/Disable PIT slave mode.

Parameters

- base – PIT peripheral base address
- bEnable – enable/disable slave mode

static inline void PIT_SetTimerPrescaler(PIT_Type *base, *pit_prescaler_value_t* ePrescaler)

Sets the PIT clock prescaler.

Parameters

- base – PIT peripheral base address
- ePrescaler – Timer prescaler value

static inline void PIT_SetTimerPeriod(PIT_Type *base, uint32_t u32PeriodCount)

Sets the timer period in units of count.

Timers begin counting from 0 until it reaches the value set by this function, then it generates an interrupt and counter resumes counting from 0 again.

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

Parameters

- base – PIT peripheral base address
- u32PeriodCount – Timer period in units of ticks, use macro definition `MSEC_TO_COUNT` to convert value in ms to count of ticks, the PIT clock rate is source clock divide prescaler.

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

Parameters

- `base` – PIT peripheral base address

Returns

Current timer counting value in ticks, use macro definition `COUNT_TO_MSEC` to convert value in ticks to count of millisecond, the PIT clock rate is source clock divide prescaler.

```
static inline void PIT_StartTimer(PIT_Type *base)
```

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

```
static inline void PIT_StopTimer(PIT_Type *base)
```

Stops the timer counting.

This function stops timer counting, and the counter remains at or returns to a 0 value.

Parameters

- `base` – PIT peripheral base address

```
static inline void PIT_EnableInterrupt(PIT_Type *base)
```

Enables the PIT interrupts.

Parameters

- `base` – PIT peripheral base address

```
static inline void PIT_DisableInterrupt(PIT_Type *base)
```

Disables the selected PIT interrupts.

Parameters

- `base` – PIT peripheral base address

```
static inline uint16_t PIT_GetStatusFlags(PIT_Type *base)
```

Gets the PIT status flags.

Parameters

- `base` – PIT peripheral base address

Returns

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

```
static inline void PIT_ClearStatusFlags(PIT_Type *base)
```

Clears the PIT status flags.

Parameters

- base – PIT peripheral base address

static inline void PIT_SetPresetFiltConfig(PIT_Type *base, const *pit_config_filt_t* psConfig)

Set FILT configurations.

Parameters

- base – PIT peripheral base address
- psConfig – Pointer to user's PIT FILT config structure

static inline void PIT_SetSyncOutConfig(PIT_Type *base, const *pit_config_ctrl2_t* psConfig)

Set Sync configurations.

Parameters

- base – PIT peripheral base address
- psConfig – Pointer to user's PIT SYNC config structure

FSL_PIT_DRIVER_VERSION

PIT driver version.

enum *_pit_prescaler_value*

PIT clock prescaler values.

Values:

enumerator kPIT_PrescalerDivBy1

Clock divided by 1

enumerator kPIT_PrescalerDivBy2

Clock divided by 2

enumerator kPIT_PrescalerDivBy4

Clock divided by 4

enumerator kPIT_PrescalerDivBy8

Clock divided by 8

enumerator kPIT_PrescalerDivBy16

Clock divided by 16

enumerator kPIT_PrescalerDivBy32

Clock divided by 32

enumerator kPIT_PrescalerDivBy64

Clock divided by 64

enumerator kPIT_PrescalerDivBy128

Clock divided by 128

enumerator kPIT_PrescalerDivBy256

Clock divided by 256

enumerator kPIT_PrescalerDivBy512

Clock divided by 512

enumerator kPIT_PrescalerDivBy1024

Clock divided by 1024

enumerator kPIT_PrescalerDivBy2048

Clock divided by 2048

enumerator kPIT_PrescalerDivBy4096

Clock divided by 4096

enumerator kPIT_PrescalerDivBy8192

Clock divided by 8192

enumerator kPIT_PrescalerDivBy16384

Clock divided by 16384

enumerator kPIT_PrescalerDivBy32768

Clock divided by 32768

enum `_pit_status_flags`

List of PIT status flags.

Values:

enumerator kPIT_Timer_RollOverFlag

Timer roll over flag

enum `_pit_syncout_mode`

List of SYNC_OUT output mode.

Values:

enumerator kPIT_Syncout_Default

SYNC_OUT takes affect when PIT counter equals to the MODULO value (default)

enumerator kPIT_Syncout_Toggle

SYNC_OUT is in toggle mode

typedef enum `_pit_prescaler_value` `pit_prescaler_value_t`

PIT clock prescaler values.

typedef enum `_pit_syncout_mode` `pit_syncout_mode_t`

List of SYNC_OUT output mode.

typedef struct `_pit_config_filt` `pit_config_filt_t`

PIT FILT configuration structure.

This structure holds the configuration settings for the PIT FILT register. 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.

typedef struct `_pit_config_ctrl2` `pit_config_ctrl2_t`

PIT CTRL2 configuration structure.

This structure holds the configuration settings for the PIT CTRL2 register. 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.

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.

struct `_pit_config_filt`

#include <fsl_pit.h> PIT FILT configuration structure.

This structure holds the configuration settings for the PIT FILT register. 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 `bFilterClock`

Filter Clock Source selection.

pit_prescaler_value_t `eFilterPrescalerPeripheral`

Sets the peripheral clock prescaler.

uint8_t `u16FilterSampleCount`

Input Filter Sample Count.

uint8_t `u16FilterSamplePeriod`

Input Filter Sample Period.

struct `_pit_config_ctrl2`

#include <fsl_pit.h> PIT CTRL2 configuration structure.

This structure holds the configuration settings for the PIT CTRL2 register. 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

uint8_t `u8StretchCount`

The cycle number to be stretched for SYNC_OUT signal.

pit_syncout_mode_t `eSyncOutSel`

Select the output mode of SYNC_OUT.

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

pit_prescaler_value_t `ePrescaler`

Clock prescaler value

bool `bEnableInterrupt`

Enable PIT Roll-Over Interrupt

bool `bEnableSlaveMode`

Enable the PIT module in slave mode, in which mode the timer will be triggered by master PIT enable.

`bool bEnableTimer`

PIT timer enable flag, which is false by default

`pit_count_clock_source_t eClockSource`

Specify the PIT count clock source

`uint32_t u32PeriodCount`

Timer period in clock cycles, Use macro definition `MSEC_TO_COUNT` to convert value in ms to count of ticks, the COP clock rate is source clock divide prescaler.

`bool bEnableNegativeEdge`

choose the polarity of Preset input.

`pit_config_filt_t sPresetFilter`

Specify the PIT preset filter source

`pit_config_ctrl2_t sSyncSource`

Specify the PIT Sync source

2.50 The Driver Change Log

2.51 PIT Peripheral and Driver Overview

2.52 PMC: Power Management Controller Driver

`static inline void PMC_SetBandgapTrim(PMC_Type *base, uint8_t u8TrimValue)`

Sets the trim value of the bandgap reference in the regulator.

Parameters

- `base` – PMC peripheral base address.
- `u8TrimValue` – The bandgap's trim value, ranges from 0 to 15.

`static inline void PMC_EnableVoltageReferenceBuffer(PMC_Type *base, bool bEnable)`

Enables/Disables a buffer that drivers the 1.2V bandgap reference to the ADC.

If the users want to calibrate the ADC using the 1.2V reference voltage, then the voltage reference buffer should be enabled. When ADC calibration is not being performed, the voltage reference buffer should be disabled to save power.

Parameters

- `base` – PMC peripheral base address.
- `bEnable` – Used to control the behaviour of voltage reference buffer.
 - **true** Enable voltage reference buffer.
 - **false** Disable voltage reference buffer.

`static inline void PMC_EnableInterrupts(PMC_Type *base, uint16_t u16Interrupts)`

Enables the interrupts, including 2.2V high voltage interrupt, 2.7V/2.65V high voltage interrupt, 2.2V low voltage interrupt, 2.7V/2.65V low voltage interrupt.

Parameters

- `base` – PMC peripheral base address.
- `u16Interrupts` – The interrupts to be enabled, should be the OR'ed value of `_pmc_interrupt_enable`.

```
static inline void PMC_DisableInterrupts(PMC_Type *base, uint16_t u16Interrupts)
```

Disables the interrupts, including 2.2V high voltage interrupt, 2.7V/2.65V high voltage interrupt, 2.2V low voltage interrupt, 2.7V/2.65V low voltage interrupt.

Parameters

- base – PMC peripheral base address.
- u16Interrupts – The interrupts to be disabled, should be the OR'ed value of `_pmc_interrupt_enable`.

```
static inline uint16_t PMC_GetStatusFlags(PMC_Type *base)
```

Gets the status flags of PMC module, such as low voltage interrupt flag, small regulator 2.7 active flag, etc.

Parameters

- base – PMC peripheral base address.

Returns

The status flags of PMC module, should be the OR'ed value of `_pmc_status_flags`.

```
static inline void PMC_ClearStatusFlags(PMC_Type *base, uint16_t u16StatusFlags)
```

Clears the status flags of PMC module, only low voltage interrupt flag, sticky 2.7V/2.65V low voltage flag, and sticky 2.2V low voltage flag can be cleared.

Parameters

- base – PMC peripheral base address.
- u16StatusFlags – The status flags to be cleared, should be the OR'ed value of `kPMC_LowVoltageInterruptFlag`, and `kPMC_Sticky2P7VLowVoltageFlag/kPMC_Sticky2P65VLowVoltageFlag`, and `kPMC_Sticky2P2VLowVoltageFlag`,

```
static inline void PMC_SetVrefTrim(PMC_Type *base, uint16_t u16TrimValue)
```

Sets the trim value of the Vref reference in the regulator.

Parameters

- base – PMC peripheral base address.
- u16TrimValue – The Vref's trim value, ranges from 0 to 31.

```
static inline void PMC_SetVcapTrim(PMC_Type *base, uint16_t u16TrimValue)
```

Sets the trim value of the Vcap reference in the regulator.

Parameters

- base – PMC peripheral base address.
- u16TrimValue – The Vcap's trim value, ranges from 0 to 15.

```
FSL_PMC_DRIVER_VERSION
```

PMC driver version.

```
enum _pmc_interrupt_enable
```

The enumeration of PMC voltage detection interrupts.

Values:

```
enumerator kPMC_2P2VLowVoltageInterruptEnable
```

If the input supply is currently dropped below the 2.2V level, generate the low voltage interrupt.

enumerator kPMC_2P2VHighVoltageInterruptEnable

If the input supply is currently raised above the 2.2V level, generate the low voltage interrupt.

enumerator kPMC_AllInterruptsEnable

enum __pmc_status_flags

The enumeration of PMC status flags.

Values:

enumerator kPMC_SmallRegulator2P7VActiveFlag

The small regulator 2.7V supply is ready to be used.

enumerator kPMC_LowVoltageInterruptFlag

The low voltage interrupt flag, used to indicate whether the low voltage interrupt is asserted.

enumerator kPMC_Sticky2P2VLowVoltageFlag

Input supply has dropped below the 2.2V threshold. This sticky flag indicates that the input supply dropped below the 2.2V level at some point.

enumerator kPMC_2P2VLowVoltageFlag

Input supply is below the 2.2V threshold.

enumerator kPMC_AllStatusFlags

2.53 The Driver Change Log

2.54 PMC Peripheral and Driver Overview

2.55 eFlexPWM: Enhanced Flexible Pulse Width Modulator Driver

void PWM_Init(PWM_Type *base, const *pwm_config_t* *psConfig)

Initialization PWM module with provided structure *pwm_config_t*.

This function can initial one or more submodules of the PWM module.

This examples shows how only initial submodule 0 without fault protection channel.

```
pwm_config_t sPwmConfig = {0};
pwm_sm_config_t sPwmSm0Config;
sPwmConfig.psPwmSubmoduleConfig[0] = &sPwmSm0Config;
PWM_GetSmDefaultConfig(&sPwmSm0Config);
PWM_Init(PWM, sPwmConfig);
```

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

Parameters

- base – PWM peripheral base address.
- psConfig – Pointer to PWM module configure structure. See *pwm_config_t*.

```
void PWM_Deinit(PWM_Type *base)
```

De-initialization a PWM module.

Parameters

- base – PWM peripheral base address

```
void PWM_GetSMDDefaultConfig(pwm_sm_config_t *psConfig)
```

Gets an default PWM submodule's configuration.

This function fills in the initialization structure member, which can make submodule generate 50% duty cycle center aligned PWM_A/B output.

The default effective values are:

```
psConfig->enableDebugMode = false;
psConfig->enableWaitMode = false;
psConfig->enableRun = false;
psConfig->sCounterConfig.eCountClockSource = kPWM_ClockSrcBusClock;
psConfig->sCounterConfig.eCountClockPrescale = kPWM_ClockPrescaleDivide1;
psConfig->sCounterConfig.eCountInitSource = kPWM_InitOnLocalSync;
psConfig->sReloadConfig.eReloadSignalSelect = kPWM_LocalReloadSignal;
psConfig->sReloadConfig.eLoclReloadEffectTime = kPWM_TakeEffectAtReloadOportunity;
psConfig->sReloadConfig.eLocalReloadOportunity = kPWM_LoadEveryOportunity;
psConfig->sReloadConfig.bEnableFullCycleReloadOportunity = true;
psConfig->sReloadConfig.bEnableHalfCycleReloadOportunity = false;
psConfig->sValRegisterConfig.u16CounterInitialValue = 0xFF00U;
psConfig->sValRegisterConfig.u16ValRegister0 = 0x0U;
psConfig->sValRegisterConfig.u16ValRegister1 = 0x00FFU;
psConfig->sValRegisterConfig.u16ValRegister2 = 0xFF80U;
psConfig->sValRegisterConfig.u16ValRegister3 = 0x80U;
psConfig->sValRegisterConfig.u16ValRegister4 = 0xFF80U;
psConfig->sValRegisterConfig.u16ValRegister5 = 0x80U;
psConfig->sForceConfig.eForceSignalSelect = kPWM_LocalSoftwareForce;
psConfig->sForceConfig.eSoftOutputFor23 = kPWM_SoftwareOutputLow;
psConfig->sForceConfig.eSoftOutputFor45 = kPWM_SoftwareOutputLow;
psConfig->sForceConfig.eForceOutput23 = kPWM_GeneratedPwm;
psConfig->sForceConfig.eForceOutput45 = kPWM_GeneratedPwm;
psConfig->sDeadTimeConfig.eMode = kPWM_Independent;
psConfig->sOutputConfig.ePwmXSignalSelect = kPWM_RawPwmX;
psConfig->sOutputConfig.bEnablePwmXOutput = true;
psConfig->sOutputConfig.bEnablePwmaOutput = true;
psConfig->sOutputConfig.bEnablePwmbOutput = true;
psConfig->sOutputConfig.ePwmxFaultState = kPWM_OutputLowOnFault;
psConfig->sOutputConfig.ePwmaFaultState = kPWM_OutputLowOnFault;
psConfig->sOutputConfig.ePwmbFaultState = kPWM_OutputLowOnFault;
```

Parameters

- psConfig – Pointer to user's PWM submodule config structure. See `pwm_sm_config_t`.

```
void PWM_GetFaultProtectionDefaultConfig(pwm_fault_protection_config_t *psConfig)
```

Gets an default fault protection channel's configuration.

The default effective values are:

```
psConfig->sFaultInput[i].eFaultActiveLevel = kPWM_Logic0;
psConfig->sFaultInput[i].bEnableAutoFaultClear = true;
psConfig->sFaultInput[i].bEnableFaultFullCycleRecovery = true;
```

Parameters

- `psConfig` – Pointer to user's PWM fault protection config structure. See `pwm_fault_protection_config_t`.

```
void PWM_SetupSMConfig(PWM_Type *base, pwm_sm_number_t eSubModule, const
    pwm_sm_config_t *psConfig)
```

Sets up the PWM submodule configure.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `psConfig` – Pointer to submodule configure structure, see `pwm_sm_config_t`.

```
static inline void PWM_SetupCounterConfig(PWM_Type *base, pwm_sm_number_t eSubModule,
    const pwm_sm_counter_config_t *psConfig)
```

Sets up the PWM submodule counter configure.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `psConfig` – Pointer to submodule counter configure structure, see `pwm_sm_counter_config_t`.

```
static inline void PWM_SetCounterInitialValue(PWM_Type *base, pwm_sm_number_t
    eSubModule, uint16_t u16InitialValue)
```

Sets the PWM submodule counter initial register value.

This function set the INIT register value, the counter will start counting from INIT register value when initial signal assert or software force set. This write value will be loaded into inner set of buffered registers according to reload logic configure.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `u16InitialValue` – The submodule number counter initialize value.

```
static inline void PWM_SetupReloadLogicConfig(PWM_Type *base, pwm_sm_number_t
    eSubModule, const
    pwm_sm_reload_logic_config_t *psConfig)
```

Sets up the PWM submodule reload logic configure.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `psConfig` – Pointer to submodule reload logic configure structure, see `pwm_sm_reload_logic_config_t`.

```
void PWM_GetValueConfig(pwm_sm_value_register_config_t *psConfig,
    pwm_sm_typical_output_mode_t eTypicalOutputMode, uint16_t
    u16PwmPeriod, uint16_t u16PwmAPulseWidth, uint16_t
    u16PwmBPulseWidth)
```

Update PWM submodule compare value configuration according to the typical output mode.

Parameters

- `psConfig` – See `pwm_sm_config_t`.

- `eTypicalOutputMode` – Typical PWM_A/B output mode. See `pwm_sm_typical_output_mode_t`.
- `u16PwmPeriod` – PWM output period value in counter ticks. This value can be got by (main counter clock in Hz) / (wanted PWM signal frequency in Hz).
- `u16PwmAPulseWidth` – PWM_A pulse width value in counter ticks. Can got by (wanted PWM duty Cycle) * `u16PwmPeriod`.
- `u16PwmBPulseWidth` – PWM_B pulse width value in counter ticks. Can got by (wanted PWM duty Cycle) * `u16PwmPeriod`.

```
static inline void PWM_SetupValRegisterConfig(PWM_Type *base, pwm_sm_number_t
                                             eSubModule, const
                                             pwm_sm_value_register_config_t *psConfig)
```

Sets up the PWM submodule VALn registers logic configure.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `psConfig` – Pointer to VALn registers configure structure, see `pwm_sm_value_register_config_t`.

```
static inline void PWM_SetValueRegister(PWM_Type *base, pwm_sm_number_t eSubModule,
                                       pwm_sm_val_register_t eRegister, uint16_t u16Value)
```

Sets the PWM submodule VALn register value.

Note: These write value will be loaded into inner set of buffered registers according to reload logic configure.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eRegister` – Value register index (range in 0~5), see `pwm_sm_val_register_t`.
- `u16Value` – The value for VALn register.

```
static inline uint16_t PWM_GetValueRegister(PWM_Type *base, pwm_sm_number_t
                                           eSubModule, pwm_sm_val_register_t eRegister)
```

Gets the PWM submodule VALn register value.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eRegister` – Value register index (range in 0~5), see `pwm_sm_val_register_t`.

Returns

The VALn register value.

```
static inline void PWM_SetFracvalRegister(PWM_Type *base, pwm_sm_number_t eSubModule,
                                         pwm_sm_fracval_register_t eRegister, uint16_t
                                         u16Value)
```

Sets the PWM submodule fractional value register value.

Note: These write value will be loaded into inner set of buffered registers according to reload logic configure.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- eRegister – Fractional value register index (range in 1~5), see `pwm_sm_val_register_t`.
- u16Value – The value for FRACVALn register.

```
static inline uint16_t PWM_GetFracvalRegister(PWM_Type *base, pwm_sm_number_t
                                             eSubModule, pwm_sm_fracval_register_t
                                             eRegister)
```

Sets the PWM submodule fractional value register value.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- eRegister – Fractional value register index (range in 1~5), see `pwm_sm_fracval_register_t`.

Returns

The VALn FRACVALn value.

```
static inline void PWM_SetValueAndFracRegister(PWM_Type *base, pwm_sm_number_t
                                              eSubModule, pwm_sm_fracval_register_t
                                              eRegister, uint32_t u32Value)
```

Set submodule register VALx and its FRAC value with 32bit access.

Parameters

- base – PWM peripheral base address.
- eSubModule – Submodule ID.
- eRegister – Fractional value register index (range in 1~5), see `pwm_sm_fracval_register_t`.
- u32Value – 32bit value for VALx and its FRAC. VALx: BIT16~BIT31. FRAC-VALx: BIT11~BIT15. RESERVED: BIT10~BIT0.

```
static inline uint32_t PWM_GetValueAndFracRegister(PWM_Type *base, pwm_sm_number_t
                                                  eSubModule, pwm_sm_fracval_register_t
                                                  eRegister)
```

Get submodule register VALx and its FRAC value with 32bit access.

Parameters

- base – PWM peripheral base address.
- eSubModule – Submodule ID.
- eRegister – Fractional value register index (range in 1~5), see `pwm_sm_fracval_register_t`.

Returns

The value of submodule register VALx and its FRAC, combined into 32bit. VALx: BIT16~BIT31. FRACVALx: BIT11~BIT15. RESERVED: BIT10~BIT0.

```
static inline void PWM_SetPwmLdok(PWM_Type *base, uint16_t u16Mask)
```

Set the PWM LDOK bit on a single or multiple submodules.

Enable this feature can make buffered CTRL[PRSC] and the INIT, FRACVAL and VAL registers values take effect after next local load signal assert. The timing of take effect can be the next PWM reload or immediately. After loading, MCTRL[LDOK] is automatically cleared and need to enable again before the next register updated.

Note: The VALx, FRACVALx, INIT, and CTRL[PRSC] registers of the corresponding submodule cannot be written while the the corresponding MCTRL[LDOK] bit is set.

Parameters

- base – PWM peripheral base address
- u16Mask – PWM submodules to set the LDOK bit, Logical OR of `_pwm_sm_enable`.

```
static inline void PWM_ClearPwmLdok(PWM_Type *base, uint16_t u16Mask)
```

Clear the PWM LDOK bit on a single or multiple submodules.

Parameters

- base – PWM peripheral base address
- u16Mask – PWM submodules to clear the LDOK bit, Logical OR of `_pwm_sm_enable`.

```
static inline void PWM_SetupForceLogicConfig(PWM_Type *base, pwm_sm_number_t  
                                             eSubModule, const pwm_sm_force_logic_config_t  
                                             *psConfig)
```

brief Sets up the PWM submodule force logic configure.

param base PWM peripheral base address. param eSubModule PWM submodule number, see `pwm_sm_number_t`. param psConfig Poniter to submodule force logic configure structure, see `pwm_sm_force_logic_config_t`.

```
static inline void PWM_SetSoftwareForce(PWM_Type *base, pwm_sm_number_t eSubModule)
```

Sets up the PWM Sub-Module to trigger a software FORCE_OUT event.

Note: Only works when the CTRL2[FORCE_SEL] select `kPWM_ForceOutOnLocalSoftware`.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.

```
void PWM_SetupDeadtimeConfig(PWM_Type *base, pwm_sm_number_t eSubModule, const  
                             pwm_sm_deadtime_logic_config_t *psConfig)
```

Sets up the PWM submodule deadtime logic configure.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- psConfig – Pointer to deadtime logic configure structure, see `pwm_sm_deadtime_logic_config_t`.

```
static inline uint16_t PWM_GetDeadtimeSampleValue(PWM_Type *base, pwm_sm_number_t
                                                eSubModule)
```

Get the sampled values of the PWM_X input at the end of each deadtime.

When use PWM_A/B in complementary mode and connect to transistor to controls the output voltage. Need insert deadtime to avoid overlap of conducting interval between the top and bottom transistor. And both transistors in complementary mode are off during deadtime. Then connect the PWM_X input to complementary transistors output, then it sampling input at the end of deadtime 0 for DT[0] and the end of deadtime 1 for DT[1]. Which DT value is not 0 indicates that there is a problem with the corresponding deadtime value. This can help to decide if there need do a deadtime correction for current complementary PWM output.

Parameters

- base – PWM peripheral base address
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.

Returns

The PWM_X input sampled values.

```
void PWM_SetupFractionalDelayConfig(PWM_Type *base, pwm_sm_number_t eSubModule,
                                   const pwm_sm_fractional_delay_logic_config_t *psConfig)
```

Sets up the PWM submodule fractional delay logic configure.

Note: The fractional delay logic can only be used when the IPBus clock is running at 100 MHz.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- psConfig – Pointer to fractional delay logic configure structure, see `pwm_sm_fractional_delay_logic_config_t`.

```
void PWM_SetupOutputConfig(PWM_Type *base, pwm_sm_number_t eSubModule, const
                           pwm_sm_output_logic_config_t *psConfig)
```

Sets up the PWM submodule output logic configure.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- psConfig – Pointer to output logic configure structure, see `pwm_sm_output_logic_config_t`.

```
static inline void PWM_EnableOutput(PWM_Type *base, uint16_t u16SubModules,
                                    pwm_sm_pwm_out_t eOutput)
```

Enables the PWM submodule pin output.

This function handles PWMX_EN/PWMA_EN/PWMB_EN bit filed of OUTEN register, whcih can enable one or more submodule pin in PWMX/A/B.

Parameters

- base – PWM peripheral base address
- u16SubModules – The submodules that enable eOutput output, logical OR of `_pwm_sm_enable`.

- `eOutput` – PWM output pin ID, see `pwm_sm_pwm_out_t`.

```
static inline void PWM_DisableOutput(PWM_Type *base, uint16_t u16SubModules,  
                                     pwm_sm_pwm_out_t eOutput)
```

Disables the PWM submodule pin output.

This function handles PWMX_EN/PWMA_EN/PWMB_EN bit filed of OUTEN register, which can disable one or more submodule pin in PWMX/A/B.

Parameters

- `base` – PWM peripheral base address
- `u16SubModules` – The submodules that disable `eOutput` output, logical OR of `_pwm_sm_enable`.
- `eOutput` – PWM output pin ID, see `pwm_sm_pwm_out_t`.

```
static inline void PWM_EnableCombinedOutput(PWM_Type *base, uint16_t u16XSubModules,  
                                           uint16_t u16ASubModules, uint16_t  
                                           u16BSubModules)
```

Enables the PWM pin combination output.

This function handles PWMX_EN/PWMA_EN/PWMB_EN bit filed of OUTEN register at the same time.

Parameters

- `base` – PWM peripheral base address
- `u16XSubModules` – The submodules that enable PWMX output, should be logical OR of `_pwm_sm_enable`.
- `u16ASubModules` – The submodules that enable PWMA output, should be logical OR of `_pwm_sm_enable`.
- `u16BSubModules` – The submodules that enable PWMB output, should be logical OR of `_pwm_sm_enable`.

```
static inline void PWM_DisableCombinedOutput(PWM_Type *base, uint16_t u16XSubModules,  
                                             uint16_t u16ASubModules, uint16_t  
                                             u16BSubModules)
```

Disables the PWM pin combination output.

This function handles PWMX_EN/PWMA_EN/PWMB_EN bit filed of OUTEN register at the same time.

Parameters

- `base` – PWM peripheral base address
- `u16XSubModules` – The submodules that disable PWMX output, should be logical OR of `_pwm_sm_enable`.
- `u16ASubModules` – The submodules that disable PWMA output, should be logical OR of `_pwm_sm_enable`.
- `u16BSubModules` – The submodules that disable PWMB output, should be logical OR of `_pwm_sm_enable`.

```
static inline void PWM_MaskOutput(PWM_Type *base, uint16_t u16SubModules,  
                                  pwm_sm_pwm_out_t eOutput)
```

Mask the PWM pin output.

This function handles MASKA/MASKB/MASKX bit filed of MASK register, which can mask one or more submodule pin in PWMX/A/B.

Note: The mask bits is buffered and can be updated until a FORCE_OUT event occurs or a software update command.

Parameters

- `base` – PWM peripheral base address.
- `u16SubModules` – The submodules that mask eOutput output, logical OR of `_pwm_sm_enable`.
- `eOutput` – PWM output pin ID, see `pwm_sm_pwm_out_t`.

```
static inline void PWM_UnmaskOutput(PWM_Type *base, uint16_t u16SubModules,
                                   pwm_sm_pwm_out_t eOutput)
```

Unmask the PWM pin output.

This function handles MASKA/MASKB/MASKX bit filed of MASK register, which can mask one or more submodule pin in PWMX/A/B.

Note: The mask bits is buffered and can be updated until a FORCE_OUT event occurs or a software update command.

Parameters

- `base` – PWM peripheral base address
- `u16SubModules` – The submodules that unmask eOutput output, logical OR of `_pwm_sm_enable`.
- `eOutput` – PWM output pin ID, see `pwm_sm_pwm_out_t`.

```
static inline void PWM_MaskCombinedOutput(PWM_Type *base, uint16_t u16XSubModules,
                                          uint16_t u16ASubModules, uint16_t
                                          u16BSubModules)
```

Mask the PWM pin combination output.

This function handles MASKA/MASKB/MASKX bit filed of MASK register at the same time.

Note: The mask bits is buffered and can be updated until a FORCE_OUT event occurs or a software update command.

Parameters

- `base` – PWM peripheral base address
- `u16XSubModules` – The submodules that mask PWMX output, should be logical OR of `_pwm_sm_enable`.
- `u16ASubModules` – The submodules that mask PWMA output, should be logical OR of `_pwm_sm_enable`.
- `u16BSubModules` – The submodules that mask PWMB output, should be logical OR of `_pwm_sm_enable`.

```
static inline void PWM_UnmaskCombinedOutput(PWM_Type *base, uint16_t u16XSubModules,
                                           uint16_t u16ASubModules, uint16_t
                                           u16BSubModules)
```

Unmask the PWM pin combination output.

This function handles MASKA/MASKB/MASKX bit filed of MASK register at the same time.

Note: The mask bits is buffered and can be updated until a FORCE_OUT event occurs or a software update command.

Parameters

- `base` – PWM peripheral base address
- `u16XSubModules` – The submodules that unmask PWMX output, should be logical OR of `_pwm_sm_enable`.
- `u16ASubModules` – The submodules that unmask PWMA output, should be logical OR of `_pwm_sm_enable`.
- `u16BSubModules` – The submodules that unmask PWMB output, should be logical OR of `_pwm_sm_enable`.

static inline void PWM_UpdateMask(PWM_Type *base, *pwm_sm_number_t* eSubModule)
Update PWM output mask bits immediately with a software command.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.

static inline void PWM_EnablePwmRunInDebug(PWM_Type *base, *pwm_sm_number_t* eSubModule, bool bEnable)

Enables/Disables the PWM submodule continue to run while the chip is in DEBUG mode.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `bEnable` – Enable the feature or not.
 - **true** Enable load feature.
 - **false** Disable load feature.

static inline void PWM_EnablePwmRunInWait(PWM_Type *base, *pwm_sm_number_t* eSubModule, bool bEnable)

Enables/Disables the PWM submodule continue to run while the chip is in WAIT mode.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `bEnable` – Enable the feature or not.
 - **true** Enable load feature.
 - **false** Disable load feature.

static inline void PWM_EnableCounters(PWM_Type *base, uint16_t u16Mask)

Starts the PWM submodule counter for a single or multiple submodules.

Sets the Run bit which enables the clocks to the PWM submodule. This function can start multiple submodules at the same time.

Parameters

- `base` – PWM peripheral base address
- `u16Mask` – PWM submodules to start run, Logical OR of `_pwm_sm_enable`.

```
static inline void PWM_DisableCounters(PWM_Type *base, uint16_t u16Mask)
```

Stops the PWM counter for a single or multiple submodules.

Clears the Run bit which resets the submodule's counter. This function can stop multiple submodules at the same time.

Parameters

- *base* – PWM peripheral base address
- *u16Mask* – PWM submodules to start run, Logical OR of *_pwm_sm_enable*.

```
static inline uint16_t PWM_GetCaptureValue(PWM_Type *base, pwm_sm_number_t
                                         eSubModule, pwm_sm_input_capture_register_t
                                         eRegister)
```

Reads PWM submodule input capture value register.

This function read the CVALn register value, stores the value captured from the submodule counter.

Parameters

- *base* – PWM peripheral base address.
- *eSubModule* – PWM submodule number, see *pwm_sm_number_t*.
- *eRegister* – PWM submodule input capture value register, see *pwm_sm_input_capture_register_t*.

Returns

The input capture value.

```
static inline uint16_t PWM_GetCaptureValueCycle(PWM_Type *base, pwm_sm_number_t
                                               eSubModule,
                                               pwm_sm_input_capture_register_t eRegister)
```

Reads PWM submodule input capture value cycle register.

This function read the CVALnCYC register value, stores the cycle number corresponding to the value captured in CVALn. This register is incremented each time the counter is loaded with the INIT value at the end of a PWM modulo cycle.

Parameters

- *base* – PWM peripheral base address.
- *eSubModule* – PWM submodule number, see *pwm_sm_number_t*.
- *eRegister* – PWM submodule input capture value register, see *pwm_sm_input_capture_register_t*.

Returns

The input capture register cycle value.

```
static inline uint16_t PWM_GetCaptureEdgeCounterVaule(PWM_Type *base, pwm_sm_number_t
                                                     eSubModule,
                                                     pwm_sm_input_capture_pin_t
                                                     eInputPin)
```

Reads the PWM submodule input capture logic edge counter value.

Each input capture logic has a edge counter, which counts both the rising and falling edges of the input capture signal and it compare signal can select as input capture trigger source.

Parameters

- *base* – PWM peripheral base address.
- *eSubModule* – PWM submodule number, see *pwm_sm_number_t*.

- `eInputPin` – PWM submodule input capture pin number, see `pwm_sm_input_capture_pin_t`.

```
void PWM_SetupInputCaptureConfig(PWM_Type *base, pwm_sm_number_t eSubModule,  
                                pwm_sm_input_capture_pin_t eInputPin, const  
                                pwm_sm_input_capture_config_t *psConfig)
```

Sets up the PWM submodule input capture configure.

Each PWM submodule has 3 pins that can be configured for use as input capture pins. This function sets up the capture parameters for each pin and enables the input capture operation.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eInputPin` – PWM submodule input capture pin number, see `pwm_sm_input_capture_pin_t`.
- `psConfig` – Pointer to input capture configure structure, see `pwm_sm_input_capture_config_t`.

```
static inline void PWM_EnableInputCapture(PWM_Type *base, pwm_sm_number_t eSubModule,  
                                         pwm_sm_input_capture_pin_t eInputPin)
```

Enables the PWM submodule input capture operation.

Enables input capture operation will start the input capture process. The enable bit is self-cleared when in one shot mode and one or more of the enabled capture circuits has had a capture event.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eInputPin` – PWM submodule input capture pin number, see `pwm_sm_input_capture_pin_t`.

```
static inline void PWM_DisableInputCapture(PWM_Type *base, pwm_sm_number_t  
                                          eSubModule, pwm_sm_input_capture_pin_t  
                                          eInputPin)
```

Disables the PWM submodule input capture operation.

The enable bit can be cleared at any time to disable input capture operation.

Parameters

- `base` – PWM peripheral base address.
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eInputPin` – PWM submodule input capture pin number, see `pwm_sm_input_capture_pin_t`.

```
static inline uint16_t PWM_GetInputValue(PWM_Type *base, pwm_sm_number_t eSubModule,  
                                         pwm_sm_input_capture_pin_t eInputPin)
```

Get the logic value currently being driven into the PWM inputs.

Parameters

- `base` – PWM peripheral base address
- `eSubModule` – PWM submodule number, see `pwm_sm_number_t`.
- `eInputPin` – PWM submodule input capture pin number, see `pwm_sm_input_capture_pin_t`.

Returns

The PWM submodule input capture pin logic value.

```
void PWM_SetupFaultProtectionConfig(PWM_Type *base, pwm_fault_protection_channel_t
    eFaultProtection, const pwm_fault_protection_config_t
    *psConfig)
```

Sets up the PWM fault protection channel configure.

Parameters

- base – PWM peripheral base address.
- eFaultProtection – PWM fault protection channel number, see `pwm_fault_protection_channel_t`.
- psConfig – Pointer to fault protection channel configure structure, see `pwm_fault_protection_config_t`.

```
static inline void PWM_SetupSMFaultInputMapping(PWM_Type *base, pwm_sm_number_t
    eSubModule, pwm_sm_pwm_out_t
    ePwmOutput, const
    pwm_sm_fault_input_mapping_t
    *psMapping)
```

Mapping fault protection channel fault input status to PWM submodule output.

Note: Each PWM output can be mapping anyone or more fault inputs. The mapped fault protection channel inputs can disable PWM output.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- ePwmOutput – PWM submodule output, see `pwm_sm_pwm_out_t`.
- psMapping – The fault input disable mapping structure, see `pwm_sm_fault_input_mapping_t`.

```
void PWM_SetupDmaConfig(PWM_Type *base, pwm_sm_number_t eSubModule, const
    pwm_sm_dma_config_t *psConfig)
```

Sets up the PWM submodule DMA configure.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- psConfig – Pointer to PWM submodule DMA configure, see `pwm_sm_dma_config_t`.

```
static inline void PWM_SetEnabledCaptureDmaSource(PWM_Type *base, pwm_sm_number_t
    eSubModule,
    pwm_sm_capture_dma_source_t
    eCaptureDmaSource)
```

Select the trigger source for enabled capture FIFOs DMA read request.

Note: This function only can be used when the `bEnableCaptureDMA` be true.

Parameters

- base – PWM peripheral base address.
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- eCaptureDmaSource – The PWM DMA capture source.

```
static inline void PWM_EnableSMInterrupts(PWM_Type *base, pwm_sm_number_t eSubModule,
                                         uint16_t u16Mask)
```

Enables the PWM submodule interrupts according to a provided mask.

This examples shows how to enable VAL 0 compare interrupt and VAL 1 compare interrupt.

```
PWM_EnableSMInterrupts(PWM, kPWM_SubModule0, kPWM_CompareVal0InterruptEnable |
kPWM_CompareVal1InterruptEnable);
```

Parameters

- base – PWM peripheral base address
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- u16Mask – The PWM submodule interrupts to enable. Logical OR of `_pwm_sm_interrupt_enable`.

```
static inline void PWM_DisbaleSMInterrupts(PWM_Type *base, pwm_sm_number_t eSubModule,
                                           uint16_t u16Mask)
```

Disables the PWM submodule interrupts according to a provided mask.

This examples shows how to disable VAL 0 compare interrupt and VAL 1 compare interrupt.

```
PWM_DisbaleSMInterrupts(PWM, kPWM_SubModule0, kPWM_CompareVal0InterruptEnable |
kPWM_CompareVal1InterruptEnable);
```

Parameters

- base – PWM peripheral base address
- eSubModule – PWM submodule number, see `pwm_sm_number_t`.
- u16Mask – The PWM submodule interrupts to enable. Logical OR of `_pwm_sm_interrupt_enable`.

```
static inline void PWM_EnableFaultInterrupts(PWM_Type *base,
                                             pwm_fault_protection_channel_t eFaultProtection,
                                             uint16_t u16Mask)
```

Enables the PWM fault protection channel interrupt according to a provided mask.

This examples shows how to enable fault pin 0 interrupt and fault pin 1 interrupt.

```
PWM_EnableFaultInterrupts(PWM, kPWM_FaultProtection0, kPWM_Fault0InterruptEnable |
kPWM_Fault1InterruptEnable);
```

Parameters

- base – PWM peripheral base address
- eFaultProtection – PWM fault protection channel number, see `pwm_fault_protection_channel_t`.
- u16Mask – The PWM fault protection channel interrupts to enable. Logical OR of `_pwm_fault_protection_interrupt_enable`.

```
static inline void PWM_DisableFaultInterrupts(PWM_Type *base,
                                              pwm_fault_protection_channel_t
                                              eFaultProtection, uint16_t u16Mask)
```

Disables the PWM fault protection channel interrupt according to a provided mask. This examples shows how to disable fault pin 0 interrupt and fault pin 1 interrupt.

```
PWM_DisableFaultInterrupts(PWM, kPWM_FaultProtection0, kPWM_Fault0InterruptEnable |
kPWM_Fault1InterruptEnable);
```

Parameters

- *base* – PWM peripheral base address
- *eFaultProtection* – PWM fault protection channel number, see `pwm_fault_protection_channel_t`.
- *u16Mask* – The PWM fault protection channel interrupts to disable. Logical OR of `_pwm_fault_protection_interrupt_enable`.

```
static inline uint16_t PWM_GetSMStatusFlags(PWM_Type *base, pwm_sm_number_t
eSubModule)
```

Gets the PWM submodule status flags.

This examples shows how to check whether the submodule VAL0 compare flag set.

```
if((PWM_GetSMStatusFlags(PWM, kPWM_SubModule0) & kPWM_CompareVal0Flag) != 0U)
{
    ...
}
```

Parameters

- *base* – PWM peripheral base address
- *eSubModule* – PWM submodule number, see `pwm_sm_number_t`.

Returns

The PWM submodule status flags. This is the logical OR of `pwm_sm_status_flags_t`.

```
static inline void PWM_ClearSMStatusFlags(PWM_Type *base, pwm_sm_number_t eSubModule,
uint16_t u16Mask)
```

Clears the PWM submodule status flags.

This examples shows how to clear the submodule VAL0 compare flag.

```
PWM_ClearSMStatusFlags(PWM, kPWM_SubModule0, kPWM_CompareVal0Flag);
```

Note: The `kPWM_RegUpdatedFlag` can't be cleared by software.

Parameters

- *base* – PWM peripheral base address
- *eSubModule* – PWM submodule number, see `pwm_sm_number_t`.
- *u16Mask* – The status flags to clear. This is the logical OR of `pwm_sm_status_flags_t`.

```
static inline uint16_t PWM_GetFaultStatusFlags(PWM_Type *base,
pwm_fault_protection_channel_t
eFaultProtection)
```

Gets the PWM fault protection status flags.

This examples shows how to check whether the fault protection channel fault input pin 0 set.

```
if((PWM_GetFaultStatusFlags(PWM, kPWM_FaultProtection0) & kPWM_FaultPin0Flag) != 0U)
{
    ...
}
```

Parameters

- base – PWM peripheral base address
- eFaultProtection – PWM fault protection channel number, see `pwm_fault_protection_channel_t`.

Returns

The PWM fault protection channel status flags. This is the logical OR of `_pwm_fault_protection_status_flags`.

```
static inline void PWM_ClearFaultStatusFlags(PWM_Type *base,
                                             pwm_fault_protection_channel_t eFaultProtection,
                                             uint16_t u16Mask)
```

Clears the PWM fault protection status flags according to a provided mask.

This examples shows how to clear the fault protection channel fault 0 flag.

```
PWM_ClearFaultStatusFlags(PWM, kPWM_FaultProtection0, kPWM_Fault0Flag);
```

Note: The `kPWM_FaultPin0ActiveFlag ~ kPWM_FaultPin3ActiveFlag` can't be cleared by software.

Parameters

- base – PWM peripheral base address
- eFaultProtection – PWM fault protection channel number, see `pwm_fault_protection_channel_t`.
- u16Mask – The PWM fault protection status flags to be clear. Logical OR of `_pwm_fault_protection_status_flags`.

FSL_PWM_DRIVER_VERSION

PWM driver version.

enum _pwm_sm_number

The enumeration for PWM submodule number.

Values:

enumerator kPWM_SubModule0
PWM Submodule 0

enumerator kPWM_SubModule1
PWM Submodule 1

enumerator kPWM_SubModule2
PWM Submodule 2

enumerator kPWM_SubModule3
PWM Submodule 3

enum _pwm_sm_enable

The enumeration for PWM submodule enable.

Values:

enumerator kPWM_SubModule0Enable
PWM Submodule 0 enable.

enumerator kPWM_SubModule1Enable
PWM Submodule 1 enable.

enumerator kPWM_SubModule2Enable
PWM Submodule 2 enable.

enumerator kPWM_SubModule3Enable
PWM Submodule 3 enable.

enumerator kPWM_ALLSubModuleEnable

enum _pwm_sm_count_clock_source

The enumeration for PWM submodule clock source.

Values:

enumerator kPWM_ClockSrcBusClock
The IPBus clock is used as the source clock

enumerator kPWM_ClockSrcExternalClock
EXT_CLK is used as the source clock

enumerator kPWM_ClockSrcSubmodule0Clock
Clock of the submodule 0 (AUX_CLK) is used as the source clock

enum _pwm_sm_count_clock_prescaler

The enumeration for PWM submodule prescaler factor selection for clock source.

Values:

enumerator kPWM_ClockPrescaleDivide1
PWM submodule clock frequency = fclk/1

enumerator kPWM_ClockPrescaleDivide2
PWM submodule clock frequency = fclk/2

enumerator kPWM_ClockPrescaleDivide4
PWM submodule clock frequency = fclk/4

enumerator kPWM_ClockPrescaleDivide8
PWM submodule clock frequency = fclk/8

enumerator kPWM_ClockPrescaleDivide16
PWM submodule clock frequency = fclk/16

enumerator kPWM_ClockPrescaleDivide32
PWM submodule clock frequency = fclk/32

enumerator kPWM_ClockPrescaleDivide64
PWM submodule clock frequency = fclk/64

enumerator kPWM_ClockPrescaleDivide128
PWM submodule clock frequency = fclk/128

enum `_pwm_sm_count_init_source`

The enumeration for PWM submodule counter initialization options.

Values:

enumerator `kPWM_InitOnLocalSync`

Local sync causes initialization

enumerator `kPWM_InitOnMasterReload`

Master reload from submodule 0 causes initialization

enumerator `kPWM_InitOnMasterSync`

Master sync from submodule 0 causes initialization

enumerator `kPWM_InitOnExtSync`

EXT_SYNC causes initialization

enum `_pwm_ml2_stretch_count_clock_prescaler`

The enumeration for PWM stretch IPBus clock count prescaler for `mux0_trig/mux1_trig/out0_trig/out1_trig/pwma_trig/pwmb_trig`.

Values:

enumerator `kPWM_StretchIPBusClockPrescaler1`

Stretch count is zero, no stretch.

enumerator `kPWM_StretchIPBusClockPrescaler2`

Stretch `mux0_trig/mux1_trig/out0_trig/out1_trig/pwma_trig/pwmb_trig` for 2 IPBus clock period.

enumerator `kPWM_StretchIPBusClockPrescaler4`

Stretch `mux0_trig/mux1_trig/out0_trig/out1_trig/pwma_trig/pwmb_trig` for 4 IPBus clock period.

enumerator `kPWM_StretchIPBusClockPrescaler8`

Stretch `mux0_trig/mux1_trig/out0_trig/out1_trig/pwma_trig/pwmb_trig` for 8 IPBus clock period.

enum `_pwm_sm_reload_signal_select`

The enumeration for PWM submodule local reload take effect timing.

Values:

enumerator `kPWM_LocalReloadSignal`

The local RELOAD signal is used to reload buffered-registers.

enumerator `kPWM_MasterReloadSignal`

The master RELOAD signal (from submodule 0) is used to reload buffered-registers (should not be used in submodule 0).

enum `_pwm_sm_local_reload_effect_timing`

The enumeration for PWM submodule local reload take effect timing.

Values:

enumerator `kPWM_TakeEffectAtReloadOpportunity`

Buffered-registers reload after one/more reload opportunities, and a load opportunity can generate on a PWM half or/and full cycle.

enumerator `kPWM_TakeEffectImmediately`

Buffered-registers reload with new values as soon as MCTRL[LDOK] bit is set when choose local reload.

enum `_pwm_sm_local_reload_opportunity`

The enumeration for PWM submodule reload opportunities selection under `kPWM_ReloadWithLocalReloadOpportunity`.

Values:

- enumerator `kPWM_LoadEveryOpportunity`
Every PWM submodule reload opportunity
- enumerator `kPWM_LoadEvery2Opportunity`
Every 2 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery3Opportunity`
Every 3 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery4Opportunity`
Every 4 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery5Opportunity`
Every 5 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery6Opportunity`
Every 6 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery7Opportunity`
Every 7 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery8Opportunity`
Every 8 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery9Opportunity`
Every 9 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery10Opportunity`
Every 10 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery11Opportunity`
Every 11 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery12Opportunity`
Every 12 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery13Opportunity`
Every 13 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery14Opportunity`
Every 14 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery15Opportunity`
Every 15 PWM submodule reload opportunities
- enumerator `kPWM_LoadEvery16Opportunity`
Every 16 PWM submodule reload opportunities

enum `_pwm_sm_val_compare_mode`

The enumeration for PWM submodule VALn register compare mode.

Values:

- enumerator `kPWM_CompareOnEqual`
The VALn registers and the PWM counter are compared using an “equal to” method.

enumerator kPWM_CompareOnEqualOrGreater

The VALn registers and the PWM counter are compared using an “equal to or greater than” method.

enum _pwm_sm_val_register

The enumeration for PWM submodule VAL registers.

Values:

enumerator kPWM_VAL0

PWM submodule value register 0.

enumerator kPWM_VAL1

PWM submodule value register 1.

enumerator kPWM_VAL2

PWM submodule value register 2.

enumerator kPWM_VAL3

PWM submodule value register 3.

enumerator kPWM_VAL4

PWM submodule value register 4.

enumerator kPWM_VAL5

PWM submodule value register 5.

enum _pwm_sm_force_signal_select

The enumeration for PWM submodule FORCE_OUT source which can trigger force logic output update.

Values:

enumerator kPWM_LocalSoftwareForce

The local software force signal CTRL2[FORCE] is used to force updates.

enumerator kPWM_MasterSoftwareForce

The master software force signal from submodule 0 is used to force updates.

enumerator kPWM_LocalReloadForce

The local reload signal from this submodule is used to force updates without regard to the state of LDOK.

enumerator kPWM_MasterReloadForce

The master reload signal from submodule 0 is used to force updates if LDOK is set, should not be used in submodule 0.

enumerator kPWM_LocalSyncForce

The local sync (VAL1 match event) signal from this submodule is used to force updates.

enumerator kPWM_MasterSyncForce

The master sync signal from submodule0 is used to force updates.

enumerator kPWM_ExternalForceForce

The external force signal EXT_FORCE, from outside the PWM module causes updates.

enumerator kPWM_ExternalSyncForce

The external sync signal EXT_SYNC, from outside the PWM module causes updates.

enum `_pwm_sm_force_deadtime_source`

The enumeration for PWM submodule force out logic output (PWM23 and PWM45) source, which will transfer to output logic when a FORCE_OUT signal is asserted.

Values:

enumerator `kPWM_GeneratedPwm`

Generated PWM signal is used as the deadtime logic output.

enumerator `kPWM_InvertedGeneratedPwm`

Inverted PWM signal is used as the deadtime logic output.

enumerator `kPWM_SoftwareControlValue`

Software controlled value is used as the deadtime logic output.

enumerator `kPWM_UseExternal`

PWM_EXT_A signal is used as the deadtime logic output.

enum `_pwm_sm_force_software_output_value`

The enumeration for PWM submodule software controlled force out signal value.

Values:

enumerator `kPWM_SoftwareOutputLow`

A logic 0 is supplied to the deadtime generator when chose Software controlled value as output source.

enumerator `kPWM_SoftwareOutputHigh`

A logic 1 is supplied to the deadtime generator when chose Software controlled value as output source.

enum `_pwm_sm_deadtime_logic_mode`

The enumeration for PWM submodule deadtime logic mode, which decide how the deadtime logic process the force logic output signal.

Values:

enumerator `kPWM_Independent`

The PWMA (PWM23) and PWMB (PWM45) signal from force logic transfer to output logic independent.

enumerator `kPWM_IndependentWithDoubleSwitchPwm`

The PWMA (PWM23) and PWMB (PWM45) signals from force logic will XOR first, then the XOR signal transfer to output logic independent.

enumerator `kPWM_IndependentWithSplitDoubleSwitchPwm`

The PWMA (PWM23) and PWMB (PWM45) signals from force_out logic will XOR first, then the XOR signal transfer to output logic independent.

enumerator `kPWM_ComplementaryWithPwmA`

The PWMA (PWM23) signal from force logic will transfer to output logic with complementary mode.

enumerator `kPWM_ComplementaryWithPwmB`

The PWMB (PWM45) signal from force logic will transfer to output logic with complementary mode.

enumerator `kPWM_ComplementaryWithDoubleSwitchPwm`

The PWMA (PWM23) and PWMB (PWM45) signals from force logic will XOR first, then the XOR signal transfer to output logic with complementary mode.

enum `_pwm_sm_fracval_register`

The enumeration for PWM submodule FRACVAL registers.

Values:

enumerator `kPWM_FRACVAL1`

PWM submodule fractional value register 1.

enumerator `kPWM_FRACVAL2`

PWM submodule fractional value register 2.

enumerator `kPWM_FRACVAL3`

PWM submodule fractional value register 3.

enumerator `kPWM_FRACVAL4`

PWM submodule fractional value register 4.

enumerator `kPWM_FRACVAL5`

PWM submodule fractional value register 5.

enum `_pwm_sm_mux_trigger_source`

The enumeration for PWM submodule output logic final trigger output port signal.

Values:

enumerator `kPWM_ActualCompareEvent`

Route the `PWM_OUT_TRIG` signal (OR of `VALx` compare signal) to the mux trigger output port.

enumerator `kPWM_PwmOutput`

Route the PWM output (after polarity/mask/enable control) to the mux trigger output port.

enum `_pwm_sm_pwm_output_on_fault`

The enumeration for PWM submodule output logic PWM output fault status.

Values:

enumerator `kPWM_OutputLowOnFault`

The output is forced to logic 0 state prior to consideration of output polarity/mask/enable control during fault conditions and STOP mode.

enumerator `kPWM_OutputHighOnFault`

The output is forced to logic 1 state prior to consideration of output polarity/mask/enable control during fault conditions and STOP mode.

enumerator `kPWM_OutputTristatedOnFault`

The output status be tristated during fault conditions and STOP mode.

enum `_pwm_sm_pwm_x_signal_select`

The enumeration for PWM submodule output logic PwmX signal input source (before output polarity/mask/enable control).

Values:

enumerator `kPWM_RawPwmX`

The `PWM_X` source is raw `Pwm01_fractional_delay` signal.

enumerator `kPWM_DoubleSwitch`

The `PWM_X` source is `Pwm23_fractional_delay XOR Pwm23_fractional_delay` signal.

enum `_pwm_sm_pwm_out`

The enumeration for PWM submodule PWM output.

Values:

enumerator kPWM_PwmX
The PWM output PWM_X.

enumerator kPWM_PwmB
The PWM output PWM_B.

enumerator kPWM_PwmA
The PWM output PWM_A.

enum _pwm_sm_input_capture_pin
The enumeration for PWM submodule input capture pins.

Values:

enumerator kPWM_InputCapturePwmX
The input capture pin PwmX, need disable PwmX output when enable input capture.

enumerator kPWM_InputCapturePwmA
The input capture pin PwmA, need disable PwmA output when enable input capture.

enumerator kPWM_InputCapturePwmB
The input capture pin PwmB, need disable PwmB output when enable input capture.

enum _pwm_sm_input_capture_source
The enumeration for PWM submodule input capture source.

Values:

enumerator kPWM_RawInput
The capture source is the raw input signal.

enumerator kPWM_InputEdgeCounter
The capture source is edge counter which counts rising and falling edges on the raw input signal.

enum _pwm_sm_input_capture_edge
The enumeration for PWM submodule input capture edge when choose raw input as capture source.

Values:

enumerator kPWM_Noedge
Disabled capture on source falling/falling edge.

enumerator kPWM_FallingEdge
Enable input capture, and capture on source falling edge when chose the raw input signal as capture source.

enumerator kPWM_RisingEdge
Enable input capture, and capture on source rising edge when chose the raw input signal as capture source.

enumerator kPWM_RiseAndFallEdge
Enable input capture, and capture on source rising or falling edge when chose the raw input signal as capture source.

enum _pwm_sm_input_capture_register
The enumeration for PWM submodule input capture value register.

Values:

enumerator kPWM_InpCaptureVal0
Stores the value captured from the submodule counter when the PWM_X circuitry 0 logic capture occurs.

enumerator kPWM_InpCaptureVal1

Stores the value captured from the submodule counter when the PWM_X circuitry 1 logic capture occurs.

enumerator kPWM_InpCaptureVal2

Stores the value captured from the submodule counter when the PWM_A circuitry 0 logic capture occurs.

enumerator kPWM_InpCaptureVal3

Stores the value captured from the submodule counter when the PWM_A circuitry 1 logic capture occurs.

enumerator kPWM_InpCaptureVal4

Stores the value captured from the submodule counter when the PWM_B circuitry 0 logic capture occurs.

enumerator kPWM_InpCaptureVal5

Stores the value captured from the submodule counter when the PWM_B circuitry 1 logic capture occurs.

enum _pwm_sm_input_capture_filter_count

The enumeration for input filter count Represent the number of consecutive samples that must agree prior to the input filter accepting an input transition.

Values:

enumerator kPWM_InputCaptureFilterCount3Samples
3 samples.

enumerator kPWM_InputCaptureFilterCount4Samples
4 samples.

enumerator kPWM_InputCaptureFilterCount5Samples
5 samples.

enumerator kPWM_InputCaptureFilterCount6Samples
6 samples.

enumerator kPWM_InputCaptureFilterCount7Samples
7 samples.

enumerator kPWM_InputCaptureFilterCount8Samples
8 samples.

enumerator kPWM_InputCaptureFilterCount9Samples
9 samples.

enumerator kPWM_InputCaptureFilterCount10Samples
10 samples.

enum _pwm_sm_capture_dma_source

The enumeration for the source which can trigger the DMA read requests for the capture FIFOs.

Values:

enumerator kPWM_FIFOWatermarksORDma
Selected FIFO watermarks are OR'ed together to sets the read DMA request.

enumerator kPWM_FIFOWatermarksANDDma
Selected FIFO watermarks are AND'ed together to sets the read DMA request.

enumerator kPWM_LocalSyncDma

A local sync (VAL1 match event) sets the read DMA request.

enumerator kPWM_LocalReloadDma

A local reload (STS[RF] being set) sets the read DMA request.

enum _pwm_sm_interrupt_enable

The enumeration for PWM submodule interrupt enable.

Values:

enumerator kPWM_CompareVal0InterruptEnable

PWM submodule VAL0 compare interrupt.

enumerator kPWM_CompareVal1InterruptEnable

PWM submodule VAL1 compare interrupt.

enumerator kPWM_CompareVal2InterruptEnable

PWM submodule VAL2 compare interrupt.

enumerator kPWM_CompareVal3InterruptEnable

PWM submodule VAL3 compare interrupt.

enumerator kPWM_CompareVal4InterruptEnable

PWM submodule VAL4 compare interrupt.

enumerator kPWM_CompareVal5InterruptEnable

PWM submodule VAL5 compare interrupt.

enumerator kPWM_CaptureX0InterruptEnable

PWM submodule capture X0 interrupt.

enumerator kPWM_CaptureX1InterruptEnable

PWM submodule capture X1 interrupt.

enumerator kPWM_CaptureB0InterruptEnable

PWM submodule capture B0 interrupt.

enumerator kPWM_CaptureB1InterruptEnable

PWM submodule capture B1 interrupt.

enumerator kPWM_CaptureA0InterruptEnable

PWM submodule capture A0 interrupt.

enumerator kPWM_CaptureA1InterruptEnable

PWM submodule capture A1 interrupt.

enumerator kPWM_ReloadInterruptEnable

PWM submodule reload interrupt.

enumerator kPWM_ReloadErrorInterruptEnable

PWM submodule reload error interrupt.

enumerator kPWM_ALLSubModuleInterruptEnable

enum _pwm_sm_status_flags

The enumeration for PWM submodule status flags.

Values:

enumerator kPWM_CompareVal0Flag

PWM submodule VAL0 compare flag.

enumerator kPWM_CompareVal1Flag
PWM submodule VAL1 compare flag.

enumerator kPWM_CompareVal2Flag
PWM submodule VAL2 compare flag.

enumerator kPWM_CompareVal3Flag
PWM submodule VAL3 compare flag.

enumerator kPWM_CompareVal4Flag
PWM submodule VAL4 compare flag.

enumerator kPWM_CompareVal5Flag
PWM submodule VAL5 compare flag.

enumerator kPWM_CaptureX0Flag
PWM submodule capture X0 flag.

enumerator kPWM_CaptureX1Flag
PWM submodule capture X1 flag.

enumerator kPWM_CaptureB0Flag
PWM submodule capture B0 flag.

enumerator kPWM_CaptureB1Flag
PWM submodule capture B1 flag.

enumerator kPWM_CaptureA0Flag
PWM submodule capture A0 flag.

enumerator kPWM_CaptureA1Flag
PWM submodule capture A1 flag.

enumerator kPWM_ReloadFlag
PWM submodule reload flag.

enumerator kPWM_ReloadErrorFlag
PWM submodule reload error flag.

enumerator kPWM_RegUpdatedFlag
PWM submodule registers updated flag.

enumerator kPWM_ALLSMStatusFlags

enum _pwm_sm_typical_output_mode

The enumeration for some PWM submodule PWM_A/B typical output mode.

Values:

enumerator kPWM_SignedCenterAligned
Center-aligned PWM with signed compare value.

enumerator kPWM_CenterAligned
Center-aligned PWM with unsigned compare value.

enumerator kPWM_SignedEdgeAligned
Edge-aligned PWM with signed compare value.

enumerator kPWM_EdgeAligned
Edge-aligned PWM with signed compare value.

enum `_pwm_fault_protection_channel`

The enumeration for PWM fault protection channel number.

Values:

enumerator `kPWM_FaultProtection0`
PWM fault protection channel 0

enum `_pwm_fault_protection_interrupt_enable`

The enumeration for PWM module fault protection channel interrupt enable.

Values:

enumerator `kPWM_Fault0InterruptEnable`
Fault protection channel fault 0 interrupt

enumerator `kPWM_Fault1InterruptEnable`
Fault protection channel fault 1 interrupt

enumerator `kPWM_Fault2InterruptEnable`
Fault protection channel fault 2 interrupt

enumerator `kPWM_Fault3InterruptEnable`
Fault protection channel fault 3 interrupt

enumerator `kPWM_ALLfaultInterruptEnable`

enum `_pwm_fault_protection_status_flags`

The enumeration for PWM module fault protection status flags.

Values:

enumerator `kPWM_Fault0Flag`
Fault protection channel fault 0 flag, set within two CPU cycles after a transition to active on the fault input pin 0.

enumerator `kPWM_Fault1Flag`
Fault protection channel fault 1 flag, set within two CPU cycles after a transition to active on the fault input pin 1.

enumerator `kPWM_Fault2Flag`
Fault protection channel fault 2 flag, set within two CPU cycles after a transition to active on the fault input pin 2.

enumerator `kPWM_Fault3Flag`
Fault protection channel fault 3 flag, set within two CPU cycles after a transition to active on the fault input pin 3.

enumerator `kPWM_FaultPin0ActiveFlag`
Fault protection channel fault input pin 0 active flag.

enumerator `kPWM_FaultPin1ActiveFlag`
Fault protection channel fault input pin 1 active flag.

enumerator `kPWM_FaultPin2ActiveFlag`
Fault protection channel fault input pin 2 active flag.

enumerator `kPWM_FaultPin3ActiveFlag`
Fault protection channel fault input pin 3 active flag.

enumerator `kPWM_ALLFaultStatusFlags`

enum `_pwm_fault_active_level`

The enumeration for PWM fault protection channel number.

Values:

enumerator `kPWM_Logic0`

A logic 0 on the fault input indicates a fault condition.

enumerator `kPWM_Logic1`

A logic 1 on the fault input indicates a fault condition.

typedef enum `_pwm_sm_number` `pwm_sm_number_t`

The enumeration for PWM submodule number.

typedef enum `_pwm_sm_count_clock_source` `pwm_sm_count_clock_source_t`

The enumeration for PWM submodule clock source.

typedef enum `_pwm_sm_count_clock_prescaler` `pwm_sm_count_clock_prescaler_t`

The enumeration for PWM submodule prescaler factor selection for clock source.

typedef enum `_pwm_sm_count_init_source` `pwm_sm_count_init_source_t`

The enumeration for PWM submodule counter initialization options.

typedef enum `_pwm_ml2_stretch_count_clock_prescaler`

`pwm_ml2_stretch_count_clock_prescaler_t`

The enumeration for PWM stretch IPBus clock count prescaler for mux0_trig/mux1_trig/out0_trig/out1_trig/pwma_trig/pwmb_trig.

typedef struct `_pwm_sm_counter_config` `pwm_sm_counter_config_t`

The structure for configuring PWM submodule counter logic.

typedef enum `_pwm_sm_reload_signal_select` `pwm_sm_reload_signal_select_t`

The enumeration for PWM submodule local reload take effect timing.

typedef enum `_pwm_sm_local_reload_effect_timing` `pwm_sm_local_reload_effect_timing_t`

The enumeration for PWM submodule local reload take effect timing.

typedef enum `_pwm_sm_local_reload_opportunity` `pwm_sm_local_reload_opportunity_t`

The enumeration for PWM submodule reload opportunities selection under `kPWM_ReloadWithLocalReloadOpportunity`.

typedef struct `_pwm_sm_reload_logic_config` `pwm_sm_reload_logic_config_t`

The structure for configuring PWM submodule reload logic.

typedef enum `_pwm_sm_val_compare_mode` `pwm_sm_val_compare_mode_t`

The enumeration for PWM submodule VALn register compare mode.

typedef enum `_pwm_sm_val_register` `pwm_sm_val_register_t`

The enumeration for PWM submodule VAL registers.

typedef struct `_pwm_sm_value_register_config` `pwm_sm_value_register_config_t`

The structure for configuring PWM submodule value registers.

typedef enum `_pwm_sm_force_signal_select` `pwm_sm_force_signal_select_t`

The enumeration for PWM submodule FORCE_OUT source which can trigger force logic output update.

typedef enum `_pwm_sm_force_deadtime_source` `pwm_sm_force_deadtime_source_t`

The enumeration for PWM submodule force out logic output (PWM23 and PWM45) source, which will transfer to output logic when a FORCE_OUT signal is asserted.

typedef enum `_pwm_sm_force_software_output_value` `pwm_sm_force_software_output_value_t`

The enumeration for PWM submodule software controlled force out signal value.

```
typedef struct _pwm_sm_force_logic_config pwm_sm_force_logic_config_t
```

The structure for configuring PWM submodule force logic.

```
typedef enum _pwm_sm_deadtime_logic_mode pwm_sm_deadtime_logic_mode_t
```

The enumeration for PWM submodule deadtime logic mode, which decide how the dead-time logic process the force logic output signal.

```
typedef struct _pwm_sm_deadtime_value pwm_sm_deadtime_value_t
```

The structure of the inserted dead time value, applies only to KPWM_Complementary mode.

```
typedef struct _pwm_sm_deadtime_logic_config pwm_sm_deadtime_logic_config_t
```

The structure for configuring PWM submodule force out logic, works on the deadtime logic output.

```
typedef enum _pwm_sm_fracval_register pwm_sm_fracval_register_t
```

The enumeration for PWM submodule FRACVAL registers.

```
typedef struct _pwm_sm_fractional_delay_logic_config pwm_sm_fractional_delay_logic_config_t
```

The structure for configuring PWM submodule fractional delay logic, works on the dead-time logic output.

```
typedef enum _pwm_sm_mux_trigger_source pwm_sm_mux_trigger_source_t
```

The enumeration for PWM submodule output logic final trigger output port signal.

```
typedef enum _pwm_sm_pwm_output_on_fault pwm_sm_pwm_output_on_fault_t
```

The enumeration for PWM submodule output logic PWM output fault status.

```
typedef enum _pwm_sm_pwm_x_signal_select pwm_sm_pwm_x_signal_select_t
```

The enumeration for PWM submodule output logic PwmX signal input source (before output polarity/mask/enable control).

```
typedef enum _pwm_sm_pwm_out pwm_sm_pwm_out_t
```

The enumeration for PWM submodule PWM output.

```
typedef struct _pwm_sm_output_logic_config_t pwm_sm_output_logic_config_t
```

The structure for configuring PWM submodule output logic.

```
typedef enum _pwm_sm_input_capture_pin pwm_sm_input_capture_pin_t
```

The enumeration for PWM submodule input capture pins.

```
typedef enum _pwm_sm_input_capture_source pwm_sm_input_capture_source_t
```

The enumeration for PWM submodule input capture source.

```
typedef enum _pwm_sm_input_capture_edge pwm_sm_input_capture_edge_t
```

The enumeration for PWM submodule input capture edge when choose raw input as capture source.

```
typedef enum _pwm_sm_input_capture_register pwm_sm_input_capture_register_t
```

The enumeration for PWM submodule input capture value register.

```
typedef enum _pwm_sm_input_capture_filter_count pwm_sm_input_capture_filter_count_t
```

The enumeration for input filter count Represent the number of consecutive samples that must agree prior to the input filter accepting an input transition.

```
typedef struct _pwm_sm_input_capture_config pwm_sm_input_capture_config_t
```

The structure for configuring PWM submodule input capture logic.

Note: When choosing kPWM_InputEdgeCounter as circuit 0/1 capture source, the eCircuit0CaptureEdge and eCircuit1CaptureEdge selected trigger edge will be ignored, but still

need place a value other than kPWM_Noedge in either or both of the eCaptureCircuit0 and/or CaptureCircuit1 fields in order to enable one or both of the capture registers.

typedef enum *_pwm_sm_capture_dma_source* pwm_sm_capture_dma_source_t

The enumeration for the source which can trigger the DMA read requests for the capture FIFOs.

typedef struct *_pwm_sm_capture_dma_config* pwm_sm_capture_dma_config_t

The structure for configuring PWM submodule read capture DMA.

typedef struct *_pwm_sm_dma_config* pwm_sm_dma_config_t

The structure for configuring PWM submodule DMA.

typedef struct *_pwm_sm_fault_input_mapping* pwm_sm_fault_input_mapping_t

The enumeration for PWM submodule output fault enable mask for one fault protection channel.

The structure for configuring PWM submodule fault input disable mapping.

Note: The channel 0 input 0 and channel 1 input 0 are different pins.

Note: Each PWM output can be mapping anyone or more fault inputs. The mapped fault protection channel inputs can disable PWM output.

typedef enum *_pwm_sm_status_flags* pwm_sm_status_flags_t

The enumeration for PWM submodule status flags.

typedef enum *_pwm_sm_typical_output_mode* pwm_sm_typical_output_mode_t

The enumeration for some PWM submodule PWM_A/B typical output mode.

typedef struct *_pwm_sm_config* pwm_sm_config_t

PWM submodule config structure.

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

typedef enum *_pwm_fault_protection_channel* pwm_fault_protection_channel_t

The enumeration for PWM fault protection channel number.

typedef enum *_pwm_fault_active_level* pwm_fault_input_active_level_t

The enumeration for PWM fault protection channel number.

typedef struct *_pwm_fault_protection_input_config* pwm_fault_protection_input_config_t

typedef struct *_pwm_fault_protection_config* pwm_fault_protection_config_t

The structure for configuring PWM fault protection channel, a PWM module can have multiple fault protection channels, PWM sub-module can choose to mapping any one or more fault input from fault protection channels.

typedef struct *_pwm_config* pwm_config_t

PWM module config structure which contain submodule config structure pointers and fault protection filter config structure pointers.

Note: Need use submodule structure address to init the structure pointers, when the submodule or fault protection structure pointers is NULL, it will be ignored by PWM_Init API. This can save stack space when only one or two submodules are used.

```
struct __pwm_sm_counter_config
```

```
#include <fsl_pwm.h> The structure for configuring PWM submodule counter logic.
```

Public Members

```
pwm_sm_count_clock_source_t eCountClockSource
```

Configures PWM submodule counter clock source.

```
pwm_sm_count_clock_prescaler_t eCountClockPrescaler
```

Configures PWM submodule counter clock source prescaler.

```
pwm_sm_count_init_source_t eCountInitSource
```

Configures PWM submodule counter initial source.

```
bool bEnableForceInitial
```

Enable force-controlled initialization. The assert FORCE_OUT signal can to initialize the counter without regard to the selected initial source.

```
uint16_t u16PhaseDelayValue
```

Defines the delay from the master sync signal of submodule 0 to this submodule counter (the unit of delay is the PWM clock cycle), only works when chose kPWM_InitOnMasterSync as initial source.

```
struct __pwm_sm_reload_logic_config
```

```
#include <fsl_pwm.h> The structure for configuring PWM submodule reload logic.
```

Public Members

```
pwm_sm_reload_signal_select_t eReloadSignalSelect
```

Configures PWM submodule RELOAD signal source to be local reload signal or master reload signal.

```
pwm_sm_local_reload_effect_timing_t eLocalReloadEffectTime
```

Configures PWM submodule local reload signal effective timing when choose it as RELOAD signal source.

```
bool bEnableFullCycleReloadOpportunity
```

Enable generate a reload opportunity on PWM half cycle (count from INIT value to VAL0).

```
bool bEnableHalfCycleReloadOpportunity
```

Enable generate a reload opportunity on PWM full cycle (count from INIT value to VAL1).

```
pwm_sm_local_reload_opportunity_t eLocalReloadOpportunity
```

Configures PWM submodule reload frequency when using local reload opportunities mode .

```
struct __pwm_sm_value_register_config
```

```
#include <fsl_pwm.h> The structure for configuring PWM submodule value registers.
```

Public Members

`uint16_t` `u16CounterInitialValue`

Configures PWM submodule counter initial value.

`uint16_t` `u16ValRegister0`

Configures PWM submodule value register 0 (VAL0) value.

`uint16_t` `u16ValRegister1`

Configures PWM submodule value register 1 (VAL1) value.

`uint16_t` `u16ValRegister2`

Configures PWM submodule value register 2 (VAL2) value.

`uint16_t` `u16ValRegister3`

Configures PWM submodule value register 3 (VAL3) value.

`uint16_t` `u16ValRegister4`

Configures PWM submodule value register 4 (VAL4) value.

`uint16_t` `u16ValRegister5`

Configures PWM submodule value register 5 (VAL5) value.

`struct` `_pwm_sm_force_logic_config`

#include <fsl_pwm.h> The structure for configuring PWM submodule force logic.

Public Members

`uint8_t` `bitPWM23OutputInitialVaule`

Configures PWM submodule compare output x (PwmX) initial value.

`uint8_t` `bitPWM45OutputInitialVaule`

Configures PWM submodule compare output A (PwmA) initial value.

`uint8_t` `bitPWMXOutputInitialVaule`

Configures PWM submodule compare output B (PwmB) initial value.

`pwm_sm_force_signal_select_t` `eForceSignalSelect`

Configures PWM submodule force out select update trigger source.

`pwm_sm_force_software_output_value_t` `eSoftOutputFor23`

Configures PWM submodule force out PwmA value when select software as output source.

`pwm_sm_force_software_output_value_t` `eSoftOutputFor45`

Configures PWM submodule force out PwmB value when select software as output source.

`pwm_sm_force_deadtime_source_t` `eForceOutput23`

Configures the source of Pwm23, which will be force to deadtime logic.

`pwm_sm_force_deadtime_source_t` `eForceOutput45`

Configures the source of Pwm45, which will be force to deadtime logic.

`struct` `_pwm_sm_deadtime_value`

#include <fsl_pwm.h> The structure of the inserted dead time value, applies only to KPWM_Compplementaryxxx mode.

`struct` `_pwm_sm_deadtime_logic_config`

#include <fsl_pwm.h> The structure for configuring PWM submodule force out logic, works on the deadtime logic output.

Public Members

`pwm_sm_deadtime_logic_mode_t` eMode

The mode in which Deadtime logic process the force logic output signal.

`pwm_sm_deadtime_value_t` sDeadTimeValue0

Control the deadtime during 0 to 1 transitions of the PWM_23 output (assuming normal polarity). When disable fractional delays, the maximum value is 0x7FF which represents 2047 cycles of IP bus cycles. When enable fractional delays, the maximum value is 0xFFFF which represents 2047 31/32 cycles cycles of IP bus cycles.

`pwm_sm_deadtime_value_t` sDeadTimeValue1

Control the deadtime during 0 to 1 transitions of the PWM_45 output (assuming normal polarity). When disable fractional delays, the maximum value is 0x7FF which represents 2047 cycles of IP bus cycles. When enable fractional delays, the maximum value is 0xFFFF which represents 2047 31/32 cycles cycles of IP bus cycles.

`struct _pwm_sm_fractional_delay_logic_config`

`#include <fsl_pwm.h>` The structure for configuring PWM submodule fractional delay logic, works on the deadtime logic output.

Public Members

`uint8_t` bitsFracValue1

Configures PWM submodule compare register VAL1 fractional delay value, the unit is 1/32 IP bus clock.

`bool` bEnableVal1FractionalDelay

Enable the fractional delay feature of bitsFracValue1.

`uint8_t` bitsFracValue2

Configures PWM submodule compare register VAL2 fractional delay value, the unit is 1/32 IP bus clock.

`uint8_t` bitsFracValue3

Configures PWM submodule compare register VAL3 fractional delay value, the unit is 1/32 IP bus clock.

`bool` bEnableVal23FractionalDelay

Enable the fractional delay feature of bitsFracValue2 and bitsFracValue3.

`uint8_t` bitsFracValue4

Configures PWM submodule compare register VAL4 fractional delay value, the unit is 1/32 IP bus clock.

`uint8_t` bitsFracValue5

Configures PWM submodule compare register VAL5 fractional delay value, the unit is 1/32 IP bus clock.

`bool` bEnableVal45FractionalDelay

Enable the fractional delay feature of bitsFracValue4 and bitsFracValue5.

`struct _pwm_sm_output_logic_config_t`

`#include <fsl_pwm.h>` The structure for configuring PWM submodule output logic.

Public Members

`bool` bVal0TriggerEnable

Enable VAL0 register compare event trigger.

`bool bVal1TriggerEnable`
Enable VAL1 register compare event trigger.

`bool bVal2TriggerEnable`
Enable VAL2 register compare event trigger.

`bool bVal3TriggerEnable`
Enable VAL3 register compare event trigger.

`bool bVal4TriggerEnable`
Enable VAL4 register compare event trigger.

`bool bVal5TriggerEnable`
Enable VAL5 register compare event trigger.

`bool bEnableTriggerPostScaler`
True : Trigger outputs are generated only during the final PWM period prior to a reload opportunity, false : Trigger outputs are generated during every PWM period. Configures PWM submodule mux trigger output signal 0 source.

`pwm_sm_mux_trigger_source_t eMuxTrigger0`
Configures PWM submodule mux trigger output signal 1 source.

`pwm_sm_mux_trigger_source_t eMuxTrigger1`
Configures PWM submodule PWM_X output source (before polarity/mask/enable control).

`bool bInvertPwmXOutput`
True : invert PWM_X output, false : no invert PWM_X output.

`bool bInvertPwmaOutput`
True : invert PWM_A output, false : no invert PWM_A output.

`bool bInvertPwmbOutput`
True : invert PWM_B output, false : no invert PWM_B output.

`bool bMaskPwmXOutput`
True : PWM_X output masked, false : PWM_X output normal. Mask bit is buffered, and take effect until FORCE_OUT event or software update command.

`bool bMaskPwmaOutput`
True : PWM_A output masked, false : PWM_A output normal. Mask bit is buffered, and take effect until FORCE_OUT event or software update command.

`bool bMaskPwmbOutput`
True : PWM_B output masked, false : PWM_B output normal. Mask bit is buffered, and take effect until FORCE_OUT event or software update command.

`bool bEnablePwmXOutput`
True : Enable PWM_X output. false : PWM_X is disabled and output is tristated.

`bool bEnablePwmaOutput`
True : Enable PWM_A output. false : PWM_A is disabled and output is tristated.

`bool bEnablePwmbOutput`
True : Enable PWM_B output. false : PWM_B is disabled and output is tristated. Configures PWM submodule PWM_X output during fault status (only works when fault status enable).

`pwm_sm_pwm_output_on_fault_t ePwmXFaultState`
Configures PWM submodule PWM_A output during fault status (only works when fault status enable).

pwm_sm_pwm_output_on_fault_t ePwmaFaultState

Configures PWM submodule PWM_B output during fault status (only works when fault status enable).

struct *_pwm_sm_input_capture_config*

#include <fsl_pwm.h> The structure for configuring PWM submodule input capture logic.

Note: When choosing kPWM_InputEdgeCounter as circuit 0/1 capture source, the eCircuit0CaptureEdge and eCircuit1CaptureEdge selected trigger edge will be ignored, but still need place a value other than kPWM_Noedge in either or both of the eCaptureCircuit0 and/or CaptureCircuit1 fields in order to enable one or both of the capture registers.

Public Members

bool bEnableInputCapture

True: enable the input capture process, false : disable the input capture process.

pwm_sm_input_capture_source_t eInCaptureSource

Configures capture circuit 0/1 input source

pwm_sm_input_capture_edge_t eCircuit0CaptureEdge

Configures which edge causes a capture for capture circuit 0 , will be ignore when use edge counter as capture source.

pwm_sm_input_capture_edge_t eCircuit1CaptureEdge

Configures which edge causes a capture for capture circuit 1 , will be ignore when use edge counter as capture source.

bool bEnableOneShotCapture

True: Enable one-shot capture mode, the bEnableInputCapture will self-cleared when one or more of the enabled capture circuits has had a capture event; false: Capture circuit 0/1 will perform capture continue;

uint8_t bitsCaptureFifoWatermark

Watermark level for circuit 0/1 capture FIFO. The capture flags in the status register will set if the word count in the circuit 0/1 capture FIFO is greater than this watermark level

uint8_t u8EdgeCounterCompareValue

Edge counter compare value, used only if edge counter is used as capture circuit 0/1 input source

uint8_t u8FilterPeriod

Sampling period (in IPBus clock cycles) of the input filter, set to 0 to bypass the filter.

pwm_sm_input_capture_filter_count_t eFilterCount

Filter sample count.

struct *_pwm_sm_capture_dma_config*

#include <fsl_pwm.h> The structure for configuring PWM submodule read capture DMA.

Public Members

bool bEnableCaptureDMA

Enables DMA read requests for the Capture FIFOs.

pwm_sm_capture_dma_source_t eCaptureDMASource

Select the source to enables DMA read requests for the Capture FIFOs. Will be ignored when bEnableCaptureDMA be false.

struct *_pwm_sm_dma_config*

#include <fsl_pwm.h> The structure for configuring PWM submodule DMA.

Public Members

bool bEnableWriteValDMA

STS[RF] set enables DMA write requests for VALx and FRACVALx registers.

bool bEnableReadCaptureX0DMA

STS[CFX0] set enables DMA read requests for Capture X0 FIFO. And X0 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

bool bEnableReadCaptureX1DMA

STS[CFX1] set enables DMA read requests for Capture X1 FIFO. And X1 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

bool bEnableReadCaptureA0DMA

STS[CFA0] set enables DMA read requests for Capture A0 FIFO. And A0 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

bool bEnableReadCaptureA1DMA

STS[CFA1] set enables DMA read requests for Capture A1 FIFO. And A1 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

bool bEnableReadCaptureB0DMA

STS[CFB0] set enables DMA read requests for Capture B0 FIFO. And B0 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

bool bEnableReadCaptureB1DMA

STS[CFB1] set enables DMA read requests for Capture B1 FIFO. And B1 FIFO watermark is selected for sCaptureDma *pwm_sm_capture_dma_config_t*.

pwm_sm_capture_dma_config_t sCaptureDma

DMA read requests for the capture FIFOs configure.

struct *_pwm_sm_fault_input_mapping*

#include <fsl_pwm.h> The enumeration for PWM submodule output fault enable mask for one fault protection channel.

The structure for configuring PWM submodule fault input disable mapping.

Note: The channel 0 input 0 and channel 1 input 0 are different pins.

Note: Each PWM output can be mapping anyone or more fault inputs. The mapped fault protection channel inputs can disable PWM output.

Public Members

bool bFaultInput0Mapping

Mapping fault input 0 (from fault protection channel 0) to PWM output.

`bool bFaultInput1Mapping`
Mapping fault input 1 (from fault protection channel 0) to PWM output.

`bool bFaultInput2Mapping`
Mapping fault input 2 (from fault protection channel 0) to PWM output.

`bool bFaultInput3Mapping`
Mapping fault input 3 (from fault protection channel 0) to PWM output.

`bool bFaultInput4Mapping`
Mapping fault input 4 (from fault protection channel 1) to PWM output.

`bool bFaultInput5Mapping`
Mapping fault input 5 (from fault protection channel 1) to PWM output.

`bool bFaultInput6Mapping`
Mapping fault input 6 (from fault protection channel 1) to PWM output.

`bool bFaultInput7Mapping`
Mapping fault input 7 (from fault protection channel 1) to PWM output.

`struct _pwm_sm_config`

#include <fsl_pwm.h> PWM submodule config structure.

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

Public Members

`bool enableDebugMode`
true: PWM continues to run in debug mode; false: PWM is paused in debug mode.

`bool enableWaitMode`
true: PWM continues to run in WAIT mode; false: PWM is paused in WAIT mode.

`bool enableRun`
true: PWM submodule is enabled; false: PWM submodule is disabled. Configures submodule value registers compare mode, only can be written one time.

pwm_sm_counter_config_t `sCounterConfig`
Submodule counter logic config.

pwm_sm_reload_logic_config_t `sReloadConfig`
Submodule reload control logic config.

pwm_sm_value_register_config_t `sValRegisterConfig`
Submodule value registers config.

pwm_sm_force_logic_config_t `sForceConfig`
Submodule force out logic config.

pwm_sm_deadtime_logic_config_t `sDeadTimeConfig`
Submodule deadtime logic config.

pwm_sm_fractional_delay_logic_config_t `sFracDelayConfig`
Submodule fractional logic config.

pwm_sm_output_logic_config_t `sOutputConfig`
Submodule output logic config.

pwm_sm_input_capture_config_t sInCaptureConfig[3]

Submodule input capture config for PWM_X/A/B pins.

pwm_sm_dma_config_t sDMAConfig

Submodule DMA config. PWM_X output fault input mapping, determines which fault inputs can disable PWM_X output.

pwm_sm_fault_input_mapping_t sPwmXFaultInputMapping

PWM_A output fault input mapping, determines which fault inputs can disable PWM_A output.

pwm_sm_fault_input_mapping_t sPwmAFaultInputMapping

PWM_B output fault input mapping, determines which fault inputs can disable PWM_B output.

pwm_sm_fault_input_mapping_t sPwmBFaultInputMapping

Submodule interrupt enable mask, logic OR of `_pwm_sm_interrupt_enable`.

pwm_ml2_stretch_count_clock_prescaler_t eStrBusClock

PWM stretch IPBus clock count prescaler.

struct `_pwm_fault_protection_input_config`

`#include <fsl_pwm.h>`

Public Members

pwm_fault_input_active_level_t eFaultActiveLevel

Select the active logic level of the fault input.

bool `bEnableAutoFaultClear`

True : Enable automatic fault clearing, fault recovery (PWM outputs can re-enable) occurs when FSTS[FFPINx] is clear , false : Use manual fault clearing, fault recovery (PWM outputs can re-enable) occurs when FSTS[FFLAGx] is manual clear (and FSTS[FFPINx] is clear).

bool `bEnableManualFaultClearSafeMode`

True : fault recovery (PWM outputs can re-enable) occurs when FSTS[FFLAGx] is manual clear and FSTS[FFPINx] is clear, false : fault recovery (PWM outputs can re-enable) occurs when FSTS[FFLAGx] is manual clear.

bool `bEnableFaultFullCycleRecovery`

Enable full cycle fault recovery, which make PWM outputs are re-enabled at the start of a half cycle after fault recovery occurs.

bool `bEnableFaultHalfCycleRecovery`

Enable half cycle fault recovery, which make PWM outputs are re-enabled at the start of a half cycle after fault recovery occurs.

bool `bEnableFaultNoCombinationalPath`

True : The fault inputs are combined with the filtered and latched fault signals to disable the PWM outputs , false : the filtered and latched fault signals are used to disable the PWM outputs.

bool `bEnableFaultInterrupt`

Enable the fault input interrupt.

struct `_pwm_fault_protection_config`

`#include <fsl_pwm.h>` The structure for configuring PWM fault protection channel, a PWM module can have multiple fault protection channels, PWM sub-module can choose to map any one or more fault input from fault protection channels.

Public Members

`bool bEnableFaultGlitchStretch`

Fault Glitch Stretch Enable: A logic 1 means that input fault signals will be stretched to at least 2 IPBus clock cycles.

`uint8_t bitsFaultFilterCount`

Configures PWM fault protection channel fault filter count.

`uint8_t u8FaultFilterPeriod`

Configures PWM fault protection channel fault filter period, value of 0 will bypass the filter.

`struct __pwm_config`

#include <fsl_pwm.h> PWM module config structure which contain submodule config structure pointers and fault protection filter config structure pointers.

Note: Need use submodule structure address to init the structure pointers, when the submodule or fault protection structure pointers is NULL, it will be ignored by PWM_Init API. This can save stack space when only one or two submodules are used.

Public Members

`pwm_sm_config_t *psPwmSubmoduleConfig[1]`

< PWM submodule config. PWM fault protection channel config, will take effect for all submodules.

2.56 The Driver Change Log

2.57 eFlexPWM Peripheral and Driver Overview

2.58 QSCI: Queued Serial Communications Interface Driver

`void QSCI_GetDefaultConfig(qsci_config_t *psConfig, uint32_t u32BaudRateBps, uint32_t u32SrcClockHz)`

Sets the QSCI configuration structure to default values.

The purpose of this API is to initialize the configuration structure to default value for QSCI_Init to use. Use the unchanged structure in QSCI_Init or modify the structure before calling QSCI_Init. This is an example:

```
qsci_config_t sConfig;
QSCI_GetDefaultConfig(&sConfig, 115200, 12000000U);
QSCI_Init(QSCI0, &config);
```

Parameters

- `psConfig` – Pointer to configuration structure.
- `u32BaudRateBps` – Baudrate setting.
- `u32SrcClockHz` – The clock source frequency for QSCI module.

`status_t QSCI_Init(QSCI_Type *base, qsci_config_t *psConfig)`

Initializes the QSCI instance with a user configuration structure.

This function configures the QSCI module with the customized settings. User can configure the configuration structure manually or get the default configuration by using the `QSCI_GetDefaultConfig` function. The example below shows how to use this API to configure QSCI.

```
qsci_config_t sConfig;
QSCI_GetDefaultConfig(&sConfig, 115200, 12000000U);
QSCI_Init(QSCIO, &sConfig);
```

Parameters

- `base` – QSCI peripheral base address.
- `psConfig` – Pointer to the user-defined configuration structure.

Return values

- `kStatus_QSCI_BaudrateNotSupport` – Baudrate is not supported in the current clock source.
- `kStatus_Success` – Set baudrate succeeded.

`void QSCI_Deinit(QSCI_Type *base)`

Deinitializes a QSCI instance.

This function waits for transmitting complete, then disables TX and RX.

Parameters

- `base` – QSCI peripheral base address.

`static inline uint16_t QSCI_GetStatusFlags(QSCI_Type *base)`

Gets QSCI hardware status flags.

Parameters

- `base` – QSCI peripheral base address.

Returns

QSCI status flags, can be a single flag or several flags in `_qsci_status_flags` combined by OR.

`void QSCI_ClearStatusFlags(QSCI_Type *base, uint16_t u16StatusFlags)`

Clears QSCI status flags.

This function clears QSCI status flags. Members in `kQSCI_Group0Flags` can't be cleared by this function, they are cleared or set by hardware.

Parameters

- `base` – QSCI peripheral base address.
- `u16StatusFlags` – The status flag mask, can be a single flag or several flags in `_qsci_status_flags` combined by OR.

`void QSCI_EnableInterrupts(QSCI_Type *base, uint8_t u8Interrupts)`

Enables QSCI interrupts according to the provided mask.

This function enables the QSCI interrupts according to the provided mask. The mask is a logical OR of enumeration members in `_qsci_interrupt_enable`.

Parameters

- `base` – QSCI peripheral base address.

- `u8Interrupts` – The interrupt source mask, can be a single source or several sources in `_qsci_interrupt_enable` combined by OR.

`void QSCI_DisableInterrupts(QSCI_Type *base, uint8_t u8Interrupts)`

Disables QSCI interrupts according to the provided mask.

This function disables the QSCI interrupts according to the provided mask. The mask is a logical OR of enumeration members in `_qsci_interrupt_enable`.

Parameters

- `base` – QSCI peripheral base address.
- `u8Interrupts` – The interrupt source mask, can be a single source or several sources in `_qsci_interrupt_enable` combined by OR.

`uint8_t QSCI_GetEnabledInterrupts(QSCI_Type *base)`

Gets the enabled QSCI interrupts.

This function gets the enabled QSCI interrupts. The enabled interrupts are returned as the logical OR value of the enumerators `_qsci_interrupt_enable`.

Parameters

- `base` – QSCI peripheral base address.

Returns

The interrupt source mask, can be a single source or several sources in `_qsci_interrupt_enable` combined by OR.

`static inline void QSCI_Reset(QSCI_Type *base)`

Sets the QSCI register value to reset value.

Parameters

- `base` – QSCI peripheral base address.

`static inline void QSCI_EnableTx(QSCI_Type *base, bool bEnable)`

Enables or disables the QSCI transmitter.

This function enables or disables the QSCI transmitter.

Parameters

- `base` – QSCI peripheral base address.
- `bEnable` – True to enable, false to disable.

`static inline void QSCI_EnableRx(QSCI_Type *base, bool bEnable)`

Enables or disables the QSCI receiver.

This function enables or disables the QSCI receiver.

Parameters

- `base` – QSCI peripheral base address.
- `bEnable` – True to enable, false to disable.

`static inline void QSCI_EnableStopInWait(QSCI_Type *base, bool bEnable)`

Enables/disables stop in wait.

Parameters

- `base` – QSCI peripheral base address.
- `bEnable` – true to enable, QSCI stops working in wait mode, false to disable, QSCI keeps working in wait mode

```
static inline void QSCI_Enable9bitMode(QSCI_Type *base, bool bEnable)
```

Enables/Disables 9-bit data mode for QSCI.

Parameters

- base – QSCI peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline void QSCI_EnableStandbyMode(QSCI_Type *base, bool bEnable)
```

Enables/Disables standby mode.

When QSCI is in standby mode, further receiver interrupt requests are inhibited waiting to be wake up. The wakeup mode can be configured by `QSCI_SetWakeupMode`. Hardware wakes the receiver by automatically disabling standby.

Parameters

- base – QSCI peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline void QSCI_EnableLINSlaveMode(QSCI_Type *base, bool bEnable)
```

Enable/Disable LIN slave mode.

If enabled QSCI is in LIN slave mode. When break is detected, the baudrate register is automatically adjusted to match the value measured from the sync character that follows.

Parameters

- base – QSCI peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline void QSCI_EnableStopHold(QSCI_Type *base, bool bEnable)
```

Enable/Disable stop mode hold off.

When enabled, if chip level stop mode occurs and transmitter or receiver is still busy, QSCI will hold off stop mode until both transmitter and receiver are idle.

Parameters

- base – QSCI peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline void QSCI_SetTransferMode(QSCI_Type *base, qsci_transfer_mode_t  
eTransferMode)
```

Sets the QSCI transfer mode.

Parameters

- base – QSCI peripheral base address.
- eTransferMode – The QSCI tx/rx loop mode, `kQSCI_Normal` to use normal transfer; `kQSCI_LoopInternal` to let internal tx feed back to rx, `kQSCI_SingleWire` to use single wire mode using tx pin as tx and rx.

```
static inline void QSCI_SetWakeupMode(QSCI_Type *base, qsci_wakeup_mode_t eWakeupMode)
```

Sets wakeup mode for QSCI.

Parameters

- base – QSCI peripheral base address.
- eWakeupMode – Wakeup mode, `kQSCI_WakeupOnIdleLine` or `kQSCI_WakeupOnAddressMark`.

```
static inline void QSCI_SetPolarityMode(QSCI_Type *base, qsci_polarity_mode_t ePolarityMode)
    Sets polarity mode for QSCI.
```

Parameters

- base – QSCI peripheral base address.
- ePolarityMode – Polarity mode, kQSCI_PolarityNormal or kQSCI_PolarityInvert.

```
static inline void QSCI_SetParityMode(QSCI_Type *base, qsci_parity_mode_t eParityMode)
    Sets parity mode for QSCI.
```

Parameters

- base – QSCI peripheral base address.
- eParityMode – Polarity mode, kQSCI_ParityDisabled, kQSCI_ParityEven or kQSCI_ParityOdd.

```
status_t QSCI_SetBaudRate(QSCI_Type *base, uint32_t u32BaudRateBps, uint32_t
    u32SrcClockHz)
```

Sets the QSCI instance baud rate.

This function configures the QSCI module baud rate. This function can be used to update QSCI module baud rate after the QSCI module is initialized by the QSCI_Init.

Parameters

- base – QSCI peripheral base address.
- u32BaudRateBps – QSCI baudrate to be set.
- u32SrcClockHz – QSCI clock source frequency in Hz.

Return values

- kStatus_QSCI_BaudrateNotSupport – Baudrate is not supported in the current clock source.
- kStatus_Success – Set baudrate succeeded.

```
static inline void QSCI_EnableFifo(QSCI_Type *base, bool bEnable)
    Enables/Disables transmitter/receiver FIFO.
```

Parameters

- base – QSCI peripheral base address.
- bEnable – true to enable, false to disable.

```
static inline void QSCI_SetTxWaterMark(QSCI_Type *base, qsci_tx_water_t eTxFifoWatermark)
    Sets transmitter watermark.
```

Parameters

- base – QSCI peripheral base address.
- eTxFifoWatermark – Tx water mark level.

```
static inline void QSCI_SetRxWaterMark(QSCI_Type *base, qsci_rx_water_t eRxFifoWatermark)
    Sets receiver watermark.
```

Parameters

- base – QSCI peripheral base address.
- eRxFifoWatermark – Rx water mark level.

```
static inline void QSCI_EnableTxDMA(QSCI_Type *base, bool bEnable)
```

Enables or disables the QSCI transmitter DMA request.

This function enables or disables CTRL2[TDE], to generate the DMA requests when Tx data register is empty.

Parameters

- base – QSCI peripheral base address.
- bEnable – True to enable, false to disable.

```
static inline void QSCI_EnableRxDMA(QSCI_Type *base, bool bEnable)
```

Enables or disables the QSCI receiver DMA request.

This function enables or disables CTRL2[RDE], to generate DMA requests when receiver data register is full.

Parameters

- base – QSCI peripheral base address.
- bEnable – True to enable, false to disable.

```
static inline uint32_t QSCI_GetDataRegisterAddress(QSCI_Type *base)
```

Gets the QSCI data register byte address.

This function returns the QSCI data register address, which is mainly used by DMA/eDMA.

Parameters

- base – QSCI peripheral base address.

Returns

QSCI data register byte addresses which are used both by the transmitter and the receiver.

```
static inline void QSCI_WriteByte(QSCI_Type *base, uint8_t u8Data)
```

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 room before calling this function.

Parameters

- base – QSCI peripheral base address.
- u8Data – The byte to write.

```
static inline void QSCI_SendAddress(QSCI_Type *base, uint8_t u8Address)
```

Sends an address frame in 9-bit data mode.

Parameters

- base – QSCI peripheral base address.
- u8Address – QSCI slave address.

```
static inline uint8_t QSCI_ReadByte(QSCI_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 – QSCI peripheral base address.

Returns

The byte read from QSCI data register.

```
void QSCI_WriteBlocking(QSCI_Type *base, const uint8_t *pu8Data, uint32_t u32Length)
```

Writes TX register using a blocking method.

This function polls the TX register, waits TX register to be empty or TX FIFO have room then writes data to the TX buffer.

Parameters

- base – QSCI peripheral base address.
- pu8Data – Start address of the data to write.
- u32Length – Size of the data to write.

```
status_t QSCI_ReadBlocking(QSCI_Type *base, uint8_t *pu8Data, uint32_t u32Length)
```

Reads RX data register using a blocking method.

This function polls the RX register, waits RX register to be full or RX FIFO have data, then reads data from the RX register.

Parameters

- base – QSCI peripheral base address.
- pu8Data – Start address of the buffer to store the received data.
- u32Length – Size of the buffer.

Return values

- kStatus_Fail – Receiver error occurred while receiving data.
- kStatus_QSCI_RxHardwareOverrun – Receiver overrun occurred while receiving data
- kStatus_QSCI_NoiseError – Noise error occurred while receiving data
- kStatus_QSCI_FramingError – error occurred while receiving data
- kStatus_QSCI_ParityError – Parity error occurred while receiving data
- kStatus_Success – Successfully received all data.

```
static inline void QSCI_SendBreak(QSCI_Type *base)
```

Sends one break character (10 or 11 bits of zeroes).

Parameters

- base – QSCI peripheral base address.

```
void QSCI_TransferCreateHandle(QSCI_Type *base, qsci_transfer_handle_t *psHandle,
                             qsci_transfer_callback_t pfCallback, void *pUserData)
```

Initializes the QSCI handle.

This function initializes the QSCI handle which can be used for other QSCI transactional APIs. Usually, for a specified QSCI instance, call this API once to get the initialized handle.

Parameters

- base – QSCI peripheral base address.
- psHandle – QSCI handle pointer.
- pfCallback – The callback function.
- pUserData – The parameter of the callback function.

```
void QSCI_TransferStartRingBuffer(qsci_transfer_handle_t *psHandle, uint8_t *pu8RxRingBuffer,  
                                uint16_t u16RxRingBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific QSCI handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the QSCI_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

- psHandle – QSCI handle pointer.
- pu8RxRingBuffer – Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
- u16RxRingBufferSize – Size of the ring buffer.

```
void QSCI_TransferStopRingBuffer(qsci_transfer_handle_t *psHandle)
```

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- psHandle – QSCI handle pointer.

```
uint16_t QSCI_TransferGetRxRingBufferLength(qsci_transfer_handle_t *psHandle)
```

Get the ring buffer valid data length.

Parameters

- psHandle – QSCI handle pointer.

Returns

Valid data length in ring buffer.

```
status_t QSCI_TransferSendNonBlocking(qsci_transfer_handle_t *psHandle, qsci_transfer_t  
                                     *psTransfer)
```

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 sent out, the QSCI driver calls the callback function and passes the kStatus_QSCI_TxIdle as status parameter.

Parameters

- psHandle – QSCI handle pointer.
- psTransfer – QSCI transfer structure. See qsci_transfer_t.

Return values

- kStatus_Success – Successfully start the data transmission.
- kStatus_QSCI_TxBusy – Previous transmission still not finished; data not all written to TX register yet.

```
void QSCI_TransferAbortSend(qsci_transfer_handle_t *psHandle)
```

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

- psHandle – QSCI handle pointer.

```
status_t QSCI_TransferGetSendCount(qsci_transfer_handle_t *psHandle, uint32_t *pu32Count)
```

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

- psHandle – QSCI handle pointer.
- pu32Count – Send bytes count.

Return values

- kStatus_NoTransferInProgress – No send in progress.
- kStatus_Success – Get successfully through the parameter count;

```
status_t QSCI_TransferReceiveNonBlocking(qsci_transfer_handle_t *psHandle, qsci_transfer_t *psTransfer, uint32_t *pu32ReceivedBytes)
```

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 pu32ReceivedBytes 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 QSCI driver. When the new data arrives, the receive request is serviced first. When all data is received, the QSCI driver notifies the upper layer through a callback function and passes the status parameter kStatus_QSCI_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 psTransfer->data and this function returns with the parameter pu32ReceivedBytes set to 5. For the left 5 bytes, newly arrived data is saved from the psTransfer->data[5]. When 5 bytes are received, the QSCI 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 psTransfer->data. When all data is received, the upper layer is notified.

Parameters

- psHandle – QSCI handle pointer.
- psTransfer – QSCI transfer structure, see qsci_transfer_t.
- pu32ReceivedBytes – Bytes received from the ring buffer directly.

Return values

- kStatus_Success – Successfully queue the transfer into transmit queue.
- kStatus_QSCI_RxBusy – Previous receive request is not finished.

```
void QSCI_TransferAbortReceive(qsci_transfer_handle_t *psHandle)
```

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

- psHandle – QSCI handle pointer.

status_t QSCI_TransferGetReceivedCount(*qsci_transfer_handle_t* *psHandle, uint32_t *pu32Count)

Gets the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- psHandle – QSCI handle pointer.
- pu32Count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter pu32Count;

FSL_QSCI_DRIVER_VERSION

QSCI driver version.

Status codes for the QSCI driver.

Values:

enumerator kStatus_QSCI_TxBusy
Transmitter is busy.

enumerator kStatus_QSCI_RxBusy
Receiver is busy.

enumerator kStatus_QSCI_TxIdle
Transmitter is idle.

enumerator kStatus_QSCI_RxIdle
Receiver is idle.

enumerator kStatus_QSCI_FlagCannotClearManually
Status flag can't be manually cleared.

enumerator kStatus_QSCI_RxRingBufferOverrun
QSCI RX software ring buffer overrun.

enumerator kStatus_QSCI_RxHardwareOverrun
QSCI receiver hardware overrun.

enumerator kStatus_QSCI_NoiseError
QSCI noise error.

enumerator kStatus_QSCI_FramingError
QSCI framing error.

enumerator kStatus_QSCI_ParityError
QSCI parity error.

enumerator kStatus_QSCI_BaudrateNotSupport
Baudrate is not supported in current clock source

enumerator kStatus_QSCI_IdleLineDetected
QSCI IDLE line detected.

enumerator kStatus_QSCI_Timeout
Timeout happens when waiting for status flags to change.

enum `_qsci_status_flags`

QSCI hardware status flags.

These enumerations can be ORed together to form bit masks.

Values:

enumerator `kQSCI_TxDataRegEmptyFlag`

TX data register empty flag.

enumerator `kQSCI_TxIdleFlag`

Transmission idle flag.

enumerator `kQSCI_RxDataRegFullFlag`

RX data register full flag.

enumerator `kQSCI_RxIdleLineFlag`

Rx Idle line flag.

enumerator `kQSCI_RxOverrunFlag`

RX overrun flag.

enumerator `kQSCI_RxNoiseFlag`

RX detect noise on Rx input.

enumerator `kQSCI_RxFrameErrorFlag`

Rx frame error flag, sets if logic 0 was detected for stop bit

enumerator `kQSCI_RxParityErrorFlag`

Rx parity error if parity enabled, sets upon parity error detection

enumerator `kQSCI_RxInputEdgeFlag`

RX pin active edge interrupt flag, sets when active edge detected

enumerator `kQSCI_LINSyncErrorFlag`

Only for LIN mode.

enumerator `kQSCI_TxDMARequestFlag`

Tx DMA request is ongoing.

enumerator `kQSCI_RxDMARequestFlag`

Rx DMA request is ongoing.

enumerator `kQSCI_RxActiveFlag`

enumerator `kQSCI_Group0Flags`

Members in `kQSCI_Group0Flags` can't be cleared by `QSCI_ClearStatusFlags`, they are handled by HW.

enumerator `kQSCI_Group1Flags`

Whole `kQSCI_Group1Flags` will be cleared if trying to clear any member in `kQSCI_Group1Flags` or `kQSCI_Group2Flags` in the mask.

enumerator `kQSCI_Group2Flags`

Member in `kQSCI_Group2Flags` can be cleared individually

enumerator `kQSCI_StatusAllFlags`

enum `_qsci_interrupt_enable`

QSCI interrupt enable/disable source.

These enumerations can be ORed together to form bit masks.

Values:

enumerator kQSCI_TxEmptyInterruptEnable
Transmit data register empty interrupt.

enumerator kQSCI_TxIdleInterruptEnable
Transmission idle interrupt.

enumerator kQSCI_RxFullInterruptEnable
Receive data register full interrupt.

enumerator kQSCI_RxErrorInterruptEnable
Receive error interrupt.

enumerator kQSCI_RxInputEdgeInterruptEnable
Receive input edge interrupt.

enumerator kQSCI_RxIdleLineInterruptEnable
Receive idle interrupt.

enumerator kQSCI_AllInterruptEnable

enum _qsci_transfer_mode
QSCI transmitter/receiver loop mode.

Values:

enumerator kQSCI_Normal
Normal mode, 2 signal pins, no loop.

enumerator kQSCI_LoopInternal
Loop mode with internal TXD fed back to RXD.

enumerator kQSCI_SingleWire
Use tx pin as input and output half-duplex transfer.

enum _qsci_data_bit_mode
QSCI data bit count.

Values:

enumerator kQSCI_Data8Bit
1 start bit, 8 data bit, 1 stop bit

enumerator kQSCI_Data9Bit
1 start bit, 9 data bit, 1 stop bit. This mode actually is not supported yet in driver.

enum _qsci_wakeup_mode
QSCI wakeup mode.

Values:

enumerator kQSCI_WakeupOnIdleLine
Idle condition wakes the QSCI module.

enumerator kQSCI_WakeupOnAddressMark
Address mark wakes the QSCI module.

enum _qsci_polarity_mode
QSCI signal polarity mode.

Values:

enumerator kQSCI_PolarityNormal
Normal mode, no inversion.

enumerator kQSCI_PolarityInvert
Invert transmit and receive data bits.

enum _qsci_parity_mode
QSCI parity mode.
Values:

enumerator kQSCI_ParityDisabled
Parity disabled

enumerator kQSCI_ParityEven
Parity enabled, type even, bit setting: PE | PT = 10

enumerator kQSCI_ParityOdd
Parity enabled, type odd, bit setting: PE | PT = 11

enum _qsci_tx_water
QSCI transmitter watermark level.
Values:

enumerator kQSCI_TxWater0Word
Tx interrupt sets when tx fifo empty.

enumerator kQSCI_TxWater1Word
Tx interrupt sets when tx fifo has 1 or few word.

enumerator kQSCI_TxWater2Word
Tx interrupt sets when tx fifo has 2 or few words.

enumerator kQSCI_TxWater3Word
Tx interrupt sets when tx fifo not full.

enum _qsci_rx_water
QSCI receiver watermark level.
Values:

enumerator kQSCI_RxWater1Word
Rx interrupt sets when rx fifo not empty.

enumerator kQSCI_RxWater2Word
Rx interrupt sets when rx fifo has at least 1 word.

enumerator kQSCI_RxWater3Word
Rx interrupt sets when rx fifo has at least 2 words.

enumerator kQSCI_RxWater4Word
Rx interrupt sets when rx fifo full.

typedef enum _qsci_transfer_mode qsci_transfer_mode_t
QSCI transmitter/receiver loop mode.

typedef enum _qsci_data_bit_mode qsci_data_bit_mode_t
QSCI data bit count.

typedef enum _qsci_wakeup_mode qsci_wakeup_mode_t
QSCI wakeup mode.

typedef enum _qsci_polarity_mode qsci_polarity_mode_t
QSCI signal polarity mode.

typedef enum *_qsci_parity_mode* qsci_parity_mode_t
QSCI parity mode.

typedef enum *_qsci_tx_water* qsci_tx_water_t
QSCI transmitter watermark level.

typedef enum *_qsci_rx_water* qsci_rx_water_t
QSCI receiver watermark level.

typedef struct *_qsci_config* qsci_config_t
QSCI configuration structure.

typedef struct *_qsci_transfer_handle_t* qsci_transfer_handle_t
Forward declaration of the handle typedef .

typedef void (*qsci_transfer_callback_t)(qsci_transfer_handle_t *psHandle)
QSCI interrupt transfer callback function definition.

Defines the interface of user callback function used in QSCI interrupt transfer using transactional APIs. The callback function shall be defined and declared in application level by user. Before starting QSCI transmitting or receiving by calling QSCI_TransferSendNonBlocking or QSCI_TransferReceiveNonBlocking, call QSCI_TransferCreateHandle to install the user callback. When the transmitting or receiving ends or any bus error like hardware overrun occurs, user callback will be invoked by driver.

Param psHandle

Transfer handle that contains bus status, user data.

typedef struct *_qsci_transfer* qsci_transfer_t
QSCI transfer structure.

typedef void (*qsci_isr_t)(void *handle)

qsci_isr_t s_pfQsciIsr

void *s_psQsciHandles[]

IRQn_Type const s_eQsciTXIdleIRQs[]

uint16_t QSCI_GetInstance(QSCI_Type *base)
Get the QSCI instance from peripheral base address.

Parameters

- base – QSCI peripheral base address.

Returns

QSCI instance.

void QSCI_DriverIRQHandler(uint32_t instance)
QSCI driver IRQ handler common entry.

This function provides the common IRQ request entry for QSCI.

Parameters

- instance – QSCI instance.

QSCI_RETRY_TIMES
Retry times when checking status flags.

QSCI_GET_BUS_STATUS(psHandle)
Macros to be used inside user callback.

QSCI_GET_TRANSFER_USER_DATA(psHandle)

```
struct _qsci_config
    #include <fsl_qsci.h> QSCI configuration structure.
```

Public Members

```
qsci_transfer_mode_t eTransferMode
    Transmitter/receiver loop mode.

bool bStopInWaitEnable
    Enable/disable module stops working in wait mode.

qsci_data_bit_mode_t eDataBitMode
    Number of data bits.

qsci_wakeup_mode_t eWakeupMode
    Receiver wakeup mode, idle line or addressmark.

qsci_polarity_mode_t ePolarityMode
    Polarity of transmit/receive data.

qsci_parity_mode_t eParityMode
    Parity mode, disabled (default), even, odd.

bool bEnableStopHold
    Control the stop hold enable.

bool bEnableTx
    Enable TX

bool bEnableRx
    Enable RX

bool bEnableFifo
    Enable Tx/Rx FIFO

bool bEnableTxDMA
    Enable Tx DMA

bool bEnableRxDMA
    Enable Rx DMA

qsci_tx_water_t eTxFifoWatermark
    TX FIFO watermark

qsci_rx_water_t eRxFifoWatermark
    RX FIFO watermark

uint8_t u8Interrupts
    Mask of QSCI interrupt sources to enable.

uint32_t u32BaudRateBps
    QSCI baud rate

uint32_t u32SrcClockHz
    The clock source frequency for QSCI module.
```

```
struct _qsci_transfer_handle_t
    #include <fsl_qsci.h> QSCI transfer handle.
```

Note: If user wants to use the transactional API to transfer data in interrupt way, one QSCI instance should and can only be allocated one handle.

Note: The handle is maintained by QSCI driver internally, which means the transfer state is retained and user shall not modify its state `u8TxState` or `u8RxState` in application level. If user only wish to use transactional APIs without understanding its mechanism, it is not necessary to understand these members.

Public Members

`QSCI_Type *base`

QSCI base pointer to the instance belongs to this handle.

`uint8_t *pu8TxData`

Address of remaining data to send.

`volatile uint32_t u32TxRemainingSize`

Size of the remaining data to send.

`uint32_t u32TxDataSize`

Size of the data to send out.

`uint8_t *pu8RxData`

Address of remaining data to receive.

`volatile uint32_t u32RxRemainingSize`

Size of the remaining data to receive.

`uint32_t u32RxDataSize`

Size of the data to receive.

`uint8_t *pu8RxRingBuffer`

Start address of the receiver ring buffer.

`uint16_t u16RxRingBufferSize`

Size of the ring buffer.

`volatile uint16_t u16RxRingBufferHead`

Index for the driver to store received data into ring buffer.

`volatile uint16_t u16RxRingBufferTail`

Index for the user to get data from the ring buffer.

`qsci_transfer_callback_t pfCallback`

Callback function.

`void *pUserData`

QSCI callback function parameter.

`volatile uint8_t u8TxState`

TX transfer state.

`volatile uint8_t u8RxState`

RX transfer state

`status_t busStatus`

QSCI bus status.

`struct _qsci_transfer`

`#include <fsl_qsci.h>` QSCI transfer structure.

Public Members

`uint8_t *pu8Data`

The buffer pointer of data to be transferred.

`uint32_t u32DataSize`

The byte count to be transferred.

2.59 The Driver Change Log

2.60 QSCI_EDMA: EDMA based QSCI Driver

```
void QSCI_TransferCreateHandleEDMA(QSCI_Type *base, qsci_edma_transfer_handle_t
    *psHandle, qsci_edma_transfer_callback_t pfCallback,
    void *pUserData, DMA_Type *edmaBase,
    edma_channel_t eEdmaTxChannel, edma_channel_t
    eEdmaRxChannel)
```

Initializes the QSCI edma handle.

This function initializes the QSCI edma handle which can be used for other QSCI transactional APIs. Usually, for a specified QSCI instance, call this API once to get the initialized handle.

Parameters

- `base` – QSCI peripheral base address.
- `psHandle` – Pointer to `qsci_edma_transfer_handle_t` structure.
- `pfCallback` – Callback function.
- `pUserData` – User data.
- `edmaBase` – Edma base address.
- `eEdmaTxChannel` – eDMA channel for TX transfer.
- `eEdmaRxChannel` – eDMA channel for RX transfer.

```
status_t QSCI_TransferSendEDMA(qsci_edma_transfer_handle_t *psHandle, qsci_transfer_t
    *psTransfer)
```

Initiate data transmit using EDMA.

This function initiates a data transmit process using eDMA. This is a non-blocking function, which returns right away. When all the data is sent, the send callback function is called.

Parameters

- `psHandle` – QSCI handle pointer.
- `psTransfer` – QSCI eDMA transfer structure. See `qsci_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_QSCI_TxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

status_t QSCI_TransferReceiveEDMA(*qsci_edma_transfer_handle_t* *psHandle, *qsci_transfer_t* *psTransfer)

Initiate data receive using EDMA.

This function initiates a data receive process using eDMA. This is a non-blocking function, which returns right away. When all the data is received, the receive callback function is called.

Parameters

- psHandle – Pointer to *qsci_edma_transfer_handle_t* structure.
- psTransfer – QSCI eDMA transfer structure, see *qsci_transfer_t*.

Return values

- *kStatus_Success* – if succeed, others fail.
- *kStatus_QSCI_RxBusy* – Previous transfer ongoing.
- *kStatus_InvalidArgument* – Invalid argument.

void QSCI_TransferAbortSendEDMA(*qsci_edma_transfer_handle_t* *psHandle)

Aborts the data transmit process using EDMA.

Parameters

- psHandle – Pointer to *qsci_edma_transfer_handle_t* structure.

void QSCI_TransferAbortReceiveEDMA(*qsci_edma_transfer_handle_t* *psHandle)

Aborts the data receive process using EDMA.

Parameters

- psHandle – Pointer to *qsci_edma_transfer_handle_t* structure.

status_t QSCI_TransferGetReceivedCountEDMA(*qsci_edma_transfer_handle_t* *psHandle, *uint32_t* *pu32Count)

Gets the number of received bytes.

This function gets the number of received bytes.

Parameters

- psHandle – QSCI handle pointer.
- pu32Count – Receive bytes count.

Return values

- *kStatus_NoTransferInProgress* – No receive in progress.
- *kStatus_Success* – Get successfully through the parameter count;

status_t QSCI_TransferGetSendCountEDMA(*qsci_edma_transfer_handle_t* *psHandle, *uint32_t* *pu32Count)

Gets the number of bytes written to the QSCI TX register.

This function gets the number of bytes written to the QSCI TX register by DMA.

Parameters

- psHandle – QSCI handle pointer.
- pu32Count – Send bytes count.

Return values

- *kStatus_NoTransferInProgress* – No send in progress.
- *kStatus_Success* – Get successfully through the parameter count;

FSL_QSCI_EDMA_DRIVER_VERSION

QSCI EDMA driver version.

```
typedef struct _qsci_edma_transfer_handle qsci_edma_transfer_handle_t
```

Forward declaration of the qsci edma handle typedef. .

```
typedef void (*qsci_edma_transfer_callback_t)(qsci_edma_transfer_handle_t *psHandle)
```

QSCI edma transfer callback function definition.

Defines the interface of user callback function used in QSCI edma transfer using transactional APIs. The callback function shall be defined and declared in application level by user. Before starting QSCI transmitting or receiving by calling QSCI_TransferSendEDMA or QSCI_TransferReceiveEDMA, call QSCI_TransferCreateHandleEDMA to install the user callback. When the transmitting or receiving ends, user callback will be invoked by driver.

Param psHandle

Transfer handle that contains bus status, user data.

```
struct _qsci_edma_transfer_handle
```

```
#include <fsl_qsci_edma.h> QSCI edma transfer handle.
```

This struct address should be sizeof(edma_channel_tcd_t) aligned.

Note: If user wants to use the transactional API to transfer data in edma way, one QSCI instance should and can only be allocated one handle.

Note: The handle is maintained by QSCI driver internally, which means the transfer state is retained and user shall not modify its state u8TxState or u8RxState in application level. If user only wish to use transactional APIs without understanding its mechanism, it is not necessary to understand these members.

Public Members

edma_channel_tcd_t sTxTcd

TCD for EDMA TX transfer.

edma_channel_tcd_t sRxTcd

TCD for EDMA RX transfer.

QSCI_Type *base

Pointer to the QSCI base that belongs to this handle.

qsci_edma_transfer_callback_t pfCallback

Callback function.

uint32_t u32RxDataSizeAll

Size of the data to receive.

uint32_t u32TxDataSizeAll

Size of the data to send out.

edma_handle_t sTxEdmaHandle

The eDMA TX channel used.

edma_handle_t sRxEdmaHandle

The eDMA RX channel used.

volatile uint8_t u8TxState

TX transfer state.

volatile uint8_t u8RxState

RX transfer state

status_t busStatus

QSCI bus status.

void *pUserData

User configurable pointer to any data, function, structure etc that user wish to use in the callback

2.61 QSCI Peripheral and Driver Overview

2.62 QSPI: Queued SPI Driver

void QSPI_MasterInit(QSPI_Type *base, const *qspi_master_config_t* *psConfig)

Initializes the QUEUEDSPI as Master.

Use helper function QSPI_MasterGetDefaultConfig to get ready-to-use structure.

Parameters

- base – QUEUEDSPI peripheral address.
- psConfig – Pointer to the structure *qspi_master_config_t*.

void QSPI_MasterGetDefaultConfig(*qspi_master_config_t* *psConfig, uint32_t u32ClockFreqHz)

Helper function to create ready-to-user maste init structure.

The purpose of this API is to get the configuration structure initialized for the QSPI_MasterInit. Users may use the initialized structure unchanged in the QSPI_MasterInit or modify the structure before calling the QSPI_MasterInit. Example:

```
qspi_master_config_t sMasterConfig;
QSPI_MasterGetDefaultConfig(&sMasterConfig);
```

The default values are: Example:

```
// Parameter provided by user
psConfig->u32BaudRateBps = u32BaudRateBps;
psConfig->u32ClockFrequencyHz = u32ClockFreqHz;
psConfig->eDataWidth = eDataWidth;

// Default configuration
psConfig->eClkPolarity = kQSPI_ClockPolarityActiveRisingEdge;
psConfig->eClkPhase = kQSPI_ClockPhaseSlaveSelectHighBetweenWords;
psConfig->eShiftDirection = kQSPI_MsbFirst;
psConfig->u16DelayBetweenFrameInCLK = 1U;
psConfig->bEnableWiredOrMode = false;
psConfig->bEnableModeFault = false;
psConfig->u8DmaEnableFlags = 0U; // Disable TX/RX Dma
psConfig->bEnableFIFO = false;
psConfig->bEnableStopModeHoldOff = false;
psConfig->u8Interrupts = 0U;
psConfig->bEnableModule = false;
@todo To be added
```

Parameters

- psConfig – pointer to `qspi_master_config_t` structure.
- u32ClockFreqHz – Peripheral clock frequency in Hz

```
void QSPI_SlaveInit(QSPI_Type *base, const qspi_slave_config_t *psConfig)
```

Initializes the QUEUEDSPI as slave.

Use helper function `QSPI_SlaveGetDefaultConfig` to get ready-to-use structure.

Parameters

- base – QUEUEDSPI peripheral address.
- psConfig – Pointer to the structure `qspi_slave_config_t`.

```
void QSPI_SlaveGetDefaultConfig(qspi_slave_config_t *psConfig)
```

Set the `qspi_slave_config_t` structure to default values.

The purpose of this API is to get the configuration structure initialized for the `QSPI_SlaveInit`. Users may use the initialized structure unchanged in the `QSPI_SlaveInit` or modify the structure before calling the `QSPI_SlaveInit`. Example:

```
qspi_slave_config_t slaveConfig;
QSPI_SlaveGetDefaultConfig(&slaveConfig);
```

The default values are: Example:

```
@todo
```

Parameters

- psConfig – Pointer to the `qspi_slave_config_t` structure.

```
void QSPI_Deinit(QSPI_Type *base)
```

De-initialize the QUEUEDSPI peripheral for either Master or Slave.

Parameters

- base – QUEUEDSPI peripheral address.

```
static inline void QSPI_EnableInterrupts(QSPI_Type *base, uint8_t u8Interrupts)
```

Enable one or multiple interrupts.

This function enable one or multiple interrupts.

Note: for TX and RX requests, while enabling the interrupt request the DMA request will be disabled as well. Do not use this API while QUEUEDSPI is in running state.

Parameters

- base – QUEUEDSPI peripheral address.
- u8Interrupts – The interrupt mask which is ORed by the `_qspi_interrupt_enable`.

```
static inline void QSPI_DisableInterrupts(QSPI_Type *base, uint8_t u8Interrupts)
```

Disable one or multiple interrupts.

This function

Parameters

- base – QUEUEDSPI peripheral address.
- u8Interrupts – The interrupt mask which is ORed by the `_qspi_interrupt_enable`.

```
static inline void QSPI_EnableDMA(QSPI_Type *base, uint8_t u8DmaFlags)
```

Enable one or multiple DMA.

Note that if the DMA is enabled for Transmit or Receive, make sure the interrupt is disabled for Transmit or Receive.

Parameters

- base – QUEUEDSPI peripheral address.
- u8DmaFlags – DMA Flags ORed from `_qspi_dma_enable_flags`.

```
static inline void QSPI_DisableDMA(QSPI_Type *base, uint8_t u8DmaFlags)
```

Enable one or multiple DMA.

Note that if the DMA is enabled for Transmit or Receive, make sure the interrupt is disabled for Transmit or Receive.

Parameters

- base – QUEUEDSPI peripheral address.
- u8DmaFlags – DMA Flags ORed from `_qspi_dma_enable_flags`.

```
static inline uint32_t QSPI_GetTxRegisterAddress(QSPI_Type *base)
```

Get the QUEUEDSPI transmit data register address for the DMA operation.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

The QUEUEDSPI master PUSHHR data register address.

```
static inline uint32_t QSPI_GetRxRegisterAddress(QSPI_Type *base)
```

Get the QUEUEDSPI receive data register address for the DMA operation.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

The QUEUEDSPI POPR data register address.

```
static inline uint16_t QSPI_GetStatusFlags(QSPI_Type *base)
```

Get the QUEUEDSPI status flag state.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

QUEUEDSPI status.

```
static inline void QSPI_ClearStatusFlags(QSPI_Type *base, uint16_t u16StatusFlags)
```

Clear the status flag only for the mode fault.

Clear the status flag only for mode fault.

Note: only `kQSPI_ModeFaultFlag` can be cleared by this API.

Parameters

- base – QUEUEDSPI peripheral address.
- u16StatusFlags – status flags ORed from `_qspi_status_flags`

```
static inline void QSPI_Enable(QSPI_Type *base, bool bEnable)
```

Enable or disable the QUEUEDSPI peripheral.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – true to enable module, otherwise disable module

```
uint32_t QSPI_MasterSetBaudRate(QSPI_Type *base, uint32_t u32BaudRateBps, uint32_t
                                u32SrcClockHz)
```

Set the QUEUEDSPI baud rate in bits-per-second.

This function takes in the desired baud rate, calculates the nearest possible baud rate, and returns the calculated baud rate in bits-per-second.

Parameters

- base – QUEUEDSPI peripheral address.
- u32BaudRateBps – The desired baud rate in bits-per-second.
- u32SrcClockHz – Module source input clock in Hertz.

Returns

The actual calculated baud rate.

```
static inline void QSPI_SetMasterSlaveMode(QSPI_Type *base, qspi_master_slave_mode_t
                                           eMode)
```

Set the QUEUEDSPI as master or slave.

Parameters

- base – QUEUEDSPI peripheral address.
- eMode – Mode setting of type `qspi_master_slave_mode_t`.

```
static inline bool QSPI_IsMaster(QSPI_Type *base)
```

Return whether the QUEUEDSPI module is in master mode.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

```
static inline void QSPI_SetDataShiftOrder(QSPI_Type *base, qspi_data_shift_direction_t
                                           eDataShiftOrder)
```

Set Data Shift Order as MSB first or LSB first.

Parameters

- base – QUEUEDSPI peripheral address.
- eDataShiftOrder – MSB or LSB first from `qspi_data_shift_direction_t`

```
static inline void QSPI_EnableModeFault(QSPI_Type *base, bool bEnable)
```

Enable/Disable mode fault detection.

If enable, allows the `kQSPI_ModeFaultFlag` flag to be set. If the `kQSPI_ModeFaultFlag` flag is set, disable the Mod detection does not clear the flag. If the mod detection is disabled, the level of the `SS_B` pin does not affect the operation of an enabled SPI configured as a master. If configured as a master and mod fault detection is enabled, a transaction in progress will stop if `SS_B` goes low. For an enabled SPI configured as a slave, having this feature disabled only prevents the flag from being set. It does not affect any other part of SPI operation

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – true to enable Mode Fault detection, false to disable

static inline void QSPI_SetClockPolarity(QSPI_Type *base, *qspi_clock_polarity_t* ePolarity)
Set clock polarity.

Note: module shall be disabled before change the polarity by calling QSPI_Enable.

Parameters

- base – QUEUEDSPI peripheral address.
- ePolarity – clock polarity option

static inline void QSPI_SetClockPhase(QSPI_Type *base, *qspi_clock_phase_t* eClockPhase)
Set clock phase.

Configure whether get the Slave Select signal toggle high during 2 data frames. Get the SS toggle high between data frames will lead to SPI to be triggered with transaction for the falling edge of SS signal. Otherwise, the data transaction is started on the first active SCLK edge.

Note: module shall be disabled before change the polarity by calling QSPI_Enable.

Note: Do not use kQSPI_ClockPhaseSlaveSelectHighBetweenWords in DMA mode.

Parameters

- base – QUEUEDSPI peripheral address.
- eClockPhase – Option for clock phase

static inline void QSPI_EnableWiredORMode(QSPI_Type *base, bool bEnable)

Enable/Disable Wired OR mode for SPI pins which means open-drain when enabled and push-pull when disabled.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – true to configure SPI pins as open-drain, false to configure as push-pull

static inline void QSPI_SetTransactionDataSize(QSPI_Type *base, *qspi_data_width_t* eDataWidth)
Set the transaction data width.

Parameters

- base – QUEUEDSPI peripheral address.
- eDataWidth – datawidth for bits in each data frame.

static inline void QSPI_MasterSetWaitDelay(QSPI_Type *base, uint16_t
u16WaitDelayInPeriClockCount)

For master mode, set wait delay in clock cycle with delay is set value + 1 peripheral bus clock.

This controls the time between data transactions in master mode. Delay will not be added if no word is waiting for transmitting.

Parameters

- base – QUEUEDSPI peripheral address.
- u16WaitDelayInPeriClockCount – Clock count for the delay during data frames

```
static inline void QSPI_EnableStopModeHoldOff(QSPI_Type *base, bool bEnable)
```

Enable/Disable hold off entry to stop mode is a word is being transmitted/received for Master Mode.

When enabled, this bit allows the SPI module to hold off entry to chip level stop mode if a word is being transmitted or received. Stop mode will be entered after the SPI finishes transmitting/receiving. This bit does not allow the SPI to wake the chip from stop mode in any way. The SHEN bit can only delay the entry into stop mode. This bit should not be set in slave mode because the state of SS_B (which would be controlled by an external master device) may cause the logic to hold off stop mode entry forever.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – true to enable hold-off entering stop mode if there is transmitting/receiving

```
uint32_t QSPI_GetInstance(QSPI_Type *base)
```

Helper function exported for QSPI DMA driver.

Get the instance index from the base address. User need not understand this function.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

uint32_t Index of the peripheral instance for given base address.

```
static inline qspi_ss_data_logic_level_t QSPI_MasterGetSlaveSelectLogicLevel(QSPI_Type *base)
```

For master mode, get the SS_B input logic level while true means drive High and false means drive Low.

Get the value to drive on the SS_B pin. This bit is disabled when SSB_AUTO=1 or SSB_STRB=1. Only apply for Master mode.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

true SS_B input level High

Returns

false SS_B input level Low

```
static inline void QSPI_MasterSetSlaveSelectLogicLevel(QSPI_Type *base,
qspi_ss_data_logic_level_t eLogicLevel)
```

for master mode, drive Slave Select pin logic high or low

This feature is disabled if Slave Select automatic mode is enabled or Slave Select Strobe feature is enabled

Parameters

- base – QUEUEDSPI peripheral address.
- eLogicLevel – logic level

static inline void QSPI_MasterEnableSlaveSelectOpenDrainMode(QSPI_Type *base, bool bEnable)

For master mode, Enable open drain in SSB pad pin.

Enable it means SS_B is configured for high and low drive. This mode is generally used in single master systems. Disable it means SS_B is configured as an open drain pin (only drives low output level). This mode is useful for multiple master systems

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – Enable/Disable option.

static inline void QSPI_MasterEnableSlaveSelectAutomaticMode(QSPI_Type *base, bool bEnable)

For master mode, Enable/Disable Slave Select pin automatic mode.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – Enable/Disable option.

static inline void QSPI_SetSlaveSelectDirection(QSPI_Type *base, *qspi_ss_direction_t* eDirection)

Set Input/Output mode for SSB signal.

Parameters

- base – QUEUEDSPI peripheral address.
- eDirection – options from *qspi_ss_direction_t*

static inline void QSPI_MasterEnableSlaveSelectStrobe(QSPI_Type *base, bool bEnable)

For master, set strobe mode for SSB signal.

If enabled, Slave select pulse high during data frames irrespective of Clock Phase configuration

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – Enable/Disable option.

static inline void QSPI_EnableSlaveSelectOverride(QSPI_Type *base, bool bEnable)

Enable / Disable SSB signal from Master/Slave configuration or GPIO pin state.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – Enable/Disable option.

void QSPI_SetDummyData(QSPI_Type *base, uint8_t u8DummyData)

Set up the dummy data used when there is not transmit data provided.

Parameters

- base – QUEUEDSPI peripheral address.
- u8DummyData – Data to be transferred when tx buffer is NULL.

uint8_t QSPI_GetDummyData(QSPI_Type *base)

Get the dummy data for each peripheral.

Parameters

- base – QUEUEDSPI peripheral base address.

```
static inline void QSPI_WriteData(QSPI_Type *base, uint16_t data)
```

Write data into the transmit data register without polling the status of shifting.

Parameters

- base – QUEUEDSPI peripheral address.
- data – The data to send.

```
static inline uint16_t QSPI_ReadData(QSPI_Type *base)
```

Read data from the receive data register.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

The data from the receive data register.

```
static inline void QSPI_EnableFifo(QSPI_Type *base, bool bEnable)
```

Enable or disable the QUEUEDSPI FIFOs.

This function allows the caller to disable or enable the TX and RX FIFOs together.

Parameters

- base – QUEUEDSPI peripheral address.
- bEnable – Pass true to enable, pass false to disable

```
static inline uint16_t QSPI_GetTxFIFOCount(QSPI_Type *base)
```

Get TX FIFO level.

This function gets how many words are in the TX FIFO.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

TX FIFO word count.

```
static inline uint16_t QSPI_GetRxFIFOCount(QSPI_Type *base)
```

Get RX FIFO level.

This function gets how many words are in the RX FIFO.

Parameters

- base – QUEUEDSPI peripheral address.

Returns

RX FIFO word count.

```
static inline void QSPI_SetFifoWatermarks(QSPI_Type *base, uint16_t txWatermark, uint16_t rxWatermark)
```

Set the transmit and receive FIFO watermark values.

Parameters

- base – QUEUEDSPI peripheral address.
- txWatermark – The TX FIFO watermark value. Refer to `qspi_txfifo_watermark_t` for available values.
- rxWatermark – The RX FIFO watermark value. Refer to `qspi_rxfifo_watermark_t` for available values.

```
static inline void QSPI_GetFifoWatermarks(QSPI_Type *base, uint8_t *pu8TxWatermark, uint8_t *pu8RxWatermark)
```

Get the transmit and receive FIFO watermark values.

Parameters

- base – QUEUEDSPI peripheral address.
- pu8TxWatermark – The TX FIFO watermark value.
- pu8RxWatermark – The RX FIFO watermark value.

```
static inline void QSPI_EmptyRxFifo(QSPI_Type *base)
```

Empty the QUEUEDSPI RX FIFO.

Parameters

- base – QUEUEDSPI peripheral address.

```
void QSPI_MasterTransferCreateHandle(QSPI_Type *base, qspi_master_transfer_handle_t *psHandle, qspi_master_transfer_callback_t pfCallback, void *pUserData)
```

Initialize the QUEUEDSPI master handle.

This function initializes the QUEUEDSPI handle, which can be used for other QUEUEDSPI transactional APIs. Usually, for a specified QUEUEDSPI instance, call this API once to get the initialized handle.

Note: If only use the QSPI_MasterTransferBlocking, this API is not necessary be called.

Parameters

- base – QUEUEDSPI peripheral address.
- psHandle – QUEUEDSPI handle pointer to qspi_master_transfer_handle_t.
- pfCallback – QUEUEDSPI callback.
- pUserData – Callback function parameter.

```
status_t QSPI_MasterTransferBlocking(QSPI_Type *base, qspi_transfer_t *psXfer)
```

Polling method of QUEUEDSPI master transfer.

This function transfers data using a polling method for master. This is a blocking function, which does not return until all transfers have been completed.

Parameters

- base – QUEUEDSPI peripheral address.
- psXfer – Pointer to the qspi_transfer_t structure.

Returns

status of status_t.

```
status_t QSPI_MasterTransferNonBlocking(qspi_master_transfer_handle_t *psHandle, qspi_transfer_t *psXfer)
```

Interrupt method of QUEUEDSPI master transfer.

This function transfers data using interrupts for master. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

- psHandle – QUEUEDSPI handle pointer to qspi_master_transfer_handle_t.
- psXfer – Pointer to the qspi_transfer_t structure.

Returns

status of status_t.

status_t QSPI_MasterTransferGetCount(*qspi_master_transfer_handle_t* *psHandle, uint16_t *pu16Count)

Get the master transfer count.

Parameters

- psHandle – QUEUEDSPI handle pointer to *qspi_master_transfer_handle_t*.
- pu16Count – The number of bytes transferred by using the non-blocking transaction.

Returns

status of status_t.

void QSPI_MasterTransferAbort(*qspi_master_transfer_handle_t* *psHandle)

Abort a transfer that uses interrupts for master.

Parameters

- psHandle – QUEUEDSPI handle pointer to *qspi_master_transfer_handle_t*.

void QSPI_MasterTransferHandleIRQ(*qspi_master_transfer_handle_t* *psHandle)

QUEUEDSPI Master IRQ handler function.

This function processes the QUEUEDSPI transmit and receive IRQ.

Parameters

- psHandle – QUEUEDSPI handle pointer to *qspi_master_transfer_handle_t*.

void QSPI_SlaveTransferCreateHandle(QSPI_Type *base, *qspi_slave_transfer_handle_t* *psHandle, *qspi_slave_transfer_callback_t* pfCallback, void *pUserData)

Initialize the QUEUEDSPI slave handle.

This function initializes the QUEUEDSPI handle, which can be used for other QUEUEDSPI transactional APIs. Usually, for a specified QUEUEDSPI instance, call this API once to get the initialized handle.

Parameters

- base – QUEUEDSPI peripheral base address.
- psHandle – QUEUEDSPI handle pointer to the *qspi_slave_transfer_handle_t*.
- pfCallback – QUEUEDSPI callback.
- pUserData – Callback function parameter.

status_t QSPI_SlaveTransferNonBlocking(*qspi_slave_transfer_handle_t* *psHandle, *qspi_transfer_t* *psXfer)

Interrupt driven method of QUEUEDSPI slave transfer with completion will be notified by registered callback.

This function transfers data using interrupts for slave. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

- psHandle – Pointer to the *qspi_slave_transfer_handle_t* structure which stores the transfer state.
- psXfer – Pointer to the *qspi_transfer_t* structure.

Returns

status of status_t.

`status_t` QSPI_SlaveTransferGetCount(*qspi_slave_transfer_handle_t* *psHandle, `uint16_t` *pu16Count)

Get the slave transfer count already transmitted/received.

Parameters

- psHandle – Pointer to the `qspi_slave_transfer_handle_t` structure which stores the transfer state.
- pu16Count – The number of bytes transferred by using the non-blocking transaction.

Returns

status of `status_t`.

`void` QSPI_SlaveTransferAbort(*qspi_slave_transfer_handle_t* *psHandle)

Abort a transaction.

Parameters

- psHandle – Pointer to the `qspi_slave_transfer_handle_t` structure which stores the transfer state.

`void` QSPI_SlaveTransferHandleIRQ(*qspi_slave_transfer_handle_t* *psHandle)

QUEUEDSPI slave IRQ handler function.

This function processes the QUEUEDSPI transmit and receive IRQ.

Parameters

- psHandle – Pointer to the `qspi_slave_transfer_handle_t` structure which stores the transfer state.

`FSL_QSPI_DRIVER_VERSION`

QSPI driver version.

`QSPI_TRANSFER_GET_BASE(handle)`

Extract Base Address from handle for master or slave handle.

`QSPI_TRANSFER_GET_USER_DATA(handle)`

Extract user data from handle for master or slave handle.

Status return code for the QUEUEDSPI driver. Only used in transactional layer in this driver.

.

Values:

enumerator `kStatus_QSPI_Busy`
QUEUEDSPI transfer is busy.

enumerator `kStatus_QSPI_Error`
QUEUEDSPI driver error.

enumerator `kStatus_QSPI_Idle`
QUEUEDSPI is idle.

enumerator `kStatus_QSPI_OutOfRange`
QUEUEDSPI transfer out of range.

`enum` `_qspi_status_flags`

QUEUEDSPI peripheral status flags.

Values:

enumerator kQSPI_TxEmptyFlag
Transmitter Empty Flag.

enumerator kQSPI_ModeFaultFlag
Mode Fault Flag.

enumerator kQSPI_RxOverflowFlag
Receiver Overflow Flag.

enumerator kQSPI_RxFullFlag
Receiver Full Flag.

enumerator kQSPI_AllStatusFlags

enum _qspi_interrupt_enable
QUEUEDSPI interrupt source.

Values:

enumerator kQSPI_TxInterruptEnable
SPTE interrupt enable.

enumerator kQSPI_RxInterruptEnable
SPRF interrupt enable.

enumerator kQSPI_RxOverFlowInterruptEnable
Bus error interrupt enable.

enumerator kQSPI_AllInterrupts

enum _qspi_ss_direction
options for Slave Select (SSB) signal direction.

Values:

enumerator kQSPI_SlaveSelectDirectionInput
SSB signal as input for slave mode or master mode with Mode fault enabled.

enumerator kQSPI_SlaveSelectDirectionOutput
SSB signal as output.

enum _qspi_ss_data_logic_level
logical level for Slave Select (SSB) signal data

Values:

enumerator kQSPI_SlaveSelectLogicLow
Slave select logic level low

enumerator kQSPI_SlaveSelectLogicHigh
Slave select logic level high

enum _qspi_txfifo_watermark
QUEUEDSPI Transmit FIFO watermark settings.

Values:

enumerator kQSPI_TxFifoWatermarkEmpty
Transmit interrupt active when Tx FIFO is empty

enumerator kQSPI_TxFifoWatermarkOneWord
Transmit interrupt active when Tx FIFO has one or fewer words available

enumerator kQSPI_TxFifoWatermarkTwoWord
Transmit interrupt active when Tx FIFO has two or fewer words available

enumerator kQSPI_TxFifoWatermarkThreeWord

Transmit interrupt active when Tx FIFO has three or fewer words available

enum __qspi_rxfifo_watermark

QUEUEDSPI Receive FIFO watermark settings.

Values:

enumerator kQSPI_RxFifoWatermarkOneWord

Receive interrupt active when Rx FIFO has at least one word used

enumerator kQSPI_RxFifoWatermarkTwoWord

Receive interrupt active when Rx FIFO has at least two words used

enumerator kQSPI_RxFifoWatermarkThreeWord

Receive interrupt active when Rx FIFO has at least three words used

enumerator kQSPI_RxFifowatermarkFull

Receive interrupt active when Rx FIFO is full

enum __qspi_data_width

Transfer data width in each frame.

Values:

enumerator kQSPI_Data2Bits

2 bits data width

enumerator kQSPI_Data3Bits

3 bits data width

enumerator kQSPI_Data4Bits

4 bits data width

enumerator kQSPI_Data5Bits

5 bits data width

enumerator kQSPI_Data6Bits

6 bits data width

enumerator kQSPI_Data7Bits

7 bits data width

enumerator kQSPI_Data8Bits

8 bits data width

enumerator kQSPI_Data9Bits

9 bits data width

enumerator kQSPI_Data10Bits

10 bits data width

enumerator kQSPI_Data11Bits

11 bits data width

enumerator kQSPI_Data12Bits

12 bits data width

enumerator kQSPI_Data13Bits

13 bits data width

enumerator kQSPI_Data14Bits

14 bits data width

enumerator kQSPI_Data15Bits
15 bits data width

enumerator kQSPI_Data16Bits
16 bits data width

enum `_qspi_dma_enable_flags`
QUEUEDSPI DMA configuration for Transmit and Receive.

Values:

enumerator kQSPI_DmaRx
Receive DMA Enable Flag.

enumerator kQSPI_DmaTx
Transmit DMA Enable Flag.

enum `_qspi_master_slave_mode`
QUEUEDSPI master or slave mode configuration.

Values:

enumerator kQSPI_Slave
QUEUEDSPI peripheral operates in slave mode.

enumerator kQSPI_Master
QUEUEDSPI peripheral operates in master mode.

enum `_qspi_clock_polarity`
QUEUEDSPI clock polarity configuration.

Values:

enumerator kQSPI_ClockPolarityActiveRisingEdge
CPOL=0. Active-high QUEUEDSPI clock (idles low), rising edge of SCLK starts transaction.

enumerator kQSPI_ClockPolarityActiveFallingEdge
CPOL=1. Active-low QUEUEDSPI clock (idles high), falling edge of SCLK starts transaction.

enum `_qspi_clock_phase`
QUEUEDSPI clock phase configuration.

Values:

enumerator kQSPI_ClockPhaseSlaveSelectHighBetweenWords
CPHA=0, Slave Select toggle high during data frames.

enumerator kQSPI_ClockPhaseSlaveSelectLowBetweenWords
CPHA=1, Slave Select keep low during data frames.

enum `_qspi_data_shift_direction`
QUEUEDSPI data shifter direction options for a given CTAR.

Values:

enumerator kQSPI_MsbFirst
Data transfers start with most significant bit.

enumerator kQSPI_LsbFirst
Data transfers start with least significant bit.

enum `_qspi_pcs_polarity_config`

QUEUEDSPI Peripheral Chip Select Polarity configuration.

Values:

enumerator `kQSPI_PcsActiveHigh`

Pcs Active High (idles low).

enumerator `kQSPI_PcsActiveLow`

Pcs Active Low (idles high).

enum `_qspi_master_transfer_flag`

transaction layer configuration options for each transaction

Values:

enumerator `kQSPI_MasterPCSContinuous`

Indicates whether the PCS signal de-asserts during transfer between frames, note this flag should not be used when CPHA is 0.

enumerator `kQSPI_MasterActiveAfterTransfer`

Indicates whether the PCS signal is active after the last frame transfer, note 1. this flag should not be used when CPHA is 0, 2. this flag can only be used when `kQSPI_MasterPCSContinuous` is used.

enum `_qspi_transfer_state`

QUEUEDSPI transfer state, used internally for transactional layer.

Values:

enumerator `kQSPI_Idle`

Nothing in the transmitter/receiver.

enumerator `kQSPI_Busy`

Transfer queue is not finished.

enumerator `kQSPI_Error`

Transfer error.

typedef enum `_qspi_ss_direction` `qspi_ss_direction_t`

options for Slave Select (SSB) signal direction.

typedef enum `_qspi_ss_data_logic_level` `qspi_ss_data_logic_level_t`

logical level for Slave Select (SSB) signal data

typedef enum `_qspi_txfifo_watermark` `qspi_txfifo_watermark_t`

QUEUEDSPI Transmit FIFO watermark settings.

typedef enum `_qspi_rxfifo_watermark` `qspi_rxfifo_watermark_t`

QUEUEDSPI Receive FIFO watermark settings.

typedef enum `_qspi_data_width` `qspi_data_width_t`

Transfer data width in each frame.

typedef enum `_qspi_master_slave_mode` `qspi_master_slave_mode_t`

QUEUEDSPI master or slave mode configuration.

typedef enum `_qspi_clock_polarity` `qspi_clock_polarity_t`

QUEUEDSPI clock polarity configuration.

typedef enum `_qspi_clock_phase` `qspi_clock_phase_t`

QUEUEDSPI clock phase configuration.

typedef enum *_qspi_data_shift_direction* qspi_data_shift_direction_t
 QUEUEDSPI data shifter direction options for a given CTAR.

typedef struct *_qspi_master_config* qspi_master_config_t
 QUEUEDSPI master configuration structure with all master configuration fields covered.

typedef struct *_qspi_slave_config* qspi_slave_config_t
 QUEUEDSPI slave configuration structure with all slave configuration fields covered.

typedef enum *_qspi_pcs_polarity_config* qspi_pcs_polarity_config_t
 QUEUEDSPI Peripheral Chip Select Polarity configuration.

typedef struct *_qspi_transfer* qspi_transfer_t
 QUEUEDSPI master/slave transfer structure.

typedef struct *_qspi_master_handle* qspi_master_transfer_handle_t
 Forward declaration of the *_qspi_master_handle* typedefs. .

typedef void (*qspi_master_transfer_callback_t)(qspi_master_transfer_handle_t *psHandle, status_t eCompletionStatus, void *pUserData)

Completion callback function pointer type.

Param base

QUEUEDSPI peripheral address.

Param psHandle

Pointer to the handle for the QUEUEDSPI master.

Param eCompletionStatus

Success or error code describing whether the transfer completed.

Param pUserData

Arbitrary pointer-dataSized value passed from the application.

typedef struct *_qspi_slave_transfer_handle* qspi_slave_transfer_handle_t
 Forward declaration of the *_qspi_slave_transfer_handle* typedefs. .

typedef void (*qspi_slave_transfer_callback_t)(qspi_slave_transfer_handle_t *psHandle, status_t eCompletionStatus, void *pUserData)

Completion callback function pointer type.

Param base

QUEUEDSPI peripheral address.

Param handle

Pointer to the handle for the QUEUEDSPI slave.

Param status

Success or error code describing whether the transfer completed.

Param pUserData

Arbitrary pointer-dataSized value passed from the application.

void QSPI_DriverIRQHandler(uint32_t instance)

QSPI driver IRQ handler common entry.

This function provides the common IRQ request entry for QSPI.

Parameters

- instance – QSPI instance.

QSPI_DUMMY_DATA

User Configuraiton item dummy data filled into Output signal if there is no Tx data.

Dummy data used for Tx if there is no txData.

`struct _qspi_master_config`
#include <fsl_queued_spi.h> QUEUEDSPI master configuration structure with all master configuration fields covered.

Public Members

`uint32_t u32BaudRateBps`
Baud Rate for QUEUEDSPI.

`qspi_data_width_t eDataWidth`
Data width in SPI transfer

`qspi_clock_polarity_t eClkPolarity`
Clock polarity.

`qspi_clock_phase_t eClkPhase`
Clock phase.

`qspi_data_shift_direction_t eShiftDirection`
MSB or LSB data shift direction.

`bool bEnableWiredOrMode`
SPI pin configuration, when enabled the SPI pins are configured as open-drain drivers with the pull-ups disabled.

`bool bEnableModeFault`
Enable/Disable mode fault detect for Slave Select Signal

`bool bEnableStrobe`
Enable/Disable strobe between data frames irrespective of clock pahse setting

`bool bEnableSlaveSelAutoMode`
Enable/Disable Slave Select Auto mode.

`bool bEnableSlaveSelOpenDrain`
Enable the open-drain mode of SPI Pins, otherwise Push-Pull

`uint16_t u16DelayBetweenFrameInCLK`
The delay between frame.

`bool bEnableFIFO`
Enable / Disable FIFO for Transmit/Receive

`qspi_txfifo_watermark_t eTxFIFOWatermark`
Watermark config for Transmit FIFO

`qspi_rxfifo_watermark_t eRxFIFOWatermark`
Watermark config for Receive FIFO

`bool bEnableModule`
Enable / Disable module

`uint8_t u8Interrupts`
Interrupt enabled ORed from `_qspi_interrupt_enable`

`uint8_t u8DmaEnableFlags`
Configure DMA Enable/Disable for Transmit/Receive

`struct _qspi_slave_config`
#include <fsl_queued_spi.h> QUEUEDSPI slave configuration structure with all slave configuration fields covered.

Public Members

qspi_data_width_t eDataWidth

Data width in SPI transfer

qspi_clock_polarity_t eClkPolarity

Clock polarity.

qspi_clock_phase_t eClkPhase

Clock phase.

qspi_data_shift_direction_t eShiftDirection

MSB or LSB data shift direction.

bool bEnableWiredOrMode

SPI pin configuration, when enabled the SPI pins are configured as open-drain drivers with the pull-ups disabled.

bool bEnableModeFault

Enable/Disable mode fault detect for Slave Select Signal

bool bEnableSlaveSelOverride

Enable/Disable override Slave Select (SS) signal with Master/Slave Mode config

bool bEnableFIFO

Enable / Disable FIFO for Transmit/Receive

qspi_txfifo_watermark_t eTxFIFOWatermark

Watermark config for Transmit FIFO

qspi_rxfifo_watermark_t eRxFIFOWatermark

Watermark config for Receive FIFO

bool bEnableModule

Enable/Disable Module

uint8_t u8DmaEnableFlags

Configure DMA Enable/Disable for Transmit/Receive

struct *_qspi_transfer*

#include <fsl_queued_spi.h> QUEUEDSPI master/slave transfer structure.

Public Members

void *pTxData

Transmit buffer.

void *pRxData

Receive buffer.

volatile uint16_t u16DataSize

Transfer bytes.

uint8_t u8ConfigFlags

Transfer configuration flags; set from *_qspi_master_transfer_flag*. This is not used in slave transfer.

struct *_qspi_master_handle*

#include <fsl_queued_spi.h> QUEUEDSPI master transfer handle structure used for transactional API.

Public Members

QSPI_Type *base

Base address for the QSPI peripheral

qspi_data_width_t eDataWidth

The desired number of bits per frame.

volatile bool bIsPcsActiveAfterTransfer

Indicates whether the PCS signal is active after the last frame transfer, This is not used in slave transfer.

uint8_t *volatile pu8TxData

Send buffer.

uint8_t *volatile pu8RxData

Receive buffer.

volatile uint16_t u16RemainingSendByteCount

A number of bytes remaining to send.

volatile uint16_t u16RemainingReceiveByteCount

A number of bytes remaining to receive.

uint16_t u16TotalByteCount

A number of transfer bytes

volatile uint8_t u8State

QUEUEDSPI transfer state, see *_qspi_transfer_state*.

qspi_master_transfer_callback_t pfCallback

Completion callback.

void *pUserData

Callback user data.

volatile uint16_t u16ErrorCount

Error count for slave transfer, this is not used in master transfer.

2.63 QSPI Peripheral and Driver Overview

2.64 QSPI_EDMA: EDMA based QSPI Driver

FSL_QSPI_EDMA_DRIVER_VERSION

QSPI EDMA driver version.

typedef struct *_qspi_master_edma_handle* *qspi_master_edma_handle_t*

Forward declaration of the *_qspi_master_edma_handle* typedefs.

typedef struct *_qspi_master_edma_handle* *qspi_slave_edma_handle_t*

Forward declaration of the *_qspi_master_edma_handle* typedefs.

typedef void (**qspi_edma_transfer_callback_t*)(*qspi_master_edma_handle_t* *psHandle, *status_t* eCompletionStatus, void *pUserData)

Completion callback function pointer type.

Param base

QUEUEDSPI peripheral base address.

Param psHandle

Pointer to the handle for the QUEUEDSPI master.

Param eCompletionStatus

Success or error code describing whether the transfer completed.

Param pUserData

Arbitrary pointer-dataSized value passed from the application.

```
void QSPI_MasterTransferCreateHandleEDMA(QSPI_Type *base, qspi_master_edma_handle_t
    *psHandle, qspi_edma_transfer_callback_t
    pfCallback, void *pUserData, DMA_Type
    *psEdmaBase, edma_channel_t eEdmaTxChannel,
    edma_channel_t eEdmaRxChannel)
```

Initialize the QUEUEDSPI master EDMA handle.

This function initializes the QUEUEDSPI EDMA master handle which can be used for QUEUEDSPI EDMA master transactional APIs. Usually, for a specified QUEUEDSPI instance, call this API once to get the initialized handle.

Parameters

- base – QUEUEDSPI peripheral base address.
- psHandle – QUEUEDSPI handle pointer to qspi_master_edma_handle_t.
- pfCallback – QUEUEDSPI callback.
- pUserData – callback function parameter.
- psEdmaBase – base address for the EDMA
- eEdmaTxChannel – Channel of the EDMA used for QSPI Tx
- eEdmaRxChannel – Channel of the EDMA used for QSPI Rx

```
status_t QSPI_MasterTransferEDMA(qspi_master_edma_handle_t *psHandle, qspi_transfer_t
    *psXfer)
```

EDMA method of QUEUEDSPI master transfer.

This function transfers data using EDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: : The transfer data size should be even, if the transfer data width is larger than 8.

Parameters

- psHandle – pointer to qspi_master_edma_handle_t structure which stores the transfer state.
- psXfer – pointer to qspi_transfer_t structure.

Returns

status of status_t.

```
status_t QSPI_MasterTransferGetCountEDMA(qspi_master_edma_handle_t *psHandle, uint16_t
    *pu16Count)
```

Get the master EDMA transfer count.

Parameters

- psHandle – Pointer to the qspi_master_edma_handle_t structure which stores the transfer state.
- pu16Count – The number of bytes transferred by using the EDMA transaction.

Returns

status of status_t.

```
void QSPI_MasterTransferAbortEDMA(qspi_master_edma_handle_t *psHandle)
```

Abort a transfer that uses EDMA for master.

Parameters

- psHandle – Pointer to the *qspi_master_edma_handle_t* structure which stores the transfer state.

```
void QSPI_SlaveTransferCreateHandleEDMA(QSPI_Type *base, qspi_slave_edma_handle_t
                                        *psHandle, qspi_edma_transfer_callback_t
                                        pfCallback, void *pUserData, DMA_Type
                                        *psEdmaBase, edma_channel_t eEdmaTxChannel,
                                        edma_channel_t eEdmaRxChannel)
```

Initialize the QUEUEDSPI slave EDMA handle.

This function initializes the QUEUEDSPI EDMA handle which can be used for other QUEUEDSPI transactional APIs. Usually, for a specified QUEUEDSPI instance, call this API once to get the initialized handle.

Parameters

- base – QUEUEDSPI peripheral base address.
- psHandle – QUEUEDSPI handle pointer to *qspi_slave_edma_handle_t*.
- pfCallback – QUEUEDSPI callback.
- pUserData – callback function parameter.
- psEdmaBase – base address for the EDMA
- eEdmaTxChannel – Channel of the EDMA used for QSPI Tx
- eEdmaRxChannel – Channel of the EDMA used for QSPI Rx

```
status_t QSPI_SlaveTransferEDMA(qspi_slave_edma_handle_t *psHandle, qspi_transfer_t
                                *psXfer)
```

EDMA method of QUEUEDSPI slave transfer.

This function transfers data using EDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: : The transfer data size should be even if the transfer data width is larger than 8.

Parameters

- psHandle – pointer to *qspi_slave_edma_handle_t* structure which stores the transfer state.
- psXfer – pointer to *qspi_transfer_t* structure.

Returns

status of status_t.

```
status_t QSPI_SlaveTransferGetCountEDMA(qspi_slave_edma_handle_t *psHandle, uint16_t
                                        *pu16Count)
```

Get the slave EDMA transfer count.

Parameters

- psHandle – Pointer to the *qspi_slave_edma_handle_t* structure which stores the transfer state.

- `u16Count` – The number of bytes transferred by using the EDMA transaction.

Returns

status of `status_t`.

```
void QSPI_SlaveTransferAbortEDMA(qspi_slave_edma_handle_t *psHandle)
```

Abort a transfer that uses EDMA for slave.

Parameters

- `psHandle` – Pointer to the `qspi_slave_edma_handle_t` structure which stores the transfer state.

```
struct _qspi_master_edma_handle
```

#include <fsl_queued_spi_edma.h> QUEUEDSPI master EDMA transfer handle structure used for transactional API. This struct should be `sizeof(edma_channel_tcd_t)` aligned.

Public Members

```
QSPI_Type *base
```

Base address of the QSPI Peripheral

```
volatile uint8_t u8State
```

QUEUEDSPI transfer state , defined in `_qspi_transfer_state`.

```
uint16_t u16TotalByteCount
```

A number of transfer bytes.

```
qspi_data_width_t eDataWidth
```

The desired number of bits per frame.

```
uint16_t u16TxDummyData
```

Used if `txData` is NULL.

```
uint16_t u16RxDummyData
```

Used if `rxData` is NULL.

```
edma_handle_t sTxHandle
```

`edma_handle_t` handle point used for transmitting data.

```
edma_handle_t sRxHandle
```

`edma_handle_t` handle point used for receiving data.

```
bool bIsTxInProgress
```

Indicates whether the transmit is in progress.

```
bool bIsRxInProgress
```

Indicates whether the receive is in progress.

```
qspi_edma_transfer_callback_t pfCallback
```

Completion callback.

```
void *pUserData
```

Callback user data.

```
volatile bool bIsPcsActiveAfterTransfer
```

Indicates whether the PCS signal is active after the last frame transfer, This is not used in slave transfer.

2.65 QTMR: Quad Timer Driver

`void QTMR_Init(TMR_Type *base, const qtmr_config_t *psConfig)`

Initialization Quad Timer module with provided structure.

This function can initial one or more channels of the Quad Timer module.

This examples shows how only initial channel 0.

```
qtmr_config_t sConfig = {0};
qtmr_channel_config_t sChannel0Config;
sConfig.psChannelConfig[0] = &sChannel0Config;
QTMR_GetChannelDefaultConfig(&sChannel0Config);
QTMR_Init(QTMR, sConfig);
```

Note: This API should be called at the beginning of the application using the Quad Timer module.

Parameters

- `base` – Quad Timer peripheral base address.
- `psConfig` – Pointer to user's Quad Timer config structure. See `qtmr_config_t`.

`void QTMR_Deinit(TMR_Type *base)`

De-initialization Quad Timer module.

Parameters

- `base` – Quad Timer peripheral base address.

`void QTMR_GetChannelDefaultConfig(qtmr_channel_config_t *psConfig)`

Gets an available pre-defined options for Quad Timer channel module's configuration.

This function initializes the channel configuration structure with a free run 16bit timer work setting. The default values are:

```
psConfig->sInputConfig.ePrimarySource = kQTMR_PrimarySrcIPBusClockDivide2;
psConfig->sInputConfig.eSecondarySource = kQTMR_SecondarySrcInputPin0;
psConfig->sInputConfig.eSecondarySourceCaptureMode = kQTMR_SecondarySrcCaptureNoCapture;
psConfig->sInputConfig.bEnableSecondarySrcFaultFunction = false;
psConfig->sInputConfig.eEnableInputInvert = false;
psConfig->sCountConfig.eCountMode = kQTMR_CountPrimarySrcRiseEdge;
psConfig->sCountConfig.eCountLength = kQTMR_CountLengthUntilRollOver;
psConfig->sCountConfig.eCountDir = kQTMR_CountDirectionUp;
psConfig->sCountConfig.eCountTimes = kQTMR_CountTimesRepeat;
psConfig->sCountConfig.eCountLoadMode = kQTMR_CountLoadNormal;
psConfig->sCountConfig.eCountPreload1 = kQTMR_CountPreloadNoLoad;
psConfig->sCountConfig.eCountPreload2 = kQTMR_CountPreloadNoLoad;
psConfig->sOutputConfig.eOutputMode = kQTMR_OutputAssertWhenCountActive;
psConfig->sOutputConfig.eOutputValueOnForce = kQTMR_OutputValueClearOnForce;
psConfig->sOutputConfig.bEnableOutputInvert = false;
psConfig->sOutputConfig.bEnableSwForceOutput = false;
psConfig->sOutputConfig.bEnableOutputPin = false;
psConfig->sCooperationConfig.bEnableMasterReInit = false;
psConfig->sCooperationConfig.bEnableMasterForceOFLAG = false;
psConfig->sCooperationConfig.bEnableMasterMode = false;
psConfig->eDebugMode = kQTMR_DebugRunNormal;
psConfig->u16EnabledInterruptMask = 0x0U;
psConfig->u16EnabledDMAMask = 0x0U;
psConfig->u16Comp1 = 0x0U;
```

(continues on next page)

(continued from previous page)

```

psConfig->u16Comp2 = 0x0U;
psConfig->u16Comp1Preload = 0x0U;
psConfig->u16Comp1Preload = 0x0U;
psConfig->u16Load = 0x0U;
psConfig->u16Count = 0x0U;
psConfig->bEnableChannel = false;

```

Parameters

- psConfig – Pointer to user’s Quad Timer channel config structure. See `qtmr_channel_config_t`.

```
void QTMR_SetupChannleConfig(TMR_Type *base, qtmr_channel_number_t eChannelNumber,
                           const qtmr_channel_config_t *psConfig)
```

Setup a Quad Timer channel with provided structure.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- psConfig – Pointer to user’s Quad Timer channel config structure. See `qtmr_channel_config_t`.

```
static inline void QTMR_SetPrimaryCountSource(TMR_Type *base, qtmr_channel_number_t
                                             eChannelNumber,
                                             qtmr_channel_primary_count_source_t
                                             ePrimarySource)
```

Sets primary input source.

This function select the primary input source, it can select from “input pin 0~3”, “channel output

0~3” and “IP bus clock prescaler”.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- ePrimarySource – The primary input source. See `qtmr_channel_primary_count_source_t`.

```
static inline void QTMR_SetSecondaryCountSource(TMR_Type *base, qtmr_channel_number_t
                                              eChannelNumber,
                                              qtmr_channel_secondary_count_source_t
                                              source)
```

Sets secondary input source.

This function select the secondary input source, it can select from “input pin 0~3”.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.

- `source` – The Secondary input source. See `qtmr_channel_secondary_count_source_t`.

```
void QTMR_SetSecondarySourceInputCaptureMode(TMR_Type *base, qtmr_channel_number_t
                                              eChannelNumber,
                                              qtmr_channel_secondary_source_capture_mode_t
                                              eCaptureMode)
```

Sets secondary input capture mode.

This function select the capture mode for secondary input, it can select from “disable capture”, “capture on

rising/falling edge” and “capture on both edges”. Need enable capture mode when input edge interrupt is needed.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eCaptureMode` – The capture mode of secondary input. See `qtmr_channel_secondary_source_capture_mode_t`.

```
static inline void QTMR_EnableSecondarySourceFault(TMR_Type *base, qtmr_channel_number_t
                                                  eChannelNumber, bool bEnable)
```

Enables/Disables secondary input source signal fault feature.

Enable fault feature will make secondary input acts as a fault signal so that the channel output signal (OFLAG) is cleared when the secondary input is set.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `bEnable` – Enable the feature or not.
 - **true** Enable secondary source fault feature.
 - **false** Disable secondary source fault feature.

```
static inline void QTMR_EnableInputInvert(TMR_Type *base, qtmr_channel_number_t
                                          eChannelNumber, bool bEnable)
```

Enables/Disables input pin signal polarity invert feature.

This function enables/disables input pin signal polarity invert feature.

Note: Invert feature only affects “input pin 0~3”, and acts on the channel input node, not the input pin, so it only affect current channel and not share by other channel

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `bEnable` – Enable the feature or not.
 - **true** Invert input pin signal polarity.

- **false** No invert for input pin signal polarity.

```
static inline void QTMR_SetInputFilter(TMR_Type *base, qtmr_input_pin_t ePin, uint8_t
                                     u8Count, uint8_t u8Period)
```

Sets input filter for one input pin.

Sets input filter if the input signal is noisy.

Note: The input filter acts on the input pin directly, so the input filter config will affect all channels that select this input pin as source. Turning on the input filter (setting `FILT_PER` to a non-zero value) introduces a latency of $((u8Count + 3) \times u8Period) + 2$ IP bus clock periods.

Parameters

- `base` – Quad Timer peripheral base address.
- `ePin` – Quad Timer input pin number. See `qtmr_input_pin_t`.
- `u8Count` – Range is 0~7, represent the number of consecutive samples that must agree prior to the input filter accepting an input transition. Actual consecutive samples numbers is $(u8Count + 3)$.
- `u8Period` – Represent the sampling period (in IP bus clock cycles) of the input pin signals. Each input is sampled multiple times at the rate specified by this field. If `u8Period` is 0, then the input pin filter is bypassed.

```
static inline uint16_t QTMR_GetInputPinValueInSecondarySource(TMR_Type *base,
                                                            qtmr_channel_number_t
                                                            eChannelNumber)
```

Gets the external input signal value selected via the secondary input source.

This function read the value of the secondary input source, the input pin IPS and filtering have been applied to the read back value.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

The state of the current state of the external input pin selected via the secondary count source after application of IPS and filtering.

```
void QTMR_SetCountMode(TMR_Type *base, qtmr_channel_number_t eChannelNumber,
                      qtmr_channel_count_mode_t eCountMode)
```

Sets channel count mode.

This function select channel basic count mode which trigger by primary input or/and secondary input events.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eCountMode` – The mode of operation for the count. See `qtmr_channel_count_mode_t`.

```
static inline void QTMR_SetCountLength(TMR_Type *base, qtmr_channel_number_t
                                     eChannelNumber, qtmr_channel_count_length_t
                                     eLength)
```

Sets channel count length.

This function select channel single count length from “until roll over” or “until compare”. “until roll over” means count until 0xFFFF, “until compare” means count until reach COMP1 (for count up) or COMP2 (for count up) value (unless the output signal is in alternating compare mode, this mode make channel use COMP1 and COMP2 alternately).

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See *qtmr_channel_number_t*.
- eLength – The channel count length. See *qtmr_channel_count_length_t*.

```
static inline void QTMR_SetCountDirection(TMR_Type *base, qtmr_channel_number_t
                                          eChannelNumber, qtmr_channel_count_direction_t
                                          eDirection)
```

Sets channel count direction.

This function select channel count direction from “count up” or “count down”. Under normal count mode, this function decide the count direction directly, when chose “secondary specifies direction” count mode, count direction decide by “the secondary input level” XOR with “the function selection”.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See *qtmr_channel_number_t*.
- eDirection – The channel count direction. See *qtmr_channel_count_direction_t*.

```
static inline qtmr_channel_count_direction_t QTMR_GetCountDirection(TMR_Type *base,
                                                                    qtmr_channel_number_t
                                                                    eChannelNumber)
```

Gets channel count direction.

This function read the channel count direction of the last count during quadrature encoded count mode.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See *qtmr_channel_number_t*.

Returns

The direction of the last count. Value see *qtmr_channel_count_direction_t*.

```
static inline void QTMR_SetCountTimes(TMR_Type *base, qtmr_channel_number_t
                                      eChannelNumber, qtmr_channel_count_times_t
                                      eTimes)
```

Sets channel count times.

This function select channel count times from “once” or “repeatedly”. If select “once” with “until compare”, channel will stop when reach COMP1 (for count up) or COMP2 (for count up) (unless the output signal is in alternating compare mode, this mode will make channel reaching COMP1, re-initializes then count reaching COMP2, and then stops).

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eTimes` – The channel count times. See `qtmr_channel_count_times_t`.

```
static inline void QTMR_SetCountLoadMode(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber,
                                         qtmr_channel_count_load_mode_t eLoadMode)
```

Sets channel count load mode.

This function select channel count re-initialized load mode from “normal” or “alternative”. “normal” means channel counter re-initialized from LOAD register when compare event, “alternative” means channel counter can re-initialized from LOAD (count up) or CMPLD2 (count down) when compare event.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eLoadMode` – The channel count load mode. See `qtmr_channel_count_load_mode_t`.

```
static inline void QTMR_SetCompare1PreloadControl(TMR_Type *base, qtmr_channel_number_t
                                                  eChannelNumber,
                                                  qtmr_channel_count_preload_mode_t
                                                  ePreloadMode)
```

Sets channel preload mode for compare register 1.

This function select channel preload mode for compare register 1. Default the COMP1 register never preload, when enabled, the COMP1 can preload from CMPLD1 register when COMP1 or COMP2 compare event.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `ePreloadMode` – The compare register 1 preload mode. See `qtmr_channel_count_preload_mode_t`.

```
static inline void QTMR_SetCompare2PreloadControl(TMR_Type *base, qtmr_channel_number_t
                                                  eChannelNumber,
                                                  qtmr_channel_count_preload_mode_t
                                                  ePreloadMode)
```

Sets channel preload mode for compare register 2.

This function select channel preload mode for compare register 2. Default the COMP2 register never preload, when enabled, the COMP2 can preload from CMPLD2 register when COMP1 or COMP2 compare event.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `ePreloadMode` – The compare register 2 preload mode. See `qtmr_channel_count_preload_mode_t`.

```
static inline void QTMR_SetCompare1PreloadValue(TMR_Type *base, qtmr_channel_number_t
                                                eChannelNumber, uint16_t
                                                u16Comp1Preload)
```

Sets channel compare register 1 preload register value.

This function set the CMPLD1 register value. The COMP1 can preload from CMPLD1 register when preload mode is not “never preload”.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Comp1Preload – Value for Channel compare register 1 preload register.

```
static inline void QTMR_SetCompare2PreloadValue(TMR_Type *base, qtmr_channel_number_t
                                                eChannelNumber, uint16_t
                                                u16Comp2Preload)
```

Sets channel compare register 2 preload register value.

This function set the CMPLD2 register value. The COMP2 can preload from CMPLD2 register when preload mode is not “never preload”.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Comp2Preload – Value for Channel compare register 2 preload register.

```
static inline void QTMR_SetLoadValue(TMR_Type *base, qtmr_channel_number_t
                                     eChannelNumber, uint16_t u16Load)
```

Sets channel load register value.

This function set the LOAD register value. The channel will re-initialize the counter value with this register after counter compare or overflow event.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Load – Value used to initialize the counter after counter compare or overflow event.

```
static inline void QTMR_SetCompare1Value(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber, uint16_t u16Comp1)
```

Sets channel count compare register 1.

This function set the COMP1 register value. It use to trigger compare event in count up mode or alternating compare mode.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Comp1 – Value for Channel compare register 1.

```
static inline void QTMR_SetCompare2Value(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber, uint16_t u16Comp2)
```

Sets channel count compare register 2.

This function set the COMP2 register value. It use to trigger compare event in count down mode or alternating compare mode.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Comp2 – Value for Channel compare register 2.

```
static inline uint16_t QTMR_ReadCaptureValue(TMR_Type *base, qtmr_channel_number_t
                                             eChannelNumber)
```

Gets channel capture register value.

This function read the CAPT register value, which store the real-time channel counter value when input capture event.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

The value captured from the channel counter.

```
static inline uint16_t QTMR_GetHoldValue(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber)
```

Gets channel hold register value.

This function read the HOLD register value, which stores the channel counter's values of specific channels whenever any of the four channels within a module is read.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

The channel counter value when any read operation occurs.

```
static inline void QTMR_SetCounterValue(TMR_Type *base, qtmr_channel_number_t
                                        eChannelNumber, uint16_t u16Count)
```

Sets channel counter register value.

This function set the CNTR register value, the channel will start counting based on this value.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- u16Count – The channel counter initialize value.

```
static inline uint16_t QTMR_GetCounterValue(TMR_Type *base, qtmr_channel_number_t
                                          eChannelNumber)
```

Reads channel counter register value.

This function read the CNTR register value, which stores the channel real-time channel counting value. This read operation will trigger HOLD register update.

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.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

The real-time channel counter value.

```
static inline void QTMR_SetOutputMode(TMR_Type *base, qtmr_channel_number_t
                                      eChannelNumber, qtmr_channel_output_mode_t
                                      eOutputMode)
```

Sets Channel output signal (OFLAG) work mode.

This function select channel output signal (OFLAG) work mode base on different channel event.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eOutputMode` – The mode of operation for the OFLAG output signal. See `qtmr_channel_output_mode_t`.

```
static inline void QTMR_SetOutputValueOnForce(TMR_Type *base, qtmr_channel_number_t
                                              eChannelNumber,
                                              qtmr_channel_output_value_on_force_t eValue)
```

Sets the value of output signal when a force event occurs.

This function config the value of output signal when a force event occurs. Force events can be a software command or compare event from a master channel.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `eValue` – The value of output signal when force event occur. See `qtmr_channel_output_value_on_force_t`.

```
static inline void QTMR_EnableOutputInvert(TMR_Type *base, qtmr_channel_number_t
                                          eChannelNumber, bool bEnable)
```

Enables/Disables output signal polarity invert feature.

This function enables/disables the invert feature of output signal (OFLAG).

Parameters

- `base` – Quad Timer peripheral base address.

- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- bEnable – Enable the feature or not.
 - **true** Invert output signal polarity.
 - **false** No invert for output signal polarity.

```
static inline void QTMR_EnableSwForceOutput(TMR_Type *base, qtmr_channel_number_t
                                           eChannelNumber)
```

Enables software triggers a FORCE command to output signal.

This function uses a software command to trigger force event, which can force the current value of SCTRL[VAL] bit to be written to the OFLAG output.

Note: This function can be called only if the counter is disabled.

```
QTMR_SetOutputValueOnForce(QTMR, kQTMR_Channel0, kQTMR_OutputValueSetOnForce);
QTMR_EnableSwForceOutput(QTMR, kQTMR_Channel0);
```

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.

```
static inline void QTMR_EnableOutputPin(TMR_Type *base, qtmr_channel_number_t
                                       eChannelNumber, bool bEnable)
```

Enables/Disables output signal (OFLAG) drive on the external pin feature.

This function enables/disables output signal (OFLAG) drive on the external pin feature.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- bEnable – Enable the feature or not.
 - **true** The output signal is driven on the external pin.
 - **false** The external pin is configured as an input.

```
static inline void QTMR_EnableMasterMode(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber, bool bEnable)
```

Enables/Disables channel master mode.

This function enables/disables channel master mode.

Note: Master channel can broadcast compare event to all channels within the module to re-initialize channel and/or force channel output signal.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- bEnable – Enable the feature or not.

- **true** Enables channel master mode.
- **false** Disables channel master mode.

```
static inline void QTMR_EnableMasterForceOFLAG(TMR_Type *base, qtmr_channel_number_t  
                                              eChannelNumber, bool bEnable)
```

Enables/Disables force the channel output signal (OFLAG) state by master channel compare event.

This function enables/disables the compare event from master channel within the same module to force the state of this channel OFLAG output signal.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- bEnable – Enable the feature or not.
 - **true** Enables OFLAG state to be forced by master channel compare event.
 - **false** Disables OFLAG state to be forced by master channel compare event.

```
static inline void QTMR_EnableMasterReInit(TMR_Type *base, qtmr_channel_number_t  
                                          eChannelNumber, bool bEnable)
```

Enables/Disables channel be re-initialized by master channel compare event feature.

This function enables/disables the compare event from master channel within the same module to force the re-initialization of this channel.

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.
- bEnable – Enable the feature or not.
 - **true** Enables channel be re-initialized by master channel compare event.
 - **false** Disables channel be re-initialized by master channel compare event.

```
static inline void QTMR_EnableDma(TMR_Type *base, qtmr_channel_number_t  
                                  eChannelNumber, uint16_t u16Mask)
```

Enables the Quad Timer DMA request according to a provided mask.

This function enables the Quad Timer DMA request according to a provided mask. The mask is a logical OR of enumerators members. See `_qtmr_channel_dma_enable`. This examples shows how to enable compare 1 register preload DMA request and compare 2 register preload DMA request.

```
QTMR_EnableDma((QTMR, kQTMR_Channel0, kQTMR_Compare1PreloadDmaEnable | kQTMR_  
->Compare2PreloadDmaEnable);
```

Parameters

- base – Quad Timer peripheral base address.
- eChannelNumber – Quad Timer channel number. See `qtmr_channel_number_t`.

- `u16Mask` – The QTMR DMA requests to enable. Logical OR of `_qtmr_channel_dma_enable`.

```
static inline void QTMR_DisableDma(TMR_Type *base, qtmr_channel_number_t
                                eChannelNumber, uint16_t u16Mask)
```

Disables the Quad Timer DMA request according to a provided mask.

This function disables the Quad Timer DMA request according to a provided mask. The mask is a logical OR of enumerators members. See `_qtmr_channel_dma_enable`. This examples shows how to disable compare 1 register preload DMA request and compare 2 register preload DMA request.

```
QTMR_DisableDma((QTMR, kQTMR_Channel0, kQTMR_Compare1PreloadDmaEnable | kQTMR_
↳ Compare2PreloadDmaEnable);
```

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `u16Mask` – The QTMR DMA requests to disable. Logical OR of `_qtmr_channel_dma_enable`.

```
static inline void QTMR_EnableInterrupts(TMR_Type *base, qtmr_channel_number_t
                                        eChannelNumber, uint16_t u16Mask)
```

Enables the Quad Timer interrupts according to a provided mask.

This function enables the Quad Timer interrupts according to a provided mask. The mask is a logical OR of enumerators members. See `_qtmr_channel_interrupt_enable`. This examples shows how to enable compare 1 interrupt and compare 2 interrupt.

```
QTMR_EnableInterrupts((QTMR, kQTMR_Channel0, kQTMR_Compare1InterruptEnable |
↳ kQTMR_Compare2InterruptEnable);
```

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.
- `u16Mask` – The QTMR DMA interrupts to enable. Logical OR of `_qtmr_channel_interrupt_enable`.

```
static inline void QTMR_DisableInterrupts(TMR_Type *base, qtmr_channel_number_t
                                          eChannelNumber, uint16_t u16Mask)
```

Disables the Quad Timer interrupts according to a provided mask.

This function disables the Quad Timer interrupts according to a provided mask. The mask is a logical OR of enumerators members. See `_qtmr_channel_interrupt_enable`. This examples shows how to disable compare 1 interrupt and compare 2 interrupt.

```
QTMR_DisableInterrupts((QTMR, kQTMR_Channel0, kQTMR_Compare1InterruptEnable |
↳ kQTMR_Compare2InterruptEnable);
```

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.

- `u16Mask` – The QTMR DMA interrupts to disable. Logical OR of `_qtmr_channel_interrupt_enable`.

```
static inline uint16_t QTMR_GetStatusFlags(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber)
```

Gets the Quad Timer status flags.

This function gets all QTMR channel status flags. The flags are returned as the logical OR value of the enumerators `_qtmr_channel_status_flags`. To check for a specific status, compare the return value with enumerators in the `_qtmr_channel_status_flags`. For example, to check whether the compare flag set.

```
if((QTMR_GetStatusFlags(QTMR, kQTMR_Channel0) & kQTMR_CompareFlag) != 0U)
{
    ...
}
```

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

The QTMR status flags which is the logical OR of the enumerators `_qtmr_channel_status_flags`.

```
static inline void QTMR_ClearStatusFlags(TMR_Type *base, qtmr_channel_number_t
                                         eChannelNumber, uint16_t u16Mask)
```

Clears the Quad Timer status flags.

This function clears QTMR channel status flags with a provide mask. The mask is a logical OR of enumerators `_qtmr_channel_status_flags`. This examples shows how to clear compare 1 flag and compare 2 flag.

```
QTMR_ClearStatusFlags((QTMR, kQTMR_Channel0, kQTMR_Compare1Flag | kQTMR_
↳ Compare2Flag);
```

Parameters

- `base` – Quad Timer peripheral base address
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`
- `u16Mask` – The QTMR status flags to clear. Logical OR of `_qtmr_channel_status_flags`

```
static inline void QTMR_SetDebugActions(TMR_Type *base, qtmr_channel_number_t
                                       eChannelNumber, qtmr_channel_debug_action_t
                                       eDebugMode)
```

Sets channel debug actions.

This function selects the certain actions which will perform when the chip entering debug mode.

Parameters

- `base` – Quad Timer peripheral base address.
- `eChannelNumber` – Quad Timer channel number. See `qtmr_channel_number_t`.

- `eDebugMode` – The Quad Timer channel actions in response to the chip entering debug mode. See `qtmr_channel_debug_action_t`.

```
static inline void QTMR_EnableChannels(TMR_Type *base, uint16_t u16Mask)
```

Enables the Quad Timer channels according to a provided mask.

This function enables the Quad Timer channels according to a provided mask. The mask is a logical OR of enumerators `_qtmr_channel_enable`. This examples shows how to enable channel 0 and channel 1.

```
QTMR_EnableChannels(QTMR, kQTMR_Channel0Enable | kQTMR_Channel1Enable);
```

Note: If one channel has effective count mode, it will start its counter as soon as the channel be enabled.

Parameters

- `base` – Quad Timer peripheral base address.
- `u16Mask` – The QTMR channels to enable. Logical OR of `_qtmr_channel_enable`.

```
static inline void QTMR_DisableChannels(TMR_Type *base, uint16_t u16Mask)
```

Disables the Quad Timer channels according to a provided mask.

This function disables the Quad Timer channels according to a provided mask. The mask is a logical OR of enumerators `_qtmr_channel_enable`. This examples shows how to disable channel 0 and channel 1.

```
QTMR_DisableChannels(QTMR, kQTMR_Channel0Enable | kQTMR_Channel1Enable);
```

Parameters

- `base` – Quad Timer peripheral base address.
- `u16Mask` – The QTMR channels to enable. Logical OR of `_qtmr_channel_enable`.

```
static inline uint32_t TMR_GetCaptureRegAddr(TMR_Type *base, qtmr_channel_number_t nChannel)
```

Gets the TMR capture register address. This API is used to provide the transfer address for TMR capture transfer.

Parameters

- `base` – TMR base pointer
- `nChannel` – Quad Timer channel number. See `qtmr_channel_number_t`.

Returns

capture register address

```
FSL_QTMR_DRIVER_VERSION
```

QTMR driver version.

```
enum _qtmr_input_pin
```

The enumeration for Quad Timer module input pin source.

Values:

```
enumerator kQTMR_InputPin0
```

Quad Timer input pin 0.

enumerator kQTMR_InputPin1

Quad Timer input pin 1.

enumerator kQTMR_InputPin2

Quad Timer input pin 2.

enumerator kQTMR_InputPin3

Quad Timer input pin 3.

enum _qtmr_channel_number

The enumeration for Quad Timer module channel number.

Values:

enumerator kQTMR_Channel0

Quad Timer Channel 0.

enumerator kQTMR_Channel1

Quad Timer Channel 1.

enumerator kQTMR_Channel2

Quad Timer Channel 2.

enumerator kQTMR_Channel3

Quad Timer Channel 3.

enum _qtmr_channel_primary_count_source

The enumeration for Quad Timer channel primary input source.

Values:

enumerator kQTMR_PrimarySrcInputPin0

Quad Timer input pin 0.

enumerator kQTMR_PrimarySrcInputPin1

Quad Timer input pin 1.

enumerator kQTMR_PrimarySrcInputPin2

Quad Timer input pin 2.

enumerator kQTMR_PrimarySrcInputPin3

Quad Timer input pin 3.

enumerator kQTMR_PrimarySrcChannel0Output

Quad Timer channel 0 output.

enumerator kQTMR_PrimarySrcChannel1Output

Quad Timer channel 1 output.

enumerator kQTMR_PrimarySrcChannel2Output

Quad Timer channel 2 output.

enumerator kQTMR_PrimarySrcChannel3Output

Quad Timer channel 3 output.

enumerator kQTMR_PrimarySrcIPBusClockDivide1

IP bus clock divide by 1.

enumerator kQTMR_PrimarySrcIPBusClockDivide2

IP bus clock divide by 2.

enumerator kQTMR_PrimarySrcIPBusClockDivide4

IP bus clock divide by 4.

enumerator kQTMR_PrimarySrcIPBusClockDivide8
IP bus clock divide by 8.

enumerator kQTMR_PrimarySrcIPBusClockDivide16
IP bus clock divide by 16.

enumerator kQTMR_PrimarySrcIPBusClockDivide32
IP bus clock divide by 32.

enumerator kQTMR_PrimarySrcIPBusClockDivide64
IP bus clock divide by 64.

enumerator kQTMR_PrimarySrcIPBusClockDivide128
IP bus clock divide by 128.

enum _qtmr_channel_secondary_count_source

The enumeration for Quad Timer channel secondary input source.

Values:

enumerator kQTMR_SecondarySrcInputPin0
Quad Timer input pin 0.

enumerator kQTMR_SecondarySrcInputPin1
Quad Timer input pin 1.

enumerator kQTMR_SecondarySrcInputPin2
Quad Timer input pin 2.

enumerator kQTMR_SecondarySrcInputPin3
Quad Timer input pin 3.

enum _qtmr_channel_secondary_source_capture_mode

The enumeration for Quad Timer channel secondary input source capture mode.

Values:

enumerator kQTMR_SecondarySrcCaptureNoCapture
Secondary source capture is disabled.

enumerator kQTMR_SecondarySrcCaptureRisingEdge
Secondary source capture on rising edge.

enumerator kQTMR_SecondarySrcCaptureFallingEdge
Secondary source capture on falling edge.

enumerator kQTMR_SecondarySrcCaptureRisingAndFallingEdge
Secondary source capture on both edges.

enumerator kQTMR_SecondarySrcCaptureRisingEdgeWithReload
Secondary source capture on rising edge while cause the channel to be reloaded.

enumerator kQTMR_SecondarySrcCaptureFallingEdgeWithReload
Secondary source capture on falling edge while cause the channel to be reloaded.

enumerator kQTMR_SecondarySrcCaptureRisingAndFallingEdgeWithReload
Secondary source capture on both edges while cause the channel to be reloaded.

enum _qtmr_channel_count_mode

The enumeration for Quad Timer channel count mode.

When “channel output 0~3” or “IP bus clock prescaler” is chosen, active edge is the rising edge. When “input pin 0~3” is chosen, active edge and active level is determined by input invert feature (IPS). Disable input invert feature means active edge is rising edge, active

level is high level, enable input invert feature means active edge is falling edge, active level is low level.

Values:

enumerator kQTMR_CountNoOperation

No operation.

enumerator kQTMR_CountPrimarySrcRiseEdge

Count active edge of primary input source.

enumerator kQTMR_CountPrimarySrcRiseAndFallEdge

Count rising and falling edges of primary input source.

enumerator kQTMR_CountPrimarySrcRiseEdgeSecondarySrcInHigh

Count active edge of primary input source when secondary input is at a active level.

enumerator kQTMR_CountPrimarySecondarySrcInQuadDecode

Quadrature count mode, uses primary and secondary sources.

enumerator kQTMR_CountPrimarySrcRiseEdgeSecondarySrcDir

Count active edge of primary input source; secondary input source specifies count direction.

enumerator kQTMR_CountPrimarySrcRiseEdgeSecondarySrcRiseEdgeTrig

The active edge of secondary input source triggers count active edge of primary input source, and the channel counter will stop upon receiving a second trigger event while it's still counting from the first trigger event.

enumerator kQTMR_CountCascadeWithOtherChannel

Cascaded count mode, the channel will count as compare events occur in the selected source channel (use a special high-speed signal path rather than the OFLAG output signal). The active edge of secondary input source triggers count active edge of primary input source, and the channel counter will re-initialized upon receiving a second trigger event while it's still counting from the first trigger event.

enumerator kQTMR_CountPrimarySrcRiseEdgeSecondarySrcRiseEdgeTrigWithReInit

enum _qtmr_channel_count_length

The enumeration for Quad Timer channel count length.

Values:

enumerator kQTMR_CountLengthUntilRollOver

Count until roll over at \$FFFF.

enumerator kQTMR_CountLengthUntilCompare

Count until compare.

enum _qtmr_channel_count_direction

The enumeration for Quad Timer channel count direction.

Values:

enumerator kQTMR_CountDirectionUp

Count direction up.

enumerator kQTMR_CountDirectionDown

Count direction down.

enum _qtmr_channel_count_times

The enumeration for Quad Timer channel count times.

Values:

enumerator kQTMR_CountTimesRepeat
Count repeatedly.

enumerator kQTMR_CountTimesOnce
Count time once.

enum _qtmr_channel_count_load_mode

The enumeration for Quad Timer channel count load mode.

Values:

enumerator kQTMR_CountLoadNormal

Count can be re-initialized only with the LOAD register when match event occurs.

enumerator kQTMR_CountLoadAlternative

Channel can be re-initialized with the LOAD register when count up and a match with COMP1 occurs, or with CMPLD2 register when count down and a match with COMP2 occurs.

enum _qtmr_channel_count_preload_mode

The enumeration for Quad Timer channel COMP1 & COMP2 preload mode.

Values:

enumerator kQTMR_CountPreloadNoLoad

Not load CMPLDn into COMPn register when compare event occurs.

enumerator kQTMR_CountPreloadOnComp1CompareEvent

Load CMPLDn register into COMPn when occurs a successful comparison of channel counter value and the COMP1 register.

enumerator kQTMR_CountPreloadOnComp2CompareEvent

Load CMPLDn register into COMPn when occurs a successful comparison of channel counter value and the COMP2 register.

enum _qtmr_channel_output_mode

The enumeration for Quad Timer channel output signal (OFLAG signal) work mode.

Values:

enumerator kQTMR_OutputAssertWhenCountActive

OFLAG output assert while counter is active.

enumerator kQTMR_OutputClearOnCompare

OFLAG output clear on successful compare.

enumerator kQTMR_OutputSetOnCompare

OFLAG output set on successful compare.

enumerator kQTMR_OutputToggleOnCompare

OFLAG output toggle on successful compare.

enumerator kQTMR_OutputToggleOnAltCompareReg

OFLAG output toggle using alternating compare registers.

enumerator kQTMR_OutputSetOnCompareClearOnSecSrcActiveEdge

OFLAG output set on compare, clear on secondary source input edge.

enumerator kQTMR_OutputSetOnCompareClearOnCountRoll

OFLAG output set on compare, clear on counter rollover.

enumerator kQTMR_OutputGateClockOutWhenCountActive

OFLAG output gated while count is active.

enum `_qtmr_qtmr_channel_output_value_on_force`

The enumeration for Quad Timer channel output signal (OFLAG) value on force event occur.

Values:

enumerator `kQTMR_OutputValueClearOnForce`

OFLAG output clear (low) when software triggers a FORCE command or master channel force the OFLAG (EEOF need set).

enumerator `kQTMR_OutputValueSetOnForce`

OFLAG output set (high) when software triggers a FORCE command or master channel force the OFLAG (EEOF need set).

enum `_qtmr_channel_debug_action`

The enumeration for Quad Timer channel run options when the chip entering debug mode.

Values:

enumerator `kQTMR_DebugRunNormal`

Continue with normal operation.

enumerator `kQTMR_DebugHaltCounter`

Halt counter.

enumerator `kQTMR_DebugForceOutToZero`

Force output to logic 0.

enumerator `kQTMR_DebugHaltCountForceOutZero`

Halt counter and force output to logic 0.

enum `_qtmr_channel_interrupt_enable`

The enumeration for Quad Timer channel 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.

enumerator `kQTMR_ALLInterruptEnable`

enum `_qtmr_channel_status_flags`

The enumeration for Quad Timer channel work status.

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.

enumerator kQTMR_StatusAllFlags

enum _qtmr_channel_enable
The enumeration for Quad Timer channel enable.

Values:

enumerator kQTMR_Channel0Enable
Channel 0 enable.

enumerator kQTMR_Channel1Enable
Channel 1 enable.

enumerator kQTMR_Channel2Enable
Channel 2 enable.

enumerator kQTMR_Channel3Enable
Channel 3 enable.

enumerator kQTMR_ALLChannelEnable

enum _qtmr_channel_dma_enable
The enumeration for Quad Timer channel DMA trigger source.

Values:

enumerator kQTMR_InputEdgeFlagDmaEnable
Input edge flag setting will trigger DMA read request for CAPT register.

enumerator kQTMR_Compare1PreloadDmaEnable
Channel load CMPLD1 register into COMP1 will trigger DMA write request for CMPLD1.

enumerator kQTMR_Compare2PreloadDmaEnable
Channel load CMPLD2 register into COMP2 will trigger DMA write request for CMPLD2.

enumerator kQTMR_AllDMAEnable

typedef enum _qtmr_input_pin qtmr_input_pin_t
The enumeration for Quad Timer module input pin source.

typedef enum _qtmr_channel_number qtmr_channel_number_t
The enumeration for Quad Timer module channel number.

typedef enum _qtmr_channel_primary_count_source qtmr_channel_primary_count_source_t
The enumeration for Quad Timer channel primary input source.

typedef enum _qtmr_channel_secondary_count_source qtmr_channel_secondary_count_source_t
The enumeration for Quad Timer channel secondary input source.

typedef enum _qtmr_channel_secondary_source_capture_mode
qtmr_channel_secondary_source_capture_mode_t
The enumeration for Quad Timer channel secondary input source capture mode.

`typedef enum _qtmr_channel_count_mode qtmr_channel_count_mode_t`

The enumeration for Quad Timer channel count mode.

When “channel output 0~3” or “IP bus clock prescaler” is chosen, active edge is the rising edge. When “input pin 0~3” is chosen, active edge and active level is determined by input invert feature (IPS). Disable input invert feature means active edge is rising edge, active level is high level, enable input invert feature means active edge is falling edge, active level is low level.

`typedef enum _qtmr_channel_count_length qtmr_channel_count_length_t`

The enumeration for Quad Timer channel count length.

`typedef enum _qtmr_channel_count_direction qtmr_channel_count_direction_t`

The enumeration for Quad Timer channel count direction.

`typedef enum _qtmr_channel_count_times qtmr_channel_count_times_t`

The enumeration for Quad Timer channel count times.

`typedef enum _qtmr_channel_count_load_mode qtmr_channel_count_load_mode_t`

The enumeration for Quad Timer channel count load mode.

`typedef enum _qtmr_channel_count_preload_mode qtmr_channel_count_preload_mode_t`

The enumeration for Quad Timer channel COMP1 & COMP2 preload mode.

`typedef enum _qtmr_channel_output_mode qtmr_channel_output_mode_t`

The enumeration for Quad Timer channel output signal (OFLAG signal) work mode.

`typedef enum _qtmr_qtmr_channel_output_value_on_force`

`qtmr_channel_output_value_on_force_t`

The enumeration for Quad Timer channel output signal (OFLAG) value on force event occur.

`typedef enum _qtmr_channel_debug_action qtmr_channel_debug_action_t`

The enumeration for Quad Timer channel run options when the chip entering debug mode.

`typedef struct _qtmr_channel_input_config qtmr_channel_input_config_t`

The structure for configuring Quad Timer channel input signal.

`typedef struct _qtmr_channel_count_config qtmr_channel_count_config_t`

The structure for configuring Quad Timer channel counting behaviors.

`typedef struct _qtmr_channel_output_config qtmr_channel_output_config_t`

The structure for configuring Quad Timer channel output signal (OFLAG).

`typedef struct _qtmr_channel_cooperation_config qtmr_channel_cooperation_config_t`

The structure for configuring Quad Timer channel cooperation mode with other channels.

`typedef struct _qtmr_channel_config qtmr_channel_config_t`

Quad Timer channel configuration covering all channel configurable fields.

`typedef struct _qtmr_input_pin_filter_config qtmr_input_pin_filter_config_t`

The structure for configuring Quad Timer module input pin filter.

`typedef struct _qtmr_config qtmr_config_t`

Quad Timer module configuration which contain channel config structure pointers and input pin filter config structure pointers.

Note: Need use channel structure address to init the structure pointers, when the channel or input pin structure pointers is NULL, it will be ignored by QTMR_Init API. This can save stack space when only one or two channels are used.

`struct _qtmr_channel_input_config`

`#include <fsl_qtmr.h>` The structure for configuring Quad Timer channel input signal.

Public Members

qtmr_channel_primary_count_source_t ePrimarySource

Specify the primary input source.

qtmr_channel_secondary_count_source_t eSecondarySource

Specify the secondary input source.

bool bEnableSecondarySrcFaultFunction

true: The selected secondary input acts as a fault signal which can clear the channel output signal when it is set, false: Fault function disabled.

bool eEnableInputInvert

true: Invert input signal value when select input pin as primary or/and secondary input source false: no operation.

struct *_qtmr_channel_count_config*

#include <fsl_qtmr.h> The structure for configuring Quad Timer channel counting behaviors.

Public Members

qtmr_channel_count_mode_t eCountMode

Configures channel count mode.

qtmr_channel_count_length_t eCountLength

Configures channel count length.

qtmr_channel_count_direction_t eCountDir

Configures channel count direction.

qtmr_channel_count_times_t eCountTimes

Configures channel count times.

qtmr_channel_count_load_mode_t eCountLoadMode

Configures channel count load mode.

struct *_qtmr_channel_output_config*

#include <fsl_qtmr.h> The structure for configuring Quad Timer channel output signal (OFLAG).

Public Members

qtmr_channel_output_mode_t eOutputMode

Configures channel output signal work mode.

qtmr_channel_output_value_on_force_t eOutputValueOnForce

The value of output signal when force event occur.

bool bEnableOutputInvert

True: the polarity of output signal will be inverted, false: The output signal is not inverted.

bool bEnableSwForceOutput

True: forces the current value of eOFLAGValueOnForce to output signal. false: no operation.

bool bEnableOutputPin

True: the output signal is driven on the external pin. false: the external pin is configured as an input.

struct `_qtmr_channel_cooperation_config`

#include <fsl_qtmr.h> The structure for configuring Quad Timer channel cooperation mode with other channels.

Public Members

bool `bEnableMasterReInit`

true: Master channel within the module can re-initialize this channel when it has a compare event, false: no operation.

bool `bEnableMasterForceOFLAG`

true: Master channel within the module can force this channel OFLAG signal when it has a compare event, false: no operation.

bool `bEnableMasterMode`

true: This channel is configured as mater channel, it can broadcast compare event to all channels within the module to re-initialize channel and/or force channel output signal, false: no operation.

struct `_qtmr_channel_config`

#include <fsl_qtmr.h> Quad Timer channel configuration covering all channel configurable fields.

Public Members

`qtmr_channel_input_config_t` `sInputConfig`

Configures channel input signal.

`qtmr_channel_count_config_t` `sCountConfig`

Configures channel count work mode.

`qtmr_channel_output_config_t` `sOutputConfig`

Configures channel output signal (OFLAG) work mode.

`qtmr_channel_debug_action_t` `eDebugMode`

Configures channel operation in chip debug mode.

uint16_t `u16EnabledInterruptMask`

The mask of the interrupts to be enabled, should be the OR'ed of `_qtmr_channel_interrupt_enable`.

uint16_t `u16EnabledDMAMask`

The mask of the interrupts to be enabled, should be the OR'ed of `_qtmr_channel_dma_enable`.

uint16_t `u16Comp1`

Value for Channel compare register 1.

uint16_t `u16Comp2`

Value for Channel compare register 2.

uint16_t `u16Comp1Preload`

Value for Channel compare 1 preload register.

uint16_t `u16Comp2Preload`

Value for Channel compare 2 preload register.

uint16_t `u16Load`

Value for Channel load register.

uint16_t u16Count

Value for Channel counter value register.

bool bEnableChannel

True: enable the channel prescaler (if it is being used) and counter false: disable channel.

struct __qtmr_input_pin_filter_config

#include <fsl_qtmr.h> The structure for configuring Quad Timer module input pin filter.

Public Members

uint8_t u8Period

Value for input filter sample period.

uint8_t u8Count

Value for input filter sample count (sample count = count +3).

struct __qtmr_config

#include <fsl_qtmr.h> Quad Timer module configuration which contain channel config structure pointers and input pin filter config structure pointers.

Note: Need use channel structure address to init the structure pointers, when the channel or input pin structure pointers is NULL, it will be ignored by QTMR_Init API. This can save stack space when only one or two channels are used.

2.66 The Driver Change Log

2.67 QTMR Peripheral and Driver Overview

2.68 The Driver Change Log

2.69 SIM: System Integration Module Driver

FSL_SIM_DRIVER_VERSION

SIM driver version.

2.70 The Driver Change Log

```
static inline void SIM_SetWaitModeOperation(SIM_Type *base, sim_wait_mode_operation_t
                                             eOperation)
```

Sets the operation of wait mode, enable/disable the entry of wait mode.

Parameters

- base – SIM peripheral base address.
- eOperation – Used to enable/disable the wait mode, please refer to sim_wait_mode_operation_t.

```
static inline void SIM_SetStopModeOperation(SIM_Type *base, sim_stop_mode_operation_t
                                           eOperation)
```

Sets the operation of stop mode, enable/disable the entry of stop mode.

Parameters

- base – SIM peripheral base address.
- eOperation – Used to enable/disable the stop mode, please refer to `sim_stop_mode_operation_t`.

```
static inline void SIM_EnterLPMode(SIM_Type *base)
```

Enters into LPMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: Please make sure the power mode register is not set as write protected before invoking this function.

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

```
static inline void SIM_ExitLPMode(SIM_Type *base)
```

Exits from LPMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: Please make sure the power mode register is not set as write protected before invoking this function.

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

```
static inline void SIM_EnterVLPMMode(SIM_Type *base)
```

Enters into VLPMMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: Please make sure the power mode register is not set as write protected before invoking this function. If both set to enter LPMode and VLPMMode, the VLPMMode has higher priority.

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

```
static inline void SIM_ExitVLPMMode(SIM_Type *base)
```

Exits from VLPMMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: Please make sure the power mode register is not set as write protected before invoking this function.

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

```
static inline bool SIM_IsInLPMMode(SIM_Type *base)
```

Indicates whether the chip is in LPMMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

Return values

- true – The chip is in LPMMode.
- false – The chip is not in LPMMode.

```
static inline bool SIM_IsInVLPMMode(SIM_Type *base)
```

Indicates whether the chip is in VLPMMode when the advanced power mode is enabled(register FOPT[1] bit is set).

Note: This function is useful only when the FTFE module's FOPT[0] bit is set(advanced power mode is enabled).

Parameters

- base – SIM peripheral base address.

Return values

- true – The chip is in VLPMMode.
- false – The chip is not in VLPMMode.

```
static inline void SIM_TriggerSoftwareReset(SIM_Type *base)
```

Triggers the software reset for device.

Parameters

- base – SIM base address.

```
static inline uint16_t SIM_GetResetStatusFlags(SIM_Type *base)
```

Gets the cause of the most recent reset.

Note: At any given time, the only one reset source is indicated. When multiple reset source assert simultaneously, the reset source with the highest precedence is indicated. The precedence from highest to lowest is POR, external reset, COP loss of reference reset, COP CPU time-out reset, software reset, COP window time-out reset. The POR is always set during a power-on reset. However, POR is cleared and the external reset is set if the external reset pin is asserted or remains asserted after the power-on reset has de-asserted.

Parameters

- base – SIM peripheral base address.

Returns

The current reset status flags, should be the OR'ed value of `_sim_reset_status_flags`.

```
static inline void SIM_TriggerPeripheralSoftwareReset(SIM_Type *base,
                                                    sim_swReset_peri_index_t ePeriIndex)
```

Triggers the software reset of specific peripheral.

Parameters

- base – SIM peripheral base address.
- ePeriIndex – The index of the peripheral to be reset.

```
static inline void SIM_EnableResetPadCellInputFilter(SIM_Type *base, bool bEnable)
```

Enables/Disables the input filter on external reset padcell.

If the input filter is enabled, the filter will remove transient signals on the input at the expense of an increased input delay.

Note: If the input filter is enabled, the filter will affect all input functions supported by that padcell, including GPIO.

Parameters

- base – SIM peripheral base address.
- bEnable – Used to control the behaviour of input filter.
 - **true** Enable the input filter on external input padcell.
 - **false** Disable the input filter on external input padcell.

```
static inline void SIM_SetInternalPeriInput(SIM_Type *base, sim_internal_peri_index_t eIndex,
                                           sim_internal_peri_input_t eInput)
```

Sets internal peripheral inputs, some peripheral inputs have the ability to be connected to either XBAR outputs or GPIO.

Parameters

- base – SIM base address.
- eIndex – The internal peripherals that supply multi-inputs.
- eInput – The specific input that connected to the selected internal peripheral.

```
static inline void SIM_SetXbarInputPWMSelection(SIM_Type *base,
                                               sim_xbar_input_pwm_index_t eIndex,
                                               sim_xbar_input_pwm_selection_t eSelection)
```

Selects the Xbar input from PWMA and PWMB.

Parameters

- base – SIM base address.
- eIndex – SIM PWM select register field index.
- eSelection – Xbar input pwm selection.

```
static inline void SIM_SetXbarInputAdcTmrSelection(SIM_Type *base,
                                                  sim_xbar_input_adc_tmr_index_t eIndex,
                                                  sim_xbar_input_adc_tmr_selection_t
                                                  eSelection)
```

Selects the Xbar input from ADC and TMR A/B.

Parameters

- base – SIM base address.
- eIndex – SIM ADC and TMR select register field index.
- eSelection – Xbar input ADC and TMR selection.

```
static inline void SIM_SetSmallRegulator1P2VControlMode(SIM_Type *base,
                                                         sim_small_regulator_1P2V_control_mode_t
                                                         eControlMode)
```

Sets the control mode of small regulator 1.2V supply, the available control modes are normal mode, standby mode, etc.

Note: This function is useful only when the flash module's FOPT[0] bit is 0.

Parameters

- base – SIM peripheral base address.
- eControlMode – The control mode to be set, please refer to `sim_small_regulator_1P2V_control_mode_t`.

```
static inline void SIM_SetSmallRegulator2P7VControlMode(SIM_Type *base,
                                                         sim_small_regulator_2P7V_control_mode_t
                                                         eControlMode)
```

Sets the control mode of small regulator 2.7 supply, the available control modes are normal mode, standby mode, etc.

Note: This function is useful only when the flash module's FOPT[0] bit is 0.

Parameters

- base – SIM peripheral base address.
- eControlMode – The control mode to be set, please refer to `sim_small_regulator_2P7V_control_mode_t`.

```
static inline void SIM_SetLargeRegulatorControlMode(SIM_Type *base,
                                                     sim_large_regulator_control_mode_t
                                                     eControlMode)
```

Sets the control mode of large regulator, the available control mode are normal mode, standby mode, etc.

Note: This function is useful only when the flash module's FOPT[0] bit is 0.

Parameters

- base – SIM peripheral base address.
- eControlMode – The control mode to be set, please refer to `sim_large_regulator_control_mode_t`.

```
static inline void SIM_SetRegisterProtectionMode(SIM_Type *base,  
                                                sim_write_protection_module_t eModule,  
                                                sim_write_protection_mode_t eMode)
```

Sets the write protection mode of the selected register.

Parameters

- base – SIM peripheral base address.
- eModule – The module to be set, please refer to `sim_write_protection_module_t`.
- eMode – The specific write protection mode to be set, please refer to `sim_write_protection_mode_t`.

```
static inline uint32_t SIM_GetJTAGID(SIM_Type *base)
```

Gets JTAG ID, the JTAG ID is 32bits width.

Parameters

- base – SIM base address.

Returns

The 32bits width JTAG ID.

```
static inline uint8_t SIM_GetPMCBandgapTrim(SIM_Type *base)
```

Gets the trim vlaue of the bandgap inside the PMC module, the available range is 0 to 15.

Parameters

- base – SIM peripheral base address.

Returns

The trim value of PMC's bandgap, ranges from 0 to 15.

```
static inline uint16_t SIM_Get200KHzROSCFreqTrim(SIM_Type *base)
```

Gets the trim value of 200KHz Relaxation Osillator Frequency, the available range is 0 to 511.

Parameters

- base – SIM peripheral base address.

Returns

The trim value of 200 khz Relaxation Oscillator frequency, ranges from 0 to 511.

```
static inline void SIM_SetIOShortAddressValue(SIM_Type *base, uint32_t  
                                              u32IOShortAddressValue)
```

Sets the I/O short address location value which specifies the memory referenced through the I/O short address mode.

The I/O short address mode allows the instrution to specify the lower 6 bits of the address. And the upper 18 bits of the address can be controlled by invoking this function.

Note: The pipeline delay between setting the related register set and using short I/O adrrsing with the new value is five cycles.

Parameters

- base – SIM base address.
- u32IOShortAddressValue – The value of I/O short address location, this address value should be 24 bits width.

```
static inline uint16_t SIM_GetSoftwareControlData(SIM_Type *base,
                                                sim_software_contrl_register_index_t
                                                eIndex)
```

Gets the software control data by the software control register index.

Parameters

- base – SIM base address.
- eIndex – SIM software control register index.

Returns

Software control registers value.

```
static inline void SIM_SetSoftwareControlData(SIM_Type *base,
                                             sim_software_contrl_register_index_t eIndex,
                                             uint16_t u16Value)
```

Sets the software control data by the software control register index, the data is for general-purpose use by software.

Parameters

- base – SIM base address.
- eIndex – SIM software control register index.
- u16Value – Software control registers value.

```
static inline void SIM_SetOnCEClockOperationMode(SIM_Type *base,
                                                sim_onceclk_operation_mode_t
                                                eOperationMode)
```

Sets the operation mode of the OnCE clock, the available operation modes are always enabled and enabled when the core TAP is enabled.

Parameters

- base – SIM peripheral base address.
- eOperationMode – The operation mode of OnCE clock, please refer to `sim_onceclk_operation_mode_t`.

```
static inline void SIM_SetDMAOperationMode(SIM_Type *base, sim_dma_operation_mode_t
                                           eOperationMode)
```

Sets the operation mode of DMA, such as disabled, enabled in run mode only, etc.

Parameters

- base – SIM peripheral base address.
- eOperationMode – The operation mode to be set, please refer to `sim_dma_operation_mode_t`.

```
static inline sim_boot_mode_t SIM_GetBootMode(SIM_Type *base)
```

Gets the device's boot mode, the available boot modes are ROM boot and NVM flash boot.

Parameters

- base – SIM peripheral base address.

Returns

The device's boot mode, please refer to `sim_boot_mode_t`.

```
static inline void SIM_EnableADCScanControlReorder(SIM_Type *base, bool bEnable)
```

Enables/Disables the ADC scan control register reorder feature.

Parameters

- base – SIM peripheral base address.
- bEnable – Used to control the ADC scan control register reorder feature.
 - **true** Enable the re-ordering of ADC scan control bits.
 - **false** ADC scan control register works in normal order.

```
static inline void SIM_SetMasterPIT(SIM_Type *base, sim_master_pit_selection_t eMasterPit)
```

Sets master programmable interval timer.

Parameters

- base – SIM peripheral base address.
- eMasterPit – The master PIT to be selected, please refer to `sim_master_pit_selection_t`.

```
static inline void SIM_SetBootOverride(SIM_Type *base, sim_boot_override_mode_t eMode)
```

Sets the boot over ride mode, this API can be used to determine the boot option in the next reset excluding POR.

Parameters

- base – SIM peripheral base address.
- eMode – The boot over ride mode.

```
enum _sim_reset_status_flags
```

The enumeration of system reset status flags, such as power on reset, software reset, etc.

Values:

```
enumerator kSIM_PowerONResetFlag
```

The Power on reset caused the most recent reset.

```
enumerator kSIM_ExternalResetFlag
```

The external reset caused the most recent reset, that means the external reset pin was asserted or remained asserted after the power-on reset de-asserted.

```
enumerator kSIM_COPLossOfReferenceResetFlag
```

The computer operating properly module signaled a PLL loss of reference clock reset caused the most recent reset.

```
enumerator kSIM_COPCPUTimeOutResetFlag
```

The computer operating properly module signaled a CPU time-out reset caused the most recent reset.

```
enumerator kSIM_SoftwareResetFlag
```

The previous system reset occurred as a result of a software reset

```
enumerator kSIM_COPWindowTimeOutResetFlag
```

The previous system reset occurred as a result of a cop_window reset.

```
enum _sim_stop_mode_operation
```

The enumeration of stop mode operation can be used to enable/disable stop mode enter.

Values:

```
enumerator kSIM_STOPInstructionEnterStopMode
```

Stop mode is entered when the DSC core executes a STOP instruction.

enumerator kSIM_STOPInstructionNotEnterStopMode

The DSC core STOP instruction does not cause entry into stop mode.

enumerator kSIM_STOPInstructionEnterStopModeWriteProtect

Stop mode is entered when the DSC core executes a STOP instruction, and the related register bit field is write protected until the next reset.

enumerator kSIM_STOPInstructionNotEnterStopModeWriteProtect

The DSC core STOP instruction does not cause entry into stop mode, and the related register bit field is write protected until the next reset.

enum _sim_wait_mode_operation

The enumeration of wait mode operation can be used to enable/disable wait mode enter.

Values:

enumerator kSIM_WAITInstructionEnterWaitMode

Wait mode is entered when the DSC core executes a WAIT instruction.

enumerator kSIM_WAITInstructionNotEnterWaitMode

The DSC core WAIT instruction does not cause entry into wait mode.

enumerator kSIM_WAITInstructionEnterWaitModeWriteProtect

Wait mode is entered when the DSC core executes a WAIT instruction, and the related register bit field is write protected until the next reset.

enumerator kSIM_WAITInstructionNotEnterWaitModeWriteProtect

The DSC core WAIT instruction does not cause entry into wait mode, and the related register bit field is write protected until the next reset.

enum _sim_onceclk_operation_mode

The enumeration of OnCE clock operation mode, such as enabled when core TAP is enabled and always enabled.

Values:

enumerator kSIM_OnCEClkEnabledWhenCoreTapEnabled

The OnCE clock to the DSC core is enabled when the core TAP is enabled.

enumerator kSIM_OnCEClkAlwaysEnabled

The OnCE clock to the DSC core is always enabled.

enum _sim_dma_operation_mode

The enumeration of dma operation mode, this enumeration can be used to disable/enable DMA module in different power modes.

Values:

enumerator kSIM_DMADisable

DMA module is disabled.

enumerator kSIM_DMAEnableAtRunModeOnly

DMA module is enabled in run mode only.

enumerator kSIM_DMAEnableAtRunModeWaitMode

DMA module is enabled in run and wait modes only.

enumerator kSIM_DMAEnableAtAllPowerModes

DMA module is enabled in all power modes.

enumerator kSIM_DMADisableWriteProtect

DMA module is disabled and the related register bit field is write protected until the next reset.

enumerator kSIM_DMAEnableAtRunModeOnlyWriteProtect

DMA module is enabled in run mode only and the related bit field is write protected until the next reset.

enumerator kSIM_DMAEnableAtRunModeWaitModeWriteProtect

DMA module is enabled in run and wait modes only and the related register bit field is write protected until the next reset.

enumerator kSIM_DMAEnableAtAllPowerModesWriteProtect

DMA module is enabled in all low power modes and the related register bit field is write protected until the next reset.

enum _sim_boot_mode

The enumeration of device's boot mode, including ROM boot and NVM flash boot.

Values:

enumerator kSIM_BootFromNVMFlash

Indicates the chip is boot from NVM Flash.

enumerator kSIM_BootFromROM

Indicates the chip is boot from ROM.

enum _sim_small_regulator_1P2V_control_mode

The enumeration of small regulator 1P2V control mode, such as normal mode and standby mode.

Values:

enumerator kSIM_SmallRegulator1P2VInNormalMode

Small regulator 1.2V supply placed in normal mode.

enumerator kSIM_SmallRegulator1P2VInStandbyMode

Small regulator 1.2V supply placed in standby mode.

enumerator kSIM_SmallRegulator1P2VInNormalModeWriteProtect

Small regulator 1.2V supply placed in normal mode, and the related register bit field is write protected until the next reset.

enumerator kSIM_SmallRegulator1P2VInStandbyModeWriteProtect

Small regulator 1.2V supply placed in standby mode, and the related register bit field is write protected until the next reset.

enum _sim_small_regulator_2P7V_control_mode

The enumeration of small regulator 2P7V control mode, such as normal mode, standby mode, powerdown mode, etc.

Values:

enumerator kSIM_SmallRegulator2P7VInNormalMode

Small regulator 2.7V supply placed in normal mode.

enumerator kSIM_SmallRegulator2P7VInStandbyMode

Small regulator 2.7V supply placed in standby mode.

enumerator kSIM_SmallRegulator2P7VInPowerdownMode

Small regulator 2.7V supply placed in powerdown mode.

enumerator kSIM_SmallRegulator2P7VInNormalModeWriteProtect

Small regulator 2.7V supply placed in normal mode and the related bit field is write protected until chip reset.

enumerator kSIM_SmallRegulator2P7VInStandbyModeWriteProtect

Small regulator 2.7V supply placed in standby mode and the related bit field is write protected until chip reset.

enumerator kSIM_SmallRegulator2P7VInPowerdownModeWriteProtect

Small regulator placed in powerdown mode and the related bit field is write protected until chip reset.

enum _sim_large_regulator_control_mode

The enumeration of large regulator control mode, such as normal mode, standby mode.

Values:

enumerator kSIM_LargeRegulatorInNormalMode

Large regulator placed in normal mode.

enumerator kSIM_LargeRegulatorInStandbyMode

Large regulator placed in standby mode.

enumerator kSIM_LargeRegulatorInNormalModeWriteProtect

Large regulator placed in normal mode, and the related register bit field is write protected until chip reset.

enumerator kSIM_LargeRegulatorInStandbyModeWriteProtect

Large regulator placed in standby mode, and the related register bit field is write protected until chip reset.

enum _sim_write_protection_module

The enumeration of modules that support various protection mode.

Values:

enumerator kSIM_GPIOInternalPeripheralSelectProtection

Used to control the protection mode GPSn and IPSn registers in the SIM, all XBAR, EVTG, GPIO_n_PER, GPIO_n_PPMODE, GPIO_n_DRIVE.

enumerator kSIM_PeripheralClockEnableProtection

Used to control the protection mode of PCEn, SDn, PSWRn, and PCR register.

enumerator kSIM_GPIOPortDProtection

Used to control the protection mode of GPIO_D_PER, GPIO_D_PPMODE, and GPIO_D_DRIVE register.

enumerator kSIM_PowerModeControlWriteProtection

Used to control the protection mode of the PWRMODE register.

enum _sim_write_protection_mode

The enumeration of write protection mode, such as write protection off, write protection on, etc.

Values:

enumerator kSIM_WriteProtectionOff

Write protection off.

enumerator kSIM_WriteProtectionOn

Write protection on.

enumerator kSIM_WriteProtectionOffAndLocked

Write protection off and locked until chip reset.

enumerator kSIM_WriteProtectionOnAndLocked

Write protection on and locked until chip reset.

enum `_sim_lpi2c_trigger_selection`

The enumeration of lpi2c trigger selection, including slave trigger and master trigger.

Values:

enumerator `kSIM_Lpi2cSlaveTrigger`

Selects slave trigger.

enumerator `kSIM_Lpi2cMasterTrigger`

Selects master trigger.

enum `_sim_master_pit_selection`

The enumeration of master pit.

Values:

enumerator `kSIM_PIT0MasterPIT1Slave`

PIT0 is master PIT and PIT1 is slave PIT.

enumerator `kSIM_PIT1MasterPIT0Slave`

PIT0 is master PIT and PIT1 is slave PIT.

typedef enum `_sim_stop_mode_operation` `sim_stop_mode_operation_t`

The enumeration of stop mode operation can be used to enable/disable stop mode enter.

typedef enum `_sim_wait_mode_operation` `sim_wait_mode_operation_t`

The enumeration of wait mode operation can be used to enable/disable wait mode enter.

typedef enum `_sim_onceclk_operation_mode` `sim_onceclk_operation_mode_t`

The enumeration of OnCE clock operation mode, such as enabled when core TAP is enabled and always enabled.

typedef enum `_sim_dma_operation_mode` `sim_dma_operation_mode_t`

The enumeration of dma operation mode, this enumeration can be used to disable/enable DMA module in different power modes.

typedef enum `_sim_boot_mode` `sim_boot_mode_t`

The enumeration of device's boot mode, including ROM boot and NVM flash boot.

typedef enum `_sim_small_regulator_1P2V_control_mode`

`sim_small_regulator_1P2V_control_mode_t`

The enumeration of small regulator 1P2V control mode, such as normal mode and standby mode.

typedef enum `_sim_small_regulator_2P7V_control_mode`

`sim_small_regulator_2P7V_control_mode_t`

The enumeration of small regulator 2P7V control mode, such as normal mode, standby mode, powerdown mode, etc.

typedef enum `_sim_large_regulator_control_mode` `sim_large_regulator_control_mode_t`

The enumeration of large regulator control mode, such as normal mode, standby mode.

typedef enum `_sim_write_protection_module` `sim_write_protection_module_t`

The enumeration of modules that support various protection mode.

typedef enum `_sim_write_protection_mode` `sim_write_protection_mode_t`

The enumeration of write protection mode, such as write protection off, write protection on, etc.

typedef enum `_sim_lpi2c_trigger_selection` `sim_lpi2c_trigger_selection_t`

The enumeration of lpi2c trigger selection, including slave trigger and master trigger.

```
typedef enum _sim_master_pit_selection sim_master_pit_selection_t
```

The enumeration of master pit.

```
FSL_COMPONENT_ID
```

```
SIM_RESET_STATUS_MASK
```

The macro of REST status bit field mask.

```
SIM_PWR_SR27_CONTROL_MODE_MASK
```

The definition of the short regulator control mode bit field mask.

```
SIM_PWR_SR27_CONTROL_MODE_SHIFT
```

The definition of the short regulator control mode bit field shift.

```
SIM_PWR_SR27_CONTROL_MODE(x)
```

The macro that can be used to set the bit field of PWR register's short regulator bit field.

```
SIM_PROT_BIT_FIELD_MASK(moduleName)
```

The definition of the PORT register bit filed mask.

```
SIM_PORT_SET_MODE_PROTECTION_MODE(moduleName, protectionMode)
```

The macro that can be used to set module's protection mode.

2.71 SIM Peripheral and Driver Overview

2.72 XBAR: Inter-Peripheral Crossbar Switch Driver

```
void XBARA_Init(XBARA_Type *base)
```

Initializes the XBARA module.

This function un-gates the XBARA clock.

Parameters

- base – XBARA peripheral address.

```
void XBARA_Deinit(XBARA_Type *base)
```

Shuts down the XBARA module.

This function disables XBARA clock.

Parameters

- base – XBARA peripheral address.

```
static inline void XBARA_SetSignalsConnection(XBARA_Type *base, xbar_input_signal_t eInput,
                                             xbar_output_signal_t eOutput)
```

Sets a connection between the selected XBARA_IN[*] input and the XBARA_OUT[*] output signal.

This function connects the XBARA input to the selected XBARA output. If more than one XBARA module is available, only the inputs and outputs from the same module can be connected.

Example:

```
XBARA_SetSignalsConnection(XBARA, kXBARA_InputPIT_TRG0, kXBARA_
↪OutputDMAMUX18);
```

Parameters

- base – XBARA peripheral address.
- eInput – XBARA input signal.
- eOutput – XBARA output signal.

```
static inline void XBARA_SetActiveEdgeDetectMode(XBARA_Type *base, xbar_output_signal_t
                                                eOutput, xbara_active_edge_t
                                                eActiveEdgeMode)
```

Sets active edge detection mode for the XBARA_OUT[*] output signal.

Parameters

- base – XBARA peripheral address.
- eOutput – XBARA output signal.
- eActiveEdgeMode – Active edge mode.

```
static inline void XBARA_SetInterruptDMARequestMode(XBARA_Type *base,
                                                    xbar_output_signal_t eOutput,
                                                    xbara_request_t eRequest)
```

Sets DMA, Interrupt or disabled request generation mode for the XBARA_OUT[*] output signal.

Parameters

- base – XBARA peripheral address.
- eOutput – XBARA output signal.
- eRequest – Request type.

```
void XBARA_SetOutputSignalConfig(XBARA_Type *base, xbar_output_signal_t eOutput, const
                                xbara_control_config_t *psControlConfig)
```

Configures the XBARA output signal edge detection and interrupt/dma features.

This function configures an XBARA control register. The active edge detection and the DMA/IRQ function on the corresponding XBARA output can be set.

Example:

```
xbara_control_config_t userConfig;
userConfig.activeEdge = kXBARA_EdgeRising;
userConfig.requestType = kXBARA_RequestInterruptEnable;
XBARA_SetOutputSignalConfig(XBARA, kXBARA_OutputDMAMUX18, &userConfig);
```

Note: Only a subset of the XBARA output signal can be called with this API. On debug mode code will check whether the output signal eOutput satisfy the requirement.

Parameters

- base – XBARA peripheral address.
- eOutput – XBARA output number.
- psControlConfig – Pointer to structure that keeps configuration of control register.

```
uint16_t XBARA_GetStatusFlags(XBARA_Type *base)
```

Gets the active edge detection status for all XBAR output signal supporting this feature.

This function gets the active edge detect status of all XBARA_OUTs. If the active edge occurs, the return value is asserted. When the interrupt or the DMA functionality is enabled for

the XBARA_OUTx, this field is 1 when the interrupt or DMA request is asserted and 0 when the interrupt or DMA request has been cleared.

Parameters

- base – XBARA peripheral address.

Returns

ORed value from all status flag from xbara_status_flag_t.

```
static inline void XBARA_ClearStatusFlags(XBARA_Type *base, uint16_t u16Flags)
```

Clear the edge detection status flags of relative mask.

Parameters

- base – XBARA peripheral address.
- u16Flags – status flags composed from ORed xbara_status_flag_t indicating flags to be cleared.

```
FSL_XBAR_DRIVER_VERSION
```

XBAR driver version.

```
enum _xbara_active_edge
```

XBARA active edge for detection.

Values:

```
enumerator kXBARA_EdgeNone
```

Edge detection status bit never asserts.

```
enumerator kXBARA_EdgeRising
```

Edge detection status bit asserts on rising edges.

```
enumerator kXBARA_EdgeFalling
```

Edge detection status bit asserts on falling edges.

```
enumerator kXBARA_EdgeRisingAndFalling
```

Edge detection status bit asserts on rising and falling edges.

```
enum _xbara_request
```

XBARA DMA and interrupt configurations. Note it only apply for a subset of XBARA output signal.

Values:

```
enumerator kXBARA_RequestDisable
```

Interrupt and DMA are disabled.

```
enumerator kXBARA_RequestDMAEnable
```

DMA enabled, interrupt disabled.

```
enumerator kXBARA_RequestInterruptEnable
```

Interrupt enabled, DMA disabled.

```
enum _xbara_status_flag
```

XBARA status flags.

This provides constants for the XBARA status flags for use in the XBARA functions. The enumerator value is designed to make sure Flags in same register can be created with register value to write/read register.

Values:

```
enumerator kXBARA_EdgeDetectionOut0Flag
```

XBAR_OUT0 active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut1Flag`
XBAR_OUT1 active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut2Flag`
XBAR_OUT2 active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_EdgeDetectionOut3Flag`
XBAR_OUT3 active edge interrupt flag, sets when active edge detected.

enumerator `kXBARA_AllStatusFlags`

typedef enum `_xbara_active_edge` `xbara_active_edge_t`
XBARA active edge for detection.

typedef enum `_xbara_request` `xbara_request_t`
XBARA DMA and interrupt configurations. Note it only apply for a subset of XBARA output signal.

typedef enum `_xbara_status_flag` `xbara_status_flag_t`
XBARA status flags.

This provides constants for the XBARA status flags for use in the XBARA functions. The enumerator value is designed to make sure Flags in same register can be created with register value to write/read register.

typedef struct `_xbara_control_config` `xbara_control_config_t`
Defines the configuration structure of the XBARA control register.

This structure keeps the configuration of XBARA control register for one output. Control registers are available only for a few outputs. Not every XBARA module has control registers.

`XBARA_SELx(base, output)`
Macro function to extract the XBAR select register address for a given xbar output signal.

`XBARA_CTRLx(base, output)`
Macro function to extract the XBAR Ctrl register address for a given xbar output signal.

`XBARA_SELx_SELn_SHIFT(output)`
Macro function to get SELn field shift in XBARA_SELx register for a given output signal.

`XBARA_SELx_SELn_MASK(output)`
Macro function to get SELn field mask in XBARA_SELx register for a given output signal.

`XBARA_SELx_SELn(output, input_signal)`
Macro function to create SELn field value in XBARA_SELx register for given output signal and input signal value `input_signal`, see `xbar_input_signal_t`.

`XBARA_CTRLx_DIENn_MASK(output)`
Macro function to get DIENn field mask in XBARA_CTRLx register for a given output signal.

`XBARA_CTRLx_DIENn_SHIFT(output)`
Macro function to get DIENn field shift in XBARA_CTRLx register for a given output signal.

`XBARA_CTRLx_DIENn(output, x)`
Macro function to create DIENn field value in XBARA_CTRLx register for given output signal and DMA/Interrupt mode `x`, see `xbara_request_t`.

`XBARA_CTRLx_EDGEen_MASK(output)`
Macro function to get EDGEen field mask in XBARA_CTRLx register for a given output signal.

`XBARA_CTRLx_EDGEen_SHIFT(output)`
Macro function to get EDGEen field shift in XBARA_CTRLx register for a given output signal.

`XBARA_CTRLx_EDGEn(output, x)`

Macro function to create EDGEn field value in XBARA_CTRLx register for given output signal and edge mode x, see `xbara_active_edge_t`.

`XBARA_CTRLx_STS_MASK`

Macro value for the Status bits in CTRL register.

`struct _xbara_control_config`

#include <fsl_xbara.h> Defines the configuration structure of the XBARA control register.

This structure keeps the configuration of XBARA control register for one output. Control registers are available only for a few outputs. Not every XBARA module has control registers.

Public Members

`xbara_active_edge_t` eActiveEdge

Active edge to be detected.

`xbara_request_t` eRequestType

Selects DMA/Interrupt request.

2.73 The Driver Change Log

2.74 XBAR Peripheral and Driver Overview

Chapter 3

Middleware

Chapter 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

[Readme](#)

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 corepkcs11

PKCS #11 key management library.

Readme

4.1.8 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme