Kinetis SDK v.2.0 API Reference Manual

NXP Semiconductors

Document Number: KSDK20KV58APIRM
Rev. 0
Jul 2016

Contents

Contents

Section Number	Title	Page Number

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Section Number | Title | Page Number

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

Chapter **PWM: Pulse Width Modulator**

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Chapter 1
Introduction

The Kinetis Software Development Kit (KSDK) 2.0 is a collection of software enablement, for NXP Kinetis Microcontrollers, that includes peripheral drivers, high-level stacks including USB and lwIP, integration with WolfSSL and mbed TLS cryptography libraries, other middleware packages (multicore support and FatFS), and integrated RTOS support for FreeRTOS, µC/OS-II, and µC/OS-III. In addition to the base enablement, the KSDK is augmented with demo applications, driver example projects, and API documentation to help users quickly leverage the support of the Kinetis SDK. The Kinetis Expert (KEx) Web UI is available to provide access to all Kinetis SDK packages. See the *Kinetis SDK v.2.0.0 Release Notes* (document KSDK200RN) and the supported Devices section at www.nxp.com/ksdk for details.

The Kinetis SDK is built with the following runtime software components:

- ARM® and DSP standard libraries, and CMSIS-compliant device header files which provide direct access to the peripheral registers.
- Open-source peripheral drivers that provide stateless, high-performance, ease-of-use APIs. Communication drivers provide higher-level transactional APIs for a higher-performance option.
- Open-source RTOS wrapper driver built on on top of KSDK peripheral drivers and leverage native RTOS services to better comply to the RTOS cases.
- Real time operation systems (RTOS) including FreeRTOS OS, µC/OS-II, and µC/OS-III.
- Stacks and middleware in source or object formats including:
  - A USB device, host, and OTG stack with comprehensive USB class support.
  - CMSIS-DSP, a suite of common signal processing functions.
  - FatFs, a FAT file system for small embedded systems.
  - Encryption software utilizing the mmCAU hardware acceleration.
  - SDMMC, a software component supporting SD Cards and eMMC.
  - mbedTLS, cryptographic SSL/TLS libraries.
  - lwIP, a light-weight TCP/IP stack.
  - WolfSSL, a cryptography and SSL/TLS library.
  - EMV L1 that complies to EMV-v4.3_Book_1 specification.
  - DMA Manager, a software component used for managing on-chip DMA channel resources.
  - The Kinetis SDK comes complete with software examples demonstrating the usage of the peripheral drivers, RTOS wrapper drivers, middleware and RTOSes.

  All demo applications and driver examples are provided with projects for the following toolchains:
  - Atollic TrueSTUDIO
  - GNU toolchain for ARM® Cortex® -M with Cmake build system
  - IAR Embedded Workbench
  - Keil MDK
  - Kinetis Design Studio

The peripheral drivers and RTOS driver wrappers can be used across multiple devices within the Kinetis product family without modification. The configuration items for each driver are encapsulated into C language data structures. Kinetis device-specific configuration information is provided as part of the KSDK and need not be modified by the user. If necessary, the user is able to modify the peripheral driver and RTOS wrapper driver configuration during runtime. The driver examples demonstrate how to configure the drivers by passing the proper configuration data to the APIs. The Kinetis SDK folder structure is organized to reduce the total number of includes required to compile a project.

Deliverable	Location
Examples	<install_dir>/examples/
Demo Applications	<install_dir>/examples/<board_name>/demo_apps/
Driver Examples	<install_dir>/examples/<board_name>/driver_examples/
Documentation	<install_dir>/doc/
USB Documentation	<install_dir>/doc/usb/
lwIP Documentation	<install_dir>/doc/tcpip/lwip/
Middleware	<install_dir>/middleware/
DMA Manager	<install_dir>/dma_manager_<version>/
FatFS	<install_dir>/middleware/fatfs_<version>
lwIP TCP/IP	<install_dir>/middleware/lwip_<version>/
MMCAU	<install_dir>/mmcau_<version>/
SD MMC Support	<install_dir>/sdmmc_<version>/
USB Stack	<install_dir>/middleware/usb_<version>
Drivers	<install_dir>/<device_name>/drivers/
CMSIS Standard ARM Cortex-M Headers, math and DSP Libraries	<install_dir>/<device_name>/CMSIS/
Device Startup and Linker	<install_dir>/<device_name>/<toolchain>/
SDK Utilities	<install_dir>/<device_name>/utilities/
RTOS Kernels	<install_dir>/rtos/

Table 2: KSDK Folder Structure

The rest of this document describes the API references in detail for the peripheral drivers and RTOS wrapper drivers. For the latest version of this and other Kinetis SDK documents, see the kex.nxp.com/apidoc.

Chapter 2
Driver errors status

- kStatus_DSPI_Error = 601
- kStatus_EDMA_QueueFull = 5100
- kStatus_EDMA_Busy = 5101
- kStatus_ENET_RxFrameError = 4000
- kStatus_ENET_RxFrameFail = 4001
- kStatus_ENET_RxFrameEmpty = 4002
- kStatus_ENET_TxFrameBusy = 4003
- kStatus_ENET_TxFrameFail = 4004
- #kStatus_ENET_PtpTsRingFull = 4005
- #kStatus_ENET_PtpTsRingEmpty = 4006
- kStatus_SMC_StopAbort = 3900
- kStatus_NOTIFIER_ErrorNotificationBefore = 9800
- kStatus_NOTIFIER_ErrorNotificationAfter = 9801
- kStatus_DMAMGR_ChannelOccupied = 5200
- kStatus_DMAMGR_ChannelNotUsed = 5201
- kStatus_DMAMGR_NoFreeChannel = 5202
- kStatus_DMAMGR_ChannelNotMatchSource = 5203

Chapter 3
Architectural Overview

This chapter provides the architectural overview for the Kinetis Software Development Kit (KSDK). It describes each layer within the architecture and its associated components.

Overview

The Kinetis SDK architecture consists of five key components listed below.

1. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) CORE compliance device-specific header files, SOC Header, and CMSIS math/DSP libraries.
2. Peripheral Drivers
3. Real-time Operating Systems (RTOS)
4. Stacks and Middleware that integrate with the Kinetis SDK
5. Demo Applications based on the Kinetis SDK

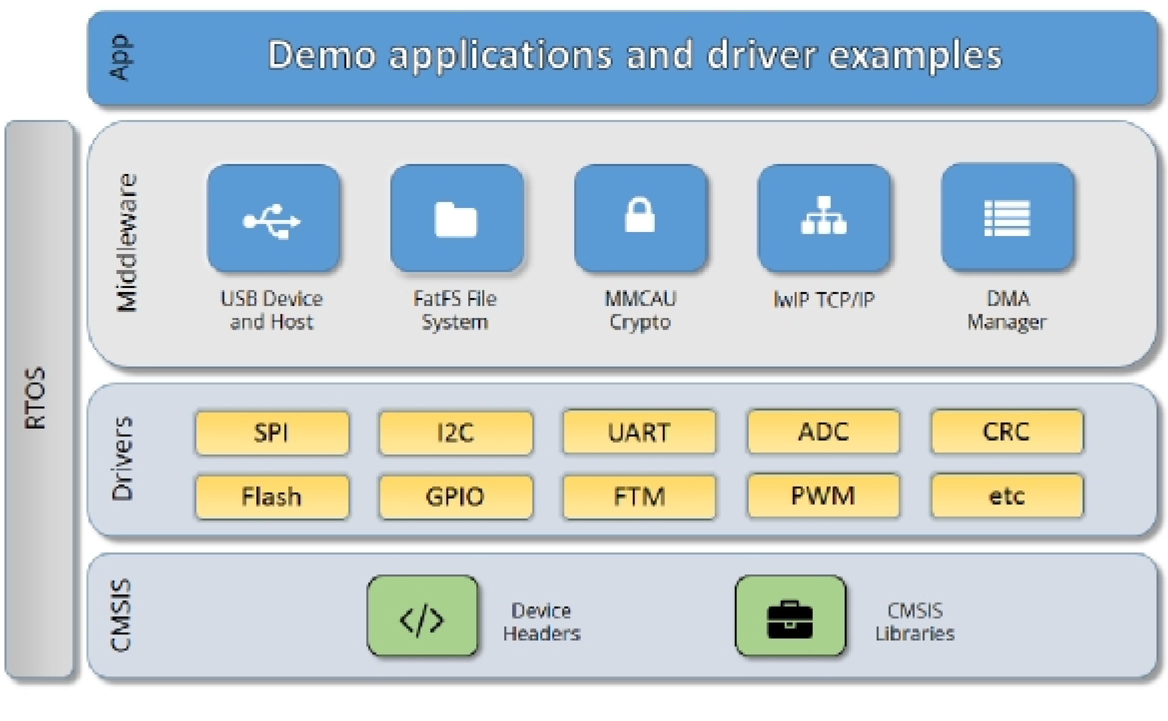

Figure 1: KSDK Block Diagram

Kinetis MCU header files

Each supported Kinetis MCU device in the KSDK has an overall System-on Chip (SoC) memory-mapped header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers. The overall SoC header file provides a access to the peripheral registers through pointers and predefined bit masks. In addition to the overall SoC memory-mapped header file, the KSDK includes a feature header file for each device. The feature header file allows NXP to deliver a single software driver for a given peripheral. The feature file ensures that the driver is properly compiled for the target SOC.

CMSIS Support

Along with the SoC header files and peripheral extension header files, the KSDK also includes common CMSIS header files for the ARM Cortex-M core and the math and DSP libraries from the latest CMSIS release. The CMSIS DSP library source code is also included for reference.

KSDK Peripheral Drivers

The KSDK peripheral drivers mainly consist of low-level functional APIs for the Kinetis MCU product family on-chip peripherals and also of high-level transactional APIs for some bus drivers/DMA driver/e-DMA driver to quickly enable the peripherals and perform transfers.

All KSDK peripheral drivers only depend on the CMSIS headers, device feature files, fsl_common.h, and fsl_clock.h files so that users can easily pull selected drivers and their dependencies into projects. With the exception of the clock/power-relevant peripherals, each peripheral has its own driver. Peripheral drivers handle the peripheral clock gating/ungating inside the drivers during initialization and deinitialization respectively.

Low-level functional APIs provide common peripheral functionality, abstracting the hardware peripheral register accesses into a set of stateless basic functional operations. These APIs primarily focus on the control, configuration, and function of basic peripheral operations. The APIs hide the register access details and various MCU peripheral instantiation differences so that the application can be abstracted from the low-level hardware details. The API prototypes are intentionally similar to help ensure easy portability across supported KSDK devices.

Transactional APIs provide a quick method for customers to utilize higher-level functionality of the peripherals. The transactional APIs utilize interrupts and perform asynchronous operations without user intervention. Transactional APIs operate on high-level logic that requires data storage for internal operation context handling. However, the Peripheral Drivers do not allocate this memory space. Rather, the user passes in the memory to the driver for internal driver operation. Transactional APIs ensure the NVIC is enabled properly inside the drivers. The transactional APIs do not meet all customer needs, but provide a baseline for development of custom user APIs.

Note that the transactional drivers never disable an NVIC after use. This is due to the shared nature of interrupt vectors on Kinetis devices. It's up to the user to ensure that NVIC interrupts are properly disabled after usage is complete.

Interrupt handling for transactional APIs

A double weak mechanism is introduced for drivers with transactional API. The double weak indicates two levels of weak vector entries. See the examples below:

```
        PUBWEAK SPI0_IRQHandler
        PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

```
LDR     R0, =SPI0_DriverIRQHandler
BX      R0
```

The first level of the weak implementation are the functions defined in the vector table. In the devices/<DEVICE_NAME>/<TOOLCHAIN>/startup_<DEVICE_NAME>.s/.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0_DriverIRQHandler) jumps to itself (B .). The KSDK drivers with transactional APIs provide the reimplementation of the second layer function inside of the peripheral driver. If the KSDK drivers with transactional APIs are linked into the image, the SPI0_DriverIRQHandler is replaced with the function implemented in the KSDK SPI driver.

The reason for implementing the double weak functions is to provide a better user experience when using the transactional APIs. For drivers with a transactional function, call the transactional APIs and the drivers complete the interrupt-driven flow. Users are not required to redefine the vector entries out of the box. At the same time, if users are not satisfied by the second layer weak function implemented in the KSDK drivers, users can redefine the first layer weak function and implement their own interrupt handler functions to suit their implementation.

The limitation of the double weak mechanism is that it cannot be used for peripherals that share the same vector entry. For this use case, redefine the first layer weak function to enable the desired peripheral interrupt functionality. For example, if the MCU's UART0 and UART1 share the same vector entry, redefine the UART0_UART1_IRQHandler according to the use case requirements.

Feature Header Files

The peripheral drivers are designed to be reusable regardless of the peripheral functional differences from one Kinetis MCU device to another. An overall Peripheral Feature Header File is provided for the KSDK-supported MCU device to define the features or configuration differences for each Kinetis sub-family device.

Application

See the *Getting Started with Kinetis SDK (KSDK) v2.0* document (KSDK20GSUG).

Chapter 4
Trademarks

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions

Freescale, the Freescale logo, Kinetis, and Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Tower is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM, ARM powered logo, Keil, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

Chapter 5
ADC16: 16-bit SAR Analog-to-Digital Converter Driver

5.1 Overview

The KSDK provides a peripheral driver for the 16-bit SAR Analog-to-Digital Converter (ADC16) module of Kinetis devices.

5.2 Typical use case

5.2.1 Polling Configuration

```
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_Init(DEMO_ADC16_INSTANCE);
    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Configure(DEMO_ADC16_INSTANCE, &adc16ConfigStruct);
    ADC16_EnableHardwareTrigger(DEMO_ADC16_INSTANCE, false);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    if (kStatus_Success == ADC16_DoAutoCalibration(DEMO_ADC16_INSTANCE))
    {
        PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
    }
    else
    {
        PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
    }
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION

    adc16ChannelConfigStruct.channelNumber = DEMO_ADC16_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
      false;
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif // FSL_FEATURE_ADC16_HAS_DIFF_MODE

    while(1)
    {
        GETCHAR(); // Input any key in terminal console.
        ADC16_ChannelConfigure(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);
        while (kADC16_ChannelConversionDoneFlag !=
      ADC16_ChannelGetStatusFlags(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP))
        {
        }
        PRINTF("ADC Value: %d\r\n", ADC16_ChannelGetConversionValue(DEMO_ADC16_INSTANCE,
      DEMO_ADC16_CHANNEL_GROUP));
    }
```

5.2.2 Interrupt Configuration

```
    volatile bool g_Adc16ConversionDoneFlag = false;
    volatile uint32_t g_Adc16ConversionValue;
    volatile uint32_t g_Adc16InterruptCount = 0U;
```

```
    // ...

    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_Init(DEMO_ADC16_INSTANCE);
    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Configure(DEMO_ADC16_INSTANCE, &adc16ConfigStruct);
    ADC16_EnableHardwareTrigger(DEMO_ADC16_INSTANCE, false);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    if (ADC16_DoAutoCalibration(DEMO_ADC16_INSTANCE))
    {
        PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
    }
    else
    {
        PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
    }
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION

    adc16ChannelConfigStruct.channelNumber = DEMO_ADC16_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
      true; // Enable the interrupt.
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif // FSL_FEATURE_ADC16_HAS_DIFF_MODE

    while(1)
    {
        GETCHAR(); // Input a key in the terminal console.
        g_Adc16ConversionDoneFlag = false;
        ADC16_ChannelConfigure(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);
        while (!g_Adc16ConversionDoneFlag)
        {
        }
        PRINTF("ADC Value: %d\r\n", g_Adc16ConversionValue);
        PRINTF("ADC Interrupt Count: %d\r\n", g_Adc16InterruptCount);
    }

    // ...

    void DEMO_ADC16_IRQHandler(void)
    {
        g_Adc16ConversionDoneFlag = true;
        // Read the conversion result to clear the conversion completed flag.
        g_Adc16ConversionValue = ADC16_ChannelConversionValue(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP
      );
        g_Adc16InterruptCount++;
    }
```

Files

- file fsl_adc16.h

Data Structures

- struct adc16_config_t
  ADC16 converter configuration. More...
- struct adc16_hardware_compare_config_t
  ADC16 Hardware comparison configuration. More...
- struct adc16_channel_config_t
  ADC16 channel conversion configuration. More...

Enumerations

- enum _adc16_channel_status_flags { kADC16_ChannelConversionDoneFlag = ADC_SC1_COCO_MASK }
  Channel status flags.
- enum _adc16_status_flags { kADC16_ActiveFlag = ADC_SC2_ADACT_MASK }
  Converter status flags.
- enum adc16_clock_divider_t {
  kADC16_ClockDivider1 = 0U,
  kADC16_ClockDivider2 = 1U,
  kADC16_ClockDivider4 = 2U,
  kADC16_ClockDivider8 = 3U }
  Clock divider for the converter.
- enum adc16_resolution_t {
  kADC16_Resolution8or9Bit = 0U,
  kADC16_Resolution12or13Bit = 1U,
  kADC16_Resolution10or11Bit = 2U,
  kADC16_ResolutionSE8Bit = kADC16_Resolution8or9Bit,
  kADC16_ResolutionSE12Bit = kADC16_Resolution12or13Bit,
  kADC16_ResolutionSE10Bit = kADC16_Resolution10or11Bit }
  Converter's resolution.
- enum adc16_clock_source_t {
  kADC16_ClockSourceAlt0 = 0U,
  kADC16_ClockSourceAlt1 = 1U,
  kADC16_ClockSourceAlt2 = 2U,
  kADC16_ClockSourceAlt3 = 3U,
  kADC16_ClockSourceAsynchronousClock = kADC16_ClockSourceAlt3 }
  Clock source.
- enum adc16_long_sample_mode_t {
  kADC16_LongSampleCycle24 = 0U,
  kADC16_LongSampleCycle16 = 1U,
  kADC16_LongSampleCycle10 = 2U,
  kADC16_LongSampleCycle6 = 3U,
  kADC16_LongSampleDisabled = 4U }
  Long sample mode.
- enum adc16_reference_voltage_source_t {
  kADC16_ReferenceVoltageSourceVref = 0U,
  kADC16_ReferenceVoltageSourceValt = 1U }
  Reference voltage source.
- enum adc16_hardware_compare_mode_t {
  kADC16_HardwareCompareMode0 = 0U,
  kADC16_HardwareCompareMode1 = 1U,
  kADC16_HardwareCompareMode2 = 2U,
  kADC16_HardwareCompareMode3 = 3U }
  Hardware compare mode.

Driver version

- #define FSL_ADC16_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  ADC16 driver version 2.0.0.

Initialization

- void ADC16_Init (ADC_Type ∗base, const adc16_config_t ∗config)
  Initializes the ADC16 module.
- void ADC16_Deinit (ADC_Type ∗base)
  De-initializes the ADC16 module.
- void ADC16_GetDefaultConfig (adc16_config_t ∗config)
  Gets an available pre-defined settings for the converter's configuration.

Advanced Features

- static void ADC16_EnableHardwareTrigger (ADC_Type ∗base, bool enable)
  Enables the hardware trigger mode.
- void ADC16_SetHardwareCompareConfig (ADC_Type ∗base, const adc16_hardware_compare_config_t ∗config)
  Configures the hardware compare mode.
- uint32_t ADC16_GetStatusFlags (ADC_Type ∗base)
  Gets the status flags of the converter.
- void ADC16_ClearStatusFlags (ADC_Type ∗base, uint32_t mask)
  Clears the status flags of the converter.

Conversion Channel

- void ADC16_SetChannelConfig (ADC_Type ∗base, uint32_t channelGroup, const adc16_channel_config_t ∗config)
  Configures the conversion channel.
- static uint32_t ADC16_GetChannelConversionValue (ADC_Type ∗base, uint32_t channelGroup)
  Gets the conversion value.
- uint32_t ADC16_GetChannelStatusFlags (ADC_Type ∗base, uint32_t channelGroup)
  Gets the status flags of channel.

5.3 Data Structure Documentation

5.3.1 struct adc16_config_t

Data Fields

- adc16_reference_voltage_source_t referenceVoltageSource
  Select the reference voltage source.
- adc16_clock_source_t clockSource
  Select the input clock source to converter.
- bool enableAsynchronousClock
  Enable the asynchronous clock output.
- adc16_clock_divider_t clockDivider
  Select the divider of input clock source.

- adc16_resolution_t resolution
  Select the sample resolution mode.
- adc16_long_sample_mode_t longSampleMode
  Select the long sample mode.
- bool enableHighSpeed
  Enable the high-speed mode.
- bool enableLowPower
  Enable low power.
- bool enableContinuousConversion
  Enable continuous conversion mode.

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource

5.3.1.0.0.1.2 adc16_clock_source_t adc16_config_t::clockSource

5.3.1.0.0.1.3 bool adc16_config_t::enableAsynchronousClock

5.3.1.0.0.1.4 adc16_clock_divider_t adc16_config_t::clockDivider

5.3.1.0.0.1.5 adc16_resolution_t adc16_config_t::resolution

5.3.1.0.0.1.6 adc16_long_sample_mode_t adc16_config_t::longSampleMode

5.3.1.0.0.1.7 bool adc16_config_t::enableHighSpeed

5.3.1.0.0.1.8 bool adc16_config_t::enableLowPower

5.3.1.0.0.1.9 bool adc16_config_t::enableContinuousConversion

5.3.2 struct adc16_hardware_compare_config_t

Data Fields

- adc16_hardware_compare_mode_t hardwareCompareMode
  Select the hardware compare mode.
- int16_t value1
  Setting value1 for hardware compare mode.
- int16_t value2
  Setting value2 for hardware compare mode.

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode

See "adc16_hardware_compare_mode_t".

5.3.2.0.0.2.2 int16_t adc16_hardware_compare_config_t::value1

5.3.2.0.0.2.3 int16_t adc16_hardware_compare_config_t::value2

5.3.3 struct adc16_channel_config_t

Data Fields

- uint32_t channelNumber
  Setting the conversion channel number.
- bool enableInterruptOnConversionCompleted
  Generate a interrupt request once the conversion is completed.

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 uint32_t adc16_channel_config_t::channelNumber

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

5.3.3.0.0.3.2 bool adc16_channel_config_t::enableInterruptOnConversionCompleted

5.4 Macro Definition Documentation

5.4.1 #define FSL_ADC16_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

5.5 Enumeration Type Documentation

5.5.1 enum _adc16_channel_status_flags

Enumerator

kADC16_ChannelConversionDoneFlag Conversion done.

5.5.2 enum _adc16_status_flags

Enumerator

kADC16_ActiveFlag Converter is active.

5.5.3 enum adc16_clock_divider_t

Enumerator

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

kADC16_ClockDivider2 For divider 2 from the input clock to the module.
kADC16_ClockDivider4 For divider 4 from the input clock to the module.
kADC16_ClockDivider8 For divider 8 from the input clock to the module.

5.5.4 enum adc16_resolution_t

Enumerator

kADC16_Resolution8or9Bit Single End 8-bit or Differential Sample 9-bit.
kADC16_Resolution12or13Bit Single End 12-bit or Differential Sample 13-bit.
kADC16_Resolution10or11Bit Single End 10-bit or Differential Sample 11-bit.
kADC16_ResolutionSE8Bit Single End 8-bit.
kADC16_ResolutionSE12Bit Single End 12-bit.
kADC16_ResolutionSE10Bit Single End 10-bit.

5.5.5 enum adc16_clock_source_t

Enumerator

kADC16_ClockSourceAlt0 Selection 0 of the clock source.
kADC16_ClockSourceAlt1 Selection 1 of the clock source.
kADC16_ClockSourceAlt2 Selection 2 of the clock source.
kADC16_ClockSourceAlt3 Selection 3 of the clock source.
kADC16_ClockSourceAsynchronousClock Using internal asynchronous clock.

5.5.6 enum adc16_long_sample_mode_t

Enumerator

kADC16_LongSampleCycle24 20 extra ADCK cycles, 24 ADCK cycles total.
kADC16_LongSampleCycle16 12 extra ADCK cycles, 16 ADCK cycles total.
kADC16_LongSampleCycle10 6 extra ADCK cycles, 10 ADCK cycles total.
kADC16_LongSampleCycle6 2 extra ADCK cycles, 6 ADCK cycles total.
kADC16_LongSampleDisabled Disable the long sample feature.

5.5.7 enum adc16_reference_voltage_source_t

Enumerator

kADC16_ReferenceVoltageSourceVref For external pins pair of VrefH and VrefL.
kADC16_ReferenceVoltageSourceValt For alternate reference pair of ValtH and ValtL.

5.5.8 enum adc16_hardware_compare_mode_t

Enumerator

kADC16_HardwareCompareMode0 x < value1.
kADC16_HardwareCompareMode1 x > value1.
kADC16_HardwareCompareMode2 if value1 <= value2, then x < value1 || x > value2; else, value1 > x > value2.
kADC16_HardwareCompareMode3 if value1 <= value2, then value1 <= x <= value2; else x >= value1 || x <= value2.

5.6 Function Documentation

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

Parameters

base	ADC16 peripheral base address.
config	Pointer to configuration structure. See "adc16_config_t".

5.6.2 void ADC16_Deinit (ADC_Type * *base*)

Parameters

base	ADC16 peripheral base address.

5.6.3 void ADC16_GetDefaultConfig (adc16_config_t * *config*)

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

```
*   config->referenceVoltageSource     = kADC16_ReferenceVoltageSourceVref
  ;
*   config->clockSource                = kADC16_ClockSourceAsynchronousClock
  ;
*   config->enableAsynchronousClock    = true;
*   config->clockDivider               = kADC16_ClockDivider8;
*   config->resolution                 = kADC16_ResolutionSE12Bit;
*   config->longSampleMode             = kADC16_LongSampleDisabled;
*   config->enableHighSpeed            = false;
*   config->enableLowPower             = false;
*   config->enableContinuousConversion = false;
*
```

Parameters

config	Pointer to the configuration structure.

5.6.4 static void ADC16_EnableHardwareTrigger (ADC_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	ADC16 peripheral base address.
enable	Switcher of the hardware trigger feature. "true" means enabled, "false" means not enabled.

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

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

Parameters

base	ADC16 peripheral base address.
config	Pointer to the "adc16_hardware_compare_config_t" structure. Passing "NULL" disables the feature.

5.6.6 uint32_t ADC16_GetStatusFlags (ADC_Type ∗ *base*)

Parameters

base	ADC16 peripheral base address.

Returns

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

5.6.7 void ADC16_ClearStatusFlags (ADC_Type ∗ *base*, uint32_t *mask*)

Parameters

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

5.6.8 void ADC16_SetChannelConfig (ADC_Type * *base*, uint32_t *channelGroup*, const adc16_channel_config_t * *config*)

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

Note that the "Channel Group" has a detailed description. To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC has more than one group of status and control registers, one for each conversion. The channel group parameter indicates which group of registers are used, for example channel group 0 is for Group A registers and channel group 1 is for Group B registers. The channel groups are used in a "ping-pong" approach to control the ADC operation. At any point, only one of the channel groups is actively controlling ADC conversions. The channel group 0 is used for both software and hardware trigger modes. Channel groups 1 and greater indicate potentially multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual about the number of SC1n registers (channel groups) specific to this device. None of the channel groups 1 or greater are used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

base	ADC16 peripheral base address.
channelGroup	Channel group index.
config	Pointer to the "adc16_channel_config_t" structure for the conversion channel.

5.6.9 static uint32_t ADC16_GetChannelConversionValue (ADC_Type * *base*, uint32_t *channelGroup*) [inline], [static]

Parameters

base	ADC16 peripheral base address.
channelGroup	Channel group index.

Returns

Conversion value.

5.6.10 uint32_t ADC16_GetChannelStatusFlags (ADC_Type ∗ *base*, uint32_t *channelGroup*)

Parameters

base	ADC16 peripheral base address.
channelGroup	Channel group index.

Returns

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

Chapter 6
AOI: Crossbar AND/OR/INVERT Driver

6.1 Overview

The SDK provides Peripheral driver for the Crossbar AND/OR/INVERT (AOI) block of Kinetis devices.

The AOI module supports a configurable number of event outputs, where each event output represents a user-programmed combinational boolean function based on four event inputs. The key features of this module include:

- Four dedicated inputs for each event output
- User-programmable combinational boolean function evaluation for each event output
- Memory-mapped device connected to a slave peripheral (IPS) bus
- Configurable number of event outputs

6.2 Function groups

6.2.1 AOI Initialization

To initialize the AOI driver, call the AOI_Init() function and pass a baseaddr pointer.

```
// Initialize AOI module.
status = AOI_Init(AOI);
```

6.2.2 AOI Get Set Operation

The AOI module provides a universal boolean function generator using a four-term sum of products expression with each product term containing true or complement values of the four selected event inputs (A, B, C, D). The AOI is a highly programmable module for creating combinational boolean outputs for use as hardware triggers. 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. To configure the selected AOI module event, call the API of the AOI_SetEventLogicConfig() function. To get current event state configure, call the API of AOI_GetEventLogicConfig() function.The AOI module does not support any special modes of operation.

```
/*
EVENTn
= (0,An,~An,1) & (0,Bn,~Bn,1) & (0,Cn,~Cn,1) & (0,Dn,~Dn,1)// product term 0
| (0,An,~An,1) & (0,Bn,~Bn,1) & (0,Cn,~Cn,1) & (0,Dn,~Dn,1)// product term 1
| (0,An,~An,1) & (0,Bn,~Bn,1) & (0,Cn,~Cn,1) & (0,Dn,~Dn,1)// product term 2
| (0,An,~An,1) & (0,Bn,~Bn,1) & (0,Cn,~Cn,1) & (0,Dn,~Dn,1)// product term 3
*/
    aoi_event_config_t demoEventLogicStruct;

    /* Configure the AOI event */
```

```
    demoEventLogicStruct.PT0AC = kAOI_InvInputSignal;  /* CMP0 output*/
    demoEventLogicStruct.PT0BC = kAOI_InputSignal;     /* PIT0 output*/
    demoEventLogicStruct.PT0CC = kAOI_LogicOne;
    demoEventLogicStruct.PT0DC = kAOI_LogicOne;

    demoEventLogicStruct.PT1AC = kAOI_LogicZero;
    demoEventLogicStruct.PT1BC = kAOI_LogicOne;
    demoEventLogicStruct.PT1CC = kAOI_LogicOne;
    demoEventLogicStruct.PT1DC = kAOI_LogicOne;

    demoEventLogicStruct.PT2AC = kAOI_LogicZero;
    demoEventLogicStruct.PT2BC = kAOI_LogicOne;
    demoEventLogicStruct.PT2CC = kAOI_LogicOne;
    demoEventLogicStruct.PT2DC = kAOI_LogicOne;

    demoEventLogicStruct.PT3AC = kAOI_LogicZero;
    demoEventLogicStruct.PT3BC = kAOI_LogicOne;
    demoEventLogicStruct.PT3CC = kAOI_LogicOne;
    demoEventLogicStruct.PT3DC = kAOI_LogicOne;

    AOI_SetEventLogicConfig(AOI, kAOI_Event0, &demoEventLogicStruct);
```

6.3 Typical use case

AOI module is designed to be integrated in conjuction with one or more inter-peripheral crossbar switch (XBAR) modules. A crossbar switch is typically used to select the 4∗n AOI inputs from among available peripheral outputs and GPIO signals. The n EVENTn outputs from the AOI module are typically used as additional inputs to a second crossbar switch, adding to it the ability to connect to its outputs an arbitrary 4-input boolean function of its other inputs.

This is an example to initialize and configure the AOI driver for a possible use case. Because the AOI module function is directly connected with an XBAR (Inter-peripheral crossbar) module, other peripheral (PIT, CMP and XBAR) drivers are used to show full functionality of AOI module.

For example:

```
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "fsl_cmp.h"
#include "fsl_pit.h"
#include "fsl_aoi.h"
#include "fsl_xbara.h"
#include "fsl_xbarb.h"
#include "board.h"
#include "app.h"

/*******************************************************************************
 * Definitions
 ******************************************************************************/

/*******************************************************************************
 * Prototypes
 ******************************************************************************/
volatile bool xbaraInterrupt = false;
/*******************************************************************************
 * Code
 ******************************************************************************/

int main (void)
{
    cmp_config_t cmpConfig;
    cmp_dac_config_t cmpdacConfig;
```

```
pit_config_t pitConfig;

aoi_event_config_t aoiEventLogicStruct;

xbara_control_config_t xbaraConfig;

/* Init board hardware.*/
BOARD_InitHardware();

cmpdacConfig.referenceVoltageSource =
  kCMP_VrefSourceVin2;
cmpdacConfig.DACValue = 32U; /* Set DAC output value */

CMP_GetDefaultConfig(&cmpConfig);
CMP_Init(BOARD_CMP_BASEADDR, &cmpConfig);
/* Set input plus is CMP_channel1, input minus is CMP_DAC out */
CMP_SetInputChannels(BOARD_CMP_BASEADDR, 1, 7);

CMP_SetDACConfig(BOARD_CMP_BASEADDR, &cmpdacConfig);

/* Enable falling interrupt */
CMP_EnableInterrupts(BOARD_CMP_BASEADDR,
  kCMP_OutputFallingInterruptEnable);
EnableIRQ(BOARD_CMP_IRQ);

PIT_Init(BOARD_PIT_BASEADDR);
pitConfig.enableRunInDebug = false;
PIT_Configure(BOARD_PIT_BASEADDR, &pitConfig);
/* Set period is 500ms */
PIT_SetTimerPeriod(BOARD_PIT_BASEADDR, BOARD_PIT_CHANNEL,  USEC_TO_COUNT(500000u,
  CLOCK_GetFreq(kCLOCK_BusClk)));
PIT_EnableInterrupts(BOARD_PIT_BASEADDR, BOARD_PIT_CHANNEL,
  kPIT_TimerInterruptEnable);
EnableIRQ(BOARD_PIT_IRQ);
PIT_StartTimer(BOARD_PIT_BASEADDR, BOARD_PIT_CHANNEL);

/* Configure the AOI event */
aoiEventLogicStruct.PT0AC = kAOI_InvInputSignal;  /* CMP0 output*/
aoiEventLogicStruct.PT0BC = kAOI_InputSignal;     /* PIT0 output*/
aoiEventLogicStruct.PT0CC = kAOI_LogicOne;
aoiEventLogicStruct.PT0DC = kAOI_LogicOne;

aoiEventLogicStruct.PT1AC = kAOI_LogicZero;
aoiEventLogicStruct.PT1BC = kAOI_LogicOne;
aoiEventLogicStruct.PT1CC = kAOI_LogicOne;
aoiEventLogicStruct.PT1DC = kAOI_LogicOne;

aoiEventLogicStruct.PT2AC = kAOI_LogicZero;
aoiEventLogicStruct.PT2BC = kAOI_LogicOne;
aoiEventLogicStruct.PT2CC = kAOI_LogicOne;
aoiEventLogicStruct.PT2DC = kAOI_LogicOne;

aoiEventLogicStruct.PT3AC = kAOI_LogicZero;
aoiEventLogicStruct.PT3BC = kAOI_LogicOne;
aoiEventLogicStruct.PT3CC = kAOI_LogicOne;
aoiEventLogicStruct.PT3DC = kAOI_LogicOne;
/* Init AOI module. */
AOI_Init(BOARD_AOI_BASEADDR);
AOI_SetEventLogicConfig(BOARD_AOI_BASEADDR,
  kAOI_Event0, &aoiEventLogicStruct);

/* Init XBAR module. */
XBARA_Init(BOARD_XBARA_BASEADDR);
XBARB_Init(BOARD_XBARB_BASEADDR);

/* Configure the XBARA signal connections */
XBARA_SetSignalsConnection(BOARD_XBARA_BASEADDR, kXBARA_InputPIT_TRG0,
```

```
      kXBARA_OutputDMAMUX18);

    /* Configure the XBARA interrupt */
    xbaraConfig.activeEdge = kXBARA_EdgeRising;
    xbaraConfig.requestType = kXBARA_RequestInterruptEnalbe;
    XBARA_SetOutputSignalConfig(BOARD_XBARA_BASEADDR, kXBARA_OutputDMAMUX18, &
      xbaraConfig);

    /* Configure the XBARA signal connections */
    XBARB_SetSignalsConnection(BOARD_XBARB_BASEADDR, kXBARB_InputCMP0_Output,
      kXBARB_OutputAOI_IN0);
    XBARB_SetSignalsConnection(BOARD_XBARB_BASEADDR, kXBARB_InputPIT_TRG0,
      kXBARB_OutputAOI_IN1);
    XBARA_SetSignalsConnection(BOARD_XBARA_BASEADDR, kXBARA_InputAND_OR_INVERT_0,
       kXBARA_OutputDMAMUX18);
    /* Enable at the NVIC. */
    EnableIRQ(BOARD_XBARA_IRQ);

    PRINTF("XBAR and AOI Demo: Start...\r\n");

    while(1)
    {
      if(xbaraInterrupt)
      {
        xbaraInterrupt = false;
        PRINTF("XBARA interrupt occurred\r\n\r\n");
      }
    }
}

void CMP0_IRQHandler(void)
{
    if(CMP_GetStatusFlags(BOARD_CMP_BASEADDR) &
      kCMP_OutputFallingEventFlag)
    {
        CMP_ClearStatusFlags(BOARD_CMP_BASEADDR, kCMP_OutputFallingEventFlag);
    }
}

void PIT0_IRQHandler(void)
{
    if(PIT_GetStatusFlags(BOARD_PIT_BASEADDR, BOARD_PIT_CHANNEL) &
      kPIT_TimerFlag)
    {
        PIT_ClearStatusFlags(BOARD_PIT_BASEADDR, BOARD_PIT_CHANNEL, kPIT_TimerFlag);
    }
}

void XBARA_IRQHandler(void)
{
    /* Clear interrupt flag */
    XBARA_ClearStatusFlags(XBARA, kXBARA_EdgeDetectionOut0);
    xbaraInterrupt = true;
}
```

Files

- file fsl_aoi.h

Data Structures

- struct aoi_event_config_t
  AOI event configuration structure. More...

Macros

- #define AOI AOI0
  AOI peripheral address.

Enumerations

- enum aoi_input_config_t {
  kAOI_LogicZero = 0x0U,
  kAOI_InputSignal = 0x1U,
  kAOI_InvInputSignal = 0x2U,
  kAOI_LogicOne = 0x3U }
  AOI input configurations.
- enum aoi_event_t {
  kAOI_Event0 = 0x0U,
  kAOI_Event1 = 0x1U,
  kAOI_Event2 = 0x2U,
  kAOI_Event3 = 0x3U }
  AOI event indexes, where an event is the collection of the four product terms (0, 1, 2, and 3) and the four signal inputs (A, B, C, and D).

Driver version

- #define FSL_AOI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

AOI Initialization

- void AOI_Init (AOI_Type *base)
  Initializes an AOI instance for operation.
- void AOI_Deinit (AOI_Type *base)
  Deinitializes an AOI instance for operation.

AOI Get Set Operation

- void AOI_GetEventLogicConfig (AOI_Type *base, aoi_event_t event, aoi_event_config_t *config)
  Gets the Boolean evaluation associated.
- void AOI_SetEventLogicConfig (AOI_Type *base, aoi_event_t event, const aoi_event_config_-t *eventConfig)
  Configures an AOI event.

6.4 Data Structure Documentation

6.4.1 struct aoi_event_config_t

Defines structure _aoi_event_config and use the AOI_SetEventLogicConfig() function to make whole event configuration.

Data Fields

- aoi_input_config_t PT0AC
  Product term 0 input A.
- aoi_input_config_t PT0BC
  Product term 0 input B.
- aoi_input_config_t PT0CC
  Product term 0 input C.
- aoi_input_config_t PT0DC
  Product term 0 input D.
- aoi_input_config_t PT1AC
  Product term 1 input A.
- aoi_input_config_t PT1BC
  Product term 1 input B.
- aoi_input_config_t PT1CC
  Product term 1 input C.
- aoi_input_config_t PT1DC
  Product term 1 input D.
- aoi_input_config_t PT2AC
  Product term 2 input A.
- aoi_input_config_t PT2BC
  Product term 2 input B.
- aoi_input_config_t PT2CC
  Product term 2 input C.
- aoi_input_config_t PT2DC
  Product term 2 input D.
- aoi_input_config_t PT3AC
  Product term 3 input A.
- aoi_input_config_t PT3BC
  Product term 3 input B.
- aoi_input_config_t PT3CC
  Product term 3 input C.
- aoi_input_config_t PT3DC
  Product term 3 input D.

6.5 Macro Definition Documentation

6.5.1 #define FSL_AOI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

6.6 Enumeration Type Documentation

6.6.1 enum aoi_input_config_t

The selection item represents the Boolean evaluations.

Enumerator

kAOI_LogicZero Forces the input to logical zero.
kAOI_InputSignal Passes the input signal.
kAOI_InvInputSignal Inverts the input signal.

kAOI_LogicOne Forces the input to logical one.

6.6.2 enum aoi_event_t

Enumerator

kAOI_Event0 Event 0 index.
kAOI_Event1 Event 1 index.
kAOI_Event2 Event 2 index.
kAOI_Event3 Event 3 index.

6.7 Function Documentation

6.7.1 void AOI_Init (AOI_Type * *base*)

This function un-gates the AOI clock.

Parameters

base	AOI peripheral address.

6.7.2 void AOI_Deinit (AOI_Type * *base*)

This function shutdowns AOI module.

Parameters

base	AOI peripheral address.

6.7.3 void AOI_GetEventLogicConfig (AOI_Type * *base*, aoi_event_t *event*, aoi_event_config_t * *config*)

This function returns the Boolean evaluation associated.

Example:

```
aoi_event_config_t demoEventLogicStruct;

AOI_GetEventLogicConfig(AOI, kAOI_Event0, &demoEventLogicStruct);
```

Parameters

base	AOI peripheral address.
event	Index of the event which will be set of type aoi_event_t.
config	Selected input configuration .

6.7.4 void AOI_SetEventLogicConfig (AOI_Type ∗ *base,* aoi_event_t *event,* const aoi_event_config_t ∗ *eventConfig*)

This function configures an AOI event according to the aoiEventConfig structure. This function configures all inputs (A, B, C, and D) of all product terms (0, 1, 2, and 3) of a desired event.

Example:

```
aoi_event_config_t demoEventLogicStruct;

demoEventLogicStruct.PT0AC = kAOI_InvInputSignal;
demoEventLogicStruct.PT0BC = kAOI_InputSignal;
demoEventLogicStruct.PT0CC = kAOI_LogicOne;
demoEventLogicStruct.PT0DC = kAOI_LogicOne;

demoEventLogicStruct.PT1AC = kAOI_LogicZero;
demoEventLogicStruct.PT1BC = kAOI_LogicOne;
demoEventLogicStruct.PT1CC = kAOI_LogicOne;
demoEventLogicStruct.PT1DC = kAOI_LogicOne;

demoEventLogicStruct.PT2AC = kAOI_LogicZero;
demoEventLogicStruct.PT2BC = kAOI_LogicOne;
demoEventLogicStruct.PT2CC = kAOI_LogicOne;
demoEventLogicStruct.PT2DC = kAOI_LogicOne;

demoEventLogicStruct.PT3AC = kAOI_LogicZero;
demoEventLogicStruct.PT3BC = kAOI_LogicOne;
demoEventLogicStruct.PT3CC = kAOI_LogicOne;
demoEventLogicStruct.PT3DC = kAOI_LogicOne;

AOI_SetEventLogicConfig(AOI, kAOI_Event0, demoEventLogicStruct);
```

Parameters

base	AOI peripheral address.
event	Event which will be configured of type aoi_event_t.
eventConfig	Pointer to type aoi_event_config_t structure. The user is responsible for filling out the members of this structure and passing the pointer to this function.

Chapter 7
Clock Driver

7.1 Overview

The KSDK provides APIs for Kinetis devices clock operation.

7.2 Get frequency

There is a centralized function CLOCK_GetFreq to get different types of clock frequency by passing in clock name, for example, pass in kCLOCK_CoreSysClk to get core clock, pass in kCLOCK_BusClk to get bus clock. Beside, there are also separate functions to get frequency, for example, use CLOCK_GetCoreSysClkFreq to get core clock frequency, use CLOCK_GetBusClkFreq to get bus clock frequency, use these separate functions could reduce image size.

7.3 External clock frequency

The external clock EXTAL0/EXTAL1/EXTAL32 are decided by board level design. Clock driver uses variables g_xtal0Freq/g_xtal1Freq/g_xtal32Freq to save these clock frequency. Correspondingly, the APIs CLOCK_SetXtal0Freq, CLOCK_SetXtal1Freq and CLOCK_SetXtal32Freq are used to set these variables.

Upper layer must set these values correctly, for example, after OSC0(SYSOSC) is initialized using CLOCK_InitOsc0 or CLOCK_InitSysOsc, upper layer should call CLOCK_SetXtal0Freq too. Otherwise, the clock frequency get functions may not get valid value. This is useful for multi-core platforms, only one core calls CLOCK_InitOsc0 to initialize OSC0, other cores only need to call CLOCK_SetXtal0Freq.

Modules

- Multipurpose Clock Generator (MCG)

Files

- file fsl_clock.h

Data Structures

- struct sim_clock_config_t
  SIM configuration structure for clock setting. More...
- struct oscer_config_t
  OSC configuration for OSCERCLK. More...
- struct osc_config_t
  OSC Initialization Configuration Structure. More...
- struct mcg_pll_config_t
  MCG PLL configuration. More...
- struct mcg_config_t
  MCG mode change configuration structure. More...

Macros

- #define DMAMUX_CLOCKS
  Clock ip name array for DMAMUX.
- #define HSADC_CLOCKS
  Clock ip name array for HSADC.
- #define ENET_CLOCKS
  Clock ip name array for ENET.
- #define PORT_CLOCKS
  Clock ip name array for PORT.
- #define FLEXBUS_CLOCKS
  Clock ip name array for FLEXBUS.
- #define ENC_CLOCKS
  Clock ip name array for ENC.
- #define EWM_CLOCKS
  Clock ip name array for EWM.
- #define PIT_CLOCKS
  Clock ip name array for PIT.
- #define DSPI_CLOCKS
  Clock ip name array for DSPI.
- #define LPTMR_CLOCKS
  Clock ip name array for LPTMR.
- #define FTM_CLOCKS
  Clock ip name array for FTM.
- #define EDMA_CLOCKS
  Clock ip name array for EDMA.
- #define FLEXCAN_CLOCKS
  Clock ip name array for FLEXCAN.
- #define DAC_CLOCKS
  Clock ip name array for DAC.
- #define ADC16_CLOCKS
  Clock ip name array for ADC16.
- #define XBARA_CLOCKS
  Clock ip name array for XBARA.
- #define XBARB_CLOCKS
  Clock ip name array for XBARB.
- #define AOI_CLOCKS
  Clock ip name array for AOI.
- #define TRNG_CLOCKS
  Clock ip name array for TRNG.
- #define MPU_CLOCKS
  Clock ip name array for MPU.
- #define PWM_CLOCKS
  Clock ip name array for PWM.
- #define UART_CLOCKS
  Clock ip name array for UART.
- #define CRC_CLOCKS
  Clock ip name array for CRC.
- #define I2C_CLOCKS
  Clock ip name array for I2C.
- #define PDB_CLOCKS
  Clock ip name array for PDB.

- #define CMP_CLOCKS
  Clock ip name array for CMP.
- #define FTF_CLOCKS
  Clock ip name array for FTF.
- #define LPO_CLK_FREQ 1000U
  LPO clock frequency.
- #define SYS_CLK kCLOCK_CoreSysClk
  Peripherals clock source definition.

Enumerations

- enum clock_name_t {
  kCLOCK_CoreSysClk,
  kCLOCK_PlatClk,
  kCLOCK_BusClk,
  kCLOCK_FlexBusClk,
  kCLOCK_FlashClk,
  kCLOCK_FastPeriphClk,
  kCLOCK_PllFllSelClk,
  kCLOCK_Er32kClk,
  kCLOCK_Osc0ErClk,
  kCLOCK_Osc1ErClk,
  kCLOCK_Osc0ErClkUndiv,
  kCLOCK_McgFixedFreqClk,
  kCLOCK_McgInternalRefClk,
  kCLOCK_McgFllClk,
  kCLOCK_McgPll0Clk,
  kCLOCK_McgPll1Clk,
  kCLOCK_McgExtPllClk,
  kCLOCK_McgPeriphClk,
  kCLOCK_McgIrc48MClk,
  kCLOCK_LpoClk }
  Clock name used to get clock frequency.
- enum clock_ip_name_t
  Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.
- enum osc_mode_t {
  kOSC_ModeExt = 0U,
  kOSC_ModeOscLowPower = MCG_C2_EREFS0_MASK,
  kOSC_ModeOscHighGain }
  OSC work mode.
- enum _osc_cap_load {
  kOSC_Cap2P = OSC_CR_SC2P_MASK,
  kOSC_Cap4P = OSC_CR_SC4P_MASK,
  kOSC_Cap8P = OSC_CR_SC8P_MASK,
  kOSC_Cap16P = OSC_CR_SC16P_MASK }
  Oscillator capacitor load setting.
- enum _oscer_enable_mode {

kOSC_ErClkEnable = OSC_CR_ERCLKEN_MASK,
kOSC_ErClkEnableInStop = OSC_CR_EREFSTEN_MASK }
OSCERCLK enable mode.
- enum mcg_fll_src_t {
kMCG_FllSrcExternal,
kMCG_FllSrcInternal }
MCG FLL reference clock source select.
- enum mcg_irc_mode_t {
kMCG_IrcSlow,
kMCG_IrcFast }
MCG internal reference clock select.
- enum mcg_dmx32_t {
kMCG_Dmx32Default,
kMCG_Dmx32Fine }
MCG DCO Maximum Frequency with 32.768 kHz Reference.
- enum mcg_drs_t {
kMCG_DrsLow,
kMCG_DrsMid,
kMCG_DrsMidHigh,
kMCG_DrsHigh }
MCG DCO range select.
- enum mcg_pll_ref_src_t {
kMCG_PllRefOsc0,
kMCG_PllRefOsc1 }
MCG PLL reference clock select.
- enum mcg_clkout_src_t {
kMCG_ClkOutSrcOut,
kMCG_ClkOutSrcInternal,
kMCG_ClkOutSrcExternal }
MCGOUT clock source.
- enum mcg_atm_select_t {
kMCG_AtmSel32k,
kMCG_AtmSel4m }
MCG Automatic Trim Machine Select.
- enum mcg_oscsel_t {
kMCG_OscselOsc,
kMCG_OscselRtc }
MCG OSC Clock Select.
- enum mcg_pll_clk_select_t { kMCG_PllClkSelPll0 }
MCG PLLCS select.
- enum mcg_monitor_mode_t {
kMCG_MonitorNone,
kMCG_MonitorInt,
kMCG_MonitorReset }
MCG clock monitor mode.
- enum _mcg_status {

kStatus_MCG_ModeUnreachable = MAKE_STATUS(kStatusGroup_MCG, 0),
kStatus_MCG_ModeInvalid = MAKE_STATUS(kStatusGroup_MCG, 1),
kStatus_MCG_AtmBusClockInvalid = MAKE_STATUS(kStatusGroup_MCG, 2),
kStatus_MCG_AtmDesiredFreqInvalid = MAKE_STATUS(kStatusGroup_MCG, 3),
kStatus_MCG_AtmIrcUsed = MAKE_STATUS(kStatusGroup_MCG, 4),
kStatus_MCG_AtmHardwareFail = MAKE_STATUS(kStatusGroup_MCG, 5),
kStatus_MCG_SourceUsed = MAKE_STATUS(kStatusGroup_MCG, 6) }
MCG status.

- enum _mcg_status_flags_t {
kMCG_Osc0LostFlag = (1U << 0U),
kMCG_Osc0InitFlag = (1U << 1U),
kMCG_Pll0LostFlag = (1U << 5U),
kMCG_Pll0LockFlag = (1U << 6U) }
MCG status flags.
- enum _mcg_irclk_enable_mode {
kMCG_IrclkEnable = MCG_C1_IRCLKEN_MASK,
kMCG_IrclkEnableInStop = MCG_C1_IREFSTEN_MASK }
MCG internal reference clock (MCGIRCLK) enable mode definition.
- enum _mcg_pll_enable_mode {
kMCG_PllEnableIndependent = MCG_C5_PLLCLKEN0_MASK,
kMCG_PllEnableInStop = MCG_C5_PLLSTEN0_MASK }
MCG PLL clock enable mode definition.
- enum mcg_mode_t {
kMCG_ModeFEI = 0U,
kMCG_ModeFBI,
kMCG_ModeBLPI,
kMCG_ModeFEE,
kMCG_ModeFBE,
kMCG_ModeBLPE,
kMCG_ModePBE,
kMCG_ModePEE,
kMCG_ModeError }
MCG mode definitions.

Functions

- static void CLOCK_EnableClock (clock_ip_name_t name)
Enable the clock for specific IP.
- static void CLOCK_DisableClock (clock_ip_name_t name)
Disable the clock for specific IP.
- static void CLOCK_SetEr32kClock (uint32_t src)
Set ERCLK32K source.
- static void CLOCK_SetEnetTime0Clock (uint32_t src)
Set EMVSIM clock source.
- static void CLOCK_SetTraceClock (uint32_t src, uint32_t divValue, uint32_t fracValue)
Set debug trace clock source.
- static void CLOCK_SetPllFllSelClock (uint32_t src)
Set PLLFLLSEL clock source.

- static void CLOCK_SetClkOutClock (uint32_t src)
  Set CLKOUT source.
- static void CLOCK_SetOutDiv (uint32_t outdiv1, uint32_t outdiv2, uint32_t outdiv3, uint32_t outdiv4)
  System clock divider.
- uint32_t CLOCK_GetFreq (clock_name_t clockName)
  Gets the clock frequency for a specific clock name.
- uint32_t CLOCK_GetCoreSysClkFreq (void)
  Get the core clock or system clock frequency.
- uint32_t CLOCK_GetFastPeriphClkFreq (void)
  Get the fast peripheral clock frequency.
- uint32_t CLOCK_GetFlexBusClkFreq (void)
  Get the flexbus clock frequency.
- uint32_t CLOCK_GetBusClkFreq (void)
  Get the bus clock frequency.
- uint32_t CLOCK_GetFlashClkFreq (void)
  Get the flash clock frequency.
- uint32_t CLOCK_GetPllFllSelClkFreq (void)
  Get the output clock frequency selected by SIM[PLLFLLSEL].
- uint32_t CLOCK_GetEr32kClkFreq (void)
  Get the external reference 32K clock frequency (ERCLK32K).
- uint32_t CLOCK_GetOsc0ErClkUndivFreq (void)
  Get the OSC0 external reference undivided clock frequency (OSC0ERCLK_UNDIV).
- uint32_t CLOCK_GetOsc0ErClkFreq (void)
  Get the OSC0 external reference clock frequency (OSC0ERCLK).
- void CLOCK_SetSimConfig (sim_clock_config_t const *config)
  Set the clock configure in SIM module.
- static void CLOCK_SetSimSafeDivs (void)
  Set the system clock dividers in SIM to safe value.

Variables

- uint32_t g_xtal0Freq
  External XTAL0 (OSC0) clock frequency.
- uint32_t g_xtal32Freq
  External XTAL32/EXTAL32/RTC_CLKIN clock frequency.

Driver version

- #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))
  CLOCK driver version 2.2.0.

MCG frequency functions.

- uint32_t CLOCK_GetOutClkFreq (void)
  Gets the MCG output clock (MCGOUTCLK) frequency.
- uint32_t CLOCK_GetFllFreq (void)
  Gets the MCG FLL clock (MCGFLLCLK) frequency.
- uint32_t CLOCK_GetInternalRefClkFreq (void)
  Gets the MCG internal reference clock (MCGIRCLK) frequency.
- uint32_t CLOCK_GetFixedFreqClkFreq (void)

Gets the MCG fixed frequency clock (MCGFFCLK) frequency.
- uint32_t CLOCK_GetPll0Freq (void)
  Gets the MCG PLL0 clock (MCGPLL0CLK) frequency.

MCG clock configuration.

- static void CLOCK_SetLowPowerEnable (bool enable)
  Enables or disables the MCG low power.
- status_t CLOCK_SetInternalRefClkConfig (uint8_t enableMode, mcg_irc_mode_t ircs, uint8_t fcrdiv)
  Configures the Internal Reference clock (MCGIRCLK).
- status_t CLOCK_SetExternalRefClkConfig (mcg_oscsel_t oscsel)
  Selects the MCG external reference clock.
- void CLOCK_EnablePll0 (mcg_pll_config_t const ∗config)
  Enables the PLL0 in FLL mode.
- static void CLOCK_DisablePll0 (void)
  Disables the PLL0 in FLL mode.
- uint32_t CLOCK_CalcPllDiv (uint32_t refFreq, uint32_t desireFreq, uint8_t ∗prdiv, uint8_t ∗vdiv)
  Calculates the PLL divider setting for a desired output frequency.

MCG clock lock monitor functions.

- void CLOCK_SetOsc0MonitorMode (mcg_monitor_mode_t mode)
  Sets the OSC0 clock monitor mode.
- void CLOCK_SetPll0MonitorMode (mcg_monitor_mode_t mode)
  Sets the PLL0 clock monitor mode.
- uint32_t CLOCK_GetStatusFlags (void)
  Gets the MCG status flags.
- void CLOCK_ClearStatusFlags (uint32_t mask)
  Clears the MCG status flags.

OSC configuration

- static void OSC_SetExtRefClkConfig (OSC_Type ∗base, oscer_config_t const ∗config)
  Configures the OSC external reference clock (OSCERCLK).
- static void OSC_SetCapLoad (OSC_Type ∗base, uint8_t capLoad)
  Sets the capacitor load configuration for the oscillator.
- void CLOCK_InitOsc0 (osc_config_t const ∗config)
  Initializes the OSC0.
- void CLOCK_DeinitOsc0 (void)
  Deinitializes the OSC0.

External clock frequency

- static void CLOCK_SetXtal0Freq (uint32_t freq)
  Sets the XTAL0 frequency based on board settings.
- static void CLOCK_SetXtal32Freq (uint32_t freq)
  Sets the XTAL32/RTC_CLKIN frequency based on board settings.

MCG auto-trim machine.

- status_t CLOCK_TrimInternalRefClk (uint32_t extFreq, uint32_t desireFreq, uint32_t ∗actualFreq, mcg_atm_select_t atms)
  Auto trims the internal reference clock.

MCG mode functions.

- mcg_mode_t CLOCK_GetMode (void)
  Gets the current MCG mode.
- status_t CLOCK_SetFeiMode (mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FEI mode.
- status_t CLOCK_SetFeeMode (uint8_t frdiv, mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FEE mode.
- status_t CLOCK_SetFbiMode (mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FBI mode.
- status_t CLOCK_SetFbeMode (uint8_t frdiv, mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FBE mode.
- status_t CLOCK_SetBlpiMode (void)
  Sets the MCG to BLPI mode.
- status_t CLOCK_SetBlpeMode (void)
  Sets the MCG to BLPE mode.
- status_t CLOCK_SetPbeMode (mcg_pll_clk_select_t pllcs, mcg_pll_config_t const ∗config)
  Sets the MCG to PBE mode.
- status_t CLOCK_SetPeeMode (void)
  Sets the MCG to PEE mode.
- status_t CLOCK_ExternalModeToFbeModeQuick (void)
  Switches the MCG to FBE mode from the external mode.
- status_t CLOCK_InternalModeToFbiModeQuick (void)
  Switches the MCG to FBI mode from internal modes.
- status_t CLOCK_BootToFeiMode (mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FEI mode during system boot up.
- status_t CLOCK_BootToFeeMode (mcg_oscsel_t oscsel, uint8_t frdiv, mcg_dmx32_t dmx32, mcg_drs_t drs, void(∗fllStableDelay)(void))
  Sets the MCG to FEE mode during system bootup.
- status_t CLOCK_BootToBlpiMode (uint8_t fcrdiv, mcg_irc_mode_t ircs, uint8_t ircEnableMode)
  Sets the MCG to BLPI mode during system boot up.
- status_t CLOCK_BootToBlpeMode (mcg_oscsel_t oscsel)
  Sets the MCG to BLPE mode during sytem boot up.
- status_t CLOCK_BootToPeeMode (mcg_oscsel_t oscsel, mcg_pll_clk_select_t pllcs, mcg_pll_config_t const ∗config)
  Sets the MCG to PEE mode during system boot up.
- status_t CLOCK_SetMcgConfig (mcg_config_t const ∗config)
  Sets the MCG to a target mode.

7.4 Data Structure Documentation

7.4.1 struct sim_clock_config_t

Data Fields

- uint8_t pllFllSel
  PLL/FLL/IRC48M selection.
- uint8_t er32kSrc
  ERCLK32K source selection.
- uint32_t clkdiv1
  SIM_CLKDIV1.

7.4.1.0.0.4 Field Documentation

7.4.1.0.0.4.1 uint8_t sim_clock_config_t::pllFllSel

7.4.1.0.0.4.2 uint8_t sim_clock_config_t::er32kSrc

7.4.1.0.0.4.3 uint32_t sim_clock_config_t::clkdiv1

7.4.2 struct oscer_config_t

Data Fields

- uint8_t enableMode
  OSCERCLK enable mode.
- uint8_t erclkDiv
  Divider for OSCERCLK.

7.4.2.0.0.5 Field Documentation

7.4.2.0.0.5.1 uint8_t oscer_config_t::enableMode

OR'ed value of _oscer_enable_mode.

7.4.2.0.0.5.2 uint8_t oscer_config_t::erclkDiv

7.4.3 struct osc_config_t

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

1. freq: The external frequency.
2. workMode: The OSC module mode.

Data Fields

- uint32_t freq
  External clock frequency.
- uint8_t capLoad
  Capacitor load setting.
- osc_mode_t workMode
  OSC work mode setting.
- oscer_config_t oscerConfig
  Configuration for OSCERCLK.

7.4.3.0.0.6 Field Documentation

7.4.3.0.0.6.1 uint32_t osc_config_t::freq

7.4.3.0.0.6.2 uint8_t osc_config_t::capLoad

7.4.3.0.0.6.3 osc_mode_t osc_config_t::workMode

7.4.3.0.0.6.4 oscer_config_t osc_config_t::oscerConfig

7.4.4 struct mcg_pll_config_t

Data Fields

- uint8_t enableMode
  Enable mode.
- uint8_t prdiv
  Reference divider PRDIV.
- uint8_t vdiv
  VCO divider VDIV.

7.4.4.0.0.7 Field Documentation

7.4.4.0.0.7.1 uint8_t mcg_pll_config_t::enableMode

OR'ed value of _mcg_pll_enable_mode.

7.4.4.0.0.7.2 uint8_t mcg_pll_config_t::prdiv

7.4.4.0.0.7.3 uint8_t mcg_pll_config_t::vdiv

7.4.5 struct mcg_config_t

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

1. frdiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by frdiv is in the 31.25 kHz to 39.0625 kHz range.
2. The PLL reference clock divider PRDIV: PLL reference clock frequency after PRDIV should be in

the FSL_FEATURE_MCG_PLL_REF_MIN to FSL_FEATURE_MCG_PLL_REF_MAX range.

Data Fields

- mcg_mode_t mcgMode
  MCG mode.
- uint8_t irclkEnableMode
  MCGIRCLK enable mode.
- mcg_irc_mode_t ircs
  Source, MCG_C2[IRCS].
- uint8_t fcrdiv
  Divider, MCG_SC[FCRDIV].
- uint8_t frdiv
  Divider MCG_C1[FRDIV].
- mcg_drs_t drs
  DCO range MCG_C4[DRST_DRS].
- mcg_dmx32_t dmx32
  MCG_C4[DMX32].
- mcg_pll_config_t pll0Config
  MCGPLL0CLK configuration.

7.4.5.0.0.8 Field Documentation

7.4.5.0.0.8.1 mcg_mode_t mcg_config_t::mcgMode

7.4.5.0.0.8.2 uint8_t mcg_config_t::irclkEnableMode

7.4.5.0.0.8.3 mcg_irc_mode_t mcg_config_t::ircs

7.4.5.0.0.8.4 uint8_t mcg_config_t::fcrdiv

7.4.5.0.0.8.5 uint8_t mcg_config_t::frdiv

7.4.5.0.0.8.6 mcg_drs_t mcg_config_t::drs

7.4.5.0.0.8.7 mcg_dmx32_t mcg_config_t::dmx32

7.4.5.0.0.8.8 mcg_pll_config_t mcg_config_t::pll0Config

7.5 Macro Definition Documentation

7.5.1 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))

7.5.2 #define DMAMUX_CLOCKS

Value:

```
{                    \
        kCLOCK_Dmamux0 \
    }
```

7.5.3 #define HSADC_CLOCKS

Value:

```
{                                \
        kCLOCK_Hsadc0, kCLOCK_Hsadc1 \
    }
```

7.5.4 #define ENET_CLOCKS

Value:

```
{                \
        kCLOCK_Enet0 \
    }
```

7.5.5 #define PORT_CLOCKS

Value:

```
{                                                                         \
        kCLOCK_PortA, kCLOCK_PortB, kCLOCK_PortC, kCLOCK_PortD, kCLOCK_PortE \
    }
```

7.5.6 #define FLEXBUS_CLOCKS

Value:

```
{                   \
        kCLOCK_Flexbus0 \
    }
```

7.5.7 #define ENC_CLOCKS

Value:

```
{               \
        kCLOCK_Enc0 \
    }
```

7.5.8 #define EWM_CLOCKS

Value:

```
{                \
        kCLOCK_Ewm0 \
    }
```

7.5.9 #define PIT_CLOCKS

Value:

```
{                \
        kCLOCK_Pit0 \
    }
```

7.5.10 #define DSPI_CLOCKS

Value:

```
{                                          \
        kCLOCK_Spi0, kCLOCK_Spi1, kCLOCK_Spi2 \
    }
```

7.5.11 #define LPTMR_CLOCKS

Value:

```
{                  \
        kCLOCK_Lptmr0 \
    }
```

7.5.12 #define FTM_CLOCKS

Value:

```
{                                                       \
        kCLOCK_Ftm0, kCLOCK_Ftm1, kCLOCK_Ftm2, kCLOCK_Ftm3 \
    }
```

7.5.13 #define EDMA_CLOCKS

Value:

```
{              \
        kCLOCK_Dma0 \
    }
```

7.5.14 #define FLEXCAN_CLOCKS

Value:

```
{                                                    \
        kCLOCK_Flexcan0, kCLOCK_Flexcan1, kCLOCK_Flexcan2 \
    }
```

7.5.15 #define DAC_CLOCKS

Value:

```
{              \
        kCLOCK_Dac0 \
    }
```

7.5.16 #define ADC16_CLOCKS

Value:

```
{               \
        kCLOCK_Adc0  \
    }
```

7.5.17 #define XBARA_CLOCKS

Value:

```
{               \
        kCLOCK_XbarA \
    }
```

7.5.18 #define XBARB_CLOCKS

Value:

```
{                \
        kCLOCK_XbarB \
    }
```

7.5.19 #define AOI_CLOCKS

Value:

```
{               \
        kCLOCK_Aoi0 \
    }
```

7.5.20 #define TRNG_CLOCKS

Value:

```
{                \
        kCLOCK_Trng0 \
    }
```

7.5.21 #define MPU_CLOCKS

Value:

```
{               \
        kCLOCK_Mpu0 \
    }
```

7.5.22 #define PWM_CLOCKS

Value:

```
{                                                                           \
        {                                                                       \
            kCLOCK_Pwm0_Sm0, kCLOCK_Pwm0_Sm1, kCLOCK_Pwm0_Sm2, kCLOCK_Pwm0_Sm3 \
        }                                                                       \
        ,                                                                       \
        {                                                                       \
            kCLOCK_Pwm1_Sm0, kCLOCK_Pwm1_Sm1, kCLOCK_Pwm1_Sm2, kCLOCK_Pwm1_Sm3 \
        }                                                                       \
    }
```

7.5.23 #define UART_CLOCKS

Value:

```
{                                                                                  \
        kCLOCK_Uart0, kCLOCK_Uart1, kCLOCK_Uart2, kCLOCK_Uart3, kCLOCK_Uart4, kCLOCK_Uart5 \
    }
```

7.5.24 #define CRC_CLOCKS

Value:

```
{              \
        kCLOCK_Crc0 \
    }
```

7.5.25 #define I2C_CLOCKS

Value:

```
{                            \
        kCLOCK_I2c0, kCLOCK_I2c1 \
    }
```

7.5.26 #define PDB_CLOCKS

Value:

```
{                            \
        kCLOCK_Pdb0, kCLOCK_Pdb1 \
    }
```

7.5.27 #define CMP_CLOCKS

Value:

```
{                                                      \
        kCLOCK_Cmp0, kCLOCK_Cmp1, kCLOCK_Cmp2, kCLOCK_Cmp3 \
    }
```

7.5.28 #define FTF_CLOCKS

Value:

```
{                \
        kCLOCK_Ftf0 \
    }
```

7.5.29 #define SYS_CLK kCLOCK_CoreSysClk

7.6 Enumeration Type Documentation

7.6.1 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock.
kCLOCK_PlatClk Platform clock.
kCLOCK_BusClk Bus clock.
kCLOCK_FlexBusClk FlexBus clock.
kCLOCK_FlashClk Flash clock.
kCLOCK_FastPeriphClk Fast peripheral clock.
kCLOCK_PllFllSelClk The clock after SIM[PLLFLLSEL].
kCLOCK_Er32kClk External reference 32K clock (ERCLK32K)
kCLOCK_Osc0ErClk OSC0 external reference clock (OSC0ERCLK)
kCLOCK_Osc1ErClk OSC1 external reference clock (OSC1ERCLK)
kCLOCK_Osc0ErClkUndiv OSC0 external reference undivided clock(OSC0ERCLK_UNDIV).
kCLOCK_McgFixedFreqClk MCG fixed frequency clock (MCGFFCLK)
kCLOCK_McgInternalRefClk MCG internal reference clock (MCGIRCLK)
kCLOCK_McgFllClk MCGFLLCLK.
kCLOCK_McgPll0Clk MCGPLL0CLK.
kCLOCK_McgPll1Clk MCGPLL1CLK.
kCLOCK_McgExtPllClk EXT_PLLCLK.
kCLOCK_McgPeriphClk MCG peripheral clock (MCGPCLK)
kCLOCK_McgIrc48MClk MCG IRC48M clock.
kCLOCK_LpoClk LPO clock.

7.6.2 enum clock_ip_name_t

7.6.3 enum osc_mode_t

Enumerator

kOSC_ModeExt Use an external clock.

kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

7.6.4 enum _osc_cap_load

Enumerator

kOSC_Cap2P 2 pF capacitor load
kOSC_Cap4P 4 pF capacitor load
kOSC_Cap8P 8 pF capacitor load
kOSC_Cap16P 16 pF capacitor load

7.6.5 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.
kOSC_ErClkEnableInStop Enable in stop mode.

7.6.6 enum mcg_fll_src_t

Enumerator

kMCG_FllSrcExternal External reference clock is selected.
kMCG_FllSrcInternal The slow internal reference clock is selected.

7.6.7 enum mcg_irc_mode_t

Enumerator

kMCG_IrcSlow Slow internal reference clock selected.
kMCG_IrcFast Fast internal reference clock selected.

7.6.8 enum mcg_dmx32_t

Enumerator

kMCG_Dmx32Default DCO has a default range of 25%.
kMCG_Dmx32Fine DCO is fine-tuned for maximum frequency with 32.768 kHz reference.

7.6.9 enum mcg_drs_t

Enumerator

kMCG_DrsLow Low frequency range.
kMCG_DrsMid Mid frequency range.
kMCG_DrsMidHigh Mid-High frequency range.
kMCG_DrsHigh High frequency range.

7.6.10 enum mcg_pll_ref_src_t

Enumerator

kMCG_PllRefOsc0 Selects OSC0 as PLL reference clock.
kMCG_PllRefOsc1 Selects OSC1 as PLL reference clock.

7.6.11 enum mcg_clkout_src_t

Enumerator

kMCG_ClkOutSrcOut Output of the FLL is selected (reset default)
kMCG_ClkOutSrcInternal Internal reference clock is selected.
kMCG_ClkOutSrcExternal External reference clock is selected.

7.6.12 enum mcg_atm_select_t

Enumerator

kMCG_AtmSel32k 32 kHz Internal Reference Clock selected
kMCG_AtmSel4m 4 MHz Internal Reference Clock selected

7.6.13 enum mcg_oscsel_t

Enumerator

kMCG_OscselOsc Selects System Oscillator (OSCCLK)
kMCG_OscselRtc Selects 32 kHz RTC Oscillator.

7.6.14 enum mcg_pll_clk_select_t

Enumerator

kMCG_PllClkSelPll0 PLL0 output clock is selected.

7.6.15 enum mcg_monitor_mode_t

Enumerator

kMCG_MonitorNone Clock monitor is disabled.
kMCG_MonitorInt Trigger interrupt when clock lost.
kMCG_MonitorReset System reset when clock lost.

7.6.16 enum _mcg_status

Enumerator

kStatus_MCG_ModeUnreachable Can't switch to target mode.
kStatus_MCG_ModeInvalid Current mode invalid for the specific function.
kStatus_MCG_AtmBusClockInvalid Invalid bus clock for ATM.
kStatus_MCG_AtmDesiredFreqInvalid Invalid desired frequency for ATM.
kStatus_MCG_AtmIrcUsed IRC is used when using ATM.
kStatus_MCG_AtmHardwareFail Hardware fail occurs during ATM.
kStatus_MCG_SourceUsed Can't change the clock source because it is in use.

7.6.17 enum _mcg_status_flags_t

Enumerator

kMCG_Osc0LostFlag OSC0 lost.
kMCG_Osc0InitFlag OSC0 crystal initialized.
kMCG_Pll0LostFlag PLL0 lost.
kMCG_Pll0LockFlag PLL0 locked.

7.6.18 enum _mcg_irclk_enable_mode

Enumerator

kMCG_IrclkEnable MCGIRCLK enable.
kMCG_IrclkEnableInStop MCGIRCLK enable in stop mode.

7.6.19 enum _mcg_pll_enable_mode

Enumerator

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

kMCG_PllEnableInStop MCGPLLCLK enable in STOP mode.

7.6.20 enum mcg_mode_t

Enumerator

kMCG_ModeFEI FEI - FLL Engaged Internal.
kMCG_ModeFBI FBI - FLL Bypassed Internal.
kMCG_ModeBLPI BLPI - Bypassed Low Power Internal.
kMCG_ModeFEE FEE - FLL Engaged External.
kMCG_ModeFBE FBE - FLL Bypassed External.
kMCG_ModeBLPE BLPE - Bypassed Low Power External.
kMCG_ModePBE PBE - PLL Bypassed External.
kMCG_ModePEE PEE - PLL Engaged External.
kMCG_ModeError Unknown mode.

7.7 Function Documentation

7.7.1 static void CLOCK_EnableClock (clock_ip_name_t *name*) [inline], [static]

Parameters

name	Which clock to enable, see clock_ip_name_t.

7.7.2 static void CLOCK_DisableClock (clock_ip_name_t *name*) [inline], [static]

Parameters

name	Which clock to disable, see clock_ip_name_t.

7.7.3 static void CLOCK_SetEr32kClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set ERCLK32K clock source.

7.7.4 static void CLOCK_SetEnetTime0Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set enet timestamp clock source.

7.7.5 static void CLOCK_SetTraceClock (uint32_t *src,* uint32_t *divValue,* uint32_t *fracValue*) [inline], [static]

Parameters

src	The value to set debug trace clock source.

7.7.6 static void CLOCK_SetPllFllSelClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set PLLFLLSEL clock source.

7.7.7 static void CLOCK_SetClkOutClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set CLKOUT source.

7.7.8 static void CLOCK_SetOutDiv (uint32_t *outdiv1,* uint32_t *outdiv2,* uint32_t *outdiv3,* uint32_t *outdiv4*) [inline], [static]

Set the SIM_CLKDIV1[OUTDIV1], SIM_CLKDIV1[OUTDIV2], SIM_CLKDIV1[OUTDIV3], SIM_-CLKDIV1[OUTDIV4].

Parameters

outdiv1	Clock 1 output divider value.
outdiv2	Clock 2 output divider value.
outdiv3	Clock 3 output divider value.
outdiv4	Clock 4 output divider value.

7.7.9 uint32_t CLOCK_GetFreq (clock_name_t *clockName*)

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

Parameters

clockName	Clock names defined in clock_name_t

Returns

Clock frequency value in Hertz

7.7.10 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Clock frequency in Hz.

7.7.11 uint32_t CLOCK_GetFastPeriphClkFreq (void)

Returns

Clock frequency in Hz.

7.7.12 uint32_t CLOCK_GetFlexBusClkFreq (void)

Returns

Clock frequency in Hz.

7.7.13 uint32_t CLOCK_GetBusClkFreq (void)

Returns

Clock frequency in Hz.

7.7.14 uint32_t CLOCK_GetFlashClkFreq (void)

Returns

Clock frequency in Hz.

7.7.15 uint32_t CLOCK_GetPllFllSelClkFreq (void)

Returns

Clock frequency in Hz.

7.7.16 uint32_t CLOCK_GetEr32kClkFreq (void)

Returns

Clock frequency in Hz.

7.7.17 uint32_t CLOCK_GetOsc0ErClkUndivFreq (void)

Returns

Clock frequency in Hz.

7.7.18 uint32_t CLOCK_GetOsc0ErClkFreq (void)

Returns

Clock frequency in Hz.

7.7.19 void CLOCK_SetSimConfig (sim_clock_config_t const * *config*)

This function sets system layer clock settings in SIM module.

Parameters

config	Pointer to the configure structure.

7.7.20 static void CLOCK_SetSimSafeDivs (void) [inline], [static]

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

Parameters

config	Pointer to the configure structure.

7.7.21 uint32_t CLOCK_GetOutClkFreq (void)

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

Returns

The frequency of MCGOUTCLK.

7.7.22 uint32_t CLOCK_GetFllFreq (void)

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

Returns

The frequency of MCGFLLCLK.

7.7.23 uint32_t CLOCK_GetInternalRefClkFreq (void)

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

Returns

The frequency of MCGIRCLK.

7.7.24 uint32_t CLOCK_GetFixedFreqClkFreq (void)

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

Returns

The frequency of MCGFFCLK.

7.7.25 uint32_t CLOCK_GetPll0Freq (void)

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

Returns

The frequency of MCGPLL0CLK.

7.7.26 static void CLOCK_SetLowPowerEnable (bool *enable*) [inline], [static]

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

Parameters

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

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

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

Parameters

enableMode	MCGIRCLK enable mode, OR'ed value of _mcg_irclk_enable_mode.
ircs	MCGIRCLK clock source, choose fast or slow.
fcrdiv	Fast IRC divider setting (`FCRDIV`).

Return values

kStatus_MCG_Source-Used	Because the internall reference clock is used as a clock source, the confuration should not be changed. Otherwise, a glitch occurs.
kStatus_Success	MCGIRCLK configuration finished successfully.

7.7.28 status_t CLOCK_SetExternalRefClkConfig (mcg_oscsel_t *oscsel*)

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

Parameters

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

Return values

kStatus_MCG_Source-Used	Because the external reference clock is used as a clock source, the confuration should not be changed. Otherwise, a glitch occurs.

kStatus_Success	External reference clock set successfully.

7.7.29 void CLOCK_EnablePll0 (mcg_pll_config_t const * *config*)

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

Parameters

config	Pointer to the configuration structure.

7.7.30 static void CLOCK_DisablePll0 (void) [inline], [static]

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

7.7.31 uint32_t CLOCK_CalcPllDiv (uint32_t *refFreq*, uint32_t *desireFreq*, uint8_t * *prdiv*, uint8_t * *vdiv*)

This function calculates the correct reference clock divider (`PRDIV`) and VCO divider (`VDIV`) to generate a desired PLL output frequency. It returns the closest frequency match with the corresponding `PRDIV/-VDIV` returned from parameters. If a desired frequency is not valid, this function returns 0.

Parameters

refFreq	PLL reference clock frequency.
desireFreq	Desired PLL output frequency.
prdiv	PRDIV value to generate desired PLL frequency.
vdiv	VDIV value to generate desired PLL frequency.

Returns

Closest frequency match that the PLL was able generate.

7.7.32 void CLOCK_SetOsc0MonitorMode (mcg_monitor_mode_t *mode*)

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

Parameters

mode	Monitor mode to set.

7.7.33 void CLOCK_SetPll0MonitorMode (mcg_monitor_mode_t *mode*)

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

Parameters

mode	Monitor mode to set.

7.7.34 uint32_t CLOCK_GetStatusFlags (void)

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

Example:

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

mcgFlags = CLOCK_GetStatusFlags();

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

Returns

Logical OR value of the _mcg_status_flags_t.

7.7.35 void CLOCK_ClearStatusFlags (uint32_t *mask*)

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

Example:

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

CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Parameters

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

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

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

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

OSC_SetExtRefClkConfig(OSC, &config);
```

Parameters

base	OSC peripheral address.
config	Pointer to the configuration structure.

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

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

Parameters

base	OSC peripheral address.
capLoad	OR'ed value for the capacitor load option, see _osc_cap_load.

Example:

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

7.7.38 void CLOCK_InitOsc0 (osc_config_t const * *config*)

This function initializes the OSC0 according to the board configuration.

Parameters

config	Pointer to the OSC0 configuration structure.

7.7.39 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

7.7.40 static void CLOCK_SetXtal0Freq (uint32_t *freq*) [inline], [static]

Parameters

freq	The XTAL0/EXTAL0 input clock frequency in Hz.

7.7.41 static void CLOCK_SetXtal32Freq (uint32_t *freq*) [inline], [static]

Parameters

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

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

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

Parameters

extFreq	External clock frequency, which should be a bus clock.
desireFreq	Frequency to trim to.
actualFreq	Actual frequency after trimming.

atms	Trim fast or slow internal reference clock.

Return values

kStatus_Success	ATM success.
kStatus_MCG_AtmBus-ClockInvalid	The bus clock is not in allowed range for the ATM.
kStatus_MCG_Atm-DesiredFreqInvalid	MCGIRCLK could not be trimmed to the desired frequency.
kStatus_MCG_AtmIrc-Used	Could not trim because MCGIRCLK is used as a bus clock source.
kStatus_MCG_Atm-HardwareFail	Hardware fails while trimming.

7.7.43 mcg_mode_t CLOCK_GetMode (void)

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

Returns

Current MCG mode or error code; See mcg_mode_t.

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

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

Parameters

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

Note

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

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

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

Parameters

frdiv	FLL reference clock divider setting, FRDIV.
dmx32	DMX32 in FEE mode.
drs	The DCO range selection.
fllStableDelay	Delay function to make sure FLL is stable. Passing NULL does not cause a delay.

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

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

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

Parameters

dmx32	DMX32 in FBI mode.
drs	The DCO range selection.
fllStableDelay	Delay function to make sure FLL is stable. If the FLL is not used in FBI mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

Note

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

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

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

Parameters

frdiv	FLL reference clock divider setting, FRDIV.
dmx32	DMX32 in FBE mode.
drs	The DCO range selection.
fllStableDelay	Delay function to make sure FLL is stable. If the FLL is not used in FBE mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.

kStatus_Success	Switched to the target mode successfully.

7.7.48 status_t CLOCK_SetBlpiMode (void)

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

7.7.49 status_t CLOCK_SetBlpeMode (void)

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

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

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

Parameters

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

Note

1. The parameter `pllcs` selects the PLL. For platforms with only one PLL, the parameter pllcs is kept for interface compatibility.
2. The parameter `config` is the PLL configuration structure. On some platforms, it is possible to choose the external PLL directly, which renders the configuration structure not necessary. In this case, pass in NULL. For example: CLOCK_SetPbeMode(kMCG_OscselOsc, kMCG_PllClkSelExtPll, NULL);

7.7.51 status_t CLOCK_SetPeeMode (void)

This function sets the MCG to PEE mode.

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

Note

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

7.7.52 status_t CLOCK_ExternalModeToFbeModeQuick (void)

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

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

Return values

kStatus_Success	Switched successfully.
kStatus_MCG_Mode-Invalid	If the current mode is not an external mode, do not call this function.

7.7.53 status_t CLOCK_InternalModeToFbiModeQuick (void)

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

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

Return values

kStatus_Success	Switched successfully.
kStatus_MCG_Mode-Invalid	If the current mode is not an internal mode, do not call this function.

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

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

Parameters

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

Note

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

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

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

Parameters

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

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

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

Parameters

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

Return values

kStatus_MCG_Source-Used	Could not change MCGIRCLK setting.
kStatus_Success	Switched to the target mode successfully.

7.7.57 status_t CLOCK_BootToBlpeMode (mcg_oscsel_t *oscsel*)

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

Parameters

oscsel	OSC clock select, MCG_C7[OSCSEL].

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

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

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

Parameters

oscsel	OSC clock select, MCG_C7[OSCSEL].

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

Return values

kStatus_MCG_Mode-Unreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

7.7.59 status_t CLOCK_SetMcgConfig (mcg_config_t const ∗ *config*)

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

Parameters

config	Pointer to the target MCG mode configuration structure.

Returns

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

Note

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

7.8 Variable Documentation

7.8.1 uint32_t g_xtal0Freq

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

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

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

7.8.2 uint32_t g_xtal32Freq

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

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the CLOCK_SetXtal32Freq to get a valid clock frequency.

7.9 Multipurpose Clock Generator (MCG)

The KSDK provides a peripheral driver for the MCG module of Kinetis devices.

7.9.1 Function description

MCG driver provides these functions:

- Functions to get the MCG clock frequency.
- Functions to configure the MCG clock, such as PLLCLK and MCGIRCLK.
- Functions for the MCG clock lock lost monitor.
- Functions for the OSC configuration.
- Functions for the MCG auto-trim machine.
- Functions for the MCG mode.

7.9.1.1 MCG frequency functions

MCG module provides clocks, such as MCGOUTCLK, MCGIRCLK, MCGFFCLK, MCGFLLCLK and MCGPLLCLK. The MCG driver provides functions to get the frequency of these clocks, such as CLOCK_GetOutClkFreq(), CLOCK_GetInternalRefClkFreq(), CLOCK_GetFixedFreqClkFreq(), CLOCK_GetFllFreq(), CLOCK_GetPll0Freq(), CLOCK_GetPll1Freq(), and CLOCK_GetExtPllFreq(). These functions get the clock frequency based on the current MCG registers.

7.9.1.2 MCG clock configuration

The MCG driver provides functions to configure the internal reference clock (MCGIRCLK), the external reference clock, and MCGPLLCLK.

The function CLOCK_SetInternalRefClkConfig() configures the MCGIRCLK, including the source and the driver. Do not change MCGIRCLK when the MCG mode is BLPI/FBI/PBI because the MCGIRCLK is used as a system clock in these modes and changing settings makes the system clock unstable.

The function CLOCK_SetExternalRefClkConfig() configures the external reference clock source (MCG_C7[OSCSEL]). Do not call this function when the MCG mode is BLPE/FBE/PBE/FEE/PEE because the external reference clock is used as a clock source in these modes. Changing the external reference clock source requires at least a 50 micro seconds wait. The function CLOCK_SetExternalRefClkConfig() implements a for loop delay internally. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 50 micro seconds delay. However, when the system clock is slow, the delay time may significantly increase. This for loop count can be optimized for better performance for specific cases.

The MCGPLLCLK is disabled in FBE/FEE/FBI/FEI modes by default. Applications can enable the MCGPLLCLK in these modes using the functions CLOCK_EnablePll0() and CLOCK_EnablePll1(). To enable the MCGPLLCLK, the PLL reference clock divider(PRDIV) and the PLL VCO divider(VDIV) must be set to a proper value. The function CLOCK_CalcPllDiv() helps to get the PRDIV/VDIV.

7.9.1.3 MCG clock lock monitor functions

The MCG module monitors the OSC and the PLL clock lock status. The MCG driver provides the functions to set the clock monitor mode, check the clock lost status, and clear the clock lost status.

7.9.1.4 OSC configuration

The MCG is needed together with the OSC module to enable the OSC clock. The function CLOCK_InitOsc0() CLOCK_InitOsc1 uses the MCG and OSC to initialize the OSC. The OSC should be configured based on the board design.

7.9.1.5 MCG auto-trim machine

The MCG provides an auto-trim machine to trim the MCG internal reference clock based on the external reference clock (BUS clock). During clock trimming, the MCG must not work in FEI/FBI/BLPI/PBI/PEI modes. The function CLOCK_TrimInternalRefClk() is used for the auto clock trimming.

7.9.1.6 MCG mode functions

The function CLOCK_GetMcgMode returns the current MCG mode. The MCG can only switch between the neighbouring modes. If the target mode is not current mode's neighbouring mode, the application must choose the proper switch path. For example, to switch to PEE mode from FEI mode, use FEI -> FBE -> PBE -> PEE.

For the MCG modes, the MCG driver provides three kinds of functions:

The first type of functions involve functions CLOCK_SetXxxMode, such as CLOCK_SetFeiMode(). These functions only set the MCG mode from neighbouring modes. If switching to the target mode directly from current mode is not possible, the functions return an error.

The second type of functions are the functions CLOCK_BootToXxxMode, such as CLOCK_BootToFeiMode(). These functions set the MCG to specific modes from reset mode. Because the source mode and target mode are specific, these functions choose the best switch path. The functions are also useful to set up the system clock during boot up.

The third type of functions is the CLOCK_SetMcgConfig(). This function chooses the right path to switch to the target mode. It is easy to use, but introduces a large code size.

Whenever the FLL settings change, there should be a 1 millisecond delay to ensure that the FLL is stable. The function CLOCK_SetMcgConfig() implements a for loop delay internally to ensure that the FLL is stable. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 1 millisecond delay. However, when the system clock is slow, the delay time may increase significantly. The for loop count can be optimized for better performance according to a specific case.

7.9.2 Typical use case

The function CLOCK_SetMcgConfig is used to switch between any modes. However, this heavy-light function introduces a large code size. This section shows how to use the mode function to implement a quick and light-weight switch between typical specific modes. Note that the step to enable the external clock is not included in the following steps. T Enable the corresponding clock before using it as a clock source.

7.9.2.1 Switch between BLPI and FEI

Use case	Steps	Functions
BLPI -> FEI	BLPI -> FBI	CLOCK_InternalModeToFbiModeQuick(...)
	FBI -> FEI	CLOCK_SetFeiMode(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig(...)
FEI -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig(...)
	FEI -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPowerEnable(true)

7.9.2.2 Switch between BLPI and FEE

Use case	Steps	Functions
BLPI -> FEE	BLPI -> FBI	CLOCK_InternalModeToFbiModeQuick(...)
	Change external clock source if need	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig(...)
	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPowerEnable(true)

7.9.2.3 Switch between BLPI and PEE

Use case	Steps	Functions
BLPI -> PEE	BLPI -> FBI	CLOCK_InternalModeToFbiModeQuick(...)
	Change external clock source if need	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) // fllStableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPI	PEE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPowerEnable(true)

7.9.2.4 Switch between BLPE and PEE

This table applies when using the same external clock source (MCG_C7[OSCSEL]) in BLPE mode and PEE mode.

Use case	Steps	Functions
BLPE -> PEE	BLPE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

If using different external clock sources (MCG_C7[OSCSEL]) in BLPE mode and PEE mode, call the CLOCK_SetExternalRefClkConfig() in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
BLPE -> PEE	BLPE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)

	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

7.9.2.5 Switch between BLPE and FEE

This table applies when using the same external clock source (MCG_C7[OSCSEL]) in BLPE mode and FEE mode.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	FBE -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	PEE -> FBE	CLOCK_SetPbeMode(...)
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

If using different external clock sources (MCG_C7[OSCSEL]) in BLPE mode and FEE mode, call the CLOCK_SetExternalRefClkConfig() in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)

	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

7.9.2.6 Switch between BLPI and PEI

Use case	Steps	Functions
BLPI -> PEI	BLPI -> PBI	CLOCK_SetPbiMode(...)
	PBI -> PEI	CLOCK_SetPeiMode(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config(...)
PEI -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config
	PEI -> FBI	CLOCK_InternalModeToFbi-ModeQuick(...)
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

Chapter 8
CMP: Analog Comparator Driver

8.1 Overview

The KSDK provides a peripheral driver for the Analog Comparator (CMP) module of Kinetis devices.

The CMP driver is a basic comparator with advanced features. The APIs for the basic comparator enable the CMP to compare the two voltages of the two input channels and create the output of the comparator result. The APIs for advanced features can be used as the plug-in functions based on the basic comparator. They can process the comparator's output with hardware support.

8.2 Typical use case

8.2.1 Polling Configuration

```
int main(void)
{
    cmp_config_t mCmpConfigStruct;
    cmp_dac_config_t mCmpDacConfigStruct;

    // ...

    // Configures the comparator.
    CMP_Init(DEMO_CMP_INSTANCE);
    CMP_GetDefaultConfig(&mCmpConfigStruct);
    CMP_Configure(DEMO_CMP_INSTANCE, &mCmpConfigStruct);

    // Configures the DAC channel.
    mCmpDacConfigStruct.referenceVoltageSource =
      kCMP_VrefSourceVin2; // VCC.
    mCmpDacConfigStruct.DACValue = 32U; // Half voltage of logic high-level.
    CMP_SetDACConfig(DEMO_CMP_INSTANCE, &mCmpDacConfigStruct);
    CMP_SetInputChannels(DEMO_CMP_INSTANCE, DEMO_CMP_USER_CHANNEL, DEMO_CMP_DAC_CHANNEL
      );

    while (1)
    {
        if (0U != (kCMP_OutputAssertEventFlag &
      CMP_GetStatusFlags(DEMO_CMP_INSTANCE)))
        {
            // Do something.
        }
        else
        {
            // Do something.
        }
    }
}
```

8.2.2 Interrupt Configuration

```
volatile uint32_t g_CmpFlags = 0U;
```

```
// ...

void DEMO_CMP_IRQ_HANDLER_FUNC(void)
{
    g_CmpFlags = CMP_GetStatusFlags(DEMO_CMP_INSTANCE);
    CMP_ClearStatusFlags(DEMO_CMP_INSTANCE, kCMP_OutputRisingEventFlag |
      kCMP_OutputFallingEventFlag);
    if (0U != (g_CmpFlags & kCMP_OutputRisingEventFlag))
    {
        // Do something.
    }
    else if (0U != (g_CmpFlags & kCMP_OutputFallingEventFlag))
    {
        // Do something.
    }
}

int main(void)
{
    cmp_config_t mCmpConfigStruct;
    cmp_dac_config_t mCmpDacConfigStruct;

    // ...
    EnableIRQ(DEMO_CMP_IRQ_ID);
    // ...

    // Configures the comparator.
    CMP_Init(DEMO_CMP_INSTANCE);
    CMP_GetDefaultConfig(&mCmpConfigStruct);
    CMP_Configure(DEMO_CMP_INSTANCE, &mCmpConfigStruct);

    // Configures the DAC channel.
    mCmpDacConfigStruct.referenceVoltageSource =
      kCMP_VrefSourceVin2; // VCC.
    mCmpDacConfigStruct.DACValue = 32U; // Half voltage of logic high-level.
    CMP_SetDACConfig(DEMO_CMP_INSTANCE, &mCmpDacConfigStruct);
    CMP_SetInputChannels(DEMO_CMP_INSTANCE, DEMO_CMP_USER_CHANNEL, DEMO_CMP_DAC_CHANNEL
      );

    // Enables the output rising and falling interrupts.
    CMP_EnableInterrupts(DEMO_CMP_INSTANCE,
      kCMP_OutputRisingInterruptEnable |
      kCMP_OutputFallingInterruptEnable);

    while (1)
    {
    }
}
```

Files

- file fsl_cmp.h

Data Structures

- struct cmp_config_t
  Configures the comparator. More...
- struct cmp_filter_config_t
  Configures the filter. More...
- struct cmp_dac_config_t
  Configures the internal DAC. More...

Enumerations

- enum _cmp_interrupt_enable {
  kCMP_OutputRisingInterruptEnable = CMP_SCR_IER_MASK,
  kCMP_OutputFallingInterruptEnable = CMP_SCR_IEF_MASK }
  Interrupt enable/disable mask.
- enum _cmp_status_flags {
  kCMP_OutputRisingEventFlag = CMP_SCR_CFR_MASK,
  kCMP_OutputFallingEventFlag = CMP_SCR_CFF_MASK,
  kCMP_OutputAssertEventFlag = CMP_SCR_COUT_MASK }
  Status flags' mask.
- enum cmp_hysteresis_mode_t {
  kCMP_HysteresisLevel0 = 0U,
  kCMP_HysteresisLevel1 = 1U,
  kCMP_HysteresisLevel2 = 2U,
  kCMP_HysteresisLevel3 = 3U }
  CMP Hysteresis mode.
- enum cmp_reference_voltage_source_t {
  kCMP_VrefSourceVin1 = 0U,
  kCMP_VrefSourceVin2 = 1U }
  CMP Voltage Reference source.

Driver version

- #define FSL_CMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  CMP driver version 2.0.0.

Initialization

- void CMP_Init (CMP_Type ∗base, const cmp_config_t ∗config)
  Initializes the CMP.
- void CMP_Deinit (CMP_Type ∗base)
  De-initializes the CMP module.
- static void CMP_Enable (CMP_Type ∗base, bool enable)
  Enables/disables the CMP module.
- void CMP_GetDefaultConfig (cmp_config_t ∗config)
  Initializes the CMP user configuration structure.
- void CMP_SetInputChannels (CMP_Type ∗base, uint8_t positiveChannel, uint8_t negativeChannel)
  Sets the input channels for the comparator.

Advanced Features

- void CMP_SetFilterConfig (CMP_Type ∗base, const cmp_filter_config_t ∗config)
  Configures the filter.
- void CMP_SetDACConfig (CMP_Type ∗base, const cmp_dac_config_t ∗config)
  Configures the internal DAC.
- void CMP_EnableInterrupts (CMP_Type ∗base, uint32_t mask)
  Enables the interrupts.
- void CMP_DisableInterrupts (CMP_Type ∗base, uint32_t mask)
  Disables the interrupts.

Results

- uint32_t CMP_GetStatusFlags (CMP_Type *base)
  Gets the status flags.
- void CMP_ClearStatusFlags (CMP_Type *base, uint32_t mask)
  Clears the status flags.

8.3 Data Structure Documentation

8.3.1 struct cmp_config_t

Data Fields

- bool enableCmp
  Enable the CMP module.
- cmp_hysteresis_mode_t hysteresisMode
  CMP Hysteresis mode.
- bool enableHighSpeed
  Enable High-speed (HS) comparison mode.
- bool enableInvertOutput
  Enable the inverted comparator output.
- bool useUnfilteredOutput
  Set the compare output(COUT) to equal COUTA(true) or COUT(false).
- bool enablePinOut
  The comparator output is available on the associated pin.

8.3.1.0.0.9 Field Documentation

8.3.1.0.0.9.1 bool cmp_config_t::enableCmp

8.3.1.0.0.9.2 cmp_hysteresis_mode_t cmp_config_t::hysteresisMode

8.3.1.0.0.9.3 bool cmp_config_t::enableHighSpeed

8.3.1.0.0.9.4 bool cmp_config_t::enableInvertOutput

8.3.1.0.0.9.5 bool cmp_config_t::useUnfilteredOutput

8.3.1.0.0.9.6 bool cmp_config_t::enablePinOut

8.3.2 struct cmp_filter_config_t

Data Fields

- uint8_t filterCount
  Filter Sample Count.
- uint8_t filterPeriod
  Filter Sample Period.

8.3.2.0.0.10 Field Documentation

8.3.2.0.0.10.1 uint8_t cmp_filter_config_t::filterCount

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

8.3.2.0.0.10.2 uint8_t cmp_filter_config_t::filterPeriod

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

8.3.3 struct cmp_dac_config_t

Data Fields

- cmp_reference_voltage_source_t referenceVoltageSource
  Supply voltage reference source.
- uint8_t DACValue
  Value for the DAC Output Voltage.

8.3.3.0.0.11 Field Documentation

8.3.3.0.0.11.1 cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource

8.3.3.0.0.11.2 uint8_t cmp_dac_config_t::DACValue

Available range is 0-63.

8.4 Macro Definition Documentation

8.4.1 #define FSL_CMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

8.5 Enumeration Type Documentation

8.5.1 enum _cmp_interrupt_enable

Enumerator

kCMP_OutputRisingInterruptEnable Comparator interrupt enable rising.
kCMP_OutputFallingInterruptEnable Comparator interrupt enable falling.

8.5.2 enum _cmp_status_flags

Enumerator

kCMP_OutputRisingEventFlag Rising-edge on the comparison output has occurred.
kCMP_OutputFallingEventFlag Falling-edge on the comparison output has occurred.

kCMP_OutputAssertEventFlag Return the current value of the analog comparator output.

8.5.3 enum cmp_hysteresis_mode_t

Enumerator

kCMP_HysteresisLevel0 Hysteresis level 0.
kCMP_HysteresisLevel1 Hysteresis level 1.
kCMP_HysteresisLevel2 Hysteresis level 2.
kCMP_HysteresisLevel3 Hysteresis level 3.

8.5.4 enum cmp_reference_voltage_source_t

Enumerator

kCMP_VrefSourceVin1 Vin1 is selected as a resistor ladder network supply reference Vin.
kCMP_VrefSourceVin2 Vin2 is selected as a resistor ladder network supply reference Vin.

8.6 Function Documentation

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

This function initializes the CMP module. The operations included are:

- Enabling the clock for CMP module.
- Configuring the comparator.
- Enabling the CMP module. Note: For some devices, multiple CMP instance share the same clock gate. In this case, to enable the clock for any instance enables all the CMPs. Check the chip reference manual for the clock assignment of the CMP.

Parameters

base	CMP peripheral base address.
config	Pointer to configuration structure.

8.6.2 void CMP_Deinit (CMP_Type ∗ *base*)

This function de-initializes the CMP module. The operations included are:

- Disabling the CMP module.
- Disabling the clock for CMP module.

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

Parameters

base	CMP peripheral base address.

8.6.3 static void CMP_Enable (CMP_Type ∗ *base,* bool *enable*) [inline], [static]

Parameters

base	CMP peripheral base address.
enable	Enables or disables the module.

8.6.4 void CMP_GetDefaultConfig (cmp_config_t ∗ *config*)

This function initializes the user configuration structure to these default values:

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

Parameters

config	Pointer to the configuration structure.

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

This function sets the input channels for the comparator. Note that two input channels cannot be set as same 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.
positive-Channel	Positive side input channel number. Available range is 0-7.
negative-Channel	Negative side input channel number. Available range is 0-7.

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

Parameters

base	CMP peripheral base address.
config	Pointer to the configuration structure.

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

Parameters

base	CMP peripheral base address.
config	Pointer to the configuration structure. "NULL" disables the feature.

8.6.8 void CMP_EnableInterrupts (CMP_Type ∗ *base*, uint32_t *mask*)

Parameters

base	CMP peripheral base address.
mask	Mask value for interrupts. See "_cmp_interrupt_enable".

8.6.9 void CMP_DisableInterrupts (CMP_Type ∗ *base*, uint32_t *mask*)

Parameters

base	CMP peripheral base address.
mask	Mask value for interrupts. See "_cmp_interrupt_enable".

8.6.10 uint32_t CMP_GetStatusFlags (CMP_Type * *base*)

Parameters

base	CMP peripheral base address.

Returns

Mask value for the asserted flags. See "_cmp_status_flags".

8.6.11 void CMP_ClearStatusFlags (CMP_Type * *base,* uint32_t *mask*)

Parameters

base	CMP peripheral base address.
mask	Mask value for the flags. See "_cmp_status_flags".

Chapter 9
CRC: Cyclic Redundancy Check Driver

9.1 Overview

The Kinetis SDK provides the Peripheral driver for the Cyclic Redundancy Check (CRC) module of Kinetis devices.

The cyclic redundancy check (CRC) module generates 16/32-bit CRC code for error detection. The CRC module provides a programmable polynomial, seed, and other parameters required to implement a 16-bit or 32-bit CRC standard.

9.2 CRC Driver Initialization and Configuration

CRC_Init() function enables the clock gate for the CRC module in the Kinetis SIM module and fully (re-)configures the CRC module according to configuration structure. The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting new checksum computation, the seed shall be set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed shall be set to the intermediate checksum value as obtained from previous calls to CRC_Get16bitResult() or CRC_Get32bitResult() function. After CRC_Init(), one or multiple CRC_WriteData() calls follow to update checksum with data, then CRC_Get16bitResult() or CRC_Get32bitResult() follows to read the result. The crcResult member of configuration structure determines if CRC_Get16bitResult() or CRC_Get32bitResult() return value is final checksum or intermediate checksum. CRC_Init() can be called as many times as required, thus, allows for runtime changes of CRC protocol.

CRC_GetDefaultConfig() function can be used to set the module configuration structure with parameters for CRC-16/CCIT-FALSE protocol.

9.3 CRC Write Data

The CRC_WriteData() function is used to add data to actual CRC. Internally it tries to use 32-bit reads and writes for all aligned data in the user buffer and it uses 8-bit reads and writes for all unaligned data in the user buffer. This function can update CRC with user supplied data chunks of arbitrary size, so one can update CRC byte by byte or with all bytes at once. Prior call CRC configuration function CRC_Init() fully specifies the CRC module configuration for CRC_WriteData() call.

9.4 CRC Get Checksum

The CRC_Get16bitResult() or CRC_Get32bitResult() function is used to read the CRC module data register. Depending on prior CRC module usage the return value is either intermediate checksum or final checksum. Example: for 16-bit CRCs the following call sequences can be used:

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() to get final checksum.

CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_Get16bitResult() to get final checksum.

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() to get intermediate checksum.

CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_Get16bitResult() to get intermediate checksum.

9.5 Comments about API usage in RTOS

If multiple RTOS tasks share the CRC module to compute checksums with different data and/or protocols, the following needs to be implemented by the user:

The triplets

CRC_Init() / CRC_WriteData() / CRC_Get16bitResult() or CRC_Get32bitResult()

shall be protected by RTOS mutex to protect CRC module against concurrent accesses from different tasks. Example:

```
CRC_Module_RTOS_Mutex_Lock;
CRC_Init();
CRC_WriteData();
CRC_Get16bitResult();
CRC_Module_RTOS_Mutex_Unlock;
```

9.6 Comments about API usage in interrupt handler

All APIs can be used from interrupt handler although execution time shall be considered (interrupt latency of equal and lower priority interrupts increases). Protection against concurrent accesses from different interrupt handlers and/or tasks shall be assured by the user.

9.7 CRC Driver Examples

9.7.1 Simple examples

Simple example with default CRC-16/CCIT-FALSE protocol

```
crc_config_t config;
CRC_Type *base;
uint8_t data[] = {0x00, 0x01, 0x02, 0x03, 0x04};
uint16_t checksum;

base = CRC0;
CRC_GetDefaultConfig(base, &config); /* default gives CRC-16/CCIT-FALSE */
CRC_Init(base, &config);
CRC_WriteData(base, data, sizeof(data));
checksum = CRC_Get16bitResult(base);
```

Simple example with CRC-32 protocol configuration

```
crc_config_t config;
uint32_t checksum;

config.polynomial = 0x04C11DB7u;
config.seed = 0xFFFFFFFFu;
config.crcBits = kCrcBits32;
config.reflectIn = true;
```

```
config.reflectOut = true;
config.complementChecksum = true;
config.crcResult = kCrcFinalChecksum;

CRC_Init(base, &config);
/* example: update by 1 byte at time */
while (dataSize)
{
    uint8_t c = GetCharacter();
    CRC_WriteData(base, &c, 1);
    dataSize--;
}
checksum = CRC_Get32bitResult(base);
```

9.7.2 Advanced examples

Per-partes data updates with context switch between. Assuming we have 3 tasks/threads, each using CRC module to compute checksums of different protocol, with context switches.

Firstly, we prepare 3 CRC module init functions for 3 different protocols: CRC-16 (ARC), CRC-16/-CCIT-FALSE and CRC-32. Table below lists the individual protocol specifications. See also: http://reveng.sourceforge.net/crc-catalogue/

	CRC-16/CCIT-FALSE	CRC-16	CRC-32
Width	16 bits	16 bits	32 bits
Polynomial	0x1021	0x8005	0x04C11DB7
Initial seed	0xFFFF	0x0000	0xFFFFFFFF
Complement checksum	No	No	Yes
Reflect In	No	Yes	Yes
Reflect Out	No	Yes	Yes

Corresponding init functions:

```
void InitCrc16_CCIT(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;

    config.polynomial = 0x1021;
    config.seed = seed;
    config.reflectIn = false;
    config.reflectOut = false;
    config.complementChecksum = false;
    config.crcBits = kCrcBits16;
    config.crcResult = isLast?kCrcFinalChecksum:
      kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}

void InitCrc16(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;
```

```
    config.polynomial = 0x8005;
    config.seed = seed;
    config.reflectIn = true;
    config.reflectOut = true;
    config.complementChecksum = false;
    config.crcBits = kCrcBits16;
    config.crcResult = isLast?kCrcFinalChecksum:
      kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}

void InitCrc32(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;

    config.polynomial = 0x04C11DB7U;
    config.seed = seed;
    config.reflectIn = true;
    config.reflectOut = true;
    config.complementChecksum = true;
    config.crcBits = kCrcBits32;
    config.crcResult = isLast?kCrcFinalChecksum:
      kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}
```

The following context switches show possible API usage:

```
uint16_t checksumCrc16;
uint32_t checksumCrc32;
uint16_t checksumCrc16Ccit;

checksumCrc16 = 0x0;
checksumCrc32 = 0xFFFFFFFFU;
checksumCrc16Ccit = 0xFFFFU;

/* Task A bytes[0-3] */
InitCrc16(base, checksumCrc16, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc16 = CRC_Get16bitResult(base);

/* Task B bytes[0-3] */
InitCrc16_CCIT(base, checksumCrc16Ccit, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc16Ccit = CRC_Get16bitResult(base);

/* Task C 4 bytes[0-3] */
InitCrc32(base, checksumCrc32, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task B add final 5 bytes[4-8] */
InitCrc16_CCIT(base, checksumCrc16Ccit, true);
CRC_WriteData(base, &data[4], 5);
checksumCrc16Ccit = CRC_Get16bitResult(base);

/* Task C 3 bytes[4-6] */
InitCrc32(base, checksumCrc32, false);
CRC_WriteData(base, &data[4], 3);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task A 3 bytes[4-6] */
InitCrc16(base, checksumCrc16, false);
```

```
CRC_WriteData(base, &data[4], 3);
checksumCrc16 = CRC_Get16bitResult(base);

/* Task C add final 2 bytes[7-8] */
InitCrc32(base, checksumCrc32, true);
CRC_WriteData(base, &data[7], 2);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task A add final 2 bytes[7-8] */
InitCrc16(base, checksumCrc16, true);
CRC_WriteData(base, &data[7], 2);
checksumCrc16 = CRC_Get16bitResult(base);
```

Files

- file fsl_crc.h

Data Structures

- struct crc_config_t
  CRC protocol configuration. More...

Macros

- #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1
  Default configuration structure filled by CRC_GetDefaultConfig().

Enumerations

- enum crc_bits_t {
  kCrcBits16 = 0U,
  kCrcBits32 = 1U }
  CRC bit width.
- enum crc_result_t {
  kCrcFinalChecksum = 0U,
  kCrcIntermediateChecksum = 1U }
  CRC result type.

Functions

- void CRC_Init (CRC_Type *base, const crc_config_t *config)
  Enables and configures the CRC peripheral module.
- static void CRC_Deinit (CRC_Type *base)
  Disables the CRC peripheral module.
- void CRC_GetDefaultConfig (crc_config_t *config)
  Loads default values to CRC protocol configuration structure.
- void CRC_WriteData (CRC_Type *base, const uint8_t *data, size_t dataSize)
  Writes data to the CRC module.
- uint32_t CRC_Get32bitResult (CRC_Type *base)
  Reads 32-bit checksum from the CRC module.
- uint16_t CRC_Get16bitResult (CRC_Type *base)
  Reads 16-bit checksum from the CRC module.

Driver version

- #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  CRC driver version.

9.8 Data Structure Documentation

9.8.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

- uint32_t polynomial
  CRC Polynomial, MSBit first.
- uint32_t seed
  Starting checksum value.
- bool reflectIn
  Reflect bits on input.
- bool reflectOut
  Reflect bits on output.
- bool complementChecksum
  True if the result shall be complement of the actual checksum.
- crc_bits_t crcBits
  Selects 16- or 32- bit CRC protocol.
- crc_result_t crcResult
  Selects final or intermediate checksum return from CRC_Get16bitResult() or CRC_Get32bitResult()

9.8.1.0.0.12 Field Documentation

9.8.1.0.0.12.1 uint32_t crc_config_t::polynomial

Example polynomial: 0x1021 = 1_0000_0010_0001 = x^12+x^5+1

9.8.1.0.0.12.2 bool crc_config_t::reflectIn

9.8.1.0.0.12.3 bool crc_config_t::reflectOut

9.8.1.0.0.12.4 bool crc_config_t::complementChecksum

9.8.1.0.0.12.5 crc_bits_t crc_config_t::crcBits

9.9 Macro Definition Documentation

9.9.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Version 2.0.1.

Current version: 2.0.1

Change log:

- Version 2.0.1
  - move DATA and DATALL macro definition from header file to source file

9.9.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as defeault.

9.10 Enumeration Type Documentation

9.10.1 enum crc_bits_t

Enumerator

kCrcBits16 Generate 16-bit CRC code.
kCrcBits32 Generate 32-bit CRC code.

9.10.2 enum crc_result_t

Enumerator

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

9.11 Function Documentation

9.11.1 void CRC_Init (CRC_Type ∗ *base*, const crc_config_t ∗ *config*)

This functions enables the clock gate in the Kinetis SIM module for the CRC peripheral. It also configures the CRC module and starts checksum computation by writing the seed.

Parameters

base	CRC peripheral address.
config	CRC module configuration structure

9.11.2 static void CRC_Deinit (CRC_Type ∗ *base*) `[inline]`, `[static]`

This functions disables the clock gate in the Kinetis SIM module for the CRC peripheral.

Parameters

base	CRC peripheral address.

9.11.3 void CRC_GetDefaultConfig (crc_config_t * *config*)

Loads default values to CRC protocol configuration structure. The default values are:

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

Parameters

config	CRC protocol configuration structure

9.11.4 void CRC_WriteData (CRC_Type * *base,* const uint8_t * *data,* size_t *dataSize*)

Writes input data buffer bytes to CRC data register. The configured type of transpose is applied.

Parameters

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

9.11.5 uint32_t CRC_Get32bitResult (CRC_Type * *base*)

Reads CRC data register (intermediate or final checksum). The configured type of transpose and complement are applied.

Parameters

base	CRC peripheral address.

Returns

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

9.11.6 uint16_t CRC_Get16bitResult (CRC_Type * *base*)

Reads CRC data register (intermediate or final checksum). The configured type of transpose and complement are applied.

Parameters

base	CRC peripheral address.

Returns

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

Chapter 10
DAC: Digital-to-Analog Converter Driver

10.1 Overview

The KSDK provides a peripheral driver for the Digital-to-Analog Converter (DAC) module of Kinetis devices.

The DAC driver includes a basic DAC module (converter) and a DAC buffer.

The basic DAC module supports operations unique to the DAC converter in each DAC instance. The APIs in this part are used in the initialization phase, which is necessary for enabling the DAC module in the application. The APIs enable/disable the clock, enable/disable the module, and configure the converter. Call the initial APIs to prepare the DAC module for the application.

The DAC buffer operates the DAC hardware buffer. The DAC module supports a hardware buffer to keep a group of DAC values to be converted. This feature supports updating the DAC output value automatically by triggering the buffer read pointer to move in the buffer. Use the APIs to configure the hardware buffer's trigger mode, watermark, work mode, and use size. Additionally, the APIs operate the DMA, interrupts, flags, the pointer (the index of the buffer), item values, and so on.

Note that the most functional features are designed for the DAC hardware buffer.

10.2 Typical use case

10.2.1 Working as a basic DAC without the hardware buffer feature.

```
// ...

// Configures the DAC.
DAC_GetDefaultConfig(&dacConfigStruct);
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);
DAC_Enable(DEMO_DAC_INSTANCE, true);
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U);

// ...

DAC_SetBufferValue(DEMO_DAC_INSTANCE, 0U, dacValue);
```

10.2.2 Working with the hardware buffer.

```
// ...

EnableIRQ(DEMO_DAC_IRQ_ID);

// ...

// Configures the DAC.
DAC_GetDefaultConfig(&dacConfigStruct);
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);
DAC_Enable(DEMO_DAC_INSTANCE, true);
```

```
    // Configures the DAC buffer.
    DAC_GetDefaultBufferConfig(&dacBufferConfigStruct);
    DAC_SetBufferConfig(DEMO_DAC_INSTANCE, &dacBufferConfigStruct);
    DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U); // Make sure the read pointer
       to the start.
    for (index = 0U, dacValue = 0; index < DEMO_DAC_USED_BUFFER_SIZE; index++, dacValue += (0xFFFU /
      DEMO_DAC_USED_BUFFER_SIZE))
    {
        DAC_SetBufferValue(DEMO_DAC_INSTANCE, index, dacValue);
    }
    // Clears flags.
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferWatermarkInterruptFlag = false;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferReadPointerTopPositionInterruptFlag = false;
    g_DacBufferReadPointerBottomPositionInterruptFlag = false;

    // Enables interrupts.
    mask = 0U;
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    mask |= kDAC_BufferWatermarkInterruptEnable;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    mask |= kDAC_BufferReadPointerTopInterruptEnable |
      kDAC_BufferReadPointerBottomInterruptEnable;
    DAC_EnableBuffer(DEMO_DAC_INSTANCE, true);
    DAC_EnableBufferInterrupts(DEMO_DAC_INSTANCE, mask);

// ISR for the DAC interrupt.
void DEMO_DAC_IRQ_HANDLER_FUNC(void)
{
    uint32_t flags = DAC_GetBufferStatusFlags(DEMO_DAC_INSTANCE);

#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    if (kDAC_BufferWatermarkFlag == (kDAC_BufferWatermarkFlag & flags))
    {
        g_DacBufferWatermarkInterruptFlag = true;
    }
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    if (kDAC_BufferReadPointerTopPositionFlag == (
      kDAC_BufferReadPointerTopPositionFlag & flags))
    {
        g_DacBufferReadPointerTopPositionInterruptFlag = true;
    }
    if (kDAC_BufferReadPointerBottomPositionFlag == (
      kDAC_BufferReadPointerBottomPositionFlag & flags))
    {
        g_DacBufferReadPointerBottomPositionInterruptFlag = true;
    }
    DAC_ClearBufferStatusFlags(DEMO_DAC_INSTANCE, flags); /* Clear flags. */
}
```

Files

- file fsl_dac.h

Data Structures

- struct dac_config_t
  DAC module configuration. More...
- struct dac_buffer_config_t
  DAC buffer configuration. More...

Enumerations

- enum _dac_buffer_status_flags {
kDAC_BufferReadPointerTopPositionFlag = DAC_SR_DACBFRPTF_MASK,
kDAC_BufferReadPointerBottomPositionFlag = DAC_SR_DACBFRPBF_MASK }
  DAC buffer flags.
- enum _dac_buffer_interrupt_enable {
kDAC_BufferReadPointerTopInterruptEnable = DAC_C0_DACBTIEN_MASK,
kDAC_BufferReadPointerBottomInterruptEnable = DAC_C0_DACBBIEN_MASK }
  DAC buffer interrupts.
- enum dac_reference_voltage_source_t {
kDAC_ReferenceVoltageSourceVref1 = 0U,
kDAC_ReferenceVoltageSourceVref2 = 1U }
  DAC reference voltage source.
- enum dac_buffer_trigger_mode_t {
kDAC_BufferTriggerByHardwareMode = 0U,
kDAC_BufferTriggerBySoftwareMode = 1U }
  DAC buffer trigger mode.
- enum dac_buffer_work_mode_t {
kDAC_BufferWorkAsNormalMode = 0U,
kDAC_BufferWorkAsOneTimeScanMode }
  DAC buffer work mode.

Driver version

- #define FSL_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  DAC driver version 2.0.1.

Initialization

- void DAC_Init (DAC_Type *base, const dac_config_t *config)
  Initializes the DAC module.
- void DAC_Deinit (DAC_Type *base)
  De-initializes the DAC module.
- void DAC_GetDefaultConfig (dac_config_t *config)
  Initializes the DAC user configuration structure.
- static void DAC_Enable (DAC_Type *base, bool enable)
  Enables the DAC module.

Buffer

- static void DAC_EnableBuffer (DAC_Type *base, bool enable)
  Enables the DAC buffer.
- void DAC_SetBufferConfig (DAC_Type *base, const dac_buffer_config_t *config)
  Configures the CMP buffer.
- void DAC_GetDefaultBufferConfig (dac_buffer_config_t *config)
  Initializes the DAC buffer configuration structure.
- static void DAC_EnableBufferDMA (DAC_Type *base, bool enable)
  Enables the DMA for DAC buffer.
- void DAC_SetBufferValue (DAC_Type *base, uint8_t index, uint16_t value)

Sets the value for items in the buffer.
- static void DAC_DoSoftwareTriggerBuffer (DAC_Type ∗base)
  Triggers the buffer by software and updates the read pointer of the DAC buffer.
- static uint8_t DAC_GetBufferReadPointer (DAC_Type ∗base)
  Gets the current read pointer of the DAC buffer.
- void DAC_SetBufferReadPointer (DAC_Type ∗base, uint8_t index)
  Sets the current read pointer of the DAC buffer.
- void DAC_EnableBufferInterrupts (DAC_Type ∗base, uint32_t mask)
  Enables interrupts for the DAC buffer.
- void DAC_DisableBufferInterrupts (DAC_Type ∗base, uint32_t mask)
  Disables interrupts for the DAC buffer.
- uint32_t DAC_GetBufferStatusFlags (DAC_Type ∗base)
  Gets the flags of events for the DAC buffer.
- void DAC_ClearBufferStatusFlags (DAC_Type ∗base, uint32_t mask)
  Clears the flags of events for the DAC buffer.

10.3 Data Structure Documentation

10.3.1 struct dac_config_t

Data Fields

- dac_reference_voltage_source_t referenceVoltageSource
  Select the DAC reference voltage source.
- bool enableLowPowerMode
  Enable the low-power mode.

10.3.1.0.0.13 Field Documentation

10.3.1.0.0.13.1 dac_reference_voltage_source_t dac_config_t::referenceVoltageSource

10.3.1.0.0.13.2 bool dac_config_t::enableLowPowerMode

10.3.2 struct dac_buffer_config_t

Data Fields

- dac_buffer_trigger_mode_t triggerMode
  Select the buffer's trigger mode.
- dac_buffer_work_mode_t workMode
  Select the buffer's work mode.
- uint8_t upperLimit
  Set the upper limit for the buffer index.

10.3.2.0.0.14 Field Documentation

10.3.2.0.0.14.1 dac_buffer_trigger_mode_t dac_buffer_config_t::triggerMode

10.3.2.0.0.14.2 dac_buffer_work_mode_t dac_buffer_config_t::workMode

10.3.2.0.0.14.3 uint8_t dac_buffer_config_t::upperLimit

Normally, 0-15 is available for a buffer with 16 items.

10.4 Macro Definition Documentation

10.4.1 #define FSL_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

10.5 Enumeration Type Documentation

10.5.1 enum _dac_buffer_status_flags

Enumerator

kDAC_BufferReadPointerTopPositionFlag DAC Buffer Read Pointer Top Position Flag.
kDAC_BufferReadPointerBottomPositionFlag DAC Buffer Read Pointer Bottom Position Flag.

10.5.2 enum _dac_buffer_interrupt_enable

Enumerator

kDAC_BufferReadPointerTopInterruptEnable DAC Buffer Read Pointer Top Flag Interrupt Enable.
kDAC_BufferReadPointerBottomInterruptEnable DAC Buffer Read Pointer Bottom Flag Interrupt Enable.

10.5.3 enum dac_reference_voltage_source_t

Enumerator

kDAC_ReferenceVoltageSourceVref1 The DAC selects DACREF_1 as the reference voltage.
kDAC_ReferenceVoltageSourceVref2 The DAC selects DACREF_2 as the reference voltage.

10.5.4 enum dac_buffer_trigger_mode_t

Enumerator

kDAC_BufferTriggerByHardwareMode The DAC hardware trigger is selected.

kDAC_BufferTriggerBySoftwareMode The DAC software trigger is selected.

10.5.5 enum dac_buffer_work_mode_t

Enumerator

kDAC_BufferWorkAsNormalMode Normal mode.
kDAC_BufferWorkAsOneTimeScanMode One-Time Scan mode.

10.6 Function Documentation

10.6.1 void DAC_Init (DAC_Type * *base*, const dac_config_t * *config*)

This function initializes the DAC module, including:

- Enabling the clock for DAC module.
- Configuring the DAC converter with a user configuration.
- Enabling the DAC module.

Parameters

base	DAC peripheral base address.
config	Pointer to the configuration structure. See "dac_config_t".

10.6.2 void DAC_Deinit (DAC_Type * *base*)

This function de-initializes the DAC module, including:

- Disabling the DAC module.
- Disabling the clock for the DAC module.

Parameters

base	DAC peripheral base address.

10.6.3 void DAC_GetDefaultConfig (dac_config_t * *config*)

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

```
*   config->referenceVoltageSource = kDAC_ReferenceVoltageSourceVref2;
*   config->enableLowPowerMode = false;
*
```

Parameters

config	Pointer to the configuration structure. See "dac_config_t".

10.6.4 static void DAC_Enable (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	DAC peripheral base address.
enable	Enables or disables the feature.

10.6.5 static void DAC_EnableBuffer (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	DAC peripheral base address.
enable	Enables or disables the feature.

10.6.6 void DAC_SetBufferConfig (DAC_Type * *base*, const dac_buffer_config_t * *config*)

Parameters

base	DAC peripheral base address.
config	Pointer to the configuration structure. See "dac_buffer_config_t".

10.6.7 void DAC_GetDefaultBufferConfig (dac_buffer_config_t * *config*)

This function initializes the DAC buffer configuration structure to a default value. The default values are:

```
*   config->triggerMode = kDAC_BufferTriggerBySoftwareMode;
*   config->watermark   = kDAC_BufferWatermark1Word;
*   config->workMode    = kDAC_BufferWorkAsNormalMode;
*   config->upperLimit  = DAC_DATL_COUNT - 1U;
*
```

Parameters

config	Pointer to the configuration structure. See "dac_buffer_config_t".

10.6.8 static void DAC_EnableBufferDMA (DAC_Type ∗ *base,* bool *enable*) [inline], [static]

Parameters

base	DAC peripheral base address.
enable	Enables or disables the feature.

10.6.9 void DAC_SetBufferValue (DAC_Type ∗ *base,* uint8_t *index,* uint16_t *value*)

Parameters

base	DAC peripheral base address.
index	Setting the index for items in the buffer. The available index should not exceed the size of the DAC buffer.
value	Setting the value for items in the buffer. 12-bits are available.

10.6.10 static void DAC_DoSoftwareTriggerBuffer (DAC_Type ∗ *base*) [inline], [static]

This function triggers the function by software. The read pointer of the DAC buffer is updated with one step after this function is called. Changing the read pointer depends on the buffer's work mode.

Parameters

base	DAC peripheral base address.

10.6.11 static uint8_t DAC_GetBufferReadPointer (DAC_Type ∗ *base*) [inline], [static]

This function gets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by software trigger or hardware trigger.

Parameters

base	DAC peripheral base address.

Returns

The current read pointer of the DAC buffer.

10.6.12 void DAC_SetBufferReadPointer (DAC_Type * *base,* uint8_t *index*)

This function sets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by software trigger or hardware trigger. After the read pointer changes, the DAC output value also changes.

Parameters

base	DAC peripheral base address.
index	Setting index value for the pointer.

10.6.13 void DAC_EnableBufferInterrupts (DAC_Type * *base,* uint32_t *mask*)

Parameters

base	DAC peripheral base address.
mask	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

10.6.14 void DAC_DisableBufferInterrupts (DAC_Type * *base,* uint32_t *mask*)

Parameters

base	DAC peripheral base address.
mask	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

10.6.15 uint32_t DAC_GetBufferStatusFlags (DAC_Type * *base*)

Parameters

base	DAC peripheral base address.

Returns

Mask value for the asserted flags. See "_dac_buffer_status_flags".

10.6.16 void DAC_ClearBufferStatusFlags (DAC_Type * *base,* uint32_t *mask*)

Parameters

base	DAC peripheral base address.
mask	Mask value for flags. See "_dac_buffer_status_flags_t".

Chapter 11
DMAMUX: Direct Memory Access Multiplexer Driver

11.1 Overview

The KSDK provides a peripheral driver for the Direct Memory Access Multiplexer (DMAMUX) of Kinetis devices.

11.2 Typical use case

11.2.1 DMAMUX Operation

```
DMAMUX_Init(DMAMUX0);
DMAMUX_SetSource(DMAMUX0, channel, source);
DMAMUX_EnableChannel(DMAMUX0, channel);
...
DMAMUX_DisableChannel(DMAMUX, channel);
DMAMUX_Deinit(DMAMUX0);
```

Files

- file fsl_dmamux.h

Driver version

- #define FSL_DMAMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
  DMAMUX driver version 2.0.2.

DMAMUX Initialization and de-initialization

- void DMAMUX_Init (DMAMUX_Type ∗base)
  Initializes the DMAMUX peripheral.
- void DMAMUX_Deinit (DMAMUX_Type ∗base)
  Deinitializes the DMAMUX peripheral.

DMAMUX Channel Operation

- static void DMAMUX_EnableChannel (DMAMUX_Type ∗base, uint32_t channel)
  Enables the DMAMUX channel.
- static void DMAMUX_DisableChannel (DMAMUX_Type ∗base, uint32_t channel)
  Disables the DMAMUX channel.
- static void DMAMUX_SetSource (DMAMUX_Type ∗base, uint32_t channel, uint32_t source)
  Configures the DMAMUX channel source.

11.3 Macro Definition Documentation

11.3.1 #define FSL_DMAMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

11.4 Function Documentation

11.4.1 void DMAMUX_Init (DMAMUX_Type * *base*)

This function ungates the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

11.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

11.4.3 static void DMAMUX_EnableChannel (DMAMUX_Type * *base,* uint32_t *channel*) [inline], [static]

This function enables the DMAMUX channel.

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.

11.4.4 static void DMAMUX_DisableChannel (DMAMUX_Type * *base,* uint32_t *channel*) [inline], [static]

This function disables the DMAMUX channel.

Note

The user must disable DMAMUX channel before configuring it.

Parameters

base	DMAMUX peripheral base address.

channel	DMAMUX channel number.

11.4.5 static void DMAMUX_SetSource (DMAMUX_Type * *base,* uint32_t *channel,* uint32_t *source*) [inline], [static]

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.
source	Channel source which is used to trigger DMA transfer.

Chapter 12
DSPI: Serial Peripheral Interface Driver

12.1 Overview

The KSDK provides a peripheral driver for the Serial Peripheral Interface (SPI) module of Kinetis devices.

Modules

- DSPI DMA Driver
- DSPI Driver
- DSPI FreeRTOS Driver
- DSPI eDMA Driver
- DSPI µCOS/II Driver
- DSPI µCOS/III Driver

12.2 DSPI Driver

12.2.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures the DSPI module and provides the functional and transactional interfaces to build the DSPI application.

12.2.2 Typical use case

12.2.2.1 Master Operation

```
dspi_master_handle_t g_m_handle; //global variable
dspi_master_config_t  masterConfig;
masterConfig.whichCtar                                  = kDSPI_Ctar0;
masterConfig.ctarConfig.baudRate                        = baudrate;
masterConfig.ctarConfig.bitsPerFrame                    = 8;
masterConfig.ctarConfig.cpol                            =
      kDSPI_ClockPolarityActiveHigh;
masterConfig.ctarConfig.cpha                            =
      kDSPI_ClockPhaseFirstEdge;
masterConfig.ctarConfig.direction                       =
      kDSPI_MsbFirst;
masterConfig.ctarConfig.pcsToSckDelayInNanoSec          = 1000000000 /
      baudrate ;
masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec      = 1000000000 /
      baudrate ;
masterConfig.ctarConfig.betweenTransferDelayInNanoSec = 1000000000 /
       baudrate ;
masterConfig.whichPcs                                   = kDSPI_Pcs0;
masterConfig.pcsActiveHighOrLow                         =
      kDSPI_PcsActiveLow;
masterConfig.enableContinuousSCK                        = false;
masterConfig.enableRxFifoOverWrite                      = false;
masterConfig.enableModifiedTimingFormat                 = false;
masterConfig.samplePoint                                =
      kDSPI_SckToSin0Clock;
DSPI_MasterInit(base, &masterConfig, srcClock_Hz);

//srcClock_Hz = CLOCK_GetFreq(xxx);
DSPI_MasterInit(base, &masterConfig, srcClock_Hz);

DSPI_MasterTransferCreateHandle(base, &g_m_handle, NULL, NULL);

masterXfer.txData      = masterSendBuffer;
masterXfer.rxData      = masterReceiveBuffer;
masterXfer.dataSize    = transfer_dataSize;
masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 ;
DSPI_MasterTransferBlocking(base, &g_m_handle, &masterXfer);
```

12.2.2.2 Slave Operation

```
dspi_slave_handle_t g_s_handle;//global variable
/*Slave config*/
slaveConfig.whichCtar                  = kDSPI_Ctar0;
slaveConfig.ctarConfig.bitsPerFrame    = 8;
slaveConfig.ctarConfig.cpol            = kDSPI_ClockPolarityActiveHigh;
slaveConfig.ctarConfig.cpha            = kDSPI_ClockPhaseFirstEdge;
```

```
slaveConfig.enableContinuousSCK        = false;
slaveConfig.enableRxFifoOverWrite      = false;
slaveConfig.enableModifiedTimingFormat = false;
slaveConfig.samplePoint                = kDSPI_SckToSin0Clock;
DSPI_SlaveInit(base, &slaveConfig);

slaveXfer.txData      = slaveSendBuffer0;
slaveXfer.rxData      = slaveReceiveBuffer0;
slaveXfer.dataSize    = transfer_dataSize;
slaveXfer.configFlags = kDSPI_SlaveCtar0;

bool isTransferCompleted = false;
DSPI_SlaveTransferCreateHandle(base, &g_s_handle, DSPI_SlaveUserCallback, &
      isTransferCompleted);

DSPI_SlaveTransferNonBlocking(&g_s_handle, &slaveXfer);


//void DSPI_SlaveUserCallback(SPI_Type *base, dspi_slave_handle_t *handle, status_t status, void
       *isTransferCompleted)
//{
//    if (status == kStatus_Success)
//    {
//        __NOP();
//    }
//    else if (status == kStatus_DSPI_Error)
//    {
//        __NOP();
//    }
//
//    *((bool *)isTransferCompleted) = true;
//
//    PRINTF("This is DSPI slave call back . \r\n");
//}
```

Files

- file fsl_dspi.h

Data Structures

- struct dspi_command_data_config_t
  DSPI master command date configuration used for the SPIx_PUSHR. More...
- struct dspi_master_ctar_config_t
  DSPI master ctar configuration structure. More...
- struct dspi_master_config_t
  DSPI master configuration structure. More...
- struct dspi_slave_ctar_config_t
  DSPI slave ctar configuration structure. More...
- struct dspi_slave_config_t
  DSPI slave configuration structure. More...
- struct dspi_transfer_t
  DSPI master/slave transfer structure. More...
- struct dspi_master_handle_t
  DSPI master transfer handle structure used for transactional API. More...
- struct dspi_slave_handle_t
  DSPI slave transfer handle structure used for the transactional API. More...

Macros

- #define DSPI_DUMMY_DATA (0x00U)
  DSPI dummy data if there is no Tx data.
- #define DSPI_MASTER_CTAR_SHIFT (0U)
  DSPI master CTAR shift macro; used internally.
- #define DSPI_MASTER_CTAR_MASK (0x0FU)
  DSPI master CTAR mask macro; used internally.
- #define DSPI_MASTER_PCS_SHIFT (4U)
  DSPI master PCS shift macro; used internally.
- #define DSPI_MASTER_PCS_MASK (0xF0U)
  DSPI master PCS mask macro; used internally.
- #define DSPI_SLAVE_CTAR_SHIFT (0U)
  DSPI slave CTAR shift macro; used internally.
- #define DSPI_SLAVE_CTAR_MASK (0x07U)
  DSPI slave CTAR mask macro; used internally.

Typedefs

- typedef void(∗ dspi_master_transfer_callback_t)(SPI_Type ∗base, dspi_master_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.
- typedef void(∗ dspi_slave_transfer_callback_t)(SPI_Type ∗base, dspi_slave_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.

Enumerations

- enum _dspi_status {
  kStatus_DSPI_Busy = MAKE_STATUS(kStatusGroup_DSPI, 0),
  kStatus_DSPI_Error = MAKE_STATUS(kStatusGroup_DSPI, 1),
  kStatus_DSPI_Idle = MAKE_STATUS(kStatusGroup_DSPI, 2),
  kStatus_DSPI_OutOfRange = MAKE_STATUS(kStatusGroup_DSPI, 3) }
  Status for the DSPI driver.
- enum _dspi_flags {
  kDSPI_TxCompleteFlag = SPI_SR_TCF_MASK,
  kDSPI_EndOfQueueFlag = SPI_SR_EOQF_MASK,
  kDSPI_TxFifoUnderflowFlag = SPI_SR_TFUF_MASK,
  kDSPI_TxFifoFillRequestFlag = SPI_SR_TFFF_MASK,
  kDSPI_RxFifoOverflowFlag = SPI_SR_RFOF_MASK,
  kDSPI_RxFifoDrainRequestFlag = SPI_SR_RFDF_MASK,
  kDSPI_TxAndRxStatusFlag = SPI_SR_TXRXS_MASK,
  kDSPI_AllStatusFlag }
  DSPI status flags in SPIx_SR register.
- enum _dspi_interrupt_enable {

kDSPI_TxCompleteInterruptEnable = SPI_RSER_TCF_RE_MASK,
kDSPI_EndOfQueueInterruptEnable = SPI_RSER_EOQF_RE_MASK,
kDSPI_TxFifoUnderflowInterruptEnable = SPI_RSER_TFUF_RE_MASK,
kDSPI_TxFifoFillRequestInterruptEnable = SPI_RSER_TFFF_RE_MASK,
kDSPI_RxFifoOverflowInterruptEnable = SPI_RSER_RFOF_RE_MASK,
kDSPI_RxFifoDrainRequestInterruptEnable = SPI_RSER_RFDF_RE_MASK,
kDSPI_AllInterruptEnable }
DSPI interrupt source.
- enum _dspi_dma_enable {
kDSPI_TxDmaEnable = (SPI_RSER_TFFF_RE_MASK | SPI_RSER_TFFF_DIRS_MASK),
kDSPI_RxDmaEnable = (SPI_RSER_RFDF_RE_MASK | SPI_RSER_RFDF_DIRS_MASK) }
DSPI DMA source.
- enum dspi_master_slave_mode_t {
kDSPI_Master = 1U,
kDSPI_Slave = 0U }
DSPI master or slave mode configuration.
- enum dspi_master_sample_point_t {
kDSPI_SckToSin0Clock = 0U,
kDSPI_SckToSin1Clock = 1U,
kDSPI_SckToSin2Clock = 2U }
DSPI Sample Point: Controls when the DSPI master samples SIN in the Modified Transfer Format.
- enum dspi_which_pcs_t {
kDSPI_Pcs0 = 1U << 0,
kDSPI_Pcs1 = 1U << 1,
kDSPI_Pcs2 = 1U << 2,
kDSPI_Pcs3 = 1U << 3,
kDSPI_Pcs4 = 1U << 4,
kDSPI_Pcs5 = 1U << 5 }
DSPI Peripheral Chip Select (Pcs) configuration (which Pcs to configure).
- enum dspi_pcs_polarity_config_t {
kDSPI_PcsActiveHigh = 0U,
kDSPI_PcsActiveLow = 1U }
DSPI Peripheral Chip Select (Pcs) Polarity configuration.
- enum _dspi_pcs_polarity {
kDSPI_Pcs0ActiveLow = 1U << 0,
kDSPI_Pcs1ActiveLow = 1U << 1,
kDSPI_Pcs2ActiveLow = 1U << 2,
kDSPI_Pcs3ActiveLow = 1U << 3,
kDSPI_Pcs4ActiveLow = 1U << 4,
kDSPI_Pcs5ActiveLow = 1U << 5,
kDSPI_PcsAllActiveLow = 0xFFU }
DSPI Peripheral Chip Select (Pcs) Polarity.
- enum dspi_clock_polarity_t {
kDSPI_ClockPolarityActiveHigh = 0U,
kDSPI_ClockPolarityActiveLow = 1U }
DSPI clock polarity configuration for a given CTAR.
- enum dspi_clock_phase_t {

kDSPI_ClockPhaseFirstEdge = 0U,
kDSPI_ClockPhaseSecondEdge = 1U }
DSPI clock phase configuration for a given CTAR.
- enum dspi_shift_direction_t {
kDSPI_MsbFirst = 0U,
kDSPI_LsbFirst = 1U }
DSPI data shifter direction options for a given CTAR.
- enum dspi_delay_type_t {
kDSPI_PcsToSck = 1U,
kDSPI_LastSckToPcs,
kDSPI_BetweenTransfer }
DSPI delay type selection.
- enum dspi_ctar_selection_t {
kDSPI_Ctar0 = 0U,
kDSPI_Ctar1 = 1U,
kDSPI_Ctar2 = 2U,
kDSPI_Ctar3 = 3U,
kDSPI_Ctar4 = 4U,
kDSPI_Ctar5 = 5U,
kDSPI_Ctar6 = 6U,
kDSPI_Ctar7 = 7U }
DSPI Clock and Transfer Attributes Register (CTAR) selection.
- enum _dspi_transfer_config_flag_for_master {
kDSPI_MasterCtar0 = 0U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar1 = 1U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar2 = 2U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar3 = 3U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar4 = 4U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar5 = 5U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar6 = 6U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterCtar7 = 7U << DSPI_MASTER_CTAR_SHIFT,
kDSPI_MasterPcs0 = 0U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcs1 = 1U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcs2 = 2U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcs3 = 3U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcs4 = 4U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcs5 = 5U << DSPI_MASTER_PCS_SHIFT,
kDSPI_MasterPcsContinuous = 1U << 20,
kDSPI_MasterActiveAfterTransfer = 1U << 21 }
Use this enumeration for the DSPI master transfer configFlags.
- enum _dspi_transfer_config_flag_for_slave { kDSPI_SlaveCtar0 = 0U << DSPI_SLAVE_CTAR-_SHIFT }
Use this enumeration for the DSPI slave transfer configFlags.
- enum _dspi_transfer_state {
kDSPI_Idle = 0x0U,
kDSPI_Busy,

kDSPI_Error }
DSPI transfer state, which is used for DSPI transactional API state machine.

Driver version

- #define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
  DSPI driver version 2.1.2.

Initialization and deinitialization

- void DSPI_MasterInit (SPI_Type ∗base, const dspi_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes the DSPI master.
- void DSPI_MasterGetDefaultConfig (dspi_master_config_t ∗masterConfig)
  Sets the dspi_master_config_t structure to default values.
- void DSPI_SlaveInit (SPI_Type ∗base, const dspi_slave_config_t ∗slaveConfig)
  DSPI slave configuration.
- void DSPI_SlaveGetDefaultConfig (dspi_slave_config_t ∗slaveConfig)
  Sets the dspi_slave_config_t structure to default values.
- void DSPI_Deinit (SPI_Type ∗base)
  De-initializes the DSPI peripheral.
- static void DSPI_Enable (SPI_Type ∗base, bool enable)
  Enables the DSPI peripheral and sets the MCR MDIS to 0.

Status

- static uint32_t DSPI_GetStatusFlags (SPI_Type ∗base)
  Gets the DSPI status flag state.
- static void DSPI_ClearStatusFlags (SPI_Type ∗base, uint32_t statusFlags)
  Clears the DSPI status flag.

Interrupts

- void DSPI_EnableInterrupts (SPI_Type ∗base, uint32_t mask)
  Enables the DSPI interrupts.
- static void DSPI_DisableInterrupts (SPI_Type ∗base, uint32_t mask)
  Disables the DSPI interrupts.

DMA Control

- static void DSPI_EnableDMA (SPI_Type ∗base, uint32_t mask)
  Enables the DSPI DMA request.
- static void DSPI_DisableDMA (SPI_Type ∗base, uint32_t mask)
  Disables the DSPI DMA request.

- static uint32_t DSPI_MasterGetTxRegisterAddress (SPI_Type *base)
  Gets the DSPI master PUSHR data register address for the DMA operation.
- static uint32_t DSPI_SlaveGetTxRegisterAddress (SPI_Type *base)
  Gets the DSPI slave PUSHR data register address for the DMA operation.
- static uint32_t DSPI_GetRxRegisterAddress (SPI_Type *base)
  Gets the DSPI POPR data register address for the DMA operation.

Bus Operations

- static void DSPI_SetMasterSlaveMode (SPI_Type *base, dspi_master_slave_mode_t mode)
  Configures the DSPI for master or slave.
- static bool DSPI_IsMaster (SPI_Type *base)
  Returns whether the DSPI module is in master mode.
- static void DSPI_StartTransfer (SPI_Type *base)
  Starts the DSPI transfers and clears HALT bit in MCR.
- static void DSPI_StopTransfer (SPI_Type *base)
  Stops (halts) DSPI transfers and sets the HALT bit in MCR.
- static void DSPI_SetFifoEnable (SPI_Type *base, bool enableTxFifo, bool enableRxFifo)
  Enables (or disables) the DSPI FIFOs.
- static void DSPI_FlushFifo (SPI_Type *base, bool flushTxFifo, bool flushRxFifo)
  Flushes the DSPI FIFOs.
- static void DSPI_SetAllPcsPolarity (SPI_Type *base, uint32_t mask)
  Configures the DSPI peripheral chip select polarity simultaneously.
- uint32_t DSPI_MasterSetBaudRate (SPI_Type *base, dspi_ctar_selection_t whichCtar, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the DSPI baud rate in bits per second.
- void DSPI_MasterSetDelayScaler (SPI_Type *base, dspi_ctar_selection_t whichCtar, uint32_t prescaler, uint32_t scaler, dspi_delay_type_t whichDelay)
  Manually configures the delay prescaler and scaler for a particular CTAR.
- uint32_t DSPI_MasterSetDelayTimes (SPI_Type *base, dspi_ctar_selection_t whichCtar, dspi_delay_type_t whichDelay, uint32_t srcClock_Hz, uint32_t delayTimeInNanoSec)
  Calculates the delay prescaler and scaler based on the desired delay input in nanoseconds.
- static void DSPI_MasterWriteData (SPI_Type *base, dspi_command_data_config_t *command, uint16_t data)
  Writes data into the data buffer for master mode.
- void DSPI_GetDefaultDataCommandConfig (dspi_command_data_config_t *command)
  Sets the dspi_command_data_config_t structure to default values.
- void DSPI_MasterWriteDataBlocking (SPI_Type *base, dspi_command_data_config_t *command, uint16_t data)
  Writes data into the data buffer master mode and waits till complete to return.
- static uint32_t DSPI_MasterGetFormattedCommand (dspi_command_data_config_t *command)
  Returns the DSPI command word formatted to the PUSHR data register bit field.
- void DSPI_MasterWriteCommandDataBlocking (SPI_Type *base, uint32_t data)
  Writes a 32-bit data word (16-bit command appended with 16-bit data) into the data buffer master mode and waits till complete to return.
- static void DSPI_SlaveWriteData (SPI_Type *base, uint32_t data)
  Writes data into the data buffer in slave mode.
- void DSPI_SlaveWriteDataBlocking (SPI_Type *base, uint32_t data)
  Writes data into the data buffer in slave mode, waits till data was transmitted, and returns.

- static uint32_t DSPI_ReadData (SPI_Type *base)
  Reads data from the data buffer.

Transactional

- void DSPI_MasterTransferCreateHandle (SPI_Type *base, dspi_master_handle_t *handle, dspi_master_transfer_callback_t callback, void *userData)
  Initializes the DSPI master handle.
- status_t DSPI_MasterTransferBlocking (SPI_Type *base, dspi_transfer_t *transfer)
  DSPI master transfer data using polling.
- status_t DSPI_MasterTransferNonBlocking (SPI_Type *base, dspi_master_handle_t *handle, dspi_transfer_t *transfer)
  DSPI master transfer data using interrupts.
- status_t DSPI_MasterTransferGetCount (SPI_Type *base, dspi_master_handle_t *handle, size_t *count)
  Gets the master transfer count.
- void DSPI_MasterTransferAbort (SPI_Type *base, dspi_master_handle_t *handle)
  DSPI master aborts a transfer using an interrupt.
- void DSPI_MasterTransferHandleIRQ (SPI_Type *base, dspi_master_handle_t *handle)
  DSPI Master IRQ handler function.
- void DSPI_SlaveTransferCreateHandle (SPI_Type *base, dspi_slave_handle_t *handle, dspi_slave_transfer_callback_t callback, void *userData)
  Initializes the DSPI slave handle.
- status_t DSPI_SlaveTransferNonBlocking (SPI_Type *base, dspi_slave_handle_t *handle, dspi_transfer_t *transfer)
  DSPI slave transfers data using an interrupt.
- status_t DSPI_SlaveTransferGetCount (SPI_Type *base, dspi_slave_handle_t *handle, size_t *count)
  Gets the slave transfer count.
- void DSPI_SlaveTransferAbort (SPI_Type *base, dspi_slave_handle_t *handle)
  DSPI slave aborts a transfer using an interrupt.
- void DSPI_SlaveTransferHandleIRQ (SPI_Type *base, dspi_slave_handle_t *handle)
  DSPI Master IRQ handler function.

12.2.3 Data Structure Documentation

12.2.3.1 struct dspi_command_data_config_t

Data Fields

- bool isPcsContinuous
  Option to enable the continuous assertion of the chip select between transfers.
- dspi_ctar_selection_t whichCtar
  The desired Clock and Transfer Attributes Register (CTAR) to use for CTAS.
- dspi_which_pcs_t whichPcs
  The desired PCS signal to use for the data transfer.
- bool isEndOfQueue

Signals that the current transfer is the last in the queue.
- bool clearTransferCount
  Clears the SPI Transfer Counter (SPI_TCNT) before transmission starts.

12.2.3.1.0.15 Field Documentation

12.2.3.1.0.15.1 bool dspi_command_data_config_t::isPcsContinuous

12.2.3.1.0.15.2 dspi_ctar_selection_t dspi_command_data_config_t::whichCtar

12.2.3.1.0.15.3 dspi_which_pcs_t dspi_command_data_config_t::whichPcs

12.2.3.1.0.15.4 bool dspi_command_data_config_t::isEndOfQueue

12.2.3.1.0.15.5 bool dspi_command_data_config_t::clearTransferCount

12.2.3.2 struct dspi_master_ctar_config_t

Data Fields

- uint32_t baudRate
  Baud Rate for DSPI.
- uint32_t bitsPerFrame
  Bits per frame, minimum 4, maximum 16.
- dspi_clock_polarity_t cpol
  Clock polarity.
- dspi_clock_phase_t cpha
  Clock phase.
- dspi_shift_direction_t direction
  MSB or LSB data shift direction.
- uint32_t pcsToSckDelayInNanoSec
  PCS to SCK delay time in nanoseconds; setting to 0 sets the minimum delay.
- uint32_t lastSckToPcsDelayInNanoSec
  The last SCK to PCS delay time in nanoseconds; setting to 0 sets the minimum delay.
- uint32_t betweenTransferDelayInNanoSec
  `After the SCK delay time in nanoseconds; setting to 0 sets the minimum` *delay.*

12.2.3.2.0.16 Field Documentation

12.2.3.2.0.16.1 uint32_t dspi_master_ctar_config_t::baudRate

12.2.3.2.0.16.2 uint32_t dspi_master_ctar_config_t::bitsPerFrame

12.2.3.2.0.16.3 dspi_clock_polarity_t dspi_master_ctar_config_t::cpol

12.2.3.2.0.16.4 dspi_clock_phase_t dspi_master_ctar_config_t::cpha

12.2.3.2.0.16.5 dspi_shift_direction_t dspi_master_ctar_config_t::direction

12.2.3.2.0.16.6 uint32_t dspi_master_ctar_config_t::pcsToSckDelayInNanoSec

It also sets the boundary value if out of range.

12.2.3.2.0.16.7 uint32_t dspi_master_ctar_config_t::lastSckToPcsDelayInNanoSec

It also sets the boundary value if out of range.

12.2.3.2.0.16.8 uint32_t dspi_master_ctar_config_t::betweenTransferDelayInNanoSec

It also sets the boundary value if out of range.

12.2.3.3 struct dspi_master_config_t

Data Fields

- dspi_ctar_selection_t whichCtar
  The desired CTAR to use.
- dspi_master_ctar_config_t ctarConfig
  Set the ctarConfig to the desired CTAR.
- dspi_which_pcs_t whichPcs
  The desired Peripheral Chip Select (pcs).
- dspi_pcs_polarity_config_t pcsActiveHighOrLow
  The desired PCS active high or low.
- bool enableContinuousSCK
  CONT_SCKE, continuous SCK enable.
- bool enableRxFifoOverWrite
  ROOE, receive FIFO overflow overwrite enable.
- bool enableModifiedTimingFormat
  Enables a modified transfer format to be used if true.
- dspi_master_sample_point_t samplePoint
  Controls when the module master samples SIN in the Modified Transfer Format.

12.2.3.3.0.17 Field Documentation

12.2.3.3.0.17.1 dspi_ctar_selection_t dspi_master_config_t::whichCtar

12.2.3.3.0.17.2 dspi_master_ctar_config_t dspi_master_config_t::ctarConfig

12.2.3.3.0.17.3 dspi_which_pcs_t dspi_master_config_t::whichPcs

12.2.3.3.0.17.4 dspi_pcs_polarity_config_t dspi_master_config_t::pcsActiveHighOrLow

12.2.3.3.0.17.5 bool dspi_master_config_t::enableContinuousSCK

Note that the continuous SCK is only supported for CPHA = 1.

12.2.3.3.0.17.6 bool dspi_master_config_t::enableRxFifoOverWrite

If ROOE = 0, the incoming data is ignored and the data from the transfer that generated the overflow is also ignored. If ROOE = 1, the incoming data is shifted to the shift register.

12.2.3.3.0.17.7 bool dspi_master_config_t::enableModifiedTimingFormat

12.2.3.3.0.17.8 dspi_master_sample_point_t dspi_master_config_t::samplePoint

It's valid only when CPHA=0.

12.2.3.4 struct dspi_slave_ctar_config_t

Data Fields

- uint32_t bitsPerFrame
  Bits per frame, minimum 4, maximum 16.
- dspi_clock_polarity_t cpol
  Clock polarity.
- dspi_clock_phase_t cpha
  Clock phase.

12.2.3.4.0.18 Field Documentation

12.2.3.4.0.18.1 uint32_t dspi_slave_ctar_config_t::bitsPerFrame

12.2.3.4.0.18.2 dspi_clock_polarity_t dspi_slave_ctar_config_t::cpol

12.2.3.4.0.18.3 dspi_clock_phase_t dspi_slave_ctar_config_t::cpha

Slave only supports MSB and does not support LSB.

12.2.3.5 struct dspi_slave_config_t

Data Fields

- dspi_ctar_selection_t whichCtar
  The desired CTAR to use.
- dspi_slave_ctar_config_t ctarConfig
  Set the ctarConfig to the desired CTAR.
- bool enableContinuousSCK
  CONT_SCKE, continuous SCK enable.
- bool enableRxFifoOverWrite
  ROOE, receive FIFO overflow overwrite enable.
- bool enableModifiedTimingFormat
  Enables a modified transfer format to be used if true.
- dspi_master_sample_point_t samplePoint
  Controls when the module master samples SIN in the Modified Transfer Format.

12.2.3.5.0.19 Field Documentation

12.2.3.5.0.19.1 dspi_ctar_selection_t dspi_slave_config_t::whichCtar

12.2.3.5.0.19.2 dspi_slave_ctar_config_t dspi_slave_config_t::ctarConfig

12.2.3.5.0.19.3 bool dspi_slave_config_t::enableContinuousSCK

Note that the continuous SCK is only supported for CPHA = 1.

12.2.3.5.0.19.4 bool dspi_slave_config_t::enableRxFifoOverWrite

If ROOE = 0, the incoming data is ignored and the data from the transfer that generated the overflow is also ignored. If ROOE = 1, the incoming data is shifted to the shift register.

12.2.3.5.0.19.5 bool dspi_slave_config_t::enableModifiedTimingFormat

12.2.3.5.0.19.6 dspi_master_sample_point_t dspi_slave_config_t::samplePoint

It's valid only when CPHA=0.

12.2.3.6 struct dspi_transfer_t

Data Fields

- uint8_t ∗ txData
  Send buffer.
- uint8_t ∗ rxData
  Receive buffer.
- volatile size_t dataSize
  Transfer bytes.
- uint32_t configFlags
  Transfer transfer configuration flags; set from _dspi_transfer_config_flag_for_master if the transfer is

used for master or _dspi_transfer_config_flag_for_slave enumeration if the transfer is used for slave.

12.2.3.6.0.20 Field Documentation

12.2.3.6.0.20.1 uint8_t* dspi_transfer_t::txData

12.2.3.6.0.20.2 uint8_t* dspi_transfer_t::rxData

12.2.3.6.0.20.3 volatile size_t dspi_transfer_t::dataSize

12.2.3.6.0.20.4 uint32_t dspi_transfer_t::configFlags

12.2.3.7 struct _dspi_master_handle

Forward declaration of the _dspi_master_handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  The desired number of bits per frame.
- volatile uint32_t command
  The desired data command.
- volatile uint32_t lastCommand
  The desired last data command.
- uint8_t fifoSize
  FIFO dataSize.
- volatile bool isPcsActiveAfterTransfer
  Indicates whether the PCS signal is active after the last frame transfer.
- volatile bool isThereExtraByte
  Indicates whether there are extra bytes.
- uint8_t *volatile txData
  Send buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- volatile uint8_t state
  DSPI transfer state, see _dspi_transfer_state.
- dspi_master_transfer_callback_t callback
  Completion callback.
- void * userData
  Callback user data.

12.2.3.7.0.21 Field Documentation

12.2.3.7.0.21.1 uint32_t dspi_master_handle_t::bitsPerFrame

12.2.3.7.0.21.2 volatile uint32_t dspi_master_handle_t::command

12.2.3.7.0.21.3 volatile uint32_t dspi_master_handle_t::lastCommand

12.2.3.7.0.21.4 uint8_t dspi_master_handle_t::fifoSize

12.2.3.7.0.21.5 volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer

12.2.3.7.0.21.6 volatile bool dspi_master_handle_t::isThereExtraByte

12.2.3.7.0.21.7 uint8_t∗ volatile dspi_master_handle_t::txData

12.2.3.7.0.21.8 uint8_t∗ volatile dspi_master_handle_t::rxData

12.2.3.7.0.21.9 volatile size_t dspi_master_handle_t::remainingSendByteCount

12.2.3.7.0.21.10 volatile size_t dspi_master_handle_t::remainingReceiveByteCount

12.2.3.7.0.21.11 volatile uint8_t dspi_master_handle_t::state

12.2.3.7.0.21.12 dspi_master_transfer_callback_t dspi_master_handle_t::callback

12.2.3.7.0.21.13 void∗ dspi_master_handle_t::userData

12.2.3.8 struct _dspi_slave_handle

Forward declaration of the _dspi_slave_handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  The desired number of bits per frame.
- volatile bool isThereExtraByte
  Indicates whether there are extra bytes.
- uint8_t ∗volatile txData
  Send buffer.
- uint8_t ∗volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- volatile uint8_t state
  DSPI transfer state.

- volatile uint32_t errorCount
  Error count for slave transfer.
- dspi_slave_transfer_callback_t callback
  Completion callback.
- void * userData
  Callback user data.

12.2.3.8.0.22 Field Documentation

12.2.3.8.0.22.1 uint32_t dspi_slave_handle_t::bitsPerFrame

12.2.3.8.0.22.2 volatile bool dspi_slave_handle_t::isThereExtraByte

12.2.3.8.0.22.3 uint8_t* volatile dspi_slave_handle_t::txData

12.2.3.8.0.22.4 uint8_t* volatile dspi_slave_handle_t::rxData

12.2.3.8.0.22.5 volatile size_t dspi_slave_handle_t::remainingSendByteCount

12.2.3.8.0.22.6 volatile size_t dspi_slave_handle_t::remainingReceiveByteCount

12.2.3.8.0.22.7 volatile uint8_t dspi_slave_handle_t::state

12.2.3.8.0.22.8 volatile uint32_t dspi_slave_handle_t::errorCount

12.2.3.8.0.22.9 dspi_slave_transfer_callback_t dspi_slave_handle_t::callback

12.2.3.8.0.22.10 void* dspi_slave_handle_t::userData

12.2.4 Macro Definition Documentation

12.2.4.1 #define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

12.2.4.2 #define DSPI_DUMMY_DATA (0x00U)

Dummy data used for Tx if there is no txData.

12.2.4.3 #define DSPI_MASTER_CTAR_SHIFT (0U)

12.2.4.4 #define DSPI_MASTER_CTAR_MASK (0x0FU)

12.2.4.5 #define DSPI_MASTER_PCS_SHIFT (4U)

12.2.4.6 #define DSPI_MASTER_PCS_MASK (0xF0U)

12.2.4.7 #define DSPI_SLAVE_CTAR_SHIFT (0U)

12.2.4.8 #define DSPI_SLAVE_CTAR_MASK (0x07U)

12.2.5 Typedef Documentation

12.2.5.1 typedef void(* dspi_master_transfer_callback_t)(SPI_Type *base, dspi_master_handle_t *handle, status_t status, void *userData)

Parameters

base	DSPI peripheral address.
handle	Pointer to the handle for the DSPI master.
status	Success or error code describing whether the transfer completed.
userData	Arbitrary pointer-dataSized value passed from the application.

12.2.5.2 typedef void(∗ dspi_slave_transfer_callback_t)(SPI_Type ∗base, dspi_slave_handle_t ∗handle, status_t status, void ∗userData)

Parameters

base	DSPI peripheral address.
handle	Pointer to the handle for the DSPI slave.
status	Success or error code describing whether the transfer completed.
userData	Arbitrary pointer-dataSized value passed from the application.

12.2.6 Enumeration Type Documentation

12.2.6.1 enum _dspi_status

Enumerator

kStatus_DSPI_Busy DSPI transfer is busy.
kStatus_DSPI_Error DSPI driver error.
kStatus_DSPI_Idle DSPI is idle.
kStatus_DSPI_OutOfRange DSPI transfer out of range.

12.2.6.2 enum _dspi_flags

Enumerator

kDSPI_TxCompleteFlag Transfer Complete Flag.
kDSPI_EndOfQueueFlag End of Queue Flag.
kDSPI_TxFifoUnderflowFlag Transmit FIFO Underflow Flag.
kDSPI_TxFifoFillRequestFlag Transmit FIFO Fill Flag.
kDSPI_RxFifoOverflowFlag Receive FIFO Overflow Flag.
kDSPI_RxFifoDrainRequestFlag Receive FIFO Drain Flag.
kDSPI_TxAndRxStatusFlag The module is in Stopped/Running state.
kDSPI_AllStatusFlag All statuses above.

12.2.6.3 enum _dspi_interrupt_enable

Enumerator

kDSPI_TxCompleteInterruptEnable TCF interrupt enable.
kDSPI_EndOfQueueInterruptEnable EOQF interrupt enable.
kDSPI_TxFifoUnderflowInterruptEnable TFUF interrupt enable.
kDSPI_TxFifoFillRequestInterruptEnable TFFF interrupt enable, DMA disable.
kDSPI_RxFifoOverflowInterruptEnable RFOF interrupt enable.
kDSPI_RxFifoDrainRequestInterruptEnable RFDF interrupt enable, DMA disable.
kDSPI_AllInterruptEnable All above interrupts enable.

12.2.6.4 enum _dspi_dma_enable

Enumerator

kDSPI_TxDmaEnable TFFF flag generates DMA requests. No Tx interrupt request.
kDSPI_RxDmaEnable RFDF flag generates DMA requests. No Rx interrupt request.

12.2.6.5 enum dspi_master_slave_mode_t

Enumerator

kDSPI_Master DSPI peripheral operates in master mode.
kDSPI_Slave DSPI peripheral operates in slave mode.

12.2.6.6 enum dspi_master_sample_point_t

This field is valid only when the CPHA bit in the CTAR register is 0.

Enumerator

kDSPI_SckToSin0Clock 0 system clocks between SCK edge and SIN sample.
kDSPI_SckToSin1Clock 1 system clock between SCK edge and SIN sample.
kDSPI_SckToSin2Clock 2 system clocks between SCK edge and SIN sample.

12.2.6.7 enum dspi_which_pcs_t

Enumerator

kDSPI_Pcs0 Pcs[0].
kDSPI_Pcs1 Pcs[1].
kDSPI_Pcs2 Pcs[2].

kDSPI_Pcs3 Pcs[3].
kDSPI_Pcs4 Pcs[4].
kDSPI_Pcs5 Pcs[5].

12.2.6.8 enum dspi_pcs_polarity_config_t

Enumerator

kDSPI_PcsActiveHigh Pcs Active High (idles low).
kDSPI_PcsActiveLow Pcs Active Low (idles high).

12.2.6.9 enum _dspi_pcs_polarity

Enumerator

kDSPI_Pcs0ActiveLow Pcs0 Active Low (idles high).
kDSPI_Pcs1ActiveLow Pcs1 Active Low (idles high).
kDSPI_Pcs2ActiveLow Pcs2 Active Low (idles high).
kDSPI_Pcs3ActiveLow Pcs3 Active Low (idles high).
kDSPI_Pcs4ActiveLow Pcs4 Active Low (idles high).
kDSPI_Pcs5ActiveLow Pcs5 Active Low (idles high).
kDSPI_PcsAllActiveLow Pcs0 to Pcs5 Active Low (idles high).

12.2.6.10 enum dspi_clock_polarity_t

Enumerator

kDSPI_ClockPolarityActiveHigh CPOL=0. Active-high DSPI clock (idles low).
kDSPI_ClockPolarityActiveLow CPOL=1. Active-low DSPI clock (idles high).

12.2.6.11 enum dspi_clock_phase_t

Enumerator

kDSPI_ClockPhaseFirstEdge CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.
kDSPI_ClockPhaseSecondEdge CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

12.2.6.12 enum dspi_shift_direction_t

Enumerator

kDSPI_MsbFirst Data transfers start with the most significant bit.
kDSPI_LsbFirst Data transfers start with the least significant bit.

12.2.6.13 enum dspi_delay_type_t

Enumerator

kDSPI_PcsToSck Pcs-to-SCK delay.
kDSPI_LastSckToPcs The last SCK edge to Pcs delay.
kDSPI_BetweenTransfer Delay between transfers.

12.2.6.14 enum dspi_ctar_selection_t

Enumerator

kDSPI_Ctar0 CTAR0 selection option for master or slave mode; note that CTAR0 and CTAR0_SLAVE are the same register address.
kDSPI_Ctar1 CTAR1 selection option for master mode only.
kDSPI_Ctar2 CTAR2 selection option for master mode only; note that some devices do not support CTAR2.
kDSPI_Ctar3 CTAR3 selection option for master mode only; note that some devices do not support CTAR3.
kDSPI_Ctar4 CTAR4 selection option for master mode only; note that some devices do not support CTAR4.
kDSPI_Ctar5 CTAR5 selection option for master mode only; note that some devices do not support CTAR5.
kDSPI_Ctar6 CTAR6 selection option for master mode only; note that some devices do not support CTAR6.
kDSPI_Ctar7 CTAR7 selection option for master mode only; note that some devices do not support CTAR7.

12.2.6.15 enum _dspi_transfer_config_flag_for_master

Enumerator

kDSPI_MasterCtar0 DSPI master transfer use CTAR0 setting.
kDSPI_MasterCtar1 DSPI master transfer use CTAR1 setting.
kDSPI_MasterCtar2 DSPI master transfer use CTAR2 setting.
kDSPI_MasterCtar3 DSPI master transfer use CTAR3 setting.

kDSPI_MasterCtar4 DSPI master transfer use CTAR4 setting.
kDSPI_MasterCtar5 DSPI master transfer use CTAR5 setting.
kDSPI_MasterCtar6 DSPI master transfer use CTAR6 setting.
kDSPI_MasterCtar7 DSPI master transfer use CTAR7 setting.
kDSPI_MasterPcs0 DSPI master transfer use PCS0 signal.
kDSPI_MasterPcs1 DSPI master transfer use PCS1 signal.
kDSPI_MasterPcs2 DSPI master transfer use PCS2 signal.
kDSPI_MasterPcs3 DSPI master transfer use PCS3 signal.
kDSPI_MasterPcs4 DSPI master transfer use PCS4 signal.
kDSPI_MasterPcs5 DSPI master transfer use PCS5 signal.
kDSPI_MasterPcsContinuous Indicates whether the PCS signal is continuous.
kDSPI_MasterActiveAfterTransfer Indicates whether the PCS signal is active after the last frame transfer.

12.2.6.16 enum _dspi_transfer_config_flag_for_slave

Enumerator

kDSPI_SlaveCtar0 DSPI slave transfer use CTAR0 setting. DSPI slave can only use PCS0.

12.2.6.17 enum _dspi_transfer_state

Enumerator

kDSPI_Idle Nothing in the transmitter/receiver.
kDSPI_Busy Transfer queue is not finished.
kDSPI_Error Transfer error.

12.2.7 Function Documentation

12.2.7.1 void DSPI_MasterInit (SPI_Type * *base,* const dspi_master_config_t * *masterConfig,* uint32_t *srcClock_Hz*)

This function initializes the DSPI master configuration. An example use case is as follows:

```
*   dspi_master_config_t  masterConfig;
*   masterConfig.whichCtar                                  = kDSPI_Ctar0;
*   masterConfig.ctarConfig.baudRate                        = 500000000;
*   masterConfig.ctarConfig.bitsPerFrame                    = 8;
*   masterConfig.ctarConfig.cpol                            =
      kDSPI_ClockPolarityActiveHigh;
*   masterConfig.ctarConfig.cpha                            =
      kDSPI_ClockPhaseFirstEdge;
*   masterConfig.ctarConfig.direction                       =
      kDSPI_MsbFirst;
*   masterConfig.ctarConfig.pcsToSckDelayInNanoSec          = 1000000000 /
```

```
      masterConfig.ctarConfig.baudRate ;
*   masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec     = 1000000000
      / masterConfig.ctarConfig.baudRate ;
*   masterConfig.ctarConfig.betweenTransferDelayInNanoSec =
      1000000000 / masterConfig.ctarConfig.baudRate ;
*   masterConfig.whichPcs                                  = kDSPI_Pcs0;
*   masterConfig.pcsActiveHighOrLow                        =
      kDSPI_PcsActiveLow;
*   masterConfig.enableContinuousSCK                       = false;
*   masterConfig.enableRxFifoOverWrite                     = false;
*   masterConfig.enableModifiedTimingFormat                = false;
*   masterConfig.samplePoint                               =
      kDSPI_SckToSin0Clock;
*   DSPI_MasterInit(base, &masterConfig, srcClock_Hz);
*
```

Parameters

base	DSPI peripheral address.
masterConfig	Pointer to the structure dspi_master_config_t.
srcClock_Hz	Module source input clock in Hertz.

12.2.7.2 void DSPI_MasterGetDefaultConfig (dspi_master_config_t * *masterConfig*)

The purpose of this API is to get the configuration structure initialized for the DSPI_MasterInit(). Users may use the initialized structure unchanged in the DSPI_MasterInit() or modify the structure before calling the DSPI_MasterInit(). Example:

```
*  dspi_master_config_t  masterConfig;
*  DSPI_MasterGetDefaultConfig(&masterConfig);
*
```

Parameters

masterConfig	pointer to dspi_master_config_t structure

12.2.7.3 void DSPI_SlaveInit (SPI_Type * *base*, const dspi_slave_config_t * *slaveConfig*)

This function initializes the DSPI slave configuration. An example use case is as follows:

```
*   dspi_slave_config_t  slaveConfig;
*  slaveConfig->whichCtar                  = kDSPI_Ctar0;
*  slaveConfig->ctarConfig.bitsPerFrame    = 8;
*  slaveConfig->ctarConfig.cpol            =
      kDSPI_ClockPolarityActiveHigh;
*  slaveConfig->ctarConfig.cpha            =
      kDSPI_ClockPhaseFirstEdge;
*  slaveConfig->enableContinuousSCK        = false;
*  slaveConfig->enableRxFifoOverWrite      = false;
*  slaveConfig->enableModifiedTimingFormat = false;
*  slaveConfig->samplePoint                = kDSPI_SckToSin0Clock;
*   DSPI_SlaveInit(base, &slaveConfig);
*
```

Parameters

base	DSPI peripheral address.
slaveConfig	Pointer to the structure dspi_master_config_t.

12.2.7.4 void DSPI_SlaveGetDefaultConfig (dspi_slave_config_t * *slaveConfig*)

The purpose of this API is to get the configuration structure initialized for the DSPI_SlaveInit(). Users may use the initialized structure unchanged in the DSPI_SlaveInit(), or modify the structure before calling the DSPI_SlaveInit(). Example:

```
*  dspi_slave_config_t  slaveConfig;
*  DSPI_SlaveGetDefaultConfig(&slaveConfig);
*
```

Parameters

slaveConfig	Pointer to the dspi_slave_config_t structure.

12.2.7.5 void DSPI_Deinit (SPI_Type * *base*)

Call this API to disable the DSPI clock.

Parameters

base	DSPI peripheral address.

12.2.7.6 static void DSPI_Enable (SPI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	DSPI peripheral address.
enable	Pass true to enable module, false to disable module.

12.2.7.7 static uint32_t DSPI_GetStatusFlags (SPI_Type * *base*) [inline], [static]

Parameters

base	DSPI peripheral address.

Returns

DSPI status (in SR register).

12.2.7.8 static void DSPI_ClearStatusFlags (SPI_Type * *base*, uint32_t *statusFlags*) [inline], [static]

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status bit to clear. The list of status bits is defined in the dspi_status_and_interrupt_request_t. The function uses these bit positions in its algorithm to clear the desired flag state. Example usage:

```
*  DSPI_ClearStatusFlags(base, kDSPI_TxCompleteFlag|
     kDSPI_EndOfQueueFlag);
*
```

Parameters

base	DSPI peripheral address.
statusFlags	The status flag used from the type dspi_flags.

< The status flags are cleared by writing 1 (w1c).

12.2.7.9 void DSPI_EnableInterrupts (SPI_Type * *base*, uint32_t *mask*)

This function configures the various interrupt masks of the DSPI. The parameters are base and an interrupt mask. Note, for Tx Fill and Rx FIFO drain requests, enable the interrupt request and disable the DMA request.

```
*  DSPI_EnableInterrupts(base,
     kDSPI_TxCompleteInterruptEnable |
     kDSPI_EndOfQueueInterruptEnable );
*
```

Parameters

base	DSPI peripheral address.
mask	The interrupt mask; use the enum _dspi_interrupt_enable.

12.2.7.10 static void DSPI_DisableInterrupts (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

```
*  DSPI_DisableInterrupts(base,
     kDSPI_TxCompleteInterruptEnable |
     kDSPI_EndOfQueueInterruptEnable );
*
```

Parameters

base	DSPI peripheral address.
mask	The interrupt mask; use the enum _dspi_interrupt_enable.

12.2.7.11 static void DSPI_EnableDMA (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

This function configures the Rx and Tx DMA mask of the DSPI. The parameters are a base and a DMA mask.

```
*  DSPI_EnableDMA(base, kDSPI_TxDmaEnable |
     kDSPI_RxDmaEnable);
*
```

Parameters

base	DSPI peripheral address.
mask	The interrupt mask; use the enum dspi_dma_enable.

12.2.7.12 static void DSPI_DisableDMA (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

This function configures the Rx and Tx DMA mask of the DSPI. The parameters are a base and a DMA mask.

```
*  SPI_DisableDMA(base, kDSPI_TxDmaEnable | kDSPI_RxDmaEnable);
*
```

Parameters

base	DSPI peripheral address.
mask	The interrupt mask; use the enum dspi_dma_enable.

12.2.7.13 static uint32_t DSPI_MasterGetTxRegisterAddress (SPI_Type * *base*) [inline], [static]

This function gets the DSPI master PUSHR data register address because this value is needed for the DMA operation.

Parameters

base	DSPI peripheral address.

Returns

The DSPI master PUSHR data register address.

12.2.7.14 static uint32_t DSPI_SlaveGetTxRegisterAddress (SPI_Type * *base*) [inline], [static]

This function gets the DSPI slave PUSHR data register address as this value is needed for the DMA operation.

Parameters

base	DSPI peripheral address.

Returns

The DSPI slave PUSHR data register address.

12.2.7.15 static uint32_t DSPI_GetRxRegisterAddress (SPI_Type * *base*) [inline], [static]

This function gets the DSPI POPR data register address as this value is needed for the DMA operation.

Parameters

base	DSPI peripheral address.

Returns

The DSPI POPR data register address.

12.2.7.16 static void DSPI_SetMasterSlaveMode (SPI_Type * *base*, dspi_master_slave_mode_t *mode*) [inline], [static]

Parameters

base	DSPI peripheral address.
mode	Mode setting (master or slave) of type dspi_master_slave_mode_t.

12.2.7.17 static bool DSPI_IsMaster (SPI_Type * *base*) [inline], [static]

Parameters

base	DSPI peripheral address.

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

12.2.7.18 static void DSPI_StartTransfer (SPI_Type * *base*) [inline], [static]

This function sets the module to begin data transfer in either master or slave mode.

Parameters

base	DSPI peripheral address.

12.2.7.19 static void DSPI_StopTransfer (SPI_Type * *base*) [inline], [static]

This function stops data transfers in either master or slave modes.

Parameters

base	DSPI peripheral address.

12.2.7.20 static void DSPI_SetFifoEnable (SPI_Type ∗ *base,* bool *enableTxFifo,* bool *enableRxFifo*) [inline], [static]

This function allows the caller to disable/enable the Tx and Rx FIFOs (independently). Note that to disable, pass in a logic 0 (false) for the particular FIFO configuration. To enable, pass in a logic 1 (true).

Parameters

base	DSPI peripheral address.
enableTxFifo	Disables (false) the TX FIFO; else enables (true) the TX FIFO
enableRxFifo	Disables (false) the RX FIFO; else enables (true) the RX FIFO

12.2.7.21 static void DSPI_FlushFifo (SPI_Type ∗ *base,* bool *flushTxFifo,* bool *flushRxFifo*) [inline], [static]

Parameters

base	DSPI peripheral address.
flushTxFifo	Flushes (true) the Tx FIFO; else does not flush (false) the Tx FIFO
flushRxFifo	Flushes (true) the Rx FIFO; else does not flush (false) the Rx FIFO

12.2.7.22 static void DSPI_SetAllPcsPolarity (SPI_Type ∗ *base,* uint32_t *mask*) [inline], [static]

For example, PCS0 and PCS1 are set to active low and other PCS is set to active high. Note that the number of PCSs is specific to the device.

```
*  DSPI_SetAllPcsPolarity(base, kDSPI_Pcs0ActiveLow |
     kDSPI_Pcs1ActiveLow);
```

Parameters

base	DSPI peripheral address.
mask	The PCS polarity mask; use the enum _dspi_pcs_polarity.

12.2.7.23 uint32_t DSPI_MasterSetBaudRate (SPI_Type * *base,* dspi_ctar_selection_t *whichCtar,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz*)

This function takes in the desired baudRate_Bps (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate, and returns the calculated baud rate in bits-per-second. It requires that the caller also provide the frequency of the module source clock (in Hertz).

Parameters

base	DSPI peripheral address.
whichCtar	The desired Clock and Transfer Attributes Register (CTAR) of the type dspi_ctar_-selection_t
baudRate_Bps	The desired baud rate in bits per second
srcClock_Hz	Module source input clock in Hertz

Returns

The actual calculated baud rate

12.2.7.24 void DSPI_MasterSetDelayScaler (SPI_Type * *base,* dspi_ctar_selection_t *whichCtar,* uint32_t *prescaler,* uint32_t *scaler,* dspi_delay_type_t *whichDelay*)

This function configures the PCS to SCK delay pre-scalar (PcsSCK) and scalar (CSSCK), after SCK delay pre-scalar (PASC) and scalar (ASC), and the delay after transfer pre-scalar (PDT) and scalar (DT).

These delay names are available in the type dspi_delay_type_t.

The user passes the delay to configure along with the prescaler and scaler value. This allows the user to directly set the prescaler/scaler values if pre-calculated or to manually increment either value.

Parameters

base	DSPI peripheral address.
whichCtar	The desired Clock and Transfer Attributes Register (CTAR) of type dspi_ctar_-selection_t.

prescaler	The prescaler delay value (can be an integer 0, 1, 2, or 3).
scaler	The scaler delay value (can be any integer between 0 to 15).
whichDelay	The desired delay to configure; must be of type dspi_delay_type_t

12.2.7.25 uint32_t DSPI_MasterSetDelayTimes (SPI_Type * *base,* dspi_ctar_selection_t *whichCtar,* dspi_delay_type_t *whichDelay,* uint32_t *srcClock_Hz,* uint32_t *delayTimeInNanoSec*)

This function calculates the values for: PCS to SCK delay pre-scalar (PCSSCK) and scalar (CSSCK), or After SCK delay pre-scalar (PASC) and scalar (ASC), or Delay after transfer pre-scalar (PDT) and scalar (DT).

These delay names are available in the type dspi_delay_type_t.

The user passes which delay to configure along with the desired delay value in nanoseconds. The function calculates the values needed for the prescaler and scaler and returning the actual calculated delay as an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. The higher-level peripheral driver alerts the user of an out of range delay input.

Parameters

base	DSPI peripheral address.
whichCtar	The desired Clock and Transfer Attributes Register (CTAR) of type dspi_ctar_-selection_t.
whichDelay	The desired delay to configure, must be of type dspi_delay_type_t
srcClock_Hz	Module source input clock in Hertz
delayTimeIn-NanoSec	The desired delay value in nanoseconds.

Returns

The actual calculated delay value.

12.2.7.26 static void DSPI_MasterWriteData (SPI_Type * *base,* dspi_-command_data_config_t * *command,* uint16_t *data*) [inline], [static]

In master mode, the 16-bit data is appended to the 16-bit command info. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). This is an example:

```
*  dspi_command_data_config_t commandConfig;
*  commandConfig.isPcsContinuous = true;
*  commandConfig.whichCtar = kDSPICtar0;
*  commandConfig.whichPcs = kDSPIPcs0;
*  commandConfig.clearTransferCount = false;
*  commandConfig.isEndOfQueue = false;
*  DSPI_MasterWriteData(base, &commandConfig, dataWord);
```

Parameters

base	DSPI peripheral address.
command	Pointer to the command structure.
data	The data word to be sent.

12.2.7.27 void DSPI_GetDefaultDataCommandConfig (dspi_command_data_config_t ∗ *command*)

The purpose of this API is to get the configuration structure initialized for use in the DSPI_MasterWrite-_xx(). Users may use the initialized structure unchanged in the DSPI_MasterWrite_xx() or modify the structure before calling the DSPI_MasterWrite_xx(). Example:

```
*  dspi_command_data_config_t  command;
*  DSPI_GetDefaultDataCommandConfig(&command);
*
```

Parameters

command	Pointer to the dspi_command_data_config_t structure.

12.2.7.28 void DSPI_MasterWriteDataBlocking (SPI_Type ∗ *base*, dspi_command_data_config_t ∗ *command*, uint16_t *data*)

In master mode, the 16-bit data is appended to the 16-bit command info. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). This is an example:

```
*  dspi_command_config_t commandConfig;
*  commandConfig.isPcsContinuous = true;
*  commandConfig.whichCtar = kDSPICtar0;
```

```
*   commandConfig.whichPcs = kDSPIPcs1;
*   commandConfig.clearTransferCount = false;
*   commandConfig.isEndOfQueue = false;
*   DSPI_MasterWriteDataBlocking(base, &commandConfig, dataWord);
*
```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

Parameters

base	DSPI peripheral address.
command	Pointer to the command structure.
data	The data word to be sent.

12.2.7.29 static uint32_t DSPI_MasterGetFormattedCommand (dspi_command_data_-config_t * *command*) [inline], [static]

This function allows the caller to pass in the data command structure and returns the command word formatted according to the DSPI PUSHR register bit field placement. The user can then "OR" the returned command word with the desired data to send and use the function DSPI_HAL_WriteCommandData-Mastermode or DSPI_HAL_WriteCommandDataMastermodeBlocking to write the entire 32-bit command data word to the PUSHR. This helps improve performance in cases where the command structure is constant. For example, the user calls this function before starting a transfer to generate the command word. When they are ready to transmit the data, they OR this formatted command word with the desired data to transmit. This process increases transmit performance when compared to calling send functions, such as DSPI_HAL_WriteDataMastermode, which format the command word each time a data word is to be sent.

Parameters

command	Pointer to command structure.

Returns

The command word formatted to the PUSHR data register bit field.

12.2.7.30 void DSPI_MasterWriteCommandDataBlocking (SPI_Type * *base*, uint32_t *data*)

In this function, the user must append the 16-bit data to the 16-bit command information and then provide the total 32-bit word as the data to send. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). The user is responsible for appending this command with the data to send. This is an example:

```
*  dataWord = <16-bit command> | <16-bit data>;
*  DSPI_HAL_WriteCommandDataMastermodeBlocking(base, dataWord);
*
```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

For a blocking polling transfer, see methods below. Option 1: uint32_t command_to_send = DSPI_-MasterGetFormattedCommand(&command); uint32_t data0 = command_to_send | data_need_to_send-_0; uint32_t data1 = command_to_send | data_need_to_send_1; uint32_t data2 = command_to_send | data_need_to_send_2;

DSPI_MasterWriteCommandDataBlocking(base,data0); DSPI_MasterWriteCommandDataBlocking(base,data1); DSPI_MasterWriteCommandDataBlocking(base,data2);

Option 2: DSPI_MasterWriteDataBlocking(base,&command,data_need_to_send_0); DSPI_Master-WriteDataBlocking(base,&command,data_need_to_send_1); DSPI_MasterWriteDataBlocking(base,&command,data _need_to_send_2);

Parameters

base	DSPI peripheral address.
data	The data word (command and data combined) to be sent.

12.2.7.31 static void DSPI_SlaveWriteData (SPI_Type * *base*, uint32_t *data*) [inline], [static]

In slave mode, up to 16-bit words may be written.

Parameters

base	DSPI peripheral address.
data	The data to send.

12.2.7.32 void DSPI_SlaveWriteDataBlocking (SPI_Type * *base*, uint32_t *data*)

In slave mode, up to 16-bit words may be written. The function first clears the transmit complete flag, writes data into data register, and finally waits until the data is transmitted.

Parameters

base	DSPI peripheral address.
data	The data to send.

12.2.7.33 static uint32_t DSPI_ReadData (SPI_Type * *base*) [inline], [static]

Parameters

base	DSPI peripheral address.

Returns

The data from the read data buffer.

12.2.7.34 void DSPI_MasterTransferCreateHandle (SPI_Type * *base*, dspi_master_-handle_t * *handle*, dspi_master_transfer_callback_t *callback*, void * *userData*)

This function initializes the DSPI handle, which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Parameters

base	DSPI peripheral base address.
handle	DSPI handle pointer to dspi_master_handle_t.
callback	DSPI callback.
userData	Callback function parameter.

12.2.7.35 status_t DSPI_MasterTransferBlocking (SPI_Type * *base*, dspi_transfer_t * *transfer*)

This function transfers data using polling. This is a blocking function, which does not return until all transfers have been completed.

Parameters

base	DSPI peripheral base address.
transfer	Pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.2.7.36 status_t DSPI_MasterTransferNonBlocking (SPI_Type * *base*, dspi_master_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function transfers data using interrupts. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_master_handle_t structure which stores the transfer state.
transfer	Pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.2.7.37 status_t DSPI_MasterTransferGetCount (SPI_Type * *base*, dspi_master_handle_t * *handle*, size_t * *count*)

This function gets the master transfer count.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_master_handle_t structure which stores the transfer state.
count	The number of bytes transferred by using the non-blocking transaction.

Returns

status of status_t.

12.2.7.38 void DSPI_MasterTransferAbort (SPI_Type ∗ *base*, dspi_master_handle_t ∗ *handle*)

This function aborts a transfer using an interrupt.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_master_handle_t structure which stores the transfer state.

12.2.7.39 void DSPI_MasterTransferHandleIRQ (SPI_Type ∗ *base*, dspi_master_handle_t ∗ *handle*)

This function processes the DSPI transmit and receive IRQ.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_master_handle_t structure which stores the transfer state.

12.2.7.40 void DSPI_SlaveTransferCreateHandle (SPI_Type ∗ *base*, dspi_slave_handle_t ∗ *handle*, dspi_slave_transfer_callback_t *callback*, void ∗ *userData*)

This function initializes the DSPI handle, which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Parameters

handle	DSPI handle pointer to the dspi_slave_handle_t.
base	DSPI peripheral base address.
callback	DSPI callback.
userData	Callback function parameter.

12.2.7.41 status_t DSPI_SlaveTransferNonBlocking (SPI_Type ∗ *base*, dspi_slave_handle_t ∗ *handle*, dspi_transfer_t ∗ *transfer*)

This function transfers data using an interrupt. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_slave_handle_t structure which stores the transfer state.
transfer	Pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.2.7.42 status_t DSPI_SlaveTransferGetCount (SPI_Type * *base*, dspi_slave_handle_t * *handle*, size_t * *count*)

This function gets the slave transfer count.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_master_handle_t structure which stores the transfer state.
count	The number of bytes transferred by using the non-blocking transaction.

Returns

status of status_t.

12.2.7.43 void DSPI_SlaveTransferAbort (SPI_Type * *base*, dspi_slave_handle_t * *handle*)

This function aborts a transfer using an interrupt.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_slave_handle_t structure which stores the transfer state.

12.2.7.44 void DSPI_SlaveTransferHandleIRQ (SPI_Type * *base*, dspi_slave_handle_t * *handle*)

This function processes the DSPI transmit and receive IRQ.

Parameters

base	DSPI peripheral base address.
handle	Pointer to the dspi_slave_handle_t structure which stores the transfer state.

12.3 DSPI DMA Driver

12.3.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures DSPI module and provides the functional and transactional interfaces to build the DSPI application.

Files

- file fsl_dspi_dma.h

Data Structures

- struct dspi_master_dma_handle_t
  DSPI master DMA transfer handle structure used for transactional API. More...
- struct dspi_slave_dma_handle_t
  DSPI slave DMA transfer handle structure used for transactional API. More...

Typedefs

- typedef void(∗ dspi_master_dma_transfer_callback_t)(SPI_Type ∗base, dspi_master_dma_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.
- typedef void(∗ dspi_slave_dma_transfer_callback_t)(SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.

Functions

- void DSPI_MasterTransferCreateHandleDMA (SPI_Type ∗base, dspi_master_dma_handle_t ∗handle, dspi_master_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗dmaRxRegToRxDataHandle, dma_handle_t ∗dmaTxDataToIntermediaryHandle, dma_handle_t ∗dmaIntermediaryToTxRegHandle)
  Initializes the DSPI master DMA handle.
- status_t DSPI_MasterTransferDMA (SPI_Type ∗base, dspi_master_dma_handle_t ∗handle, dspi_transfer_t ∗transfer)
  DSPI master transfers data using DMA.
- void DSPI_MasterTransferAbortDMA (SPI_Type ∗base, dspi_master_dma_handle_t ∗handle)
  DSPI master aborts a transfer which is using DMA.
- status_t DSPI_MasterTransferGetCountDMA (SPI_Type ∗base, dspi_master_dma_handle_t ∗handle, size_t ∗count)
  Gets the master DMA transfer remaining bytes.

- void DSPI_SlaveTransferCreateHandleDMA (SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle, dspi_slave_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗dmaRxRegToRxDataHandle, dma_handle_t ∗dmaTxDataToTxRegHandle)
  Initializes the DSPI slave DMA handle.
- status_t DSPI_SlaveTransferDMA (SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle, dspi_transfer_t ∗transfer)
  DSPI slave transfers data using DMA.
- void DSPI_SlaveTransferAbortDMA (SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle)
  DSPI slave aborts a transfer which is using DMA.
- status_t DSPI_SlaveTransferGetCountDMA (SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle, size_t ∗count)
  Gets the slave DMA transfer remaining bytes.

12.3.2 Data Structure Documentation

12.3.2.1 struct _dspi_master_dma_handle

Forward declaration of the DSPI DMA master handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  The desired number of bits per frame.
- volatile uint32_t command
  The desired data command.
- volatile uint32_t lastCommand
  The desired last data command.
- uint8_t fifoSize
  FIFO dataSize.
- volatile bool isPcsActiveAfterTransfer
  Indicates whether the PCS signal keeps active after the last frame transfer.
- volatile bool isThereExtraByte
  Indicates whether there is an extra byte.
- uint8_t ∗volatile txData
  Send buffer.
- uint8_t ∗volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- uint32_t rxBuffIfNull
  Used if there is not rxData for DMA purpose.
- uint32_t txBuffIfNull
  Used if there is not txData for DMA purpose.
- volatile uint8_t state

DSPI transfer state, see _dspi_transfer_state.
- dspi_master_dma_transfer_callback_t callback
  Completion callback.
- void ∗ userData
  Callback user data.
- dma_handle_t ∗ dmaRxRegToRxDataHandle
  dma_handle_t handle point used for RxReg to RxData buff
- dma_handle_t ∗ dmaTxDataToIntermediaryHandle
  dma_handle_t handle point used for TxData to Intermediary
- dma_handle_t ∗ dmaIntermediaryToTxRegHandle
  dma_handle_t handle point used for Intermediary to TxReg

12.3.2.1.0.23 Field Documentation

12.3.2.1.0.23.1 uint32_t dspi_master_dma_handle_t::bitsPerFrame

12.3.2.1.0.23.2 volatile uint32_t dspi_master_dma_handle_t::command

12.3.2.1.0.23.3 volatile uint32_t dspi_master_dma_handle_t::lastCommand

12.3.2.1.0.23.4 uint8_t dspi_master_dma_handle_t::fifoSize

12.3.2.1.0.23.5 volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer

12.3.2.1.0.23.6 volatile bool dspi_master_dma_handle_t::isThereExtraByte

12.3.2.1.0.23.7 uint8_t∗ volatile dspi_master_dma_handle_t::txData

12.3.2.1.0.23.8 uint8_t∗ volatile dspi_master_dma_handle_t::rxData

12.3.2.1.0.23.9 volatile size_t dspi_master_dma_handle_t::remainingSendByteCount

12.3.2.1.0.23.10 volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount

12.3.2.1.0.23.11 uint32_t dspi_master_dma_handle_t::rxBuffIfNull

12.3.2.1.0.23.12 uint32_t dspi_master_dma_handle_t::txBuffIfNull

12.3.2.1.0.23.13 volatile uint8_t dspi_master_dma_handle_t::state

12.3.2.1.0.23.14 dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback

12.3.2.1.0.23.15 void∗ dspi_master_dma_handle_t::userData

12.3.2.2 struct _dspi_slave_dma_handle

Forward declaration of the DSPI DMA slave handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  Desired number of bits per frame.
- volatile bool isThereExtraByte
  Indicates whether there is an extra byte.
- uint8_t ∗volatile txData
  A send buffer.
- uint8_t ∗volatile rxData
  A receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- uint32_t rxBuffIfNull
  Used if there is not rxData for DMA purpose.
- uint32_t txBuffIfNull
  Used if there is not txData for DMA purpose.
- uint32_t txLastData
  Used if there is an extra byte when 16 bits per frame for DMA purpose.
- volatile uint8_t state
  DSPI transfer state.
- uint32_t errorCount
  Error count for the slave transfer.
- dspi_slave_dma_transfer_callback_t callback
  Completion callback.
- void ∗ userData
  Callback user data.
- dma_handle_t ∗ dmaRxRegToRxDataHandle
  dma_handle_t handle point used for RxReg to RxData buff
- dma_handle_t ∗ dmaTxDataToTxRegHandle
  dma_handle_t handle point used for TxData to TxReg

12.3.2.2.0.24 Field Documentation

12.3.2.2.0.24.1 uint32_t dspi_slave_dma_handle_t::bitsPerFrame

12.3.2.2.0.24.2 volatile bool dspi_slave_dma_handle_t::isThereExtraByte

12.3.2.2.0.24.3 uint8_t* volatile dspi_slave_dma_handle_t::txData

12.3.2.2.0.24.4 uint8_t* volatile dspi_slave_dma_handle_t::rxData

12.3.2.2.0.24.5 volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount

12.3.2.2.0.24.6 volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount

12.3.2.2.0.24.7 uint32_t dspi_slave_dma_handle_t::rxBuffIfNull

12.3.2.2.0.24.8 uint32_t dspi_slave_dma_handle_t::txBuffIfNull

12.3.2.2.0.24.9 uint32_t dspi_slave_dma_handle_t::txLastData

12.3.2.2.0.24.10 volatile uint8_t dspi_slave_dma_handle_t::state

12.3.2.2.0.24.11 uint32_t dspi_slave_dma_handle_t::errorCount

12.3.2.2.0.24.12 dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback

12.3.2.2.0.24.13 void* dspi_slave_dma_handle_t::userData

12.3.3 Typedef Documentation

12.3.3.1 typedef void(* dspi_master_dma_transfer_callback_t)(SPI_Type *base, dspi_master_dma_handle_t *handle, status_t status, void *userData)

Parameters

base	DSPI peripheral base address.
handle	Pointer to the handle for the DSPI master.
status	Success or error code describing whether the transfer completed.
userData	Arbitrary pointer-dataSized value passed from the application.

12.3.3.2 typedef void(∗ dspi_slave_dma_transfer_callback_t)(SPI_Type ∗base, dspi_slave_dma_handle_t ∗handle, status_t status, void ∗userData)

Parameters

base	DSPI peripheral base address.
handle	Pointer to the handle for the DSPI slave.
status	Success or error code describing whether the transfer completed.
userData	Arbitrary pointer-dataSized value passed from the application.

12.3.4 Function Documentation

12.3.4.1 void DSPI_MasterTransferCreateHandleDMA (SPI_Type ∗ *base*, dspi_master_dma_handle_t ∗ *handle*, dspi_master_dma_transfer_callback_t *callback*, void ∗ *userData*, dma_handle_t ∗ *dmaRxRegToRxDataHandle*, dma_handle_t ∗ *dmaTxDataToIntermediaryHandle*, dma_handle_t ∗ *dmaIntermediaryToTxRegHandle*)

This function initializes the DSPI DMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for dmaRxRegToRxDataHandle and Tx DMAMUX source for dmaIntermediaryToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for dmaRxRegToRxDataHandle.

Parameters

base	DSPI peripheral base address.
handle	DSPI handle pointer to dspi_master_dma_handle_t.
callback	DSPI callback.
userData	A callback function parameter.
dmaRxRegToRxDataHandle	dmaRxRegToRxDataHandle pointer to dma_handle_t.
dmaTxDataToIntermediaryHandle	dmaTxDataToIntermediaryHandle pointer to dma_handle_t.
dmaIntermediaryToTxRegHandle	dmaIntermediaryToTxRegHandle pointer to dma_handle_t.

12.3.4.2 status_t DSPI_MasterTransferDMA (SPI_Type ∗ *base*, dspi_master_dma_handle_t ∗ *handle*, dspi_transfer_t ∗ *transfer*)

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the master DMA transfer does not support the transfer_size of 1 when the bitsPerFrame is greater than 8.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_dma_handle_t structure which stores the transfer state.
transfer	A pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.3.4.3 void DSPI_MasterTransferAbortDMA (SPI_Type ∗ *base*, dspi_master_dma_handle_t ∗ *handle*)

This function aborts a transfer which is using DMA.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_dma_handle_t structure which stores the transfer state.

12.3.4.4 status_t DSPI_MasterTransferGetCountDMA (SPI_Type * *base*, dspi_master_dma_handle_t * *handle*, size_t * *count*)

This function gets the master DMA transfer remaining bytes.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_dma_handle_t structure which stores the transfer state.
count	A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

12.3.4.5 void DSPI_SlaveTransferCreateHandleDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*, dspi_slave_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *dmaRxRegToRxDataHandle*, dma_handle_t * *dmaTxDataToTxRegHandle*)

This function initializes the DSPI DMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for dmaRxRegToRxDataHandle and Tx DMAMUX source for dmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for dmaRxRegToRxDataHandle.

Parameters

base	DSPI peripheral base address.

handle	DSPI handle pointer to dspi_slave_dma_handle_t.
callback	DSPI callback.
userData	A callback function parameter.
dmaRxRegToRxDataHandle	dmaRxRegToRxDataHandle pointer to dma_handle_t.
dmaTxDataToTxRegHandle	dmaTxDataToTxRegHandle pointer to dma_handle_t.

12.3.4.6 status_t DSPI_SlaveTransferDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the slave DMA transfer does not support the transfer_size of 1 when the bitsPerFrame is greater than 8.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state.
transfer	A pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.3.4.7 void DSPI_SlaveTransferAbortDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*)

This function aborts a transfer which is using DMA.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state.

12.3.4.8 status_t DSPI_SlaveTransferGetCountDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*, size_t * *count*)

This function gets the slave DMA transfer remaining bytes.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state.
count	A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

12.4 DSPI eDMA Driver

12.4.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures DSPI module and provides the functional and transactional interfaces to build the DSPI application.

Files

- file fsl_dspi_edma.h

Data Structures

- struct dspi_master_edma_handle_t
  DSPI master eDMA transfer handle structure used for the transactional API. More...
- struct dspi_slave_edma_handle_t
  DSPI slave eDMA transfer handle structure used for the transactional API. More...

Typedefs

- typedef void(∗ dspi_master_edma_transfer_callback_t)(SPI_Type ∗base, dspi_master_edma_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.
- typedef void(∗ dspi_slave_edma_transfer_callback_t)(SPI_Type ∗base, dspi_slave_edma_handle_t ∗handle, status_t status, void ∗userData)
  Completion callback function pointer type.

Functions

- void DSPI_MasterTransferCreateHandleEDMA (SPI_Type ∗base, dspi_master_edma_handle_t ∗handle, dspi_master_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗edmaRxRegToRxDataHandle, edma_handle_t ∗edmaTxDataToIntermediaryHandle, edma_handle_t ∗edmaIntermediaryToTxRegHandle)
  Initializes the DSPI master eDMA handle.
- status_t DSPI_MasterTransferEDMA (SPI_Type ∗base, dspi_master_edma_handle_t ∗handle, dspi_transfer_t ∗transfer)
  DSPI master transfer data using eDMA.
- void DSPI_MasterTransferAbortEDMA (SPI_Type ∗base, dspi_master_edma_handle_t ∗handle)
  DSPI master aborts a transfer which is using eDMA.
- status_t DSPI_MasterTransferGetCountEDMA (SPI_Type ∗base, dspi_master_edma_handle_t ∗handle, size_t ∗count)
  Gets the master eDMA transfer count.

- void DSPI_SlaveTransferCreateHandleEDMA (SPI_Type *base, dspi_slave_edma_handle_t *handle, dspi_slave_edma_transfer_callback_t callback, void *userData, edma_handle_t *edmaRxRegToRxDataHandle, edma_handle_t *edmaTxDataToTxRegHandle)
  Initializes the DSPI slave eDMA handle.
- status_t DSPI_SlaveTransferEDMA (SPI_Type *base, dspi_slave_edma_handle_t *handle, dspi_transfer_t *transfer)
  DSPI slave transfer data using eDMA.
- void DSPI_SlaveTransferAbortEDMA (SPI_Type *base, dspi_slave_edma_handle_t *handle)
  DSPI slave aborts a transfer which is using eDMA.
- status_t DSPI_SlaveTransferGetCountEDMA (SPI_Type *base, dspi_slave_edma_handle_t *handle, size_t *count)
  Gets the slave eDMA transfer count.

12.4.2 Data Structure Documentation

12.4.2.1 struct _dspi_master_edma_handle

Forward declaration of the DSPI eDMA master handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  The desired number of bits per frame.
- volatile uint32_t command
  The desired data command.
- volatile uint32_t lastCommand
  The desired last data command.
- uint8_t fifoSize
  FIFO dataSize.
- volatile bool isPcsActiveAfterTransfer
  Indicates whether the PCS signal keeps active after the last frame transfer.
- volatile bool isThereExtraByte
  Indicates whether there is an extra byte.
- uint8_t *volatile txData
  Send buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- uint32_t rxBuffIfNull
  Used if there is not rxData for DMA purpose.
- uint32_t txBuffIfNull
  Used if there is not txData for DMA purpose.
- volatile uint8_t state

DSPI transfer state, see _dspi_transfer_state.
- dspi_master_edma_transfer_callback_t callback
  Completion callback.
- void * userData
  Callback user data.
- edma_handle_t * edmaRxRegToRxDataHandle
  edma_handle_t handle point used for RxReg to RxData buff
- edma_handle_t * edmaTxDataToIntermediaryHandle
  edma_handle_t handle point used for TxData to Intermediary
- edma_handle_t * edmaIntermediaryToTxRegHandle
  edma_handle_t handle point used for Intermediary to TxReg
- edma_tcd_t dspiSoftwareTCD [2]
  SoftwareTCD , internal used.

12.4.2.1.0.25 Field Documentation

12.4.2.1.0.25.1 uint32_t dspi_master_edma_handle_t::bitsPerFrame

12.4.2.1.0.25.2 volatile uint32_t dspi_master_edma_handle_t::command

12.4.2.1.0.25.3 volatile uint32_t dspi_master_edma_handle_t::lastCommand

12.4.2.1.0.25.4 uint8_t dspi_master_edma_handle_t::fifoSize

12.4.2.1.0.25.5 volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer

12.4.2.1.0.25.6 volatile bool dspi_master_edma_handle_t::isThereExtraByte

12.4.2.1.0.25.7 uint8_t* volatile dspi_master_edma_handle_t::txData

12.4.2.1.0.25.8 uint8_t* volatile dspi_master_edma_handle_t::rxData

12.4.2.1.0.25.9 volatile size_t dspi_master_edma_handle_t::remainingSendByteCount

12.4.2.1.0.25.10 volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount

12.4.2.1.0.25.11 uint32_t dspi_master_edma_handle_t::rxBuffIfNull

12.4.2.1.0.25.12 uint32_t dspi_master_edma_handle_t::txBuffIfNull

12.4.2.1.0.25.13 volatile uint8_t dspi_master_edma_handle_t::state

12.4.2.1.0.25.14 dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback

12.4.2.1.0.25.15 void* dspi_master_edma_handle_t::userData

12.4.2.2 struct _dspi_slave_edma_handle

Forward declaration of the DSPI eDMA slave handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  The desired number of bits per frame.
- volatile bool isThereExtraByte
  Indicates whether there is an extra byte.
- uint8_t ∗volatile txData
  Send buffer.
- uint8_t ∗volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  A number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  A number of bytes remaining to receive.
- size_t totalByteCount
  A number of transfer bytes.
- uint32_t rxBuffIfNull
  Used if there is not rxData for DMA purpose.
- uint32_t txBuffIfNull
  Used if there is not txData for DMA purpose.
- uint32_t txLastData
  Used if there is an extra byte when 16bits per frame for DMA purpose.
- volatile uint8_t state
  DSPI transfer state.
- uint32_t errorCount
  Error count for slave transfer.
- dspi_slave_edma_transfer_callback_t callback
  Completion callback.
- void ∗ userData
  Callback user data.
- edma_handle_t ∗ edmaRxRegToRxDataHandle
  edma_handle_t handle point used for RxReg to RxData buff
- edma_handle_t ∗ edmaTxDataToTxRegHandle
  edma_handle_t handle point used for TxData to TxReg
- edma_tcd_t dspiSoftwareTCD [2]
  SoftwareTCD, used internally.

12.4.2.2.0.26 Field Documentation

12.4.2.2.0.26.1 uint32_t dspi_slave_edma_handle_t::bitsPerFrame

12.4.2.2.0.26.2 volatile bool dspi_slave_edma_handle_t::isThereExtraByte

12.4.2.2.0.26.3 uint8_t* volatile dspi_slave_edma_handle_t::txData

12.4.2.2.0.26.4 uint8_t* volatile dspi_slave_edma_handle_t::rxData

12.4.2.2.0.26.5 volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount

12.4.2.2.0.26.6 volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount

12.4.2.2.0.26.7 uint32_t dspi_slave_edma_handle_t::rxBuffIfNull

12.4.2.2.0.26.8 uint32_t dspi_slave_edma_handle_t::txBuffIfNull

12.4.2.2.0.26.9 uint32_t dspi_slave_edma_handle_t::txLastData

12.4.2.2.0.26.10 volatile uint8_t dspi_slave_edma_handle_t::state

12.4.2.2.0.26.11 uint32_t dspi_slave_edma_handle_t::errorCount

12.4.2.2.0.26.12 dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback

12.4.2.2.0.26.13 void* dspi_slave_edma_handle_t::userData

12.4.3 Typedef Documentation

12.4.3.1 typedef void(* dspi_master_edma_transfer_callback_t)(SPI_Type *base, dspi_master_edma_handle_t *handle, status_t status, void *userData)

Parameters

base	DSPI peripheral base address.
handle	A pointer to the handle for the DSPI master.
status	Success or error code describing whether the transfer completed.
userData	An arbitrary pointer-dataSized value passed from the application.

12.4.3.2 typedef void(∗ dspi_slave_edma_transfer_callback_t)(SPI_Type ∗base, dspi_slave_edma_handle_t ∗handle, status_t status, void ∗userData)

Parameters

base	DSPI peripheral base address.
handle	A pointer to the handle for the DSPI slave.
status	Success or error code describing whether the transfer completed.
userData	An arbitrary pointer-dataSized value passed from the application.

12.4.4 Function Documentation

12.4.4.1 void DSPI_MasterTransferCreateHandleEDMA (SPI_Type ∗ *base*, dspi_master_edma_handle_t ∗ *handle*, dspi_master_edma_transfer_callback_t *callback*, void ∗ *userData*, edma_handle_t ∗ *edmaRxRegToRxDataHandle*, edma_handle_t ∗ *edmaTxDataToIntermediaryHandle*, edma_handle_t ∗ *edmaIntermediaryToTxRegHandle*)

This function initializes the DSPI eDMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI eDMA has separated (RX and TX as two sources) or shared (RX and TX are the same source) DMA request source. (1) For the separated DMA request source, enable and set the RX DMAMUX source for edmaRxRegToRxDataHandle and TX DMAMUX source for edmaIntermediaryToTxRegHandle. (2) For the shared DMA request source, enable and set the RX/RX DMAMUX source for the edmaRxRegToRxDataHandle.

Parameters

base	DSPI peripheral base address.
handle	DSPI handle pointer to dspi_master_edma_handle_t.
callback	DSPI callback.
userData	A callback function parameter.
edmaRxRegToRxDataHandle	edmaRxRegToRxDataHandle pointer to edma_handle_t.
edmaTxDataToIntermediaryHandle	edmaTxDataToIntermediaryHandle pointer to edma_handle_t.
edmaIntermediaryToTxRegHandle	edmaIntermediaryToTxRegHandle pointer to edma_handle_t.

12.4.4.2 status_t DSPI_MasterTransferEDMA (SPI_Type ∗ *base*, dspi_master_edma_handle_t ∗ *handle*, dspi_transfer_t ∗ *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.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_edma_handle_t structure which stores the transfer state.
transfer	A pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.4.4.3 void DSPI_MasterTransferAbortEDMA (SPI_Type ∗ *base*, dspi_master_edma_handle_t ∗ *handle*)

This function aborts a transfer which is using eDMA.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_edma_handle_t structure which stores the transfer state.

12.4.4.4 status_t DSPI_MasterTransferGetCountEDMA (SPI_Type * *base*, dspi_master_edma_handle_t * *handle*, size_t * *count*)

This function gets the master eDMA transfer count.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_master_edma_handle_t structure which stores the transfer state.
count	A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

12.4.4.5 void DSPI_SlaveTransferCreateHandleEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*, dspi_slave_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *edmaRxRegToRxDataHandle*, edma_handle_t * *edmaTxDataToTxRegHandle*)

This function initializes the DSPI eDMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI eDMA has separated (RN and TX in 2 sources) or shared (RX and TX are the same source) DMA request source. (1)For the separated DMA request source, enable and set the RX DMAMUX source for edmaRxRegToRxDataHandle and TX DMAMUX source for edmaTxDataToTxRegHandle. (2)For the shared DMA request source, enable and set the RX/RX DMAMUX source for the edmaRxRegToRxDataHandle.

Parameters

base	DSPI peripheral base address.

handle	DSPI handle pointer to dspi_slave_edma_handle_t.
callback	DSPI callback.
userData	A callback function parameter.
edmaRxRegToRxDataHandle	edmaRxRegToRxDataHandle pointer to edma_handle_t.
edmaTxDataToTxRegHandle	edmaTxDataToTxRegHandle pointer to edma_handle_t.

12.4.4.6 status_t DSPI_SlaveTransferEDMA (SPI_Type ∗ *base*, dspi_slave_edma_handle_t ∗ *handle*, dspi_transfer_t ∗ *transfer*)

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data have been transferred, the callback function is called. Note that the slave eDMA transfer doesn't support transfer_size is 1 when the bitsPerFrame is greater than 8.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_edma_handle_t structure which stores the transfer state.
transfer	A pointer to the dspi_transfer_t structure.

Returns

status of status_t.

12.4.4.7 void DSPI_SlaveTransferAbortEDMA (SPI_Type ∗ *base*, dspi_slave_edma_handle_t ∗ *handle*)

This function aborts a transfer which is using eDMA.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_edma_handle_t structure which stores the transfer state.

12.4.4.8 status_t DSPI_SlaveTransferGetCountEDMA (SPI_Type ∗ *base*, dspi_slave_edma_handle_t ∗ *handle*, size_t ∗ *count*)

This function gets the slave eDMA transfer count.

Parameters

base	DSPI peripheral base address.
handle	A pointer to the dspi_slave_edma_handle_t structure which stores the transfer state.
count	A number of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

12.5 DSPI FreeRTOS Driver

12.5.1 Overview

Files

- file fsl_dspi_freertos.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI FreeRTOS handle. More...

DSPI RTOS Operation

- status_t DSPI_RTOS_Init (dspi_rtos_handle_t ∗handle, SPI_Type ∗base, const dspi_master_config-_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes the DSPI.
- status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t ∗handle)
  Deinitializes the DSPI.
- status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t ∗handle, dspi_transfer_t ∗transfer)
  Performs the SPI transfer.

12.5.2 Data Structure Documentation

12.5.2.1 struct dspi_rtos_handle_t

DSPI µC/OS-III handle.

DSPI µC/OS-II handle.

Data Fields

- SPI_Type ∗ base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.
- SemaphoreHandle_t event
  Semaphore to notify and unblock a task when a transfer ends.
- OS_EVENT ∗ mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP ∗ event
  Semaphore to notify and unblock a task when a transfer ends.

- OS_SEM mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP event
  Semaphore to notify and unblock a task when a transfer ends.

12.5.3 Function Documentation

12.5.3.1 status_t DSPI_RTOS_Init (dspi_rtos_handle_t * *handle*, SPI_Type * *base*, const dspi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the DSPI instance to initialize.
masterConfig	A configuration structure to set-up the DSPI in master mode.
srcClock_Hz	A frequency of the input clock of the DSPI module.

Returns

status of the operation.

12.5.3.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle.

12.5.3.3 status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function performs the SPI transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS DSPI handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

12.6 DSPI µCOS/II Driver

12.6.1 Overview

Files

- file fsl_dspi_ucosii.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI FreeRTOS handle. More...

DSPI RTOS Operation

- status_t DSPI_RTOS_Init (dspi_rtos_handle_t ∗handle, SPI_Type ∗base, const dspi_master_config-_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes the DSPI.
- status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t ∗handle)
  Deinitializes the DSPI.
- status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t ∗handle, dspi_transfer_t ∗transfer)
  Performs the SPI transfer.

12.6.2 Data Structure Documentation

12.6.2.1 struct dspi_rtos_handle_t

DSPI µC/OS-III handle.

DSPI µC/OS-II handle.

Data Fields

- SPI_Type ∗ base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.
- SemaphoreHandle_t event
  Semaphore to notify and unblock a task when a transfer ends.
- OS_EVENT ∗ mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP ∗ event
  Semaphore to notify and unblock a task when a transfer ends.

- OS_SEM mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP event
  Semaphore to notify and unblock a task when a transfer ends.

12.6.3 Function Documentation

12.6.3.1 status_t DSPI_RTOS_Init (dspi_rtos_handle_t * *handle*, SPI_Type * *base*, const dspi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the DSPI instance to initialize.
masterConfig	A configuration structure to set-up the DSPI in master mode.
srcClock_Hz	A frequency of the input clock of the DSPI module.

Returns

status of the operation.

12.6.3.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle.

12.6.3.3 status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function performs the SPI transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS DSPI handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

12.7 DSPI µCOS/III Driver

12.7.1 Overview

Files

- file fsl_dspi_ucosiii.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI FreeRTOS handle. More...

DSPI RTOS Operation

- status_t DSPI_RTOS_Init (dspi_rtos_handle_t ∗handle, SPI_Type ∗base, const dspi_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes the DSPI.
- status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t ∗handle)
  Deinitializes the DSPI.
- status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t ∗handle, dspi_transfer_t ∗transfer)
  Performs the SPI transfer.

12.7.2 Data Structure Documentation

12.7.2.1 struct dspi_rtos_handle_t

DSPI µC/OS-III handle.

DSPI µC/OS-II handle.

Data Fields

- SPI_Type ∗ base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.
- SemaphoreHandle_t event
  Semaphore to notify and unblock a task when a transfer ends.
- OS_EVENT ∗ mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP ∗ event
  Semaphore to notify and unblock a task when a transfer ends.

- OS_SEM mutex
  Mutex to lock the handle during a transfer.
- OS_FLAG_GRP event
  Semaphore to notify and unblock a task when a transfer ends.

12.7.3 Function Documentation

12.7.3.1 status_t DSPI_RTOS_Init (dspi_rtos_handle_t * *handle*, SPI_Type * *base*, const dspi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the DSPI instance to initialize.
masterConfig	A configuration structure to set-up the DSPI in master mode.
srcClock_Hz	A frequency of the input clock of the DSPI module.

Returns

status of the operation.

12.7.3.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and the related RTOS context.

Parameters

handle	The RTOS DSPI handle.

12.7.3.3 status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function performs the SPI transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS DSPI handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 13
eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

13.1 Overview

The KSDK provides a peripheral driver for the enhanced Direct Memory Access (eDMA) of Kinetis devices.

13.2 Typical use case

13.2.1 eDMA Operation

```
edma_transfer_config_t transferConfig;
edma_config_t userConfig;
uint32_t transferDone = false;

EDMA_GetDefaultConfig(&userConfig);
EDMA_Init(DMA0, &userConfig);
EDMA_CreateHandle(&g_EDMA_Handle, DMA0, channel);
EDMA_SetCallback(&g_EDMA_Handle, EDMA_Callback, &transferDone);
EDMA_PrepareTransfer(&transferConfig, srcAddr, srcWidth, destAddr, destWidth,
                     bytesEachRequest, transferBytes, kEDMA_MemoryToMemory);
EDMA_SubmitTransfer(&g_EDMA_Handle, &transferConfig, true);
EDMA_StartTransfer(&g_EDMA_Handle);
/* Waits for the eDMA transfer to finish */
while (transferDone != true);
```

Files

- file fsl_edma.h

Data Structures

- struct edma_config_t
  eDMA global configuration structure. More...
- struct edma_transfer_config_t
  eDMA transfer configuration More...
- struct edma_channel_Preemption_config_t
  eDMA channel priority configuration More...
- struct edma_minor_offset_config_t
  eDMA minor offset configuration More...
- struct edma_tcd_t
  eDMA TCD. More...
- struct edma_handle_t
  eDMA transfer handle structure More...

Macros

- #define DMA_DCHPRI_INDEX(channel) (((channel) & ~0x03U) | (3 - ((channel)&0x03U)))

Compute the offset unit from DCHPRI3.
- #define DMA_DCHPRIn(base, channel) ((volatile uint8_t ∗)&(base->DCHPRI3))[DMA_DCHPRI_INDEX(channel)]
  Get the pointer of DCHPRIn.

Typedefs

- typedef void(∗ edma_callback)(struct _edma_handle ∗handle, void ∗userData, bool transferDone, uint32_t tcds)
  Define callback function for eDMA.

Enumerations

- enum edma_transfer_size_t {
  kEDMA_TransferSize1Bytes = 0x0U,
  kEDMA_TransferSize2Bytes = 0x1U,
  kEDMA_TransferSize4Bytes = 0x2U,
  kEDMA_TransferSize16Bytes = 0x4U,
  kEDMA_TransferSize32Bytes = 0x5U }
  eDMA transfer configuration
- enum edma_modulo_t {

kEDMA_ModuloDisable = 0x0U,
kEDMA_Modulo2bytes,
kEDMA_Modulo4bytes,
kEDMA_Modulo8bytes,
kEDMA_Modulo16bytes,
kEDMA_Modulo32bytes,
kEDMA_Modulo64bytes,
kEDMA_Modulo128bytes,
kEDMA_Modulo256bytes,
kEDMA_Modulo512bytes,
kEDMA_Modulo1Kbytes,
kEDMA_Modulo2Kbytes,
kEDMA_Modulo4Kbytes,
kEDMA_Modulo8Kbytes,
kEDMA_Modulo16Kbytes,
kEDMA_Modulo32Kbytes,
kEDMA_Modulo64Kbytes,
kEDMA_Modulo128Kbytes,
kEDMA_Modulo256Kbytes,
kEDMA_Modulo512Kbytes,
kEDMA_Modulo1Mbytes,
kEDMA_Modulo2Mbytes,
kEDMA_Modulo4Mbytes,
kEDMA_Modulo8Mbytes,
kEDMA_Modulo16Mbytes,
kEDMA_Modulo32Mbytes,
kEDMA_Modulo64Mbytes,
kEDMA_Modulo128Mbytes,
kEDMA_Modulo256Mbytes,
kEDMA_Modulo512Mbytes,
kEDMA_Modulo1Gbytes,
kEDMA_Modulo2Gbytes }

eDMA modulo configuration

- enum edma_bandwidth_t {
  kEDMA_BandwidthStallNone = 0x0U,
  kEDMA_BandwidthStall4Cycle = 0x2U,
  kEDMA_BandwidthStall8Cycle = 0x3U }

  Bandwidth control.
- enum edma_channel_link_type_t {
  kEDMA_LinkNone = 0x0U,
  kEDMA_MinorLink,
  kEDMA_MajorLink }

  Channel link type.
- enum _edma_channel_status_flags {

kEDMA_DoneFlag = 0x1U,
kEDMA_ErrorFlag = 0x2U,
kEDMA_InterruptFlag = 0x4U }
eDMA channel status flags.
- enum _edma_error_status_flags {
kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,
kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,
kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,
kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,
kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,
kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,
kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,
kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,
kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,
kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,
kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,
kEDMA_ValidFlag = DMA_ES_VLD_MASK }
eDMA channel error status flags.
- enum edma_interrupt_enable_t {
kEDMA_ErrorInterruptEnable = 0x1U,
kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,
kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }
eDMA interrupt source
- enum edma_transfer_type_t {
kEDMA_MemoryToMemory = 0x0U,
kEDMA_PeripheralToMemory,
kEDMA_MemoryToPeripheral }
eDMA transfer type
- enum _edma_transfer_status {
kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),
kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }
eDMA transfer status

Driver version

- #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))
eDMA driver version

eDMA initialization and de-initialization

- void EDMA_Init (DMA_Type ∗base, const edma_config_t ∗config)
Initializes the eDMA peripheral.
- void EDMA_Deinit (DMA_Type ∗base)
Deinitializes the eDMA peripheral.
- void EDMA_GetDefaultConfig (edma_config_t ∗config)
Gets the eDMA default configuration structure.

eDMA Channel Operation

- void EDMA_ResetChannel (DMA_Type *base, uint32_t channel)
  Sets all TCD registers to default values.
- void EDMA_SetTransferConfig (DMA_Type *base, uint32_t channel, const edma_transfer_config_t *config, edma_tcd_t *nextTcd)
  Configures the eDMA transfer attribute.
- void EDMA_SetMinorOffsetConfig (DMA_Type *base, uint32_t channel, const edma_minor_offset_config_t *config)
  Configures the eDMA minor offset feature.
- static void EDMA_SetChannelPreemptionConfig (DMA_Type *base, uint32_t channel, const edma_channel_Preemption_config_t *config)
  Configures the eDMA channel preemption feature.
- void EDMA_SetChannelLink (DMA_Type *base, uint32_t channel, edma_channel_link_type_t type, uint32_t linkedChannel)
  Sets the channel link for the eDMA transfer.
- void EDMA_SetBandWidth (DMA_Type *base, uint32_t channel, edma_bandwidth_t bandWidth)
  Sets the bandwidth for the eDMA transfer.
- void EDMA_SetModulo (DMA_Type *base, uint32_t channel, edma_modulo_t srcModulo, edma_modulo_t destModulo)
  Sets the source modulo and the destination modulo for the eDMA transfer.
- static void EDMA_EnableAutoStopRequest (DMA_Type *base, uint32_t channel, bool enable)
  Enables an auto stop request for the eDMA transfer.
- void EDMA_EnableChannelInterrupts (DMA_Type *base, uint32_t channel, uint32_t mask)
  Enables the interrupt source for the eDMA transfer.
- void EDMA_DisableChannelInterrupts (DMA_Type *base, uint32_t channel, uint32_t mask)
  Disables the interrupt source for the eDMA transfer.

eDMA TCD Operation

- void EDMA_TcdReset (edma_tcd_t *tcd)
  Sets all fields to default values for the TCD structure.
- void EDMA_TcdSetTransferConfig (edma_tcd_t *tcd, const edma_transfer_config_t *config, edma_tcd_t *nextTcd)
  Configures the eDMA TCD transfer attribute.
- void EDMA_TcdSetMinorOffsetConfig (edma_tcd_t *tcd, const edma_minor_offset_config_t *config)
  Configures the eDMA TCD minor offset feature.
- void EDMA_TcdSetChannelLink (edma_tcd_t *tcd, edma_channel_link_type_t type, uint32_t linkedChannel)
  Sets the channel link for the eDMA TCD.
- static void EDMA_TcdSetBandWidth (edma_tcd_t *tcd, edma_bandwidth_t bandWidth)
  Sets the bandwidth for the eDMA TCD.
- void EDMA_TcdSetModulo (edma_tcd_t *tcd, edma_modulo_t srcModulo, edma_modulo_t destModulo)
  Sets the source modulo and the destination modulo for the eDMA TCD.
- static void EDMA_TcdEnableAutoStopRequest (edma_tcd_t *tcd, bool enable)
  Sets the auto stop request for the eDMA TCD.
- void EDMA_TcdEnableInterrupts (edma_tcd_t *tcd, uint32_t mask)
  Enables the interrupt source for the eDMA TCD.

- void EDMA_TcdDisableInterrupts (edma_tcd_t *tcd, uint32_t mask)
  Disables the interrupt source for the eDMA TCD.

eDMA Channel Transfer Operation

- static void EDMA_EnableChannelRequest (DMA_Type *base, uint32_t channel)
  Enables the eDMA hardware channel request.
- static void EDMA_DisableChannelRequest (DMA_Type *base, uint32_t channel)
  Disables the eDMA hardware channel request.
- static void EDMA_TriggerChannelStart (DMA_Type *base, uint32_t channel)
  Starts the eDMA transfer by using the software trigger.

eDMA Channel Status Operation

- uint32_t EDMA_GetRemainingBytes (DMA_Type *base, uint32_t channel)
  Gets the remaining bytes from the eDMA current channel TCD.
- static uint32_t EDMA_GetErrorStatusFlags (DMA_Type *base)
  Gets the eDMA channel error status flags.
- uint32_t EDMA_GetChannelStatusFlags (DMA_Type *base, uint32_t channel)
  Gets the eDMA channel status flags.
- void EDMA_ClearChannelStatusFlags (DMA_Type *base, uint32_t channel, uint32_t mask)
  Clears the eDMA channel status flags.

eDMA Transactional Operation

- void EDMA_CreateHandle (edma_handle_t *handle, DMA_Type *base, uint32_t channel)
  Creates the eDMA handle.
- void EDMA_InstallTCDMemory (edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
  Installs the TCDs memory pool into the eDMA handle.
- void EDMA_SetCallback (edma_handle_t *handle, edma_callback callback, void *userData)
  Installs a callback function for the eDMA transfer.
- void EDMA_PrepareTransfer (edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, edma_transfer_type_t type)
  Prepares the eDMA transfer structure.
- status_t EDMA_SubmitTransfer (edma_handle_t *handle, const edma_transfer_config_t *config)
  Submits the eDMA transfer request.
- void EDMA_StartTransfer (edma_handle_t *handle)
  eDMA starts transfer.
- void EDMA_StopTransfer (edma_handle_t *handle)
  eDMA stops transfer.
- void EDMA_AbortTransfer (edma_handle_t *handle)
  eDMA aborts transfer.
- void EDMA_HandleIRQ (edma_handle_t *handle)
  eDMA IRQ handler for the current major loop transfer completion.

13.3 Data Structure Documentation

13.3.1 struct edma_config_t

Data Fields

- bool enableContinuousLinkMode
  Enable (true) continuous link mode.
- bool enableHaltOnError
  Enable (true) transfer halt on error.
- bool enableRoundRobinArbitration
  Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection.
- bool enableDebugMode
  Enable(true) eDMA debug mode.

13.3.1.0.0.27 Field Documentation

13.3.1.0.0.27.1 bool edma_config_t::enableContinuousLinkMode

Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.

13.3.1.0.0.27.2 bool edma_config_t::enableHaltOnError

Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

13.3.1.0.0.27.3 bool edma_config_t::enableDebugMode

When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

13.3.2 struct edma_transfer_config_t

This structure configures the source/destination transfer attribute. This figure shows the eDMA's transfer model:

| Transfer Size | | Minor Loop |_______________| Major loop Count 1 | Bytes | Transfer Size | | ______-______|________________|_____________________|–> Minor loop complete

| | | |_______________| Major Loop Count 2 | | | | |_______________|_____________________|–> Minor loop Complete

--—> Transfer complete

Data Fields

- uint32_t srcAddr
  Source data address.
- uint32_t destAddr
  Destination data address.
- edma_transfer_size_t srcTransferSize
  Source data transfer size.
- edma_transfer_size_t destTransferSize
  Destination data transfer size.
- int16_t srcOffset
  Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.
- int16_t destOffset
  Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.
- uint32_t minorLoopBytes
  Bytes to transfer in a minor loop.
- uint32_t majorLoopCounts
  Major loop iteration count.

13.3.2.0.0.28 Field Documentation

13.3.2.0.0.28.1 uint32_t edma_transfer_config_t::srcAddr

13.3.2.0.0.28.2 uint32_t edma_transfer_config_t::destAddr

13.3.2.0.0.28.3 edma_transfer_size_t edma_transfer_config_t::srcTransferSize

13.3.2.0.0.28.4 edma_transfer_size_t edma_transfer_config_t::destTransferSize

13.3.2.0.0.28.5 int16_t edma_transfer_config_t::srcOffset

13.3.2.0.0.28.6 int16_t edma_transfer_config_t::destOffset

13.3.2.0.0.28.7 uint32_t edma_transfer_config_t::majorLoopCounts

13.3.3 struct edma_channel_Preemption_config_t

Data Fields

- bool enableChannelPreemption
  If true: a channel can be suspended by other channel with higher priority.
- bool enablePreemptAbility
  If true: a channel can suspend other channel with low priority.
- uint8_t channelPriority
  Channel priority.

13.3.4 struct edma_minor_offset_config_t

Data Fields

- bool enableSrcMinorOffset
  Enable(true) or Disable(false) source minor loop offset.
- bool enableDestMinorOffset
  Enable(true) or Disable(false) destination minor loop offset.
- uint32_t minorOffset
  Offset for a minor loop mapping.

13.3.4.0.0.29 Field Documentation

13.3.4.0.0.29.1 bool edma_minor_offset_config_t::enableSrcMinorOffset

13.3.4.0.0.29.2 bool edma_minor_offset_config_t::enableDestMinorOffset

13.3.4.0.0.29.3 uint32_t edma_minor_offset_config_t::minorOffset

13.3.5 struct edma_tcd_t

This structure is the same as the TCD register, which is described in the chip reference manual and is used to configure scatter/gather feature as a next hardware TCD.

Data Fields

- __IO uint32_t SADDR
  SADDR register, used to save source address.
- __IO uint16_t SOFF
  SOFF register, save offset bytes every transfer.
- __IO uint16_t ATTR
  ATTR register, source/destination transfer size and modulo.
- __IO uint32_t NBYTES
  Nbytes register, minor loop length in bytes.
- __IO uint32_t SLAST
  SLAST register.
- __IO uint32_t DADDR
  DADDR register, used for destination address.
- __IO uint16_t DOFF
  DOFF register, used for destination offset.
- __IO uint16_t CITER
  CITER register, current minor loop numbers, for unfinished minor loop.
- __IO uint32_t DLAST_SGA
  DLASTSGA register, next stcd address used in scatter-gather mode.
- __IO uint16_t CSR
  CSR register, for TCD control status.
- __IO uint16_t BITER
  BITER register, begin minor loop count.

13.3.5.0.0.30 Field Documentation

13.3.5.0.0.30.1 __IO uint16_t edma_tcd_t::CITER

13.3.5.0.0.30.2 __IO uint16_t edma_tcd_t::BITER

13.3.6 struct edma_handle_t

Data Fields

- edma_callback callback
  Callback function for major count exhausted.
- void ∗ userData
  Callback function parameter.
- DMA_Type ∗ base
  eDMA peripheral base address.
- edma_tcd_t ∗ tcdPool
  Pointer to memory stored TCDs.
- uint8_t channel
  eDMA channel number.
- volatile int8_t header
  The first TCD index.
- volatile int8_t tail
  The last TCD index.
- volatile int8_t tcdUsed
  The number of used TCD slots.
- volatile int8_t tcdSize
  The total number of TCD slots in the queue.
- uint8_t flags
  The status of the current channel.

13.3.6.0.0.31 Field Documentation

13.3.6.0.0.31.1 edma_callback edma_handle_t::callback

13.3.6.0.0.31.2 void* edma_handle_t::userData

13.3.6.0.0.31.3 DMA_Type* edma_handle_t::base

13.3.6.0.0.31.4 edma_tcd_t* edma_handle_t::tcdPool

13.3.6.0.0.31.5 uint8_t edma_handle_t::channel

13.3.6.0.0.31.6 volatile int8_t edma_handle_t::header

13.3.6.0.0.31.7 volatile int8_t edma_handle_t::tail

13.3.6.0.0.31.8 volatile int8_t edma_handle_t::tcdUsed

13.3.6.0.0.31.9 volatile int8_t edma_handle_t::tcdSize

13.3.6.0.0.31.10 uint8_t edma_handle_t::flags

13.4 Macro Definition Documentation

13.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

Version 2.0.3.

13.5 Typedef Documentation

13.5.1 typedef void(* edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)

13.6 Enumeration Type Documentation

13.6.1 enum edma_transfer_size_t

Enumerator

kEDMA_TransferSize1Bytes Source/Destination data transfer size is 1 byte every time.
kEDMA_TransferSize2Bytes Source/Destination data transfer size is 2 bytes every time.
kEDMA_TransferSize4Bytes Source/Destination data transfer size is 4 bytes every time.
kEDMA_TransferSize16Bytes Source/Destination data transfer size is 16 bytes every time.
kEDMA_TransferSize32Bytes Source/Destination data transfer size is 32 bytes every time.

13.6.2 enum edma_modulo_t

Enumerator

kEDMA_ModuloDisable Disable modulo.
kEDMA_Modulo2bytes Circular buffer size is 2 bytes.
kEDMA_Modulo4bytes Circular buffer size is 4 bytes.
kEDMA_Modulo8bytes Circular buffer size is 8 bytes.
kEDMA_Modulo16bytes Circular buffer size is 16 bytes.
kEDMA_Modulo32bytes Circular buffer size is 32 bytes.
kEDMA_Modulo64bytes Circular buffer size is 64 bytes.
kEDMA_Modulo128bytes Circular buffer size is 128 bytes.
kEDMA_Modulo256bytes Circular buffer size is 256 bytes.
kEDMA_Modulo512bytes Circular buffer size is 512 bytes.
kEDMA_Modulo1Kbytes Circular buffer size is 1 K bytes.
kEDMA_Modulo2Kbytes Circular buffer size is 2 K bytes.
kEDMA_Modulo4Kbytes Circular buffer size is 4 K bytes.
kEDMA_Modulo8Kbytes Circular buffer size is 8 K bytes.
kEDMA_Modulo16Kbytes Circular buffer size is 16 K bytes.
kEDMA_Modulo32Kbytes Circular buffer size is 32 K bytes.
kEDMA_Modulo64Kbytes Circular buffer size is 64 K bytes.
kEDMA_Modulo128Kbytes Circular buffer size is 128 K bytes.
kEDMA_Modulo256Kbytes Circular buffer size is 256 K bytes.
kEDMA_Modulo512Kbytes Circular buffer size is 512 K bytes.
kEDMA_Modulo1Mbytes Circular buffer size is 1 M bytes.
kEDMA_Modulo2Mbytes Circular buffer size is 2 M bytes.
kEDMA_Modulo4Mbytes Circular buffer size is 4 M bytes.
kEDMA_Modulo8Mbytes Circular buffer size is 8 M bytes.
kEDMA_Modulo16Mbytes Circular buffer size is 16 M bytes.
kEDMA_Modulo32Mbytes Circular buffer size is 32 M bytes.
kEDMA_Modulo64Mbytes Circular buffer size is 64 M bytes.
kEDMA_Modulo128Mbytes Circular buffer size is 128 M bytes.
kEDMA_Modulo256Mbytes Circular buffer size is 256 M bytes.
kEDMA_Modulo512Mbytes Circular buffer size is 512 M bytes.
kEDMA_Modulo1Gbytes Circular buffer size is 1 G bytes.
kEDMA_Modulo2Gbytes Circular buffer size is 2 G bytes.

13.6.3 enum edma_bandwidth_t

Enumerator

kEDMA_BandwidthStallNone No eDMA engine stalls.
kEDMA_BandwidthStall4Cycle eDMA engine stalls for 4 cycles after each read/write.
kEDMA_BandwidthStall8Cycle eDMA engine stalls for 8 cycles after each read/write.

13.6.4 enum edma_channel_link_type_t

Enumerator

kEDMA_LinkNone No channel link.
kEDMA_MinorLink Channel link after each minor loop.
kEDMA_MajorLink Channel link while major loop count exhausted.

13.6.5 enum _edma_channel_status_flags

Enumerator

kEDMA_DoneFlag DONE flag, set while transfer finished, CITER value exhausted.
kEDMA_ErrorFlag eDMA error flag, an error occurred in a transfer
kEDMA_InterruptFlag eDMA interrupt flag, set while an interrupt occurred of this channel

13.6.6 enum _edma_error_status_flags

Enumerator

kEDMA_DestinationBusErrorFlag Bus error on destination address.
kEDMA_SourceBusErrorFlag Bus error on the source address.
kEDMA_ScatterGatherErrorFlag Error on the Scatter/Gather address, not 32byte aligned.
kEDMA_NbytesErrorFlag NBYTES/CITER configuration error.
kEDMA_DestinationOffsetErrorFlag Destination offset not aligned with destination size.
kEDMA_DestinationAddressErrorFlag Destination address not aligned with destination size.
kEDMA_SourceOffsetErrorFlag Source offset not aligned with source size.
kEDMA_SourceAddressErrorFlag Source address not aligned with source size.
kEDMA_ErrorChannelFlag Error channel number of the cancelled channel number.
kEDMA_ChannelPriorityErrorFlag Channel priority is not unique.
kEDMA_TransferCanceledFlag Transfer cancelled.
kEDMA_ValidFlag No error occurred, this bit is 0. Otherwise, it is 1.

13.6.7 enum edma_interrupt_enable_t

Enumerator

kEDMA_ErrorInterruptEnable Enable interrupt while channel error occurs.
kEDMA_MajorInterruptEnable Enable interrupt while major count exhausted.
kEDMA_HalfInterruptEnable Enable interrupt while major count to half value.

13.6.8 enum edma_transfer_type_t

Enumerator

kEDMA_MemoryToMemory Transfer from memory to memory.
kEDMA_PeripheralToMemory Transfer from peripheral to memory.
kEDMA_MemoryToPeripheral Transfer from memory to peripheral.

13.6.9 enum _edma_transfer_status

Enumerator

kStatus_EDMA_QueueFull TCD queue is full.
kStatus_EDMA_Busy Channel is busy and can't handle the transfer request.

13.7 Function Documentation

13.7.1 void EDMA_Init (DMA_Type ∗ *base*, const edma_config_t ∗ *config*)

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Parameters

base	eDMA peripheral base address.
config	A pointer to the configuration structure, see "edma_config_t".

Note

This function enables the minor loop map feature.

13.7.2 void EDMA_Deinit (DMA_Type ∗ *base*)

This function gates the eDMA clock.

Parameters

base	eDMA peripheral base address.

13.7.3 void EDMA_GetDefaultConfig (edma_config_t ∗ *config*)

This function sets the configuration structure to default values. The default configuration is set to the following values:

```
*   config.enableContinuousLinkMode = false;
*   config.enableHaltOnError = true;
*   config.enableRoundRobinArbitration = false;
*   config.enableDebugMode = false;
*
```

Parameters

config	A pointer to the eDMA configuration structure.

13.7.4 void EDMA_ResetChannel (DMA_Type ∗ *base*, uint32_t *channel*)

This function sets TCD registers for this channel to default values.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Note

> This function must not be called while the channel transfer is ongoing or it causes unpredictable results.
> This function enables the auto stop request feature.

13.7.5 void EDMA_SetTransferConfig (DMA_Type ∗ *base*, uint32_t *channel*, const edma_transfer_config_t ∗ *config*, edma_tcd_t ∗ *nextTcd*)

This function configures the transfer attribute, including the source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
*  edma_transfer_t config;
*  edma_tcd_t tcd;
*  config.srcAddr = ..;
*  config.destAddr = ..;
*  ...
*  EDMA_SetTransferConfig(DMA0, channel, &config, &stcd);
*
```

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
config	A pointer to the eDMA transfer configuration structure.
nextTcd	A pointer to the TCD structure. The pointer can be NULL to disable the scatter/gather feature.

Note

If the nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the eDMA_ResetChannel.

13.7.6 void EDMA_SetMinorOffsetConfig (DMA_Type ∗ *base*, uint32_t *channel*, const edma_minor_offset_config_t ∗ *config*)

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.
channel	eDMA channel number.
config	A pointer to the minor offset configuration structure.

13.7.7 static void EDMA_SetChannelPreemptionConfig (DMA_Type ∗ *base*, uint32_t *channel*, const edma_channel_Preemption_config_t ∗ *config*) [inline], [static]

This function configures the channel preemption attribute and the priority of the channel.

Parameters

base	eDMA peripheral base address.

channel	eDMA channel number
config	A pointer to the channel preemption configuration structure.

13.7.8 void EDMA_SetChannelLink (DMA_Type * *base*, uint32_t *channel*, edma_channel_link_type_t *type*, uint32_t *linkedChannel*)

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
type	A channel link type, which can be one of the following: • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink
linkedChannel	The linked channel number.

Note

Users should ensure that the DONE flag is cleared before calling this interface or the configuration is invalid.

13.7.9 void EDMA_SetBandWidth (DMA_Type * *base*, uint32_t *channel*, edma_bandwidth_t *bandWidth*)

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.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
bandWidth	A bandwidth setting, which can be one of the following: • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

13.7.10 void EDMA_SetModulo (DMA_Type ∗ *base*, uint32_t *channel*, edma_modulo_t *srcModulo*, edma_modulo_t *destModulo*)

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

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
srcModulo	A source modulo value.
destModulo	A destination modulo value.

13.7.11 static void EDMA_EnableAutoStopRequest (DMA_Type ∗ *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
enable	The command to enable (true) or disable (false).

13.7.12 void EDMA_EnableChannelInterrupts (DMA_Type ∗ *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

13.7.13 void EDMA_DisableChannelInterrupts (DMA_Type ∗ *base,* uint32_t *channel,* uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

13.7.14 void EDMA_TcdReset (edma_tcd_t ∗ *tcd*)

This function sets all fields for this TCD structure to default value.

Parameters

tcd	Pointer to the TCD structure.

Note

This function enables the auto stop request feature.

13.7.15 void EDMA_TcdSetTransferConfig (edma_tcd_t ∗ *tcd,* const edma_transfer_config_t ∗ *config,* edma_tcd_t ∗ *nextTcd*)

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TC-D registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
*   edma_transfer_t config = {
*   ...
```

```
*   }
*   edma_tcd_t tcd __aligned(32);
*   edma_tcd_t nextTcd __aligned(32);
*   EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
*
```

Parameters

tcd	Pointer to the TCD structure.
config	Pointer to eDMA transfer configuration structure.
nextTcd	Pointer to the next TCD structure. It can be NULL if user do not want to enable scatter/gather feature.

Note

TCD address should be 32 bytes aligned or it causes an eDMA error.
If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

13.7.16 void EDMA_TcdSetMinorOffsetConfig (edma_tcd_t * *tcd,* const edma_minor_offset_config_t * *config*)

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

tcd	A point to the TCD structure.
config	A pointer to the minor offset configuration structure.

13.7.17 void EDMA_TcdSetChannelLink (edma_tcd_t * *tcd,* edma_channel_link_type_t *type,* uint32_t *linkedChannel*)

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note

Ensure that DONE flag is cleared before calling this interface or the configuration is invalid.

Parameters

tcd	A pointer to the TCD structure.
type	A channel link type, which can be one of the following: • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink
linkedChannel	The linked channel number.

13.7.18 static void EDMA_TcdSetBandWidth (edma_tcd_t * *tcd,* edma_bandwidth_t *bandWidth*) [inline], [static]

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.

Parameters

tcd	A pointer to the TCD structure.
bandWidth	A bandwidth setting, which can be one of the following: • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

13.7.19 void EDMA_TcdSetModulo (edma_tcd_t * *tcd,* edma_modulo_t *srcModulo,* edma_modulo_t *destModulo*)

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

Parameters

tcd	A pointer to the TCD structure.

srcModulo	A source modulo value.
destModulo	A destination modulo value.

13.7.20 static void EDMA_TcdEnableAutoStopRequest (edma_tcd_t ∗ *tcd*, bool *enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

tcd	A pointer to the TCD structure.
enable	The command to enable (true) or disable (false).

13.7.21 void EDMA_TcdEnableInterrupts (edma_tcd_t ∗ *tcd*, uint32_t *mask*)

Parameters

tcd	A pointer to the TCD structure.
mask	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

13.7.22 void EDMA_TcdDisableInterrupts (edma_tcd_t ∗ *tcd*, uint32_t *mask*)

Parameters

tcd	A pointer to the TCD structure.
mask	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

13.7.23 static void EDMA_EnableChannelRequest (DMA_Type ∗ *base*, uint32_t *channel*) [inline], [static]

This function enables the hardware channel request.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

13.7.24 static void EDMA_DisableChannelRequest (DMA_Type ∗ *base*, uint32_t *channel*) [inline], [static]

This function disables the hardware channel request.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

13.7.25 static void EDMA_TriggerChannelStart (DMA_Type ∗ *base*, uint32_t *channel*) [inline], [static]

This function starts a minor loop transfer.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

13.7.26 uint32_t EDMA_GetRemainingBytes (DMA_Type ∗ *base*, uint32_t *channel*)

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of bytes that have not finished.

Parameters

base	eDMA peripheral base address.

channel	eDMA channel number.

Returns

Bytes have not been transferred yet for the current TCD.

Note

This function can only be used to get unfinished bytes without the next TCD or it might lead to inaccuracies.

13.7.27 static uint32_t EDMA_GetErrorStatusFlags (DMA_Type ∗ *base*) [inline], [static]

Parameters

base	eDMA peripheral base address.

Returns

The mask of the error status flags. Use _edma_error_status_flags type to decode the return variables.

13.7.28 uint32_t EDMA_GetChannelStatusFlags (DMA_Type ∗ *base,* uint32_t *channel*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Returns

The mask of the channel status flags. Use _edma_channel_status_flags type to decode the return variables.

13.7.29 void EDMA_ClearChannelStatusFlags (DMA_Type ∗ *base,* uint32_t *channel,* uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of the channel status to be cleared. Use the defined _edma_channel_status-_flags type.

13.7.30 void EDMA_CreateHandle (edma_handle_t ∗ *handle,* DMA_Type ∗ *base,* uint32_t *channel*)

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

Parameters

handle	eDMA handle pointer. The eDMA handle stores callback function and parameters.
base	eDMA peripheral base address.
channel	eDMA channel number.

13.7.31 void EDMA_InstallTCDMemory (edma_handle_t ∗ *handle,* edma_tcd_t ∗ *tcdPool,* uint32_t *tcdSize*)

This function is called after the EDMA_CreateHandle to use scatter/gather feature.

Parameters

handle	eDMA handle pointer.
tcdPool	A memory pool to store TCDs. It must be 32 bytes aligned.
tcdSize	The number of TCD slots.

13.7.32 void EDMA_SetCallback (edma_handle_t ∗ *handle,* edma_callback *callback,* void ∗ *userData*)

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

Parameters

handle	eDMA handle pointer.
callback	eDMA callback function pointer.
userData	A parameter for the callback function.

13.7.33 void EDMA_PrepareTransfer (edma_transfer_config_t * *config*, void * *srcAddr*, uint32_t *srcWidth*, void * *destAddr*, uint32_t *destWidth*, uint32_t *bytesEachRequest*, uint32_t *transferBytes*, edma_transfer_type_t *type*)

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

Parameters

config	The user configuration structure of type edma_transfer_t.
srcAddr	eDMA transfer source address.
srcWidth	eDMA transfer source address width(bytes).
destAddr	eDMA transfer destination address.
destWidth	eDMA transfer destination address width(bytes).
bytesEach-Request	eDMA transfer bytes per channel request.
transferBytes	eDMA transfer bytes to be transferred.
type	eDMA transfer type.

Note

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

13.7.34 status_t EDMA_SubmitTransfer (edma_handle_t * *handle*, const edma_transfer_config_t * *config*)

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

Parameters

handle	eDMA handle pointer.
config	Pointer to eDMA transfer configuration structure.

Return values

kStatus_EDMA_Success	It means submit transfer request succeed.
kStatus_EDMA_QueueFull	It means TCD queue is full. Submit transfer request is not allowed.
kStatus_EDMA_Busy	It means the given channel is busy, need to submit request later.

13.7.35 void EDMA_StartTransfer (edma_handle_t * *handle*)

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

Parameters

handle	eDMA handle pointer.

13.7.36 void EDMA_StopTransfer (edma_handle_t * *handle*)

This function disables the channel request to pause the transfer. Call the EDMA_StartTransfer() again to resume the transfer.

Parameters

handle	eDMA handle pointer.

13.7.37 void EDMA_AbortTransfer (edma_handle_t * *handle*)

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

handle	DMA handle pointer.

13.7.38 void EDMA_HandleIRQ (edma_handle_t * *handle*)

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

Parameters

handle	eDMA handle pointer.

Chapter 14
ENC: Quadrature Encoder/Decoder

14.1 Overview

The SDK provides Peripheral driver for the Quadrature Encoder/Decoder (ENC) module of Kinetis devices.

This section describes the programming interface of the ENC Peripheral driver. The ENC driver configures ENC module, provides functional interface for user to build enc application.

14.2 Function groups

14.2.1 Initialization and De-initialization

This function group initializes default configuration structure for the ENC counter, initialize ENC counter with the normal configuration and de-initialize ENC module. Some APIs are also created to control the features.

14.2.2 Status

This function group get/clear the ENC status.

14.2.3 Interrupts

This function group enable/disable the ENC interrupts.

14.2.4 Value Operation

This function group get the counter/hold value of positions.

14.3 Typical use case

14.3.1 Polling Configuration

```
enc_config_t mEncConfigStruct;
uint32_t mCurPosValue;

BOARD_InitHardware();

// Initialize the ENC module.
ENC_GetDefaultConfig(&mEncConfigStruct);
```

```
ENC_Init(DEMO_ENC_INSTANCE, &mEncConfigStruct);
ENC_DoSoftwareLoadInitialPositionValue(DEMO_ENC_INSTANCE);

while (1)
{
    // ...

    // This read operation would capture all the position registers to responding hold registers.
    mCurPosValue = ENC_GetPositionValue(DEMO_ENC_INSTANCE);

    PRINTF("Current position value: %ld\r\n", mCurPosValue);
    PRINTF("Current position differential value: %d\r\n", (int16_t)
      ENC_GetHoldPositionDifferenceValue(DEMO_ENC_INSTANCE));
    PRINTF("Current position revolution value: %d\r\n",
      ENC_GetHoldRevolutionValue(DEMO_ENC_INSTANCE));
    PRINTF("g_EncIndexCounter: %d\r\n", g_EncIndexCounter);
}
```

Data Structures

- struct enc_config_t
  Define user configuration structure for ENC module. More...
- struct enc_self_test_config_t
  Define configuration structure for self test module. More...

Macros

- #define FSL_ENC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

Enumerations

- enum _enc_interrupt_enable {
  kENC_HOMETransitionInterruptEnable = (1U << 0U),
  kENC_INDEXPulseInterruptEnable = (1U << 1U),
  kENC_WatchdogTimeoutInterruptEnable = (1U << 2U),
  kENC_PositionCompareInerruptEnable = (1U << 3U),
  kENC_SimultBothPhaseChangeInterruptEnable,
  kENC_PositionRollOverInterruptEnable = (1U << 5U),
  kENC_PositionRollUnderInterruptEnable = (1U << 6U) }
  Interrupt enable/disable mask.
- enum _enc_status_flags {
  kENC_HOMETransitionFlag = (1U << 0U),
  kENC_INDEXPulseFlag = (1U << 1U),
  kENC_WatchdogTimeoutFlag = (1U << 2U),
  kENC_PositionCompareFlag = (1U << 3U),
  kENC_SimultBothPhaseChangeFlag = (1U << 4U),
  kENC_PositionRollOverFlag = (1U << 5U),
  kENC_PositionRollUnderFlag = (1U << 6U),
  kENC_LastCountDirectionFlag = (1U << 7U) }
  Status flag mask.

- enum _enc_signal_status_flags {
kENC_RawHOMEStatusFlag = ENC_IMR_HOME_MASK,
kENC_RawINDEXStatusFlag = ENC_IMR_INDEX_MASK,
kENC_RawPHBStatusFlag = ENC_IMR_PHB_MASK,
kENC_RawPHAEXStatusFlag = ENC_IMR_PHA_MASK,
kENC_FilteredHOMEStatusFlag = ENC_IMR_FHOM_MASK,
kENC_FilteredINDEXStatusFlag = ENC_IMR_FIND_MASK,
kENC_FilteredPHBStatusFlag = ENC_IMR_FPHB_MASK,
kENC_FilteredPHAStatusFlag = ENC_IMR_FPHA_MASK }
Signal status flag mask.
- enum enc_home_trigger_mode_t {
kENC_HOMETriggerDisabled = 0U,
kENC_HOMETriggerOnRisingEdge,
kENC_HOMETriggerOnFallingEdge }
Define HOME signal's trigger mode.
- enum enc_index_trigger_mode_t {
kENC_INDEXTriggerDisabled = 0U,
kENC_INDEXTriggerOnRisingEdge,
kENC_INDEXTriggerOnFallingEdge }
Define INDEX signal's trigger mode.
- enum enc_decoder_work_mode_t {
kENC_DecoderWorkAsNormalMode = 0U,
kENC_DecoderWorkAsSignalPhaseCountMode }
Define type for decoder work mode.
- enum enc_position_match_mode_t {
kENC_POSMATCHOnPositionCounterEqualToComapreValue = 0U,
kENC_POSMATCHOnReadingAnyPositionCounter }
Define type for the condition of POSMATCH pulses.
- enum enc_revolution_count_condition_t {
kENC_RevolutionCountOnINDEXPulse = 0U,
kENC_RevolutionCountOnRollOverModulus }
Define type for determining how the revolution counter (REV) is incremented/decremented.
- enum enc_self_test_direction_t {
kENC_SelfTestDirectionPositive = 0U,
kENC_SelfTestDirectionNegative }
Define type for direction of self test generated signal.

Variables

- bool enc_config_t::enableReverseDirection
Enable reverse direction counting.
- enc_decoder_work_mode_t enc_config_t::decoderWorkMode
Enable signal phase count mode.
- enc_home_trigger_mode_t enc_config_t::HOMETriggerMode
Enable HOME to initialize position counters.
- enc_index_trigger_mode_t enc_config_t::INDEXTriggerMode
Enable INDEX to initialize position counters.
- bool enc_config_t::enableTRIGGERClearPositionCounter

Clear POSD, REV, UPOS and LPOS on rising edge of TRIGGER, or not.
- bool enc_config_t::enableTRIGGERClearHoldPositionCounter
  Enable update of hold registers on rising edge of TRIGGER, or not.
- bool enc_config_t::enableWatchdog
  Enable the watchdog to detect if the target is moving or not.
- uint16_t enc_config_t::watchdogTimeoutValue
  Watchdog timeout count value.
- uint16_t enc_config_t::filterCount
  Input Filter Sample Count.
- uint16_t enc_config_t::filterSamplePeriod
  Input Filter Sample Period.
- enc_position_match_mode_t enc_config_t::positionMatchMode
  The condition of POSMATCH pulses.
- uint32_t enc_config_t::positionCompareValue
  Position compare value.
- enc_revolution_count_condition_t enc_config_t::revolutionCountCondition
  Revolution Counter Modulus Enable.
- bool enc_config_t::enableModuloCountMode
  Enable Modulo Counting.
- uint32_t enc_config_t::positionModulusValue
  Position modulus value.
- uint32_t enc_config_t::positionInitialValue
  Position initial value.
- enc_self_test_direction_t enc_self_test_config_t::signalDirection
  Direction of self test generated signal.
- uint16_t enc_self_test_config_t::signalCount
  Hold the number of quadrature advances to generate.
- uint16_t enc_self_test_config_t::signalPeriod
  Hold the period of quadrature phase in IPBus clock cycles.

Initialization and De-initialization

- void ENC_Init (ENC_Type ∗base, const enc_config_t ∗config)
  Initialization for the ENC module.
- void ENC_Deinit (ENC_Type ∗base)
  De-initialization for the ENC module.
- void ENC_GetDefaultConfig (enc_config_t ∗config)
  Get an available pre-defined settings for ENC's configuration.
- void ENC_DoSoftwareLoadInitialPositionValue (ENC_Type ∗base)
  Load the initial position value to position counter.
- void ENC_SetSelfTestConfig (ENC_Type ∗base, const enc_self_test_config_t ∗config)
  Enable and configure the self test function.

Status

- uint32_t ENC_GetStatusFlags (ENC_Type ∗base)
  Get the status flags.
- void ENC_ClearStatusFlags (ENC_Type ∗base, uint32_t mask)
  Clear the status flags.
- static uint16_t ENC_GetSignalStatusFlags (ENC_Type ∗base)
  Get the signals' real-time status.

Interrupts

- void ENC_EnableInterrupts (ENC_Type *base, uint32_t mask)
  Enable the interrupts.
- void ENC_DisableInterrupts (ENC_Type *base, uint32_t mask)
  Disable the interrupts.
- uint32_t ENC_GetEnabledInterrupts (ENC_Type *base)
  Get the enabled interrupts' flags.

Value Operation

- uint32_t ENC_GetPositionValue (ENC_Type *base)
  Get the current position counter's value.
- uint32_t ENC_GetHoldPositionValue (ENC_Type *base)
  Get the hold position counter's value.
- static uint16_t ENC_GetPositionDifferenceValue (ENC_Type *base)
  Get the position difference counter's value.
- static uint16_t ENC_GetHoldPositionDifferenceValue (ENC_Type *base)
  Get the hold position difference counter's value.
- static uint16_t ENC_GetRevolutionValue (ENC_Type *base)
  Get the position revolution counter's value.
- static uint16_t ENC_GetHoldRevolutionValue (ENC_Type *base)
  Get the hold position revolution counter's value.

14.4 Data Structure Documentation

14.4.1 struct enc_config_t

Data Fields

- bool enableReverseDirection
  Enable reverse direction counting.
- enc_decoder_work_mode_t decoderWorkMode
  Enable signal phase count mode.
- enc_home_trigger_mode_t HOMETriggerMode
  Enable HOME to initialize position counters.
- enc_index_trigger_mode_t INDEXTriggerMode
  Enable INDEX to initialize position counters.
- bool enableTRIGGERClearPositionCounter
  Clear POSD, REV, UPOS and LPOS on rising edge of TRIGGER, or not.
- bool enableTRIGGERClearHoldPositionCounter
  Enable update of hold registers on rising edge of TRIGGER, or not.
- bool enableWatchdog
  Enable the watchdog to detect if the target is moving or not.
- uint16_t watchdogTimeoutValue
  Watchdog timeout count value.
- uint16_t filterCount
  Input Filter Sample Count.
- uint16_t filterSamplePeriod
  Input Filter Sample Period.

- enc_position_match_mode_t positionMatchMode
  The condition of POSMATCH pulses.
- uint32_t positionCompareValue
  Position compare value.
- enc_revolution_count_condition_t revolutionCountCondition
  Revolution Counter Modulus Enable.
- bool enableModuloCountMode
  Enable Modulo Counting.
- uint32_t positionModulusValue
  Position modulus value.
- uint32_t positionInitialValue
  Position initial value.

14.4.2 struct enc_self_test_config_t

The self test module provides a quadrature test signal to the inputs of the quadrature decoder module. This is a factory test feature. It is also useful to customers' software development and testing.

Data Fields

- enc_self_test_direction_t signalDirection
  Direction of self test generated signal.
- uint16_t signalCount
  Hold the number of quadrature advances to generate.
- uint16_t signalPeriod
  Hold the period of quadrature phase in IPBus clock cycles.

14.5 Macro Definition Documentation

14.5.1 #define FSL_ENC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

14.6 Enumeration Type Documentation

14.6.1 enum _enc_interrupt_enable

Enumerator

kENC_HOMETransitionInterruptEnable HOME interrupt enable.
kENC_INDEXPulseInterruptEnable INDEX pulse interrupt enable.
kENC_WatchdogTimeoutInterruptEnable Watchdog timeout interrupt enable.
kENC_PositionCompareInerruptEnable Position compare interrupt enable.
kENC_SimultBothPhaseChangeInterruptEnable Simultaneous PHASEA and PHASEB change interrupt enable.
kENC_PositionRollOverInterruptEnable Roll-over interrupt enable.
kENC_PositionRollUnderInterruptEnable Roll-under interrupt enable.

14.6.2 enum _enc_status_flags

These flags indicate the counter's events.

Enumerator

kENC_HOMETransitionFlag HOME signal transition interrupt request.
kENC_INDEXPulseFlag INDEX Pulse Interrupt Request.
kENC_WatchdogTimeoutFlag Watchdog timeout interrupt request.
kENC_PositionCompareFlag Position compare interrupt request.
kENC_SimultBothPhaseChangeFlag Simultaneous PHASEA and PHASEB change interrupt request.
kENC_PositionRollOverFlag Roll-over interrupt request.
kENC_PositionRollUnderFlag Roll-under interrupt request.
kENC_LastCountDirectionFlag Last count was in the up direction, or the down direction.

14.6.3 enum _enc_signal_status_flags

These flags indicate the counter's signal.

Enumerator

kENC_RawHOMEStatusFlag Raw HOME input.
kENC_RawINDEXStatusFlag Raw INDEX input.
kENC_RawPHBStatusFlag Raw PHASEB input.
kENC_RawPHAEXStatusFlag Raw PHASEA input.
kENC_FilteredHOMEStatusFlag The filtered version of HOME input.
kENC_FilteredINDEXStatusFlag The filtered version of INDEX input.
kENC_FilteredPHBStatusFlag The filtered version of PHASEB input.
kENC_FilteredPHAStatusFlag The filtered version of PHASEA input.

14.6.4 enum enc_home_trigger_mode_t

The ENC would count the trigger from HOME signal line.

Enumerator

kENC_HOMETriggerDisabled HOME signal's trigger is disabled.
kENC_HOMETriggerOnRisingEdge Use positive going edge-to-trigger initialization of position counters.
kENC_HOMETriggerOnFallingEdge Use negative going edge-to-trigger initialization of position counters.

14.6.5 enum enc_index_trigger_mode_t

The ENC would count the trigger from INDEX signal line.

Enumerator

kENC_INDEXTriggerDisabled INDEX signal's trigger is disabled.
kENC_INDEXTriggerOnRisingEdge Use positive going edge-to-trigger initialization of position counters.
kENC_INDEXTriggerOnFallingEdge Use negative going edge-to-trigger initialization of position counters.

14.6.6 enum enc_decoder_work_mode_t

The normal work mode uses the standard quadrature decoder with PHASEA and PHASEB. When in signal phase count mode, a positive transition of the PHASEA input generates a count signal while the PHASEB input and the reverse direction control the counter direction. If the reverse direction is not enabled, PHASEB = 0 means counting up and PHASEB = 1 means counting down. Otherwise, the direction is reversed.

Enumerator

kENC_DecoderWorkAsNormalMode Use standard quadrature decoder with PHASEA and PHASEB.
kENC_DecoderWorkAsSignalPhaseCountMode PHASEA input generates a count signal while PHASEB input control the direction.

14.6.7 enum enc_position_match_mode_t

Enumerator

kENC_POSMATCHOnPositionCounterEqualToComapreValue POSMATCH pulses when a match occurs between the position counters (POS) and the compare value (COMP).
kENC_POSMATCHOnReadingAnyPositionCounter POSMATCH pulses when any position counter register is read.

14.6.8 enum enc_revolution_count_condition_t

Enumerator

kENC_RevolutionCountOnINDEXPulse Use INDEX pulse to increment/decrement revolution counter.
kENC_RevolutionCountOnRollOverModulus Use modulus counting roll-over/under to increment/decrement revolution counter.

14.6.9 enum enc_self_test_direction_t

Enumerator

kENC_SelfTestDirectionPositive Self test generates the signal in positive direction.
kENC_SelfTestDirectionNegative Self test generates the signal in negative direction.

14.7 Function Documentation

14.7.1 void ENC_Init (ENC_Type ∗ *base*, const enc_config_t ∗ *config*)

This function is to make the initialization for the ENC module. It should be called firstly before any operation to the ENC with the operations like:

- Enable the clock for ENC module.
- Configure the ENC's working attributes.

Parameters

base	ENC peripheral base address.
config	Pointer to configuration structure. See to "enc_config_t".

14.7.2 void ENC_Deinit (ENC_Type ∗ *base*)

This function is to make the de-initialization for the ENC module. It could be called when ENC is no longer used with the operations like:

- Disable the clock for ENC module.

Parameters

base	ENC peripheral base address.

14.7.3 void ENC_GetDefaultConfig (enc_config_t ∗ *config*)

This function initializes the ENC configuration structure with an available settings, the default value are:

```
*   config->enableReverseDirection                = false;
*   config->decoderWorkMode                       = kENC_DecoderWorkAsNormalMode
      ;
*   config->HOMETriggerMode                       = kENC_HOMETriggerDisabled;
*   config->INDEXTriggerMode                      = kENC_INDEXTriggerDisabled;
*   config->enableTRIGGERClearPositionCounter     = false;
*   config->enableTRIGGERClearHoldPositionCounter = false;
*   config->enableWatchdog                        = false;
*   config->watchdogTimeoutValue                  = 0U;
*   config->filterCount                           = 0U;
```

```
*   config->filterSamplePeriod                    = 0U;
*   config->positionMatchMode                     =
      kENC_POSMATCHOnPositionCounterEqualToComapreValue;
*   config->positionCompareValue                  = 0xFFFFFFFFU;
*   config->revolutionCountCondition              =
      kENC_RevolutionCountOnINDEXPulse;
*   config->enableModuloCountMode                 = false;
*   config->positionModulusValue                  = 0U;
*   config->positionInitialValue                  = 0U;
*
```

Parameters

config	Pointer to a variable of configuration structure. See to "enc_config_t".

14.7.4 void ENC_DoSoftwareLoadInitialPositionValue (ENC_Type ∗ *base*)

This function is to transfer the initial position value (UINIT and LINIT) contents to position counter (UPOS and LPOS), so that to provide the consistent operation the position counter registers.

Parameters

base	ENC peripheral base address.

14.7.5 void ENC_SetSelfTestConfig (ENC_Type ∗ *base*, const enc_self_test_config_t ∗ *config*)

This function is to enable and configuration the self test function. It controls and sets the frequency of a quadrature signal generator. It provides a quadrature test signal to the inputs of the quadrature decoder module. It is a factory test feature; however, it may be useful to customers' software development and testing.

Parameters

base	ENC peripheral base address.
config	Pointer to configuration structure. See to "enc_self_test_config_t". Pass "NULL" to disable.

14.7.6 uint32_t ENC_GetStatusFlags (ENC_Type ∗ *base*)

Parameters

base	ENC peripheral base address.

Returns

Mask value of status flags. For available mask, see to "_enc_status_flags".

14.7.7 void ENC_ClearStatusFlags (ENC_Type * *base*, uint32_t *mask*)

Parameters

base	ENC peripheral base address.
mask	Mask value of status flags to be cleared. For available mask, see to "_enc_status_-flags".

14.7.8 static uint16_t ENC_GetSignalStatusFlags (ENC_Type * *base*) [inline], [static]

Parameters

base	ENC peripheral base address.

Returns

Mask value of signals' real-time status. For available mask, see to "_enc_signal_status_flags"

14.7.9 void ENC_EnableInterrupts (ENC_Type * *base*, uint32_t *mask*)

Parameters

base	ENC peripheral base address.

mask	Mask value of interrupts to be enabled. For available mask, see to "_enc_interrupt_-enable".

14.7.10 void ENC_DisableInterrupts (ENC_Type ∗ *base*, uint32_t *mask*)

Parameters

base	ENC peripheral base address.
mask	Mask value of interrupts to be disabled. For available mask, see to "_enc_interrupt_-enable".

14.7.11 uint32_t ENC_GetEnabledInterrupts (ENC_Type ∗ *base*)

Parameters

base	ENC peripheral base address.

Returns

Mask value of enabled interrupts.

14.7.12 uint32_t ENC_GetPositionValue (ENC_Type ∗ *base*)

Parameters

base	ENC peripheral base address.

Returns

Current position counter's value.

14.7.13 uint32_t ENC_GetHoldPositionValue (ENC_Type ∗ *base*)

When any of the counter registers is read, the contents of each counter register is written to the corresponding hold register. Taking a snapshot of the counters' values provides a consistent view of a system position and a velocity to be attained.

Parameters

base	ENC peripheral base address.

Returns

Hold position counter's value.

14.7.14 static uint16_t ENC_GetPositionDifferenceValue (ENC_Type * *base*) [inline], [static]

Parameters

base	ENC peripheral base address.

Returns

The position difference counter's value.

14.7.15 static uint16_t ENC_GetHoldPositionDifferenceValue (ENC_Type * *base*) [inline], [static]

When any of the counter registers is read, the contents of each counter register is written to the corresponding hold register. Taking a snapshot of the counters' values provides a consistent view of a system position and a velocity to be attained.

Parameters

base	ENC peripheral base address.

Returns

Hold position difference counter's value.

14.7.16 static uint16_t ENC_GetRevolutionValue (ENC_Type * *base*) [inline], [static]

Parameters

base	ENC peripheral base address.

Returns

The position revolution counter's value.

14.7.17 static uint16_t ENC_GetHoldRevolutionValue (ENC_Type * *base*) [inline], [static]

When any of the counter registers is read, the contents of each counter register is written to the corresponding hold register. Taking a snapshot of the counters' values provides a consistent view of a system position and a velocity to be attained.

Parameters

base	ENC peripheral base address.

Returns

Hold position revolution counter's value.

14.8 Variable Documentation

14.8.1 bool enc_config_t::enableReverseDirection

14.8.2 enc_decoder_work_mode_t enc_config_t::decoderWorkMode

14.8.3 enc_home_trigger_mode_t enc_config_t::HOMETriggerMode

14.8.4 enc_index_trigger_mode_t enc_config_t::INDEXTriggerMode

14.8.5 bool enc_config_t::enableTRIGGERClearPositionCounter

14.8.6 bool enc_config_t::enableWatchdog

14.8.7 uint16_t enc_config_t::watchdogTimeoutValue

It stores the timeout count for the quadrature decoder module watchdog timer. This field is only available when "enableWatchdog" = true. The available value is a 16-bit unsigned number.

14.8.8 uint16_t enc_config_t::filterCount

This value should be chosen to reduce the probability of noisy samples causing an incorrect transition to be recognized. The value represent the number of consecutive samples that must agree prior to the input filter accepting an input transition. A value of 0x0 represents 3 samples. A value of 0x7 represents 10 samples. The Available range is 0 - 7.

14.8.9 uint16_t enc_config_t::filterSamplePeriod

This value should be set such that the sampling period is larger than the period of the expected noise. This value represents the sampling period (in IPBus clock cycles) of the decoder input signals. The available range is 0 - 255.

14.8.10 enc_position_match_mode_t enc_config_t::positionMatchMode

14.8.11 uint32_t enc_config_t::positionCompareValue

The available value is a 32-bit number.

14.8.12 enc_revolution_count_condition_t enc_config_t::revolutionCountCondition

14.8.13 bool enc_config_t::enableModuloCountMode

14.8.14 uint32_t enc_config_t::positionModulusValue

This value would be available only when "enableModuloCountMode" = true. The available value is a 32-bit number.

14.8.15 uint32_t enc_config_t::positionInitialValue

The available value is a 32-bit number.

14.8.16 enc_self_test_direction_t enc_self_test_config_t::signalDirection

14.8.17 uint16_t enc_self_test_config_t::signalCount

The available range is 0 - 255.

14.8.18 uint16_t enc_self_test_config_t::signalPeriod

The available range is 0 - 31.

Chapter 15
ENET: Ethernet MAC Driver

15.1 Overview

The KSDK provides a peripheral driver for the 10/100 Mbps Ethernet MAC (ENET) module of Kinetis devices.

The MII interface is the interface connected with MAC and PHY. the Serial management interface - MII management interface should be set firstly before any access to external PHY chip register. So call ENET-_SetSMI() to initialize MII management interface. Use ENET_StartSMIRead(), ENET_StartSMIWrite() and ENET_ReadSMIData() to read/write phy registers. This function group sets up the MII and serial management SMI interface, gets data from the SMI interface, and starts the SMI read and write command. Use ENET_SetMII() to configure the MII before successfully get the data from the external PHY.

This group sets/gets the ENET mac address, setting the multicast group address filter. ENET_Add-MulticastGroup() should be called to add the ENET MAC to multicast group. It is important for 1588 feature to receive the PTP message.

For ENET receive side, ENET_GetRxFrameSize() must be called firstly used to get the received data size, then call ENET_ReadFrame() to get the received data. If the received error happen, call ENET_GetRx-ErrBeforeReadFrame() after ENET_GetRxFrameSize() and before ENET_ReadFrame() to get the detail error informations.

For ENET transmit side, simply call ENET_SendFrame() to send the data out. The transmit data error information only accessible for 1588 enhanced buffer descriptor mode. So when ENET_ENHANCEDB-UFFERDESCRIPTOR_MODE is defined the ENET_GetTxErrAfterSendFrame() can be used to get the detail transmit error information. The transmit error information only be updated by uDMA after the data is transmit. So ENET_GetTxErrAfterSendFrame() is recommended to be called on transmit interrupt handler.

This function group configures the PTP 1588 feature, starts/stops/gets/sets/adjusts the PTP IEEE 1588 timer, gets the receive/transmit frame timestamp, and PTP IEEE 1588 timer channel feature setting.

ENET_Ptp1588Configure() must be called when ENET_ENHANCEDBUFFERDESCRIPTOR_MODE is defined and the 1588 feature is required. The ENET_GetRxFrameTime() and ENET_GetTxFrame-Time() are called by PTP stack to get the timestamp captured by ENET driver.

15.2 Typical use case

15.2.1 ENET Initialization, receive, and transmit operation

For ENET_ENHANCEDBUFFERDESCRIPTOR_MODE not defined use case, use the legacy type buffer descriptor transmit/receive the frame:

```
enet_config_t config;
```

```
uint32_t length = 0;
uint32_t sysClock;
uint32_t phyAddr = 0;
bool link = false;
phy_speed_t speed;
phy_duplex_t duplex;
enet_status_t result;
enet_data_error_stats_t eErrorStatic;
// Prepares the buffer configuration.
enet_buffer_config_t buffCfg =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
    &RxBuffDescrip[0],  // Prepare buffers
    &TxBuffDescrip[0],  // Prepare buffers
    &RxDataBuff[0][0],  // Prepare buffers
    &TxDataBuff[0][0],  // Prepare buffers
};

sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

// Gets the default configuration.
ENET_GetDefaultConfig(&config);
PHY_Init(EXAMPLE_ENET, 0, sysClock);
// Changes the link status to PHY auto-negotiated link status.
PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
if (link)
{
    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
    config.miiSpeed = (enet_mii_speed_t)speed;
    config.miiDuplex = (enet_mii_duplex_t)duplex;
}
ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);
ENET_ActiveRead(EXAMPLE_ENET);

while (1)
{
    // Gets the frame size.
    result = ENET_GetRxFrameSize(&handle, &length);
    // Calls the ENET_ReadFrame when there is a received frame.
    if (length != 0)
    {
        // Receives a valid frame and delivers the receive buffer with the size equal to length.
        uint8_t *data = (uint8_t *)malloc(length);
        ENET_ReadFrame(EXAMPLE_ENET, &handle, data, length);
        // Delivers the data to the upper layer.
        .........
        free(data);
    }
    else if (result == kStatus_ENET_RxFrameErr)
    {
       // Updates the received buffer when an error occurs.
       ENET_GetRxErrBeforeReadFrame(&handle, &eErrStatic);
       // Updates the receive buffer.
       ENET_ReadFrame(EXAMPLE_ENET, &handle, NULL, 0);
    }

   // Sends a multicast frame when the PHY is linked up.
   if(kStatus_Success == PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link))
   {
      if(link)
      {
         ENET_SendFrame(EXAMPLE_ENET, &handle, &frame[0], ENET_DATA_LENGTH);
      }
  }
}
```

For ENET_ENHANCEDBUFFERDESCRIPTOR_MODE defined case, add the PTP IEEE 1588 configuration to enable the PTP IEEE 1588 feature. The initialization occurs as follows:

```
enet_config_t config;
uint32_t length = 0;
uint32_t sysClock;
uint32_t phyAddr = 0;
bool link = false;
phy_speed_t speed;
phy_duplex_t duplex;
enet_status_t result;
enet_data_err_stats_t eErrStatic;
enet_buffer_config_t buffCfg =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
    &RxBuffDescrip[0],
    &TxBuffDescrip[0],
    &RxDataBuff[0][0],
    &TxDataBuff[0][0],
};

sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

// Sets the PTP 1588 source.
CLOCK_SetEnetTime0Clock(2);
ptpClock = CLOCK_GetFreq(kCLOCK_Osc0ErClk);
// Prepares the PTP configuration.
enet_ptp_config_t ptpConfig =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    &g_rxPtpTsBuff[0],
    &g_txPtpTsBuff[0],
    kENET_PtpTimerChannel1,
    ptpClock,
};

// Gets the default configuration.
ENET_GetDefaultConfig(&config);

PHY_Init(EXAMPLE_ENET, 0, sysClock);
// Changes the link status to PHY auto-negotiated link status.
PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
if (link)
{
    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
    config.miiSpeed = (enet_mii_speed_t)speed;
    config.miiDuplex = (enet_mii_duplex_t)duplex;
}

ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);

// Configures the PTP 1588 feature.
ENET_Ptp1588Configure(EXAMPLE_ENET, &handle, &ptpConfig);
// Adds the device to the PTP multicast group.
ENET_AddMulticastGroup(EXAMPLE_ENET, &mGAddr[0]);

ENET_ActiveRead(EXAMPLE_ENET);
```

Files

- file fsl_enet.h

Data Structures

- struct enet_rx_bd_struct_t
  Defines the receive buffer descriptor structure for the little endian system. More...
- struct enet_tx_bd_struct_t
  Defines the enhanced transmit buffer descriptor structure for the little endian system. More...
- struct enet_data_error_stats_t
  Defines the ENET data error statistic structure. More...
- struct enet_buffer_config_t
  Defines the receive buffer descriptor configuration structure. More...
- struct enet_config_t
  Defines the basic configuration structure for the ENET device. More...
- struct enet_handle_t
  Defines the ENET handler structure. More...

Macros

- #define ENET_BUFFDESCRIPTOR_RX_ERR_MASK
  Defines the receive error status flag mask.
- #define ENET_FIFO_MIN_RX_FULL 5U
  ENET minimum receive FIFO full.
- #define ENET_RX_MIN_BUFFERSIZE 256U
  ENET minimum buffer size.
- #define ENET_BUFF_ALIGNMENT 16U
  Ethernet buffer alignment.
- #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)
  Defines the PHY address scope for the ENET.

Typedefs

- typedef void(* enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, void *userData)
  ENET callback function.

Enumerations

- enum _enet_status {
  kStatus_ENET_RxFrameError = MAKE_STATUS(kStatusGroup_ENET, 0U),
  kStatus_ENET_RxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 1U),
  kStatus_ENET_RxFrameEmpty = MAKE_STATUS(kStatusGroup_ENET, 2U),
  kStatus_ENET_TxFrameBusy,
  kStatus_ENET_TxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 4U) }
  Defines the status return codes for transaction.
- enum enet_mii_mode_t {
  kENET_MiiMode = 0U,
  kENET_RmiiMode }
  Defines the RMII or MII mode for data interface between the MAC and the PHY.

- enum enet_mii_speed_t {
  kENET_MiiSpeed10M = 0U,
  kENET_MiiSpeed100M }

  Defines the 10 Mbps or 100 Mbps speed for the MII data interface.
- enum enet_mii_duplex_t {
  kENET_MiiHalfDuplex = 0U,
  kENET_MiiFullDuplex }

  Defines the half or full duplex for the MII data interface.
- enum enet_mii_write_t {
  kENET_MiiWriteNoCompliant = 0U,
  kENET_MiiWriteValidFrame }

  Defines the write operation for the MII management frame.
- enum enet_mii_read_t {
  kENET_MiiReadValidFrame = 2U,
  kENET_MiiReadNoCompliant = 3U }

  Defines the read operation for the MII management frame.
- enum enet_special_control_flag_t {
  kENET_ControlFlowControlEnable = 0x0001U,
  kENET_ControlRxPayloadCheckEnable = 0x0002U,
  kENET_ControlRxPadRemoveEnable = 0x0004U,
  kENET_ControlRxBroadCastRejectEnable = 0x0008U,
  kENET_ControlMacAddrInsert = 0x0010U,
  kENET_ControlStoreAndFwdDisable = 0x0020U,
  kENET_ControlSMIPreambleDisable = 0x0040U,
  kENET_ControlPromiscuousEnable = 0x0080U,
  kENET_ControlMIILoopEnable = 0x0100U,
  kENET_ControlVLANTagEnable = 0x0200U }

  Defines a special configuration for ENET MAC controller.
- enum enet_interrupt_enable_t {
  kENET_BabrInterrupt = ENET_EIR_BABR_MASK,
  kENET_BabtInterrupt = ENET_EIR_BABT_MASK,
  kENET_GraceStopInterrupt = ENET_EIR_GRA_MASK,
  kENET_TxFrameInterrupt = ENET_EIR_TXF_MASK,
  kENET_TxBufferInterrupt = ENET_EIR_TXB_MASK,
  kENET_RxFrameInterrupt = ENET_EIR_RXF_MASK,
  kENET_RxBufferInterrupt = ENET_EIR_RXB_MASK,
  kENET_MiiInterrupt = ENET_EIR_MII_MASK,
  kENET_EBusERInterrupt = ENET_EIR_EBERR_MASK,
  kENET_LateCollisionInterrupt = ENET_EIR_LC_MASK,
  kENET_RetryLimitInterrupt = ENET_EIR_RL_MASK,
  kENET_UnderrunInterrupt = ENET_EIR_UN_MASK,
  kENET_PayloadRxInterrupt = ENET_EIR_PLR_MASK,
  kENET_WakeupInterrupt = ENET_EIR_WAKEUP_MASK }

  List of interrupts supported by the peripheral.
- enum enet_event_t {

kENET_RxEvent,
kENET_TxEvent,
kENET_ErrEvent,
kENET_WakeUpEvent }
Defines the common interrupt event for callback use.
- enum enet_tx_accelerator_t {
kENET_TxAccelIsShift16Enabled = ENET_TACC_SHIFT16_MASK,
kENET_TxAccelIpCheckEnabled = ENET_TACC_IPCHK_MASK,
kENET_TxAccelProtoCheckEnabled = ENET_TACC_PROCHK_MASK }
Defines the transmit accelerator configuration.
- enum enet_rx_accelerator_t {
kENET_RxAccelPadRemoveEnabled = ENET_RACC_PADREM_MASK,
kENET_RxAccelIpCheckEnabled = ENET_RACC_IPDIS_MASK,
kENET_RxAccelProtoCheckEnabled = ENET_RACC_PRODIS_MASK,
kENET_RxAccelMacCheckEnabled = ENET_RACC_LINEDIS_MASK,
kENET_RxAccelisShift16Enabled = ENET_RACC_SHIFT16_MASK }
Defines the receive accelerator configuration.

Driver version

- #define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
Defines the driver version.

Control and status region bit masks of the receive buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U
Empty bit mask.
- #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U
Software owner one mask.
- #define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U
Next buffer descriptor is the start address.
- #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U
Software owner two mask.
- #define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U
Last BD of the frame mask.
- #define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U
Received because of the promiscuous mode.
- #define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U
Broadcast packet mask.
- #define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U
Multicast packet mask.
- #define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK 0x0020U
Length violation mask.
- #define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U
Non-octet aligned frame mask.
- #define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U
CRC error mask.
- #define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U
FIFO overrun mask.

- #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U
  Frame is truncated mask.

Control and status bit masks of the transmit buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U
  Ready bit mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U
  Software owner one mask.
- #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U
  Wrap buffer descriptor mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U
  Software owner two mask.
- #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U
  Last BD of the frame mask.
- #define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U
  Transmit CRC mask.

Defines the maximum Ethernet frame size.

- #define ENET_FRAME_MAX_FRAMELEN 1518U
  Maximum Ethernet frame size.
- #define ENET_FRAME_MAX_VALNFRAMELEN 1522U
  Maximum VLAN frame size.

Initialization and De-initialization

- void ENET_GetDefaultConfig (enet_config_t ∗config)
  Gets the ENET default configuration structure.
- void ENET_Init (ENET_Type ∗base, enet_handle_t ∗handle, const enet_config_t ∗config, const enet_buffer_config_t ∗bufferConfig, uint8_t ∗macAddr, uint32_t srcClock_Hz)
  Initializes the ENET module.
- void ENET_Deinit (ENET_Type ∗base)
  Deinitializes the ENET module.
- static void ENET_Reset (ENET_Type ∗base)
  Resets the ENET module.

MII interface operation

- void ENET_SetMII (ENET_Type ∗base, enet_mii_speed_t speed, enet_mii_duplex_t duplex)
  Sets the ENET MII speed and duplex.
- void ENET_SetSMI (ENET_Type ∗base, uint32_t srcClock_Hz, bool isPreambleDisabled)
  Sets the ENET SMI(serial management interface)- MII management interface.
- static bool ENET_GetSMI (ENET_Type ∗base)
  Gets the ENET SMI- MII management interface configuration.
- static uint32_t ENET_ReadSMIData (ENET_Type ∗base)
  Reads data from the PHY register through SMI interface.
- void ENET_StartSMIRead (ENET_Type ∗base, uint32_t phyAddr, uint32_t phyReg, enet_mii_read_t operation)
  Starts an SMI (Serial Management Interface) read command.

- void ENET_StartSMIWrite (ENET_Type ∗base, uint32_t phyAddr, uint32_t phyReg, enet_mii_write_t operation, uint32_t data)
  Starts a SMI write command.

MAC Address Filter

- void ENET_SetMacAddr (ENET_Type ∗base, uint8_t ∗macAddr)
  Sets the ENET module Mac address.
- void ENET_GetMacAddr (ENET_Type ∗base, uint8_t ∗macAddr)
  Gets the ENET module Mac address.
- void ENET_AddMulticastGroup (ENET_Type ∗base, uint8_t ∗address)
  Adds the ENET device to a multicast group.
- void ENET_LeaveMulticastGroup (ENET_Type ∗base, uint8_t ∗address)
  Moves the ENET device from a multicast group.

Other basic operation

- static void ENET_ActiveRead (ENET_Type ∗base)
  Activates ENET read or receive.
- static void ENET_EnableSleepMode (ENET_Type ∗base, bool enable)
  Enables/disables the MAC to enter sleep mode.
- static void ENET_GetAccelFunction (ENET_Type ∗base, uint32_t ∗txAccelOption, uint32_t ∗rxAccelOption)
  Gets ENET transmit and receive accelerator functions from MAC controller.

Interrupts.

- static void ENET_EnableInterrupts (ENET_Type ∗base, uint32_t mask)
  Enables the ENET interrupt.
- static void ENET_DisableInterrupts (ENET_Type ∗base, uint32_t mask)
  Disables the ENET interrupt.
- static uint32_t ENET_GetInterruptStatus (ENET_Type ∗base)
  Gets the ENET interrupt status flag.
- static void ENET_ClearInterruptStatus (ENET_Type ∗base, uint32_t mask)
  Clears the ENET interrupt events status flag.

Transactional operation

- void ENET_SetCallback (enet_handle_t ∗handle, enet_callback_t callback, void ∗userData)
  Set the callback function.
- void ENET_GetRxErrBeforeReadFrame (enet_handle_t ∗handle, enet_data_error_stats_t ∗eErrorStatic)
  Gets the ENET the error statistics of a received frame.
- status_t ENET_GetRxFrameSize (enet_handle_t ∗handle, uint32_t ∗length)
  Gets the size of the read frame.
- status_t ENET_ReadFrame (ENET_Type ∗base, enet_handle_t ∗handle, uint8_t ∗data, uint32_t length)
  Reads a frame from the ENET device.
- status_t ENET_SendFrame (ENET_Type ∗base, enet_handle_t ∗handle, uint8_t ∗data, uint32_t length)

Transmits an ENET frame.
- void ENET_TransmitIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The transmit IRQ handler.
- void ENET_ReceiveIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The receive IRQ handler.
- void ENET_ErrorIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The error IRQ handler.

15.3 Data Structure Documentation

15.3.1 struct enet_rx_bd_struct_t

Data Fields

- uint16_t length
  Buffer descriptor data length.
- uint16_t control
  Buffer descriptor control and status.
- uint8_t ∗ buffer
  Data buffer pointer.

15.3.1.0.0.32 Field Documentation

15.3.1.0.0.32.1 uint16_t enet_rx_bd_struct_t::length

15.3.1.0.0.32.2 uint16_t enet_rx_bd_struct_t::control

15.3.1.0.0.32.3 uint8_t∗ enet_rx_bd_struct_t::buffer

15.3.2 struct enet_tx_bd_struct_t

Data Fields

- uint16_t length
  Buffer descriptor data length.
- uint16_t control
  Buffer descriptor control and status.
- uint8_t ∗ buffer
  Data buffer pointer.

15.3.2.0.0.33 Field Documentation

15.3.2.0.0.33.1 uint16_t enet_tx_bd_struct_t::length

15.3.2.0.0.33.2 uint16_t enet_tx_bd_struct_t::control

15.3.2.0.0.33.3 uint8_t* enet_tx_bd_struct_t::buffer

15.3.3 struct enet_data_error_stats_t

Data Fields

- uint32_t statsRxLenGreaterErr
  Receive length greater than RCR[MAX_FL].
- uint32_t statsRxAlignErr
  Receive non-octet alignment/.
- uint32_t statsRxFcsErr
  Receive CRC error.
- uint32_t statsRxOverRunErr
  Receive over run.
- uint32_t statsRxTruncateErr
  Receive truncate.

15.3.3.0.0.34 Field Documentation

15.3.3.0.0.34.1 uint32_t enet_data_error_stats_t::statsRxLenGreaterErr

15.3.3.0.0.34.2 uint32_t enet_data_error_stats_t::statsRxFcsErr

15.3.3.0.0.34.3 uint32_t enet_data_error_stats_t::statsRxOverRunErr

15.3.3.0.0.34.4 uint32_t enet_data_error_stats_t::statsRxTruncateErr

15.3.4 struct enet_buffer_config_t

Note: For the internal DMA requirements, the buffers have a corresponding alignment requirement:

1. The aligned receive and transmit buffer size must be evenly divisible by 16.
2. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by 16.
3. The aligned transmit and receive buffer start address must be evenly divisible by 16. Receive buffers should be continuous with the total size equal to "rxBdNumber * rxBuffSizeAlign". Transmit buffers should be continuous with the total size equal to "txBdNumber * txBuffSizeAlign".

Data Fields

- uint16_t rxBdNumber
  Receive buffer descriptor number.

- uint16_t txBdNumber
  Transmit buffer descriptor number.
- uint32_t rxBuffSizeAlign
  Aligned receive data buffer size.
- uint32_t txBuffSizeAlign
  Aligned transmit data buffer size.
- volatile enet_rx_bd_struct_t * rxBdStartAddrAlign
  Aligned receive buffer descriptor start address.
- volatile enet_tx_bd_struct_t * txBdStartAddrAlign
  Aligned transmit buffer descriptor start address.
- uint8_t * rxBufferAlign
  Receive data buffer start address.
- uint8_t * txBufferAlign
  Transmit data buffer start address.

15.3.4.0.0.35 Field Documentation

15.3.4.0.0.35.1 uint16_t enet_buffer_config_t::rxBdNumber

15.3.4.0.0.35.2 uint16_t enet_buffer_config_t::txBdNumber

15.3.4.0.0.35.3 uint32_t enet_buffer_config_t::rxBuffSizeAlign

15.3.4.0.0.35.4 uint32_t enet_buffer_config_t::txBuffSizeAlign

15.3.4.0.0.35.5 volatile enet_rx_bd_struct_t* enet_buffer_config_t::rxBdStartAddrAlign

15.3.4.0.0.35.6 volatile enet_tx_bd_struct_t* enet_buffer_config_t::txBdStartAddrAlign

15.3.4.0.0.35.7 uint8_t* enet_buffer_config_t::rxBufferAlign

15.3.4.0.0.35.8 uint8_t* enet_buffer_config_t::txBufferAlign

15.3.5 struct enet_config_t

Note:

1. macSpecialConfig is used for a special control configuration, A logical OR of "enet_special_control-_flag_t". For a special configuration for MAC, set this parameter to 0.
2. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO .... 3 - 192 bytes written to TX FIFO .... The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO .... txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
3. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULL and the maximum is 0xFF. If the end of the frame is stored in FIFO and the frame size if smaller than the txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.
4. When "kENET_ControlFlowControlEnable" is set in the macSpecialConfig, ensure that the pause-

Duration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.

5. When "kENET_ControlStoreAndFwdDisabled" is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
6. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The "enet_tx_accelerator_t" and "enet_rx_accelerator_t" are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.

Data Fields

- uint32_t macSpecialConfig
  Mac special configuration.
- uint32_t interrupt
  Mac interrupt source.
- uint16_t rxMaxFrameLen
  Receive maximum frame length.
- enet_mii_mode_t miiMode
  MII mode.
- enet_mii_speed_t miiSpeed
  MII Speed.
- enet_mii_duplex_t miiDuplex
  MII duplex.
- uint8_t rxAccelerConfig
  Receive accelerator, A logical OR of "enet_rx_accelerator_t".
- uint8_t txAccelerConfig
  Transmit accelerator, A logical OR of "enet_rx_accelerator_t".
- uint16_t pauseDuration
  For flow control enabled case: Pause duration.
- uint8_t rxFifoEmptyThreshold
  For flow control enabled case: when RX FIFO level reaches this value, it makes MAC generate XOFF pause frame.
- uint8_t rxFifoFullThreshold
  `For store and forward disable case, the data required in RX FIFO to notify` *the MAC receive ready status.*
- uint8_t txFifoWatermark
  `For store and forward disable case, the data required in TX FIFO` *before a frame transmit start.*

15.3.5.0.0.36 Field Documentation

15.3.5.0.0.36.1 uint32_t enet_config_t::macSpecialConfig

A logical OR of "enet_special_control_flag_t".

15.3.5.0.0.36.2 uint32_t enet_config_t::interrupt

A logical OR of "enet_interrupt_enable_t".

15.3.5.0.0.36.3 uint16_t enet_config_t::rxMaxFrameLen

15.3.5.0.0.36.4 enet_mii_mode_t enet_config_t::miiMode

15.3.5.0.0.36.5 enet_mii_speed_t enet_config_t::miiSpeed

15.3.5.0.0.36.6 enet_mii_duplex_t enet_config_t::miiDuplex

15.3.5.0.0.36.7 uint8_t enet_config_t::rxAccelerConfig

15.3.5.0.0.36.8 uint8_t enet_config_t::txAccelerConfig

15.3.5.0.0.36.9 uint16_t enet_config_t::pauseDuration

15.3.5.0.0.36.10 uint8_t enet_config_t::rxFifoEmptyThreshold

15.3.5.0.0.36.11 uint8_t enet_config_t::rxFifoFullThreshold

15.3.5.0.0.36.12 uint8_t enet_config_t::txFifoWatermark

15.3.6 struct _enet_handle

Data Fields

- volatile enet_rx_bd_struct_t ∗ rxBdBase
  Receive buffer descriptor base address pointer.
- volatile enet_rx_bd_struct_t ∗ rxBdCurrent
  The current available receive buffer descriptor pointer.
- volatile enet_tx_bd_struct_t ∗ txBdBase
  Transmit buffer descriptor base address pointer.
- volatile enet_tx_bd_struct_t ∗ txBdCurrent
  The current available transmit buffer descriptor pointer.
- uint32_t rxBuffSizeAlign
  Receive buffer size alignment.
- uint32_t txBuffSizeAlign
  Transmit buffer size alignment.
- enet_callback_t callback
  Callback function.
- void ∗ userData
  Callback function parameter.

15.3.6.0.0.37 Field Documentation

15.3.6.0.0.37.1 volatile enet_rx_bd_struct_t* enet_handle_t::rxBdBase

15.3.6.0.0.37.2 volatile enet_rx_bd_struct_t* enet_handle_t::rxBdCurrent

15.3.6.0.0.37.3 volatile enet_tx_bd_struct_t* enet_handle_t::txBdBase

15.3.6.0.0.37.4 volatile enet_tx_bd_struct_t* enet_handle_t::txBdCurrent

15.3.6.0.0.37.5 uint32_t enet_handle_t::rxBuffSizeAlign

15.3.6.0.0.37.6 uint32_t enet_handle_t::txBuffSizeAlign

15.3.6.0.0.37.7 enet_callback_t enet_handle_t::callback

15.3.6.0.0.37.8 void* enet_handle_t::userData

15.4 Macro Definition Documentation

15.4.1 #define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Version 2.0.1.

15.4.2 #define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U

15.4.3 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U

15.4.4 #define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U

15.4.5 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U

15.4.6 #define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U

15.4.7 #define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U

15.4.8 #define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U

15.4.9 #define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U

15.4.10 #define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK 0x0020U

15.4.11 #define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U

15.4.12 #define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U

15.4.13 #define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U

15.4.14 #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U

15.4.15 #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U

15.4.16 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U

15.4.17 #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U

15.4.18 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U

15.4.19 #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U

15.4.20 #define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U

15.4.21 #define ENET_BUFFDESCRIPTOR_RX_ERR_MASK

Value:

```
(ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK |
      ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK | \
     ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK |
      ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK |
      ENET_BUFFDESCRIPTOR_RX_CRC_MASK)
```

15.4.22 #define ENET_FRAME_MAX_FRAMELEN 1518U

15.4.23 #define ENET_FRAME_MAX_VALNFRAMELEN 1522U

15.4.24 #define ENET_FIFO_MIN_RX_FULL 5U

15.4.25 #define ENET_RX_MIN_BUFFERSIZE 256U

15.4.26 #define ENET_BUFF_ALIGNMENT 16U

15.4.27 #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)

15.5 Typedef Documentation

15.5.1 typedef void(∗ enet_callback_t)(ENET_Type ∗base, enet_handle_t ∗handle, enet_event_t event, void ∗userData)

15.6 Enumeration Type Documentation

15.6.1 enum _enet_status

Enumerator

kStatus_ENET_RxFrameError A frame received but data error happen.
kStatus_ENET_RxFrameFail Failed to receive a frame.
kStatus_ENET_RxFrameEmpty No frame arrive.
kStatus_ENET_TxFrameBusy Transmit buffer descriptors are under process.
kStatus_ENET_TxFrameFail Transmit frame fail.

15.6.2 enum enet_mii_mode_t

Enumerator

kENET_MiiMode MII mode for data interface.
kENET_RmiiMode RMII mode for data interface.

15.6.3 enum enet_mii_speed_t

Enumerator

kENET_MiiSpeed10M Speed 10 Mbps.
kENET_MiiSpeed100M Speed 100 Mbps.

15.6.4 enum enet_mii_duplex_t

Enumerator

kENET_MiiHalfDuplex Half duplex mode.
kENET_MiiFullDuplex Full duplex mode.

15.6.5 enum enet_mii_write_t

Enumerator

kENET_MiiWriteNoCompliant Write frame operation, but not MII-compliant.
kENET_MiiWriteValidFrame Write frame operation for a valid MII management frame.

15.6.6 enum enet_mii_read_t

Enumerator

kENET_MiiReadValidFrame Read frame operation for a valid MII management frame.
kENET_MiiReadNoCompliant Read frame operation, but not MII-compliant.

15.6.7 enum enet_special_control_flag_t

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to macSpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifoFullThreshold and txFifoWatermark in the enet_config_t.

Enumerator

kENET_ControlFlowControlEnable Enable ENET flow control: pause frame.
kENET_ControlRxPayloadCheckEnable Enable ENET receive payload length check.
kENET_ControlRxPadRemoveEnable Padding is removed from received frames.

kENET_ControlRxBroadCastRejectEnable Enable broadcast frame reject.
kENET_ControlMacAddrInsert Enable MAC address insert.
kENET_ControlStoreAndFwdDisable Enable FIFO store and forward.
kENET_ControlSMIPreambleDisable Enable SMI preamble.
kENET_ControlPromiscuousEnable Enable promiscuous mode.
kENET_ControlMIILoopEnable Enable ENET MII loop back.
kENET_ControlVLANTagEnable Enable VLAN tag frame.

15.6.8 enum enet_interrupt_enable_t

This enumeration uses one-bot encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Enumerator

kENET_BabrInterrupt Babbling receive error interrupt source.
kENET_BabtInterrupt Babbling transmit error interrupt source.
kENET_GraceStopInterrupt Graceful stop complete interrupt source.
kENET_TxFrameInterrupt TX FRAME interrupt source.
kENET_TxBufferInterrupt TX BUFFER interrupt source.
kENET_RxFrameInterrupt RX FRAME interrupt source.
kENET_RxBufferInterrupt RX BUFFER interrupt source.
kENET_MiiInterrupt MII interrupt source.
kENET_EBusERInterrupt Ethernet bus error interrupt source.
kENET_LateCollisionInterrupt Late collision interrupt source.
kENET_RetryLimitInterrupt Collision Retry Limit interrupt source.
kENET_UnderrunInterrupt Transmit FIFO underrun interrupt source.
kENET_PayloadRxInterrupt Payload Receive interrupt source.
kENET_WakeupInterrupt WAKEUP interrupt source.

15.6.9 enum enet_event_t

Enumerator

kENET_RxEvent Receive event.
kENET_TxEvent Transmit event.
kENET_ErrEvent Error event: BABR/BABT/EBERR/LC/RL/UN/PLR .
kENET_WakeUpEvent Wake up from sleep mode event.

15.6.10 enum enet_tx_accelerator_t

Enumerator

kENET_TxAccelIsShift16Enabled Transmit FIFO shift-16.
kENET_TxAccelIpCheckEnabled Insert IP header checksum.
kENET_TxAccelProtoCheckEnabled Insert protocol checksum.

15.6.11 enum enet_rx_accelerator_t

Enumerator

kENET_RxAccelPadRemoveEnabled Padding removal for short IP frames.
kENET_RxAccelIpCheckEnabled Discard with wrong IP header checksum.
kENET_RxAccelProtoCheckEnabled Discard with wrong protocol checksum.
kENET_RxAccelMacCheckEnabled Discard with Mac layer errors.
kENET_RxAccelisShift16Enabled Receive FIFO shift-16.

15.7 Function Documentation

15.7.1 void ENET_GetDefaultConfig (enet_config_t ∗ *config*)

The purpose of this API is to get the default ENET MAC controller configuration structure for ENET-_Init(). User may use the initialized structure unchanged in ENET_Init(), or modify some fields of the structure before calling ENET_Init(). Example:

```
enet_config_t config;
ENET_GetDefaultConfig(&config);
```

Parameters

config	The ENET mac controller configuration structure pointer.

15.7.2 void ENET_Init (ENET_Type ∗ *base*, enet_handle_t ∗ *handle*, const enet_config_t ∗ *config*, const enet_buffer_config_t ∗ *bufferConfig*, uint8_t ∗ *macAddr*, uint32_t *srcClock_Hz*)

This function ungates the module clock and initializes it with the ENET configuration.

Parameters

base	ENET peripheral base address.
handle	ENET handler pointer.
config	ENET mac configuration structure pointer. The "enet_config_t" type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
bufferConfig	ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization.
macAddr	ENET mac address of Ethernet device. This MAC address should be provided.
srcClock_Hz	The internal module clock source for MII clock.

Note

ENET has two buffer descriptors: legacy buffer descriptors and enhanced 1588 buffer descriptors. The legacy descriptor is used by default. To use 1588 feature, use the enhanced 1588 buffer descriptor by defining "ENET_ENHANCEDBUFFERDESCRIPTOR_MODE" and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Init().

15.7.3 void ENET_Deinit (ENET_Type ∗ *base*)

This function gates the module clock, clears ENET interrupts, and disables the ENET module.

Parameters

base	ENET peripheral base address.

15.7.4 static void ENET_Reset (ENET_Type ∗ *base*) `[inline]`, `[static]`

This function restores the ENET module to reset state. Note that this function sets all registers to reset state. As a result, the ENET module can't work after calling this function.

Parameters

base	ENET peripheral base address.

15.7.5 void ENET_SetMII (ENET_Type ∗ *base*, enet_mii_speed_t *speed*, enet_mii_duplex_t *duplex*)

Parameters

base	ENET peripheral base address.
speed	The speed of the RMII mode.
duplex	The duplex of the RMII mode.

15.7.6 void ENET_SetSMI (ENET_Type ∗ *base,* uint32_t *srcClock_Hz,* bool *isPreambleDisabled*)

Parameters

base	ENET peripheral base address.
srcClock_Hz	This is the ENET module clock frequency. Normally it's the system clock. See clock distribution.
isPreamble-Disabled	The preamble disable flag. • true Enables the preamble. • false Disables the preamble.

15.7.7 static bool ENET_GetSMI (ENET_Type ∗ *base*) [inline], [static]

This API is used to get the SMI configuration to check if the MII management interface has been set.

Parameters

base	ENET peripheral base address.

Returns

The SMI setup status true or false.

15.7.8 static uint32_t ENET_ReadSMIData (ENET_Type ∗ *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The data read from PHY

15.7.9 void ENET_StartSMIRead (ENET_Type ∗ *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_read_t *operation*)

Parameters

base	ENET peripheral base address.
phyAddr	The PHY address.
phyReg	The PHY register.
operation	The read operation.

15.7.10 void ENET_StartSMIWrite (ENET_Type ∗ *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_write_t *operation*, uint32_t *data*)

Parameters

base	ENET peripheral base address.
phyAddr	The PHY address.
phyReg	The PHY register.
operation	The write operation.
data	The data written to PHY.

15.7.11 void ENET_SetMacAddr (ENET_Type ∗ *base*, uint8_t ∗ *macAddr*)

Parameters

base	ENET peripheral base address.
macAddr	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

15.7.12 void ENET_GetMacAddr (ENET_Type ∗ *base*, uint8_t ∗ *macAddr*)

Parameters

base	ENET peripheral base address.
macAddr	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

15.7.13 void ENET_AddMulticastGroup (ENET_Type ∗ *base*, uint8_t ∗ *address*)

Parameters

base	ENET peripheral base address.
address	The six-byte multicast group address which is provided by application.

15.7.14 void ENET_LeaveMulticastGroup (ENET_Type ∗ *base*, uint8_t ∗ *address*)

Parameters

base	ENET peripheral base address.
address	The six-byte multicast group address which is provided by application.

15.7.15 static void ENET_ActiveRead (ENET_Type ∗ *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Note

This must be called after the MAC configuration and state are ready. It must be called after the ENET_Init() and ENET_Ptp1588Configure(). This should be called when the ENET receive required.

15.7.16 static void ENET_EnableSleepMode (ENET_Type ∗ *base,* bool *enable*) [inline], [static]

This function is used to set the MAC enter sleep mode. When entering sleep mode, the magic frame wakeup interrupt should be enabled to wake up MAC from the sleep mode and reset it to normal mode.

Parameters

base	ENET peripheral base address.
enable	True enable sleep mode, false disable sleep mode.

15.7.17 static void ENET_GetAccelFunction (ENET_Type ∗ *base,* uint32_t ∗ *txAccelOption,* uint32_t ∗ *rxAccelOption*) [inline], [static]

Parameters

base	ENET peripheral base address.
txAccelOption	The transmit accelerator option. The "enet_tx_accelerator_t" is recommended to be used to as the mask to get the exact the accelerator option.
rxAccelOption	The receive accelerator option. The "enet_rx_accelerator_t" is recommended to be used to as the mask to get the exact the accelerator option.

15.7.18 static void ENET_EnableInterrupts (ENET_Type ∗ *base,* uint32_t *mask*) [inline], [static]

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to enable the TX frame interrupt and RX frame interrupt, do this:

```
*     ENET_EnableInterrupts(ENET, kENET_TxFrameInterrupt |
      kENET_RxFrameInterrupt);
*
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to enable. This is a logical OR of the enumeration :: enet_interrupt_enable_t.

15.7.19 static void ENET_DisableInterrupts (ENET_Type ∗ *base,* uint32_t *mask*) [inline], [static]

This function disables the ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to disable the TX frame interrupt and RX frame interrupt, do this:

```
*     ENET_DisableInterrupts(ENET, kENET_TxFrameInterrupt |
      kENET_RxFrameInterrupt);
*
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to disable. This is a logical OR of the enumeration :: enet_interrupt_enable_t.

15.7.20 static uint32_t ENET_GetInterruptStatus (ENET_Type ∗ *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: enet_interrupt_enable_t.

15.7.21 static void ENET_ClearInterruptStatus (ENET_Type ∗ *base,* uint32_t *mask*) [inline], [static]

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the enet_interrupt_enable_t. For example, to clear the TX frame interrupt and RX frame interrupt, do this:

```
*     ENET_ClearInterruptStatus(ENET,
      kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
*
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupt source to be cleared. This is the logical OR of members of the enumeration :: enet_interrupt_enable_t.

15.7.22 void ENET_SetCallback (enet_handle_t * *handle,* enet_callback_t *callback,* void * *userData*)

This API is provided for application callback required case when ENET interrupt is enabled. This API should be called after calling ENET_Init.

Parameters

handle	ENET handler pointer. Should be provided by application.
callback	The ENET callback function.
userData	The callback function parameter.

15.7.23 void ENET_GetRxErrBeforeReadFrame (enet_handle_t * *handle,* enet_data_error_stats_t * *eErrorStatic*)

This API must be called after the ENET_GetRxFrameSize and before the ENET_ReadFrame(). If the ENET_GetRxFrameSize returns kStatus_ENET_RxFrameError, the ENET_GetRxErrBeforeReadFrame can be used to get the exact error statistics. For example:

```
*       status = ENET_GetRxFrameSize(&g_handle, &length);
*       if (status == kStatus_ENET_RxFrameError)
*       {
*           // Get the error information of the received frame.
*           ENET_GetRxErrBeforeReadFrame(&g_handle, &eErrStatic);
*           // update the receive buffer.
*           ENET_ReadFrame(EXAMPLE_ENET, &g_handle, NULL, 0);
*       }
*
```

Parameters

handle	The ENET handler structure pointer. This is the same handler pointer used in the ENET_Init.
eErrorStatic	The error statistics structure pointer.

15.7.24 status_t ENET_GetRxFrameSize (enet_handle_t * *handle*, uint32_t * *length*)

This function gets a received frame size from the ENET buffer descriptors.

Note

The FCS of the frame is automatically removed by MAC and the size is the length without the FCS. After calling ENET_GetRxFrameSize, ENET_ReadFrame() should be called to update the receive buffers If the result is not "kStatus_ENET_RxFrameEmpty".

Parameters

handle	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
length	The length of the valid frame received.

Return values

kStatus_ENET_RxFrame-Empty	No frame received. Should not call ENET_ReadFrame to read frame.
kStatus_ENET_RxFrame-Error	Data error happens. ENET_ReadFrame should be called with NULL data and NULL length to update the receive buffers.
kStatus_Success	Receive a frame Successfully then the ENET_ReadFrame should be called with the right data buffer and the captured data length input.

15.7.25 status_t ENET_ReadFrame (ENET_Type * *base*, enet_handle_t * *handle*, uint8_t * *data*, uint32_t *length*)

This function reads a frame (both the data and the length) from the ENET buffer descriptors. The ENET-_GetRxFrameSize should be used to get the size of the prepared data buffer. For example:

```
*        uint32_t length;
*        enet_handle_t g_handle;
*        //Get the received frame size firstly.
*        status = ENET_GetRxFrameSize(&g_handle, &length);
```

```
*        if (length != 0)
*        {
*            //Allocate memory here with the size of "length"
*            uint8_t *data = memory allocate interface;
*            if (!data)
*            {
*                ENET_ReadFrame(ENET, &g_handle, NULL, 0);
*                //Add the console warning log.
*            }
*            else
*            {
*               status = ENET_ReadFrame(ENET, &g_handle, data, length);
*               //Call stack input API to deliver the data to stack
*            }
*        }
*        else if (status == kStatus_ENET_RxFrameError)
*        {
*           //Update the received buffer when a error frame is received.
*            ENET_ReadFrame(ENET, &g_handle, NULL, 0);
*        }
*
```

Parameters

base	ENET peripheral base address.
handle	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to store the frame which memory size should be at least "length".
length	The size of the data buffer which is still the length of the received frame.

Returns

The execute status, successful or failure.

15.7.26 status_t ENET_SendFrame (ENET_Type ∗ *base*, enet_handle_t ∗ *handle*, uint8_t ∗ *data*, uint32_t *length*)

Note

The CRC is automatically appended to the data. Input the data to send without the CRC.

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to be send.
length	The length of the data to be send.

Return values

kStatus_Success	Send frame succeed.
kStatus_ENET_TxFrame-Busy	Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added after each call return with kStatus-_ENET_TxFrameBusy.

15.7.27 void ENET_TransmitIRQHandler (ENET_Type ∗ *base*, enet_handle_t ∗ *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

15.7.28 void ENET_ReceiveIRQHandler (ENET_Type ∗ *base*, enet_handle_t ∗ *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

15.7.29 void ENET_ErrorIRQHandler (ENET_Type ∗ *base*, enet_handle_t ∗ *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

Chapter 16
EWM: External Watchdog Monitor Driver

16.1 Overview

The KSDK provides a peripheral driver for the EWM module of Kinetis devices.

16.2 Typical use case

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.enableInterrupt = true;
config.compareLowValue = 0U;
config.compareHighValue = 0xAAU;
NVIC_EnableIRQ(WDOG_EWM_IRQn);
EWM_Init(base, &config);
```

Files

- file fsl_ewm.h

Data Structures

- struct ewm_config_t
  Describes EWM clock source. More...

Enumerations

- enum _ewm_interrupt_enable_t { kEWM_InterruptEnable = EWM_CTRL_INTEN_MASK }
  EWM interrupt configuration structure, default settings all disabled.
- enum _ewm_status_flags_t { kEWM_RunningFlag = EWM_CTRL_EWMEN_MASK }
  EWM status flags.

Driver version

- #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  EWM driver version 2.0.1.

EWM Initialization and De-initialization

- void EWM_Init (EWM_Type ∗base, const ewm_config_t ∗config)
  Initializes the EWM peripheral.
- void EWM_Deinit (EWM_Type ∗base)
  Deinitializes the EWM peripheral.
- void EWM_GetDefaultConfig (ewm_config_t ∗config)
  Initializes the EWM configuration structure.

EWM functional Operation

- static void EWM_EnableInterrupts (EWM_Type ∗base, uint32_t mask)
  Enables the EWM interrupt.
- static void EWM_DisableInterrupts (EWM_Type ∗base, uint32_t mask)
  Disables the EWM interrupt.
- static uint32_t EWM_GetStatusFlags (EWM_Type ∗base)
  Gets EWM all status flags.
- void EWM_Refresh (EWM_Type ∗base)
  Services the EWM.

16.3 Data Structure Documentation

16.3.1 struct ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

Data Fields

- bool enableEwm
  Enable EWM module.
- bool enableEwmInput
  Enable EWM_in input.
- bool setInputAssertLogic
  EWM_in signal assertion state.
- bool enableInterrupt
  Enable EWM interrupt.
- uint8_t compareLowValue
  Compare low-register value.
- uint8_t compareHighValue
  Compare high-register value.

16.4 Macro Definition Documentation

16.4.1 #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

16.5 Enumeration Type Documentation

16.5.1 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM_InterruptEnable Enable EWM to generate an interrupt.

16.5.2 enum _ewm_status_flags_t

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

Enumerator

kEWM_RunningFlag Running flag, set when EWM is enabled.

16.6 Function Documentation

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

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that except for 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.

Example:

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

Parameters

base	EWM peripheral base address
config	The configuration of EWM

16.6.2 void EWM_Deinit (EWM_Type ∗ *base*)

This function is used to shut down the EWM.

Parameters

base	EWM peripheral base address

16.6.3 void EWM_GetDefaultConfig (ewm_config_t ∗ *config*)

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

```
*   ewmConfig->enableEwm = true;
*   ewmConfig->enableEwmInput = false;
*   ewmConfig->setInputAssertLogic = false;
*   ewmConfig->enableInterrupt = false;
```

```
*    ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
*    ewmConfig->prescaler = 0;
*    ewmConfig->compareLowValue = 0;
*    ewmConfig->compareHighValue = 0xFEU;
*
```

Parameters

config	Pointer to EWM configuration structure.

See Also

ewm_config_t

16.6.4 static void EWM_EnableInterrupts (EWM_Type ∗ *base,* uint32_t *mask*) [inline], [static]

This function enables the EWM interrupt.

Parameters

base	EWM peripheral base address
mask	The interrupts to enable The parameter can be combination of the following source if defined: • kEWM_InterruptEnable

16.6.5 static void EWM_DisableInterrupts (EWM_Type ∗ *base,* uint32_t *mask*) [inline], [static]

This function enables the EWM interrupt.

Parameters

base	EWM peripheral base address
mask	The interrupts to disable The parameter can be combination of the following source if defined: • kEWM_InterruptEnable

16.6.6 static uint32_t EWM_GetStatusFlags (EWM_Type * *base*) [inline], [static]

This function gets all status flags.

Example for getting Running Flag:

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

Parameters

base	EWM peripheral base address

Returns

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

See Also

_ewm_status_flags_t

- true: a related status flag has been set.
- false: a related status flag is not set.

16.6.7 void EWM_Refresh (EWM_Type * *base*)

This function reset EWM counter to zero.

Parameters

base	EWM peripheral base address

Chapter 17
C90TFS Flash Driver

17.1 Overview

The flash provides the C90TFS Flash driver of Kinetis devices with the C90TFS Flash module inside. The flash driver provides general APIs to handle specific operations on C90TFS/FTFx Flash module. The user can use those APIs directly in the application. In addition, it provides internal functions called by the driver. Although these functions are not meant to be called from the user's application directly, the APIs can still be used.

Data Structures

- struct flash_execute_in_ram_function_config_t
  Flash execute-in-RAM function information. More...
- struct flash_swap_state_config_t
  Flash Swap information. More...
- struct flash_swap_ifr_field_config_t
  Flash Swap IFR fields. More...
- union flash_swap_ifr_field_data_t
  Flash Swap IFR field data. More...
- struct flash_operation_config_t
  Active flash information for the current operation. More...
- struct flash_config_t
  Flash driver state information. More...

Typedefs

- typedef void(∗ flash_callback_t)(void)
  A callback type used for the Pflash block.

Enumerations

- enum flash_margin_value_t {
  kFLASH_MarginValueNormal,
  kFLASH_MarginValueUser,
  kFLASH_MarginValueFactory,
  kFLASH_MarginValueInvalid }
  Enumeration for supported flash margin levels.
- enum flash_security_state_t {
  kFLASH_SecurityStateNotSecure,
  kFLASH_SecurityStateBackdoorEnabled,
  kFLASH_SecurityStateBackdoorDisabled }
  Enumeration for the three possible flash security states.

- enum flash_protection_state_t {
kFLASH_ProtectionStateUnprotected,
kFLASH_ProtectionStateProtected,
kFLASH_ProtectionStateMixed }
Enumeration for the three possible flash protection levels.
- enum flash_execute_only_access_state_t {
kFLASH_AccessStateUnLimited,
kFLASH_AccessStateExecuteOnly,
kFLASH_AccessStateMixed }
Enumeration for the three possible flash execute access levels.
- enum flash_property_tag_t {
kFLASH_PropertyPflashSectorSize = 0x00U,
kFLASH_PropertyPflashTotalSize = 0x01U,
kFLASH_PropertyPflashBlockSize = 0x02U,
kFLASH_PropertyPflashBlockCount = 0x03U,
kFLASH_PropertyPflashBlockBaseAddr = 0x04U,
kFLASH_PropertyPflashFacSupport = 0x05U,
kFLASH_PropertyPflashAccessSegmentSize = 0x06U,
kFLASH_PropertyPflashAccessSegmentCount = 0x07U,
kFLASH_PropertyFlexRamBlockBaseAddr = 0x08U,
kFLASH_PropertyFlexRamTotalSize = 0x09U,
kFLASH_PropertyDflashSectorSize = 0x10U,
kFLASH_PropertyDflashTotalSize = 0x11U,
kFLASH_PropertyDflashBlockSize = 0x12U,
kFLASH_PropertyDflashBlockCount = 0x13U,
kFLASH_PropertyDflashBlockBaseAddr = 0x14U }
Enumeration for various flash properties.
- enum _flash_execute_in_ram_function_constants {
kFLASH_ExecuteInRamFunctionMaxSizeInWords = 16U,
kFLASH_ExecuteInRamFunctionTotalNum = 2U }
Constants for execute-in-RAM flash function.
- enum flash_read_resource_option_t {
kFLASH_ResourceOptionFlashIfr,
kFLASH_ResourceOptionVersionId = 0x01U }
Enumeration for the two possible options of flash read resource command.
- enum _flash_read_resource_range {
kFLASH_ResourceRangePflashIfrSizeInBytes = 256U,
kFLASH_ResourceRangeVersionIdSizeInBytes = 8U,
kFLASH_ResourceRangeVersionIdStart = 0x00U,
kFLASH_ResourceRangeVersionIdEnd = 0x07U ,
kFLASH_ResourceRangePflashSwapIfrEnd,
kFLASH_ResourceRangeDflashIfrStart = 0x800000U,
kFLASH_ResourceRangeDflashIfrEnd = 0x8003FFU }
Enumeration for the range of special-purpose flash resource.
- enum flash_flexram_function_option_t {
kFLASH_FlexramFunctionOptionAvailableAsRam = 0xFFU,

kFLASH_FlexramFunctionOptionAvailableForEeprom = 0x00U }
Enumeration for the two possilbe options of set FlexRAM function command.
- enum _flash_acceleration_ram_property
Enumeration for acceleration RAM property.
- enum flash_swap_function_option_t {
kFLASH_SwapFunctionOptionEnable = 0x00U,
kFLASH_SwapFunctionOptionDisable = 0x01U }
Enumeration for the possible options of Swap function.
- enum flash_swap_control_option_t {
kFLASH_SwapControlOptionIntializeSystem = 0x01U,
kFLASH_SwapControlOptionSetInUpdateState = 0x02U,
kFLASH_SwapControlOptionSetInCompleteState = 0x04U,
kFLASH_SwapControlOptionReportStatus = 0x08U,
kFLASH_SwapControlOptionDisableSystem = 0x10U }
Enumeration for the possible options of Swap control commands.
- enum flash_swap_state_t {
kFLASH_SwapStateUninitialized = 0x00U,
kFLASH_SwapStateReady = 0x01U,
kFLASH_SwapStateUpdate = 0x02U,
kFLASH_SwapStateUpdateErased = 0x03U,
kFLASH_SwapStateComplete = 0x04U,
kFLASH_SwapStateDisabled = 0x05U }
Enumeration for the possible flash Swap status.
- enum flash_swap_block_status_t {
kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero,
kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero }
Enumeration for the possible flash Swap block status
- enum flash_partition_flexram_load_option_t {
kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData,
kFLASH_PartitionFlexramLoadOptionNotLoaded = 0x01U }
Enumeration for the FlexRAM load during reset option.

Flash version

- enum _flash_driver_version_constants {
kFLASH_DriverVersionName = 'F',
kFLASH_DriverVersionMajor = 2,
kFLASH_DriverVersionMinor = 1,
kFLASH_DriverVersionBugfix = 0 }
Flash driver version for ROM.
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
Constructs the version number for drivers.
- #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
Flash driver version for SDK.

Flash configuration

- #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Indicates whether to support FlexNVM in the Flash driver.

- #define FLASH_SSD_IS_FLEXNVM_ENABLED (FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT && FSL_FEATURE_FLASH_HAS_FLEX_NVM)
  Indicates whether the FlexNVM is enabled in the Flash driver.
- #define FLASH_DRIVER_IS_FLASH_RESIDENT 1
  Flash driver location.
- #define FLASH_DRIVER_IS_EXPORTED 0
  Flash Driver Export option.

Flash status

- enum _flash_status {
  kStatus_FLASH_Success = MAKE_STATUS(kStatusGroupGeneric, 0),
  kStatus_FLASH_InvalidArgument = MAKE_STATUS(kStatusGroupGeneric, 4),
  kStatus_FLASH_SizeError = MAKE_STATUS(kStatusGroupFlashDriver, 0),
  kStatus_FLASH_AlignmentError,
  kStatus_FLASH_AddressError = MAKE_STATUS(kStatusGroupFlashDriver, 2),
  kStatus_FLASH_AccessError,
  kStatus_FLASH_ProtectionViolation,
  kStatus_FLASH_CommandFailure,
  kStatus_FLASH_UnknownProperty = MAKE_STATUS(kStatusGroupFlashDriver, 6),
  kStatus_FLASH_EraseKeyError = MAKE_STATUS(kStatusGroupFlashDriver, 7),
  kStatus_FLASH_RegionExecuteOnly,
  kStatus_FLASH_ExecuteInRamFunctionNotReady,
  kStatus_FLASH_PartitionStatusUpdateFailure,
  kStatus_FLASH_SetFlexramAsEepromError,
  kStatus_FLASH_RecoverFlexramAsRamError,
  kStatus_FLASH_SetFlexramAsRamError = MAKE_STATUS(kStatusGroupFlashDriver, 13),
  kStatus_FLASH_RecoverFlexramAsEepromError,
  kStatus_FLASH_CommandNotSupported = MAKE_STATUS(kStatusGroupFlashDriver, 15),
  kStatus_FLASH_SwapSystemNotInUninitialized,
  kStatus_FLASH_SwapIndicatorAddressError }
  Flash driver status codes.
- #define kStatusGroupGeneric 0
  Flash driver status group.
- #define **kStatusGroupFlashDriver** 1
- #define MAKE_STATUS(group, code) ((((group)*100) + (code)))
  Constructs a status code value from a group and a code number.

Flash API key

- enum _flash_driver_api_keys { kFLASH_ApiEraseKey = FOUR_CHAR_CODE('k', 'f', 'e', 'k') }
  Enumeration for Flash driver API keys.
- #define FOUR_CHAR_CODE(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))
  Constructs the four character code for the Flash driver API key.

Initialization

- status_t FLASH_Init (flash_config_t *config)
  Initializes the global flash properties structure members.
- status_t FLASH_SetCallback (flash_config_t *config, flash_callback_t callback)
  Sets the desired flash callback function.
- status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t *config)
  Prepares flash execute-in-RAM functions.

Erasing

- status_t FLASH_EraseAll (flash_config_t *config, uint32_t key)
  Erases entire 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.
- status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t *config, uint32_t key)
  Erases the entire flash, including protected sectors.

Programming

- status_t FLASH_Program (flash_config_t *config, uint32_t start, uint32_t *src, uint32_t lengthInBytes)
  Programs flash with data at locations passed in through parameters.
- status_t FLASH_ProgramOnce (flash_config_t *config, uint32_t index, uint32_t *src, uint32_t lengthInBytes)
  Programs Program Once Field through parameters.

Reading

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

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

Parameters

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

Return values

kStatus_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_SetFlexramAsRamError	Failed to set flexram as RAM.
kStatus_FLASH_ExecuteInRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_AccessError	Invalid instruction codes and out-of bounds addresses.
kStatus_FLASH_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_CommandFailure	Run-time error during command execution.
kStatus_FLASH_RecoverFlexramAsEepromError	Failed to recover FlexRAM as EEPROM.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Address-Error	Address is out of range.
kStatus_FLASH_Set-FlexramAsEepromError	Failed to set flexram as eeprom.
kStatus_FLASH_-ProtectionViolation	The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_Recover-FlexramAsRamError	Failed to recover the FlexRAM as RAM.

- status_t FLASH_ReadOnce (flash_config_t ∗config, uint32_t index, uint32_t ∗dst, uint32_t length-InBytes)
  Reads the resource with data at locations passed in through parameters.

Security

- status_t FLASH_GetSecurityState (flash_config_t ∗config, flash_security_state_t ∗state)
  Returns the security state via the pointer passed into the function.
- status_t FLASH_SecurityBypass (flash_config_t ∗config, const uint8_t ∗backdoorKey)
  Allows users to bypass security with a backdoor key.

Verification

- status_t FLASH_VerifyEraseAll (flash_config_t ∗config, flash_margin_value_t margin)
  Verifies erasure of the entire flash at a specified margin level.
- 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.
- status_t FLASH_VerifyProgram (flash_config_t ∗config, uint32_t start, uint32_t lengthInBytes, const uint32_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.
- status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t ∗config, flash_margin_-value_t margin)
  Verifies whether the program flash execute-only segments have been erased to the specified read margin level.

Protection

- status_t FLASH_IsProtected (flash_config_t ∗config, uint32_t start, uint32_t lengthInBytes, flash_-protection_state_t ∗protection_state)
  Returns the protection state of the desired flash area via the pointer passed into the function.
- status_t FLASH_IsExecuteOnly (flash_config_t ∗config, uint32_t start, uint32_t lengthInBytes, flash_execute_only_access_state_t ∗access_state)

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

Properties

- status_t FLASH_GetProperty (flash_config_t *config, flash_property_tag_t whichProperty, uint32-_t *value)
  Returns the desired flash property.

Flash Protection Utilities

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

Parameters

config	Pointer to storage for the driver runtime state.
option	The option used to set FlexRAM load behavior during reset.
eepromDataSizeCode	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
flexnvmPartitionCode	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

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

- status_t FLASH_PflashSetProtection (flash_config_t *config, uint32_t protectStatus)
  Sets the PFlash Protection to the intended protection status.
- status_t FLASH_PflashGetProtection (flash_config_t *config, uint32_t *protectStatus)
  Gets the PFlash protection status.

17.2 Data Structure Documentation

17.2.1 struct flash_execute_in_ram_function_config_t

Data Fields

- uint32_t activeFunctionCount
  Number of available execute-in-RAM functions.
- uint32_t ∗ flashRunCommand
  Execute-in-RAM function: flash_run_command.
- uint32_t ∗ flashCacheClearCommand
  Execute-in-RAM function: flash_cache_clear_command.

17.2.1.0.0.38 Field Documentation

17.2.1.0.0.38.1 uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

17.2.1.0.0.38.2 uint32_t∗ flash_execute_in_ram_function_config_t::flashRunCommand

17.2.1.0.0.38.3 uint32_t∗ flash_execute_in_ram_function_config_t::flashCacheClearCommand

17.2.2 struct flash_swap_state_config_t

Data Fields

- 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.

17.2.2.0.0.39 Field Documentation

17.2.2.0.0.39.1 flash_swap_state_t flash_swap_state_config_t::flashSwapState

17.2.2.0.0.39.2 flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus

17.2.2.0.0.39.3 flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus

17.2.3 struct flash_swap_ifr_field_config_t

Data Fields

- uint16_t swapIndicatorAddress
  A Swap indicator address field.
- uint16_t swapEnableWord
  A Swap enable word field.
- uint8_t reserved0 [4]

A reserved field.

17.2.3.0.0.40 Field Documentation

17.2.3.0.0.40.1 uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress

17.2.3.0.0.40.2 uint16_t flash_swap_ifr_field_config_t::swapEnableWord

17.2.3.0.0.40.3 uint8_t flash_swap_ifr_field_config_t::reserved0[4]

17.2.4 union flash_swap_ifr_field_data_t

Data Fields

- uint32_t flashSwapIfrData [2]
  A flash Swap IFR field data .
- flash_swap_ifr_field_config_t flashSwapIfrField
  A flash Swap IFR field structure.

17.2.4.0.0.41 Field Documentation

17.2.4.0.0.41.1 uint32_t flash_swap_ifr_field_data_t::flashSwapIfrData[2]

17.2.4.0.0.41.2 flash_swap_ifr_field_config_t flash_swap_ifr_field_data_t::flashSwapIfrField

17.2.5 struct flash_operation_config_t

Data Fields

- uint32_t convertedAddress
  A converted address for the current flash type.
- uint32_t activeSectorSize
  A sector size of the current flash type.
- uint32_t activeBlockSize
  A block size of the current flash type.
- uint32_t blockWriteUnitSize
  The write unit size.
- uint32_t sectorCmdAddressAligment
  An erase sector command address alignment.
- uint32_t partCmdAddressAligment
  A program/verify part command address alignment.
- 32_t resourceCmdAddressAligment
  A read resource command address alignment.
- uint32_t checkCmdAddressAligment
  A program check command address alignment.

17.2.5.0.0.42 Field Documentation

17.2.5.0.0.42.1 uint32_t flash_operation_config_t::convertedAddress

17.2.5.0.0.42.2 uint32_t flash_operation_config_t::activeSectorSize

17.2.5.0.0.42.3 uint32_t flash_operation_config_t::activeBlockSize

17.2.5.0.0.42.4 uint32_t flash_operation_config_t::blockWriteUnitSize

17.2.5.0.0.42.5 uint32_t flash_operation_config_t::sectorCmdAddressAligment

17.2.5.0.0.42.6 uint32_t flash_operation_config_t::partCmdAddressAligment

17.2.5.0.0.42.7 uint32_t flash_operation_config_t::resourceCmdAddressAligment

17.2.5.0.0.42.8 uint32_t flash_operation_config_t::checkCmdAddressAligment

17.2.6 struct flash_config_t

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

Data Fields

- uint32_t PFlashBlockBase
  A base address of the first PFlash block.
- uint32_t PFlashTotalSize
  The size of the combined PFlash block.
- uint32_t PFlashBlockCount
  A number of PFlash blocks.
- uint32_t PFlashSectorSize
  The size in bytes of a sector of PFlash.
- flash_callback_t PFlashCallback
  The callback function for the flash API.
- uint32_t PFlashAccessSegmentSize
  A size in bytes of an access segment of PFlash.
- uint32_t PFlashAccessSegmentCount
  A number of PFlash access segments.
- uint32_t ∗ flashExecuteInRamFunctionInfo
  An information structure of the flash execute-in-RAM function.
- uint32_t FlexRAMBlockBase
  For the FlexNVM device, this is the base address of the FlexRAM For the non-FlexNVM device, this is the base address of the acceleration RAM memory.
- uint32_t FlexRAMTotalSize
  For the FlexNVM device, this is the size of the FlexRAM For the non-FlexNVM device, this is the size of the acceleration RAM memory.
- uint32_t DFlashBlockBase
  For the FlexNVM device, this is the base address of the D-Flash memory (FlexNVM memory) For the

non-FlexNVM device, this field is unused.
- uint32_t DFlashTotalSize
  For the FlexNVM device, this is the total size of the FlexNVM memory; For the non-FlexNVM device, this field is unused.
- uint32_t EEpromTotalSize
  For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM For the non-FlexNVM device, this field is unused.

17.2.6.0.0.43 Field Documentation

17.2.6.0.0.43.1 uint32_t flash_config_t::PFlashTotalSize

17.2.6.0.0.43.2 uint32_t flash_config_t::PFlashBlockCount

17.2.6.0.0.43.3 uint32_t flash_config_t::PFlashSectorSize

17.2.6.0.0.43.4 flash_callback_t flash_config_t::PFlashCallback

17.2.6.0.0.43.5 uint32_t flash_config_t::PFlashAccessSegmentSize

17.2.6.0.0.43.6 uint32_t flash_config_t::PFlashAccessSegmentCount

17.2.6.0.0.43.7 uint32_t* flash_config_t::flashExecuteInRamFunctionInfo

17.3 Macro Definition Documentation

17.3.1 #define MAKE_VERSION(*major*, *minor*, *bugfix*) (((major) << 16) | ((minor) << 8) | (bugfix))

17.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

Version 2.1.0.

17.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

17.3.4 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

17.3.5 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

17.3.6 #define kStatusGroupGeneric 0

17.3.7 #define MAKE_STATUS(*group*, *code*) ((((group)*100) + (code)))

17.3.8 #define FOUR_CHAR_CODE(*a*, *b*, *c*, *d*) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))

17.4 Enumeration Type Documentation

17.4.1 enum _flash_driver_version_constants

Enumerator

kFLASH_DriverVersionName Flash driver version name.
kFLASH_DriverVersionMajor Major flash driver version.
kFLASH_DriverVersionMinor Minor flash driver version.
kFLASH_DriverVersionBugfix Bugfix for flash driver version.

17.4.2 enum _flash_status

Enumerator

kStatus_FLASH_Success API is executed successfully.
kStatus_FLASH_InvalidArgument Invalid argument.
kStatus_FLASH_SizeError Error size.
kStatus_FLASH_AlignmentError Parameter is not aligned with the specified baseline.
kStatus_FLASH_AddressError Address is out of range.
kStatus_FLASH_AccessError Invalid instruction codes and out-of bound 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_UnknownProperty Unknown property.
kStatus_FLASH_EraseKeyError API erase key is invalid.
kStatus_FLASH_RegionExecuteOnly The current region is execute-only.
kStatus_FLASH_ExecuteInRamFunctionNotReady Execute-in-RAM function is not available.
kStatus_FLASH_PartitionStatusUpdateFailure Failed to update partition status.
kStatus_FLASH_SetFlexramAsEepromError Failed to set FlexRAM as EEPROM.
kStatus_FLASH_RecoverFlexramAsRamError Failed to recover FlexRAM as RAM.
kStatus_FLASH_SetFlexramAsRamError Failed to set FlexRAM as RAM.
kStatus_FLASH_RecoverFlexramAsEepromError Failed to recover FlexRAM as EEPROM.
kStatus_FLASH_CommandNotSupported Flash API is not supported.
kStatus_FLASH_SwapSystemNotInUninitialized Swap system is not in an uninitialzed state.
kStatus_FLASH_SwapIndicatorAddressError The swap indicator address is invalid.

17.4.3 enum _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.

Enumerator

kFLASH_ApiEraseKey Key value used to validate all flash erase APIs.

17.4.4 enum flash_margin_value_t

Enumerator

kFLASH_MarginValueNormal Use the 'normal' read level for 1s.
kFLASH_MarginValueUser Apply the 'User' margin to the normal read-1 level.
kFLASH_MarginValueFactory Apply the 'Factory' margin to the normal read-1 level.
kFLASH_MarginValueInvalid Not real margin level, Used to determine the range of valid margin level.

17.4.5 enum flash_security_state_t

Enumerator

kFLASH_SecurityStateNotSecure Flash is not secure.
kFLASH_SecurityStateBackdoorEnabled Flash backdoor is enabled.
kFLASH_SecurityStateBackdoorDisabled Flash backdoor is disabled.

17.4.6 enum flash_protection_state_t

Enumerator

kFLASH_ProtectionStateUnprotected Flash region is not protected.
kFLASH_ProtectionStateProtected Flash region is protected.
kFLASH_ProtectionStateMixed Flash is mixed with protected and unprotected region.

17.4.7 enum flash_execute_only_access_state_t

Enumerator

kFLASH_AccessStateUnLimited Flash region is unlimited.

kFLASH_AccessStateExecuteOnly Flash region is execute only.
kFLASH_AccessStateMixed Flash is mixed with unlimited and execute only region.

17.4.8 enum flash_property_tag_t

Enumerator

kFLASH_PropertyPflashSectorSize Pflash sector size property.
kFLASH_PropertyPflashTotalSize Pflash total size property.
kFLASH_PropertyPflashBlockSize Pflash block size property.
kFLASH_PropertyPflashBlockCount Pflash block count property.
kFLASH_PropertyPflashBlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflashFacSupport Pflash fac support property.
kFLASH_PropertyPflashAccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflashAccessSegmentCount Pflash access segment count property.
kFLASH_PropertyFlexRamBlockBaseAddr FlexRam block base address property.
kFLASH_PropertyFlexRamTotalSize FlexRam total size property.
kFLASH_PropertyDflashSectorSize Dflash sector size property.
kFLASH_PropertyDflashTotalSize Dflash total size property.
kFLASH_PropertyDflashBlockSize Dflash block count property.
kFLASH_PropertyDflashBlockCount Dflash block base address property.
kFLASH_PropertyDflashBlockBaseAddr EEPROM total size property.

17.4.9 enum _flash_execute_in_ram_function_constants

Enumerator

kFLASH_ExecuteInRamFunctionMaxSizeInWords The maximum size of execute-in-RAM function.
kFLASH_ExecuteInRamFunctionTotalNum Total number of execute-in-RAM functions.

17.4.10 enum flash_read_resource_option_t

Enumerator

kFLASH_ResourceOptionFlashIfr Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR.
kFLASH_ResourceOptionVersionId Select code for the version ID.

17.4.11 enum _flash_read_resource_range

Enumerator

kFLASH_ResourceRangePflashIfrSizeInBytes Pflash IFR size in byte.
kFLASH_ResourceRangeVersionIdSizeInBytes Version ID IFR size in byte.
kFLASH_ResourceRangeVersionIdStart Version ID IFR start address.
kFLASH_ResourceRangeVersionIdEnd Version ID IFR end address.
kFLASH_ResourceRangePflashSwapIfrEnd Pflash swap IFR end address.
kFLASH_ResourceRangeDflashIfrStart Dflash IFR start address.
kFLASH_ResourceRangeDflashIfrEnd Dflash IFR end address.

17.4.12 enum flash_flexram_function_option_t

Enumerator

kFLASH_FlexramFunctionOptionAvailableAsRam An option used to make FlexRAM available as RAM.
kFLASH_FlexramFunctionOptionAvailableForEeprom An option used to make FlexRAM available for EEPROM.

17.4.13 enum flash_swap_function_option_t

Enumerator

kFLASH_SwapFunctionOptionEnable An option used to enable the Swap function.
kFLASH_SwapFunctionOptionDisable An option used to disable the Swap function.

17.4.14 enum flash_swap_control_option_t

Enumerator

kFLASH_SwapControlOptionIntializeSystem An option used to initialize the Swap system.
kFLASH_SwapControlOptionSetInUpdateState An option used to set the Swap in an update state.

kFLASH_SwapControlOptionSetInCompleteState An option used to set the Swap in a complete state.
kFLASH_SwapControlOptionReportStatus An option used to report the Swap status.
kFLASH_SwapControlOptionDisableSystem An option used to disable the Swap status.

17.4.15 enum flash_swap_state_t

Enumerator

kFLASH_SwapStateUninitialized Flash Swap system is in an uninitialized state.
kFLASH_SwapStateReady Flash Swap system is in a ready state.
kFLASH_SwapStateUpdate Flash Swap system is in an update state.
kFLASH_SwapStateUpdateErased Flash Swap system is in an updateErased state.
kFLASH_SwapStateComplete Flash Swap system is in a complete state.
kFLASH_SwapStateDisabled Flash Swap system is in a disabled state.

17.4.16 enum flash_swap_block_status_t

Enumerator

kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero Swap block status is that lower half program block at zero.
kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero Swap block status is that upper half program block at zero.

17.4.17 enum flash_partition_flexram_load_option_t

Enumerator

kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData FlexRAM is loaded with valid EEPROM data during reset sequence.
kFLASH_PartitionFlexramLoadOptionNotLoaded FlexRAM is not loaded during reset sequence.

17.5 Function Documentation

17.5.1 status_t FLASH_Init (flash_config_t ∗ *config*)

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_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_-PartitionStatusUpdate-Failure	Failed to update the partition status.

17.5.2 status_t FLASH_SetCallback (flash_config_t ∗ *config*, flash_callback_t *callback*)

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.

17.5.3 status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t ∗ *config*)

Parameters

config	Pointer to the storage for the driver runtime state.

Return values

kStatus_FLASH_Success	API was executed successfully.

kStatus_FLASH_Invalid-Argument	An invalid argument is provided.

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Erase-KeyError	API erase key is invalid.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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_-PartitionStatusUpdate-Failure	Failed to update the partition status.

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

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

Parameters

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

Return values

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

17.5.6 status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t ∗ *config*, uint32_t *key*)

Parameters

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

Return values

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

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

Parameters

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

Return values

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

kStatus_FLASH_Access-Error	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.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with the specified baseline.
kStatus_FLASH_Address-Error	Address is out of range.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

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

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

Parameters

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

Return values

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

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

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

Parameters

config	A pointer to the storage for the driver runtime state.
start	The start address of the desired flash memory to be programmed. Must be word-aligned.
dst	A pointer to the destination buffer of data that is used to store data to be read.
lengthInBytes	The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
option	The resource option which indicates which area should be read back.

Return values

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

Reads the Program Once Field through parameters.

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

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

This function retrieves the current flash security status, including the security enabling state and the back-door key enabling state.

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.

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

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_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.

kStatus_FLASH_-AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_Address-Error	Address is out of range.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

17.5.14 status_t FLASH_VerifyProgram (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, const uint32_t * *expectedData*, flash_margin_value_t *margin*, uint32_t * *failedAddress*, uint32_t * *failedData*)

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

config	A pointer to the storage for the driver runtime state.
start	The start address of the desired flash memory to be verified. Must be word-aligned.
lengthInBytes	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
expectedData	A pointer to the expected data that is to be verified against.
margin	Read margin choice.
failedAddress	A pointer to the returned failing address.
failedData	A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_Address-Error	Address is out of range.
kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

17.5.15 status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t ∗ *config*, flash_margin_value_t *margin*)

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.

kStatus_FLASH_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_Access-Error	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.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_Address-Error	The address is out of range.

17.5.17 status_t FLASH_IsExecuteOnly (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, flash_execute_only_access_state_t * *access_state*)

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

Parameters

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

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_InvalidArgument	An invalid argument is provided.
kStatus_FLASH_AlignmentError	The parameter is not aligned to the specified baseline.
kStatus_FLASH_AddressError	The address is out of range.

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

Parameters

config	A pointer to the storage for the driver runtime state.
whichProperty	The desired property from the list of properties in enum flash_property_tag_t

value	A pointer to the value returned for the desired flash property.

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-UnknownProperty	An unknown property tag.

17.5.19 status_t FLASH_PflashSetProtection (flash_config_t * *config*, uint32_t *protectStatus*)

Parameters

config	A pointer to storage for the driver runtime state.
protectStatus	The expected protect status to set to the PFlash protection register. Each bit is corresponding to protection of 1/32 of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-CommandFailure	Run-time error during command execution.

17.5.20 status_t FLASH_PflashGetProtection (flash_config_t * *config*, uint32_t * *protectStatus*)

Parameters

config	A pointer to the storage for the driver runtime state.
protectStatus	Protect status returned by the PFlash IP. Each bit is corresponding to the protection of 1/32 of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

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

Chapter 18
FlexBus: External Bus Interface Driver

18.1 Overview

The KSDK provides a peripheral driver for the Crossbar External Bus Interface (FlexBus) block of Kinetis devices.

A multifunction external bus interface is provided on the device with a basic functionality to interface to slave-only devices. It can be directly connected to the following asynchronous or synchronous devices with little or no additional circuitry:

- External ROMs
- Flash memories
- Programmable logic devices
- Other simple target (slave) devices

For asynchronous devices, a simple chip-select based interface can be used. The FlexBus interface has up to six general purpose chip-selects, FB_CS[5:0]. The actual number of chip selects available depends upon the device and its pin configuration.

18.2 FlexBus functional operation

To configure the FlexBus driver, use on of the two ways to configure the flexbus_config_t structure.

1. Using the FLEXBUS_GetDefaultConfig() function.
2. Set parameters in the flexbus_config_t structure.

To initialize and configure the FlexBus driver, call the FLEXBUS_Init() function and pass a pointer to the flexbus_config_t structure.

To De-initialize the FlexBus driver, call the FLEXBUS_Deinit() function.

18.3 Typical use case and example

This example shows how to write/read to external memory (MRAM) by using the FlexBus module.

```
flexbus_config_t flexbusUserConfig;

FLEXBUS_GetDefaultConfig(&flexbusUserConfig); /* Gets the default configuration. */
/* Configure some parameters when using MRAM */
flexbusUserConfig.waitStates            = 2U;                 /* Wait 2 states */
flexbusUserConfig.chipBaseAddress       = MRAM_START_ADDRESS; /* MRAM address for using
       FlexBus */
flexbusUserConfig.chipBaseAddressMask   = 7U;                 /* 512 kilobytes memory
       size */
FLEXBUS_Init(FB, &flexbusUserConfig); /* Initializes and configures the FlexBus module */

/* Do something */

FLEXBUS_Deinit(FB);
```

Files

- file fsl_flexbus.h

Data Structures

- struct flexbus_config_t
  Configuration structure that the user needs to set. More...

Enumerations

- enum flexbus_port_size_t {
  kFLEXBUS_4Bytes = 0x00U,
  kFLEXBUS_1Byte = 0x01U,
  kFLEXBUS_2Bytes = 0x02U }
  Defines port size for FlexBus peripheral.
- enum flexbus_write_address_hold_t {
  kFLEXBUS_Hold1Cycle = 0x00U,
  kFLEXBUS_Hold2Cycles = 0x01U,
  kFLEXBUS_Hold3Cycles = 0x02U,
  kFLEXBUS_Hold4Cycles = 0x03U }
  Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum flexbus_read_address_hold_t {
  kFLEXBUS_Hold1Or0Cycles = 0x00U,
  kFLEXBUS_Hold2Or1Cycles = 0x01U,
  kFLEXBUS_Hold3Or2Cycle = 0x02U,
  kFLEXBUS_Hold4Or3Cycle = 0x03U }
  Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum flexbus_address_setup_t {
  kFLEXBUS_FirstRisingEdge = 0x00U,
  kFLEXBUS_SecondRisingEdge = 0x01U,
  kFLEXBUS_ThirdRisingEdge = 0x02U,
  kFLEXBUS_FourthRisingEdge = 0x03U }
  Address setup for FlexBus peripheral.
- enum flexbus_bytelane_shift_t {
  kFLEXBUS_NotShifted = 0x00U,
  kFLEXBUS_Shifted = 0x01U }
  Defines byte-lane shift for FlexBus peripheral.
- enum flexbus_multiplex_group1_t {
  kFLEXBUS_MultiplexGroup1_FB_ALE = 0x00U,
  kFLEXBUS_MultiplexGroup1_FB_CS1 = 0x01U,
  kFLEXBUS_MultiplexGroup1_FB_TS = 0x02U }
  Defines multiplex group1 valid signals.
- enum flexbus_multiplex_group2_t {
  kFLEXBUS_MultiplexGroup2_FB_CS4 = 0x00U,
  kFLEXBUS_MultiplexGroup2_FB_TSIZ0 = 0x01U,
  kFLEXBUS_MultiplexGroup2_FB_BE_31_24 = 0x02U }
  Defines multiplex group2 valid signals.

- enum flexbus_multiplex_group3_t {
kFLEXBUS_MultiplexGroup3_FB_CS5 = 0x00U,
kFLEXBUS_MultiplexGroup3_FB_TSIZ1 = 0x01U,
kFLEXBUS_MultiplexGroup3_FB_BE_23_16 = 0x02U }
Defines multiplex group3 valid signals.
- enum flexbus_multiplex_group4_t {
kFLEXBUS_MultiplexGroup4_FB_TBST = 0x00U,
kFLEXBUS_MultiplexGroup4_FB_CS2 = 0x01U,
kFLEXBUS_MultiplexGroup4_FB_BE_15_8 = 0x02U }
Defines multiplex group4 valid signals.
- enum flexbus_multiplex_group5_t {
kFLEXBUS_MultiplexGroup5_FB_TA = 0x00U,
kFLEXBUS_MultiplexGroup5_FB_CS3 = 0x01U,
kFLEXBUS_MultiplexGroup5_FB_BE_7_0 = 0x02U }
Defines multiplex group5 valid signals.

Driver version

- #define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
Version 2.0.0.

FlexBus functional operation

- void FLEXBUS_Init (FB_Type ∗base, const flexbus_config_t ∗config)
Initializes and configures the FlexBus module.
- void FLEXBUS_Deinit (FB_Type ∗base)
De-initializes a FlexBus instance.
- void FLEXBUS_GetDefaultConfig (flexbus_config_t ∗config)
Initializes the FlexBus configuration structure.

18.4 Data Structure Documentation

18.4.1 struct flexbus_config_t

Data Fields

- uint8_t chip
Chip FlexBus for validation.
- uint8_t waitStates
Value of wait states.
- uint32_t chipBaseAddress
Chip base address for using FlexBus.
- uint32_t chipBaseAddressMask
Chip base address mask.
- bool writeProtect
Write protected.
- bool burstWrite
Burst-Write enable.

- bool burstRead
  Burst-Read enable.
- bool byteEnableMode
  Byte-enable mode support.
- bool autoAcknowledge
  Auto acknowledge setting.
- bool extendTransferAddress
  Extend transfer start/extend address latch enable.
- bool secondaryWaitStates
  Secondary wait states number.
- flexbus_port_size_t portSize
  Port size of transfer.
- flexbus_bytelane_shift_t byteLaneShift
  Byte-lane shift enable.
- flexbus_write_address_hold_t writeAddressHold
  Write address hold or deselect option.
- flexbus_read_address_hold_t readAddressHold
  Read address hold or deselect option.
- flexbus_address_setup_t addressSetup
  Address setup setting.
- flexbus_multiplex_group1_t group1MultiplexControl
  FlexBus Signal Group 1 Multiplex control.
- flexbus_multiplex_group2_t group2MultiplexControl
  FlexBus Signal Group 2 Multiplex control.
- flexbus_multiplex_group3_t group3MultiplexControl
  FlexBus Signal Group 3 Multiplex control.
- flexbus_multiplex_group4_t group4MultiplexControl
  FlexBus Signal Group 4 Multiplex control.
- flexbus_multiplex_group5_t group5MultiplexControl
  FlexBus Signal Group 5 Multiplex control.

18.5 Macro Definition Documentation

18.5.1 #define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

18.6 Enumeration Type Documentation

18.6.1 enum flexbus_port_size_t

Enumerator

kFLEXBUS_4Bytes 32-bit port size
kFLEXBUS_1Byte 8-bit port size
kFLEXBUS_2Bytes 16-bit port size

18.6.2 enum flexbus_write_address_hold_t

Enumerator

kFLEXBUS_Hold1Cycle Hold address and attributes one cycles after FB_CSn negates on writes.
kFLEXBUS_Hold2Cycles Hold address and attributes two cycles after FB_CSn negates on writes.
kFLEXBUS_Hold3Cycles Hold address and attributes three cycles after FB_CSn negates on writes.

kFLEXBUS_Hold4Cycles Hold address and attributes four cycles after FB_CSn negates on writes.

18.6.3 enum flexbus_read_address_hold_t

Enumerator

kFLEXBUS_Hold1Or0Cycles Hold address and attributes 1 or 0 cycles on reads.
kFLEXBUS_Hold2Or1Cycles Hold address and attributes 2 or 1 cycles on reads.
kFLEXBUS_Hold3Or2Cycle Hold address and attributes 3 or 2 cycles on reads.
kFLEXBUS_Hold4Or3Cycle Hold address and attributes 4 or 3 cycles on reads.

18.6.4 enum flexbus_address_setup_t

Enumerator

kFLEXBUS_FirstRisingEdge Assert FB_CSn on first rising clock edge after address is asserted.
kFLEXBUS_SecondRisingEdge Assert FB_CSn on second rising clock edge after address is asserted.
kFLEXBUS_ThirdRisingEdge Assert FB_CSn on third rising clock edge after address is asserted.
kFLEXBUS_FourthRisingEdge Assert FB_CSn on fourth rising clock edge after address is asserted.

18.6.5 enum flexbus_bytelane_shift_t

Enumerator

kFLEXBUS_NotShifted Not shifted. Data is left-justified on FB_AD
kFLEXBUS_Shifted Shifted. Data is right justified on FB_AD

18.6.6 enum flexbus_multiplex_group1_t

Enumerator

kFLEXBUS_MultiplexGroup1_FB_ALE FB_ALE.

kFLEXBUS_MultiplexGroup1_FB_CS1 FB_CS1.
kFLEXBUS_MultiplexGroup1_FB_TS FB_TS.

18.6.7 enum flexbus_multiplex_group2_t

Enumerator

kFLEXBUS_MultiplexGroup2_FB_CS4 FB_CS4.
kFLEXBUS_MultiplexGroup2_FB_TSIZ0 FB_TSIZ0.
kFLEXBUS_MultiplexGroup2_FB_BE_31_24 FB_BE_31_24.

18.6.8 enum flexbus_multiplex_group3_t

Enumerator

kFLEXBUS_MultiplexGroup3_FB_CS5 FB_CS5.
kFLEXBUS_MultiplexGroup3_FB_TSIZ1 FB_TSIZ1.
kFLEXBUS_MultiplexGroup3_FB_BE_23_16 FB_BE_23_16.

18.6.9 enum flexbus_multiplex_group4_t

Enumerator

kFLEXBUS_MultiplexGroup4_FB_TBST FB_TBST.
kFLEXBUS_MultiplexGroup4_FB_CS2 FB_CS2.
kFLEXBUS_MultiplexGroup4_FB_BE_15_8 FB_BE_15_8.

18.6.10 enum flexbus_multiplex_group5_t

Enumerator

kFLEXBUS_MultiplexGroup5_FB_TA FB_TA.
kFLEXBUS_MultiplexGroup5_FB_CS3 FB_CS3.
kFLEXBUS_MultiplexGroup5_FB_BE_7_0 FB_BE_7_0.

18.7 Function Documentation

18.7.1 void FLEXBUS_Init (FB_Type ∗ *base*, const flexbus_config_t ∗ *config*)

This function enables the clock gate for FlexBus module. Only chip 0 is validated and set to known values. Other chips are disabled. NOTE: In this function, certain parameters, depending on external memories, must be set before using FLEXBUS_Init() function. This example shows how to set up the uart_state_t and the flexbus_config_t parameters and how to call the FLEXBUS_Init function by passing in these parameters:

```
flexbus_config_t flexbusConfig;
FLEXBUS_GetDefaultConfig(&flexbusConfig);
flexbusConfig.waitStates            = 2U;
flexbusConfig.chipBaseAddress       = 0x60000000U;
flexbusConfig.chipBaseAddressMask   = 7U;
FLEXBUS_Init(FB, &flexbusConfig);
```

Parameters

base	FlexBus peripheral address.
config	Pointer to the configuration structure

18.7.2 void FLEXBUS_Deinit (FB_Type ∗ *base*)

This function disables the clock gate of the FlexBus module clock.

Parameters

base	FlexBus peripheral address.

18.7.3 void FLEXBUS_GetDefaultConfig (flexbus_config_t ∗ *config*)

This function initializes the FlexBus configuration structure to default value. The default values are:

```
fbConfig->chip                    = 0;
fbConfig->writeProtect            = 0;
fbConfig->burstWrite              = 0;
fbConfig->burstRead               = 0;
fbConfig->byteEnableMode          = 0;
fbConfig->autoAcknowledge         = true;
fbConfig->extendTransferAddress   = 0;
fbConfig->secondaryWaitStates     = 0;
fbConfig->byteLaneShift           = kFLEXBUS_NotShifted;
fbConfig->writeAddressHold        = kFLEXBUS_Hold1Cycle;
fbConfig->readAddressHold         = kFLEXBUS_Hold1Or0Cycles;
fbConfig->addressSetup            = kFLEXBUS_FirstRisingEdge;
fbConfig->portSize                = kFLEXBUS_1Byte;
fbConfig->group1MultiplexControl = kFLEXBUS_MultiplexGroup1_FB_ALE;
fbConfig->group2MultiplexControl = kFLEXBUS_MultiplexGroup2_FB_CS4 ;
fbConfig->group3MultiplexControl = kFLEXBUS_MultiplexGroup3_FB_CS5;
fbConfig->group4MultiplexControl = kFLEXBUS_MultiplexGroup4_FB_TBST;
fbConfig->group5MultiplexControl = kFLEXBUS_MultiplexGroup5_FB_TA;
```

Parameters

config	Pointer to the initialization structure.

See Also

FLEXBUS_Init

Chapter 19
FlexCAN: Flex Controller Area Network Driver

19.1 Overview

The KSDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of Kinetis devices.

Modules

- FlexCAN Driver
- FlexCAN eDMA Driver

19.2 FlexCAN Driver

19.2.1 Overview

This section describes the programming interface of the FlexCAN driver. The FlexCAN driver configures FlexCAN module, provides a functional and transactional interfaces to build the FlexCAN application.

19.2.2 Typical use case

19.2.2.1 Message Buffer Send Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t txFrame;

/* Init FlexCAN module. */
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enable FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the transmit message buffer. */
FLEXCAN_SetTxMbConfig(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, true);

/* Prepares the transmit frame for sending. */
txFrame.format = KFLEXCAN_FrameFormatStandard;
txFrame.type   = KFLEXCAN_FrameTypeData;
txFrame.id     = FLEXCAN_ID_STD(0x123);
txFrame.length = 8;
txFrame.dataWord0 = CAN_WORD0_DATA_BYTE_0(0x11) |
                    CAN_WORD0_DATA_BYTE_1(0x22) |
                    CAN_WORD0_DATA_BYTE_2(0x33) |
                    CAN_WORD0_DATA_BYTE_3(0x44);
txFrame.dataWord1 = CAN_WORD1_DATA_BYTE_4(0x55) |
                    CAN_WORD1_DATA_BYTE_5(0x66) |
                    CAN_WORD1_DATA_BYTE_6(0x77) |
                    CAN_WORD1_DATA_BYTE_7(0x88);
/* Writes a transmit message buffer to send a CAN Message. */
FLEXCAN_WriteTxMb(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, &txFrame);

/* Waits until the transmit message buffer is empty. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX));

/* Cleans the transmit message buffer empty status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX);
```

19.2.2.2 Message Buffer Receive Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive message buffer. */
```

```
mbConfig.format     = KFLEXCAN_FrameFormatStandard;
mbConfig.type       = KFLEXCAN_FrameTypeData;
mbConfig.id         = FLEXCAN_ID_STD(0x123);
FLEXCAN_SetRxMbConfig(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &mbConfig, true);

/* Waits until the receive message buffer is full. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX));

/* Reads the received message from the receive message buffer. */
FLEXCAN_ReadRxMb(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &rxFrame);

/* Cleans the receive message buffer full status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX);
```

19.2.2.3 Receive FIFO Operation

```
uint32_t rxFifoFilter[] = {FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 1, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 1, 0)}
      ;

flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive FIFO. */
rxFifoConfig.idFilterTable = rxFifoFilter;
rxFifoConfig.idFilterType  = KFLEXCAN_RxFifoFilterTypeA;
rxFifoConfig.idFilterNum   = sizeof(rxFifoFilter) / sizeof(rxFifoFilter[0]);
rxFifoConfig.priority      = KFLEXCAN_RxFifoPrioHigh;
FlEXCAN_SetRxFifoConfig(EXAMPLE_CAN, &rxFifoConfig, true);

/* Waits until the receive FIFO becomes available. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag));

/* Reads the message from the receive FIFO. */
FlEXCAN_ReadRxFifo(EXAMPLE_CAN, &rxFrame);

/* Cleans the receive FIFO available status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag);
```

Files

- file fsl_flexcan.h

Data Structures

- struct flexcan_frame_t
  FlexCAN message frame structure. More...
- struct flexcan_config_t
  FlexCAN module configuration structure. More...

- struct flexcan_timing_config_t
  FlexCAN protocol timing characteristic configuration structure. More...
- struct flexcan_rx_mb_config_t
  FlexCAN Receive Message Buffer configuration structure. More...
- struct flexcan_rx_fifo_config_t
  FlexCAN Rx FIFO configuration structure. More...
- struct flexcan_mb_transfer_t
  FlexCAN Message Buffer transfer. More...
- struct flexcan_fifo_transfer_t
  FlexCAN Rx FIFO transfer. More...
- struct flexcan_handle_t
  FlexCAN handle structure. More...

Macros

- #define FLEXCAN_ID_STD(id) (((uint32_t)(((uint32_t)(id)) << CAN_ID_STD_SHIFT)) & CAN_ID_STD_MASK)
  FlexCAN Frame ID helper macro.
- #define FLEXCAN_ID_EXT(id)
  Extend Frame ID helper macro.
- #define FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)
  FlexCAN Rx Message Buffer Mask helper macro.
- #define FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)
  Extend Rx Message Buffer Mask helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)
  FlexCAN Rx FIFO Mask helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)
  Standard Rx FIFO Mask helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)
  Standard Rx FIFO Mask helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 21)
  Standard Rx FIFO Mask helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 13)
  Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 5)
  Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id) ((FLEXCAN_ID_STD(id) & 0x7F8) >> 3)
  Standard Rx FIFO Mask helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type A helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id) ((FLEXCAN_ID_EXT(id) &

0x1FE00000) << 3)
Extend Rx FIFO Mask helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)
  Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)
  Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 21)
  Extend Rx FIFO Mask helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide) FLEXCAN_RX_FIFO_STD-_MASK_TYPE_A(id, rtr, ide)
  FlexCAN Rx FIFO Filter helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)
  Standard Rx FIFO Filter helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)
  Standard Rx FIFO Filter helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)
  Standard Rx FIFO Filter helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)
  Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)
  Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id) FLEXCAN_RX_FIFO_STD_-MASK_TYPE_C_LOW(id)
  Standard Rx FIFO Filter helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide) FLEXCAN_RX_FIFO_EXT-_MASK_TYPE_A(id, rtr, ide)
  Extend Rx FIFO Filter helper macro Type A helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)
  Extend Rx FIFO Filter helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)
  Extend Rx FIFO Filter helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id) FLEXCAN_RX_FIFO_EXT-_MASK_TYPE_C_HIGH(id)
  Extend Rx FIFO Filter helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)
  Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)
  Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id) FLEXCAN_RX_FIFO_EXT_-MASK_TYPE_C_LOW(id)
  Extend Rx FIFO Filter helper macro Type C lower part helper macro.

Typedefs

- typedef void(∗ flexcan_transfer_callback_t)(CAN_Type ∗base, flexcan_handle_t ∗handle, status_t status, uint32_t result, void ∗userData)
  FlexCAN transfer callback function.

Enumerations

- enum _flexcan_status {
kStatus_FLEXCAN_TxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
kStatus_FLEXCAN_TxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 1),
kStatus_FLEXCAN_TxSwitchToRx,
kStatus_FLEXCAN_RxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 3),
kStatus_FLEXCAN_RxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 4),
kStatus_FLEXCAN_RxOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 5),
kStatus_FLEXCAN_RxFifoBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 6),
kStatus_FLEXCAN_RxFifoIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 7),
kStatus_FLEXCAN_RxFifoOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 8),
kStatus_FLEXCAN_RxFifoWarning = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
kStatus_FLEXCAN_ErrorStatus = MAKE_STATUS(kStatusGroup_FLEXCAN, 10),
kStatus_FLEXCAN_UnHandled = MAKE_STATUS(kStatusGroup_FLEXCAN, 11) }
  FlexCAN transfer status.
- enum flexcan_frame_format_t {
kFLEXCAN_FrameFormatStandard = 0x0U,
kFLEXCAN_FrameFormatExtend = 0x1U }
  FlexCAN frame format.
- enum flexcan_frame_type_t {
kFLEXCAN_FrameTypeData = 0x0U,
kFLEXCAN_FrameTypeRemote = 0x1U }
  FlexCAN frame type.
- enum flexcan_clock_source_t {
kFLEXCAN_ClkSrcOsc = 0x0U,
kFLEXCAN_ClkSrcPeri = 0x1U }
  FlexCAN clock source.
- enum flexcan_rx_fifo_filter_type_t {
kFLEXCAN_RxFifoFilterTypeA = 0x0U,
kFLEXCAN_RxFifoFilterTypeB,
kFLEXCAN_RxFifoFilterTypeC,
kFLEXCAN_RxFifoFilterTypeD = 0x3U }
  FlexCAN Rx Fifo Filter type.
- enum flexcan_rx_fifo_priority_t {
kFLEXCAN_RxFifoPrioLow = 0x0U,
kFLEXCAN_RxFifoPrioHigh = 0x1U }
  FlexCAN Rx FIFO priority.
- enum _flexcan_interrupt_enable {
kFLEXCAN_BusOffInterruptEnable = CAN_CTRL1_BOFFMSK_MASK,
kFLEXCAN_ErrorInterruptEnable = CAN_CTRL1_ERRMSK_MASK,
kFLEXCAN_RxWarningInterruptEnable = CAN_CTRL1_RWRNMSK_MASK,
kFLEXCAN_TxWarningInterruptEnable = CAN_CTRL1_TWRNMSK_MASK,
kFLEXCAN_WakeUpInterruptEnable = CAN_MCR_WAKMSK_MASK }
  FlexCAN interrupt configuration structure, default settings all disabled.
- enum _flexcan_flags {

kFLEXCAN_SynchFlag = CAN_ESR1_SYNCH_MASK,
kFLEXCAN_TxWarningIntFlag = CAN_ESR1_TWRNINT_MASK,
kFLEXCAN_RxWarningIntFlag = CAN_ESR1_RWRNINT_MASK,
kFLEXCAN_TxErrorWarningFlag = CAN_ESR1_TXWRN_MASK,
kFLEXCAN_RxErrorWarningFlag = CAN_ESR1_RXWRN_MASK,
kFLEXCAN_IdleFlag = CAN_ESR1_IDLE_MASK,
kFLEXCAN_FaultConfinementFlag = CAN_ESR1_FLTCONF_MASK,
kFLEXCAN_TransmittingFlag = CAN_ESR1_TX_MASK,
kFLEXCAN_ReceivingFlag = CAN_ESR1_RX_MASK,
kFLEXCAN_BusOffIntFlag = CAN_ESR1_BOFFINT_MASK,
kFLEXCAN_ErrorIntFlag = CAN_ESR1_ERRINT_MASK,
kFLEXCAN_WakeUpIntFlag = CAN_ESR1_WAKINT_MASK,
kFLEXCAN_ErrorFlag }
  FlexCAN status flags.
- enum _flexcan_error_flags {
kFLEXCAN_StuffingError = CAN_ESR1_STFERR_MASK,
kFLEXCAN_FormError = CAN_ESR1_FRMERR_MASK,
kFLEXCAN_CrcError = CAN_ESR1_CRCERR_MASK,
kFLEXCAN_AckError = CAN_ESR1_ACKERR_MASK,
kFLEXCAN_Bit0Error = CAN_ESR1_BIT0ERR_MASK,
kFLEXCAN_Bit1Error = CAN_ESR1_BIT1ERR_MASK }
  FlexCAN error status flags.
- enum _flexcan_rx_fifo_flags {
kFLEXCAN_RxFifoOverflowFlag = CAN_IFLAG1_BUF7I_MASK,
kFLEXCAN_RxFifoWarningFlag = CAN_IFLAG1_BUF6I_MASK,
kFLEXCAN_RxFifoFrameAvlFlag = CAN_IFLAG1_BUF5I_MASK }
  FlexCAN Rx FIFO status flags.

Driver version

- #define FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  FlexCAN driver version 2.1.0.

Initialization and deinitialization

- void FLEXCAN_Init (CAN_Type ∗base, const flexcan_config_t ∗config, uint32_t sourceClock_Hz)
  Initializes a FlexCAN instance.
- void FLEXCAN_Deinit (CAN_Type ∗base)
  De-initializes a FlexCAN instance.
- void FLEXCAN_GetDefaultConfig (flexcan_config_t ∗config)
  Get the default configuration structure.

Configuration.

- void FLEXCAN_SetTimingConfig (CAN_Type ∗base, const flexcan_timing_config_t ∗config)
  Sets the FlexCAN protocol timing characteristic.
- void FLEXCAN_SetRxMbGlobalMask (CAN_Type ∗base, uint32_t mask)
  Sets the FlexCAN receive message buffer global mask.
- void FLEXCAN_SetRxFifoGlobalMask (CAN_Type ∗base, uint32_t mask)
  Sets the FlexCAN receive FIFO global mask.
- void FLEXCAN_SetRxIndividualMask (CAN_Type ∗base, uint8_t maskIdx, uint32_t mask)
  Sets the FlexCAN receive individual mask.
- void FLEXCAN_SetTxMbConfig (CAN_Type ∗base, uint8_t mbIdx, bool enable)
  Configures a FlexCAN transmit message buffer.
- void FLEXCAN_SetRxMbConfig (CAN_Type ∗base, uint8_t mbIdx, const flexcan_rx_mb_config_t ∗config, bool enable)
  Configures a FlexCAN Receive Message Buffer.
- void FLEXCAN_SetRxFifoConfig (CAN_Type ∗base, const flexcan_rx_fifo_config_t ∗config, bool enable)
  Configures the FlexCAN Rx FIFO.

Status

- static uint32_t FLEXCAN_GetStatusFlags (CAN_Type ∗base)
  Gets the FlexCAN module interrupt flags.
- static void FLEXCAN_ClearStatusFlags (CAN_Type ∗base, uint32_t mask)
  Clears status flags with the provided mask.
- static void FLEXCAN_GetBusErrCount (CAN_Type ∗base, uint8_t ∗txErrBuf, uint8_t ∗rxErrBuf)
  Gets the FlexCAN Bus Error Counter value.
- static uint32_t FLEXCAN_GetMbStatusFlags (CAN_Type ∗base, uint32_t mask)
  Gets the FlexCAN Message Buffer interrupt flags.
- static void FLEXCAN_ClearMbStatusFlags (CAN_Type ∗base, uint32_t mask)
  Clears the FlexCAN Message Buffer interrupt flags.

Interrupts

- static void FLEXCAN_EnableInterrupts (CAN_Type ∗base, uint32_t mask)
  Enables FlexCAN interrupts according to provided mask.
- static void FLEXCAN_DisableInterrupts (CAN_Type ∗base, uint32_t mask)
  Disables FlexCAN interrupts according to provided mask.
- static void FLEXCAN_EnableMbInterrupts (CAN_Type ∗base, uint32_t mask)
  Enables FlexCAN Message Buffer interrupts.
- static void FLEXCAN_DisableMbInterrupts (CAN_Type ∗base, uint32_t mask)
  Disables FlexCAN Message Buffer interrupts.

Bus Operations

- static void FLEXCAN_Enable (CAN_Type ∗base, bool enable)
  Enables or disables the FlexCAN module operation.

- status_t FLEXCAN_WriteTxMb (CAN_Type ∗base, uint8_t mbIdx, const flexcan_frame_t ∗txFrame)
  Writes a FlexCAN Message to Transmit Message Buffer.
- status_t FLEXCAN_ReadRxMb (CAN_Type ∗base, uint8_t mbIdx, flexcan_frame_t ∗rxFrame)
  Reads a FlexCAN Message from Receive Message Buffer.
- status_t FLEXCAN_ReadRxFifo (CAN_Type ∗base, flexcan_frame_t ∗rxFrame)
  Reads a FlexCAN Message from Rx FIFO.

Transactional

- status_t FLEXCAN_TransferSendBlocking (CAN_Type ∗base, uint8_t mbIdx, flexcan_frame_t ∗txFrame)
  Performs a polling send transaction on the CAN bus.
- status_t FLEXCAN_TransferReceiveBlocking (CAN_Type ∗base, uint8_t mbIdx, flexcan_frame_t ∗rxFrame)
  Performs a polling receive transaction on the CAN bus.
- status_t FLEXCAN_TransferReceiveFifoBlocking (CAN_Type ∗base, flexcan_frame_t ∗rxFrame)
  Performs a polling receive transaction from Rx FIFO on the CAN bus.
- void FLEXCAN_TransferCreateHandle (CAN_Type ∗base, flexcan_handle_t ∗handle, flexcan_transfer_callback_t callback, void ∗userData)
  Initializes the FlexCAN handle.
- status_t FLEXCAN_TransferSendNonBlocking (CAN_Type ∗base, flexcan_handle_t ∗handle, flexcan_mb_transfer_t ∗xfer)
  Sends a message using IRQ.
- status_t FLEXCAN_TransferReceiveNonBlocking (CAN_Type ∗base, flexcan_handle_t ∗handle, flexcan_mb_transfer_t ∗xfer)
  Receives a message using IRQ.
- status_t FLEXCAN_TransferReceiveFifoNonBlocking (CAN_Type ∗base, flexcan_handle_t ∗handle, flexcan_fifo_transfer_t ∗xfer)
  Receives a message from Rx FIFO using IRQ.
- void FLEXCAN_TransferAbortSend (CAN_Type ∗base, flexcan_handle_t ∗handle, uint8_t mbIdx)
  Aborts the interrupt driven message send process.
- void FLEXCAN_TransferAbortReceive (CAN_Type ∗base, flexcan_handle_t ∗handle, uint8_t mbIdx)
  Aborts the interrupt driven message receive process.
- void FLEXCAN_TransferAbortReceiveFifo (CAN_Type ∗base, flexcan_handle_t ∗handle)
  Aborts the interrupt driven message receive from Rx FIFO process.
- void FLEXCAN_TransferHandleIRQ (CAN_Type ∗base, flexcan_handle_t ∗handle)
  FlexCAN IRQ handle function.

19.2.3 Data Structure Documentation

19.2.3.1 struct flexcan_frame_t

19.2.3.1.0.44 Field Documentation

19.2.3.1.0.44.1 uint32_t flexcan_frame_t::timestamp

19.2.3.1.0.44.2 uint32_t flexcan_frame_t::length

19.2.3.1.0.44.3 uint32_t flexcan_frame_t::type

19.2.3.1.0.44.4 uint32_t flexcan_frame_t::format

19.2.3.1.0.44.5 uint32_t flexcan_frame_t::reserve1

19.2.3.1.0.44.6 uint32_t flexcan_frame_t::idhit

19.2.3.1.0.44.7 uint32_t flexcan_frame_t::id

19.2.3.1.0.44.8 uint32_t flexcan_frame_t::reserve2

19.2.3.1.0.44.9 uint32_t flexcan_frame_t::dataWord0

19.2.3.1.0.44.10 uint32_t flexcan_frame_t::dataWord1

19.2.3.1.0.44.11 uint8_t flexcan_frame_t::dataByte3

19.2.3.1.0.44.12 uint8_t flexcan_frame_t::dataByte2

19.2.3.1.0.44.13 uint8_t flexcan_frame_t::dataByte1

19.2.3.1.0.44.14 uint8_t flexcan_frame_t::dataByte0

19.2.3.1.0.44.15 uint8_t flexcan_frame_t::dataByte7

19.2.3.1.0.44.16 uint8_t flexcan_frame_t::dataByte6

19.2.3.1.0.44.17 uint8_t flexcan_frame_t::dataByte5

19.2.3.1.0.44.18 uint8_t flexcan_frame_t::dataByte4

19.2.3.2 struct flexcan_config_t

Data Fields

- uint32_t baudRate
  FlexCAN baud rate in bps.
- flexcan_clock_source_t clkSrc
  Clock source for FlexCAN Protocol Engine.

- uint8_t maxMbNum
  The maximum number of Message Buffers used by user.
- bool enableLoopBack
  Enable or Disable Loop Back Self Test Mode.
- bool enableSelfWakeup
  Enable or Disable Self Wakeup Mode.
- bool enableIndividMask
  Enable or Disable Rx Individual Mask.

19.2.3.2.0.45 Field Documentation

19.2.3.2.0.45.1 uint32_t flexcan_config_t::baudRate

19.2.3.2.0.45.2 flexcan_clock_source_t flexcan_config_t::clkSrc

19.2.3.2.0.45.3 uint8_t flexcan_config_t::maxMbNum

19.2.3.2.0.45.4 bool flexcan_config_t::enableLoopBack

19.2.3.2.0.45.5 bool flexcan_config_t::enableSelfWakeup

19.2.3.2.0.45.6 bool flexcan_config_t::enableIndividMask

19.2.3.3 struct flexcan_timing_config_t

Data Fields

- uint8_t preDivider
  Clock Pre-scaler Division Factor.
- uint8_t rJumpwidth
  Re-sync Jump Width.
- uint8_t phaseSeg1
  Phase Segment 1.
- uint8_t phaseSeg2
  Phase Segment 2.
- uint8_t propSeg
  Propagation Segment.

19.2.3.3.0.46 Field Documentation

19.2.3.3.0.46.1 uint8_t flexcan_timing_config_t::preDivider

19.2.3.3.0.46.2 uint8_t flexcan_timing_config_t::rJumpwidth

19.2.3.3.0.46.3 uint8_t flexcan_timing_config_t::phaseSeg1

19.2.3.3.0.46.4 uint8_t flexcan_timing_config_t::phaseSeg2

19.2.3.3.0.46.5 uint8_t flexcan_timing_config_t::propSeg

19.2.3.4 struct flexcan_rx_mb_config_t

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.

Data Fields

- uint32_t id
  CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.
- flexcan_frame_format_t format
  CAN Frame Identifier format(Standard of Extend).
- flexcan_frame_type_t type
  CAN Frame Type(Data or Remote).

19.2.3.4.0.47 Field Documentation

19.2.3.4.0.47.1 uint32_t flexcan_rx_mb_config_t::id

19.2.3.4.0.47.2 flexcan_frame_format_t flexcan_rx_mb_config_t::format

19.2.3.4.0.47.3 flexcan_frame_type_t flexcan_rx_mb_config_t::type

19.2.3.5 struct flexcan_rx_fifo_config_t

Data Fields

- uint32_t * idFilterTable
  Pointer to FlexCAN Rx FIFO identifier filter table.
- uint8_t idFilterNum
  The quantity of filter elements.
- flexcan_rx_fifo_filter_type_t idFilterType
  The FlexCAN Rx FIFO Filter type.
- flexcan_rx_fifo_priority_t priority
  The FlexCAN Rx FIFO receive priority.

19.2.3.5.0.48 Field Documentation

19.2.3.5.0.48.1 uint32_t* flexcan_rx_fifo_config_t::idFilterTable

19.2.3.5.0.48.2 uint8_t flexcan_rx_fifo_config_t::idFilterNum

19.2.3.5.0.48.3 flexcan_rx_fifo_filter_type_t flexcan_rx_fifo_config_t::idFilterType

19.2.3.5.0.48.4 flexcan_rx_fifo_priority_t flexcan_rx_fifo_config_t::priority

19.2.3.6 struct flexcan_mb_transfer_t

Data Fields

- flexcan_frame_t * frame
  The buffer of CAN Message to be transfer.
- uint8_t mbIdx
  The index of Message buffer used to transfer Message.

19.2.3.6.0.49 Field Documentation

19.2.3.6.0.49.1 flexcan_frame_t* flexcan_mb_transfer_t::frame

19.2.3.6.0.49.2 uint8_t flexcan_mb_transfer_t::mbIdx

19.2.3.7 struct flexcan_fifo_transfer_t

Data Fields

- flexcan_frame_t * frame
  The buffer of CAN Message to be received from Rx FIFO.

19.2.3.7.0.50 Field Documentation

19.2.3.7.0.50.1 flexcan_frame_t* flexcan_fifo_transfer_t::frame

19.2.3.8 struct _flexcan_handle

FlexCAN handle structure definition.

Data Fields

- flexcan_transfer_callback_t callback
  Callback function.
- void * userData
  FlexCAN callback function parameter.
- flexcan_frame_t *volatile mbFrameBuf [CAN_WORD1_COUNT]
  The buffer for received data from Message Buffers.
- flexcan_frame_t *volatile rxFifoFrameBuf
  The buffer for received data from Rx FIFO.

- volatile uint8_t mbState [CAN_WORD1_COUNT]
  Message Buffer transfer state.
- volatile uint8_t rxFifoState
  Rx FIFO transfer state.

19.2.3.8.0.51 Field Documentation

19.2.3.8.0.51.1 flexcan_transfer_callback_t flexcan_handle_t::callback

19.2.3.8.0.51.2 void* flexcan_handle_t::userData

19.2.3.8.0.51.3 flexcan_frame_t* volatile flexcan_handle_t::mbFrameBuf[CAN_WORD1_COUNT]

19.2.3.8.0.51.4 flexcan_frame_t* volatile flexcan_handle_t::rxFifoFrameBuf

19.2.3.8.0.51.5 volatile uint8_t flexcan_handle_t::mbState[CAN_WORD1_COUNT]

19.2.3.8.0.51.6 volatile uint8_t flexcan_handle_t::rxFifoState

19.2.4 Macro Definition Documentation

19.2.4.1 #define FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

19.2.4.2 #define FLEXCAN_ID_STD(*id*) (((uint32_t)(((uint32_t)(id)) << CAN_ID_STD_SHIFT)) & CAN_ID_STD_MASK)

Standard Frame ID helper macro.

19.2.4.3 #define FLEXCAN_ID_EXT(*id*)

Value:

```
(((uint32_t)(((uint32_t)(id)) << CAN_ID_EXT_SHIFT)) & \
     (CAN_ID_EXT_MASK | CAN_ID_STD_MASK))
```

19.2.4.4 #define FLEXCAN_RX_MB_STD_MASK(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     FLEXCAN_ID_STD(id))
```

Standard Rx Message Buffer Mask helper macro.

19.2.4.5 #define FLEXCAN_RX_MB_EXT_MASK(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     FLEXCAN_ID_EXT(id))
```

19.2.4.6 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_STD(id) << 1))
```

Standard Rx FIFO Mask helper macro Type A helper macro.

19.2.4.7 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_STD(id) << 16))
```

19.2.4.8 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
     FLEXCAN_ID_STD(id))
```

19.2.4.9 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) << 21)

19.2.4.10 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) << 13)

\

19.2.4.11 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) << 5)

19.2.4.12 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) >> 3)

19.2.4.13 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(*id*, *rtr*, *ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_EXT(id) << 1))
```

19.2.4.14 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(*id*, *rtr*, *ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     ((FLEXCAN_ID_EXT(id) & 0x1FFF8000) << 1))
```

19.2.4.15 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(*id*, *rtr*, *ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
     ((FLEXCAN_ID_EXT(id) & 0x1FFF8000) >>                                  \
      15))
```

19.2.4.16 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(*id*) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) << 3)

19.2.4.17 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >>             \
     5)
```

19.2.4.18 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >>            \
     13)
```

19.2.4.19 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(*id*) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 21)

19.2.4.20 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(*id, rtr, ide*) FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)

Standard Rx FIFO Filter helper macro Type A helper macro.

19.2.4.21 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(                    \
        id, rtr, ide)
```

19.2.4.22 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(                    \
        id, rtr, ide)
```

19.2.4.23 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(          \
        id)
```

19.2.4.24 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(          \
        id)
```

19.2.4.25 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(          \
        id)
```

19.2.4.26 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(*id*) FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)

\

19.2.4.27 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(*id, rtr, ide*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)

19.2.4.28 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(                          \
        id, rtr, ide)
```

19.2.4.29 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(                           \
        id, rtr, ide)
```

19.2.4.30 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(*id*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)

\

19.2.4.31 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(          \
        id)
```

19.2.4.32 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(*id*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(           \
        id)
```

19.2.4.33 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(*id*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)

19.2.5 Typedef Documentation

19.2.5.1 typedef void(∗ flexcan_transfer_callback_t)(CAN_Type ∗base, flexcan_handle_t ∗handle, status_t status, uint32_t result, void ∗userData)

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.

19.2.6 Enumeration Type Documentation

19.2.6.1 enum _flexcan_status

Enumerator

kStatus_FLEXCAN_TxBusy Tx Message Buffer is Busy.
kStatus_FLEXCAN_TxIdle Tx Message Buffer is Idle.
kStatus_FLEXCAN_TxSwitchToRx Remote Message is send out and Message buffer changed to Receive one.
kStatus_FLEXCAN_RxBusy Rx Message Buffer is Busy.
kStatus_FLEXCAN_RxIdle Rx Message Buffer is Idle.
kStatus_FLEXCAN_RxOverflow Rx Message Buffer is Overflowed.
kStatus_FLEXCAN_RxFifoBusy Rx Message FIFO is Busy.
kStatus_FLEXCAN_RxFifoIdle Rx Message FIFO is Idle.
kStatus_FLEXCAN_RxFifoOverflow Rx Message FIFO is overflowed.
kStatus_FLEXCAN_RxFifoWarning Rx Message FIFO is almost overflowed.
kStatus_FLEXCAN_ErrorStatus FlexCAN Module Error and Status.
kStatus_FLEXCAN_UnHandled UnHadled Interrupt asserted.

19.2.6.2 enum flexcan_frame_format_t

Enumerator

kFLEXCAN_FrameFormatStandard Standard frame format attribute.
kFLEXCAN_FrameFormatExtend Extend frame format attribute.

19.2.6.3 enum flexcan_frame_type_t

Enumerator

kFLEXCAN_FrameTypeData Data frame type attribute.
kFLEXCAN_FrameTypeRemote Remote frame type attribute.

19.2.6.4 enum flexcan_clock_source_t

Enumerator

kFLEXCAN_ClkSrcOsc FlexCAN Protocol Engine clock from Oscillator.
kFLEXCAN_ClkSrcPeri FlexCAN Protocol Engine clock from Peripheral Clock.

19.2.6.5 enum flexcan_rx_fifo_filter_type_t

Enumerator

kFLEXCAN_RxFifoFilterTypeA One full ID (standard and extended) per ID Filter element.
kFLEXCAN_RxFifoFilterTypeB Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.
kFLEXCAN_RxFifoFilterTypeC Four partial 8-bit Standard or extended ID slices per ID Filter Table element.
kFLEXCAN_RxFifoFilterTypeD All frames rejected.

19.2.6.6 enum flexcan_rx_fifo_priority_t

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.

Enumerator

kFLEXCAN_RxFifoPrioLow Matching process start from Rx Message Buffer first.
kFLEXCAN_RxFifoPrioHigh Matching process start from Rx FIFO first.

19.2.6.7 enum _flexcan_interrupt_enable

This structure contains the settings for all of the FlexCAN Module interrupt configurations. Note: FlexCAN Message Buffers and Rx FIFO have their own interrupts.

Enumerator

kFLEXCAN_BusOffInterruptEnable Bus Off interrupt.

kFLEXCAN_ErrorInterruptEnable Error interrupt.
kFLEXCAN_RxWarningInterruptEnable Rx Warning interrupt.
kFLEXCAN_TxWarningInterruptEnable Tx Warning interrupt.
kFLEXCAN_WakeUpInterruptEnable Wake Up interrupt.

19.2.6.8 enum _flexcan_flags

This provides constants for the FlexCAN status flags for use in the FlexCAN functions. Note: The CPU read action clears FlEXCAN_ErrorFlag, therefore user need to read FlEXCAN_ErrorFlag and distinguish which error is occur using _flexcan_error_flags enumerations.

Enumerator

kFLEXCAN_SynchFlag CAN Synchronization Status.
kFLEXCAN_TxWarningIntFlag Tx Warning Interrupt Flag.
kFLEXCAN_RxWarningIntFlag Rx Warning Interrupt Flag.
kFLEXCAN_TxErrorWarningFlag Tx Error Warning Status.
kFLEXCAN_RxErrorWarningFlag Rx Error Warning Status.
kFLEXCAN_IdleFlag CAN IDLE Status Flag.
kFLEXCAN_FaultConfinementFlag Fault Confinement State Flag.
kFLEXCAN_TransmittingFlag FlexCAN In Transmission Status.
kFLEXCAN_ReceivingFlag FlexCAN In Reception Status.
kFLEXCAN_BusOffIntFlag Bus Off Interrupt Flag.
kFLEXCAN_ErrorIntFlag Error Interrupt Flag.
kFLEXCAN_WakeUpIntFlag Wake-Up Interrupt Flag.
kFLEXCAN_ErrorFlag All FlexCAN Error Status.

19.2.6.9 enum _flexcan_error_flags

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN_ErrorFlag in _flexcan_flags enumerations to ditermine which error is generated.

Enumerator

kFLEXCAN_StuffingError Stuffing Error.
kFLEXCAN_FormError Form Error.
kFLEXCAN_CrcError Cyclic Redundancy Check Error.
kFLEXCAN_AckError Received no ACK on transmission.
kFLEXCAN_Bit0Error Unable to send dominant bit.
kFLEXCAN_Bit1Error Unable to send recessive bit.

19.2.6.10 enum _flexcan_rx_fifo_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.

Enumerator

kFLEXCAN_RxFifoOverflowFlag Rx FIFO overflow flag.
kFLEXCAN_RxFifoWarningFlag Rx FIFO almost full flag.
kFLEXCAN_RxFifoFrameAvlFlag Frames available in Rx FIFO flag.

19.2.7 Function Documentation

19.2.7.1 void FLEXCAN_Init (CAN_Type ∗ *base*, const flexcan_config_t ∗ *config*, uint32_t *sourceClock_Hz*)

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_Init function by passing in these parameters:

```
*   flexcan_config_t flexcanConfig;
*   flexcanConfig.clkSrc            = kFLEXCAN_ClkSrcOsc;
*   flexcanConfig.baudRate          = 125000U;
*   flexcanConfig.maxMbNum          = 16;
*   flexcanConfig.enableLoopBack    = false;
*   flexcanConfig.enableSelfWakeup  = false;
*   flexcanConfig.enableIndividMask = false;
*   flexcanConfig.enableDoze        = false;
*   FLEXCAN_Init(CAN0, &flexcanConfig, 8000000UL);
*
```

Parameters

base	FlexCAN peripheral base address.
config	Pointer to user-defined configuration structure.
sourceClock_-Hz	FlexCAN Protocol Engine clock source frequency in Hz.

19.2.7.2 void FLEXCAN_Deinit (CAN_Type ∗ *base*)

This function disable the FlexCAN module clock and set all register value to reset value.

Parameters

base	FlexCAN peripheral base address.

19.2.7.3 void FLEXCAN_GetDefaultConfig (flexcan_config_t * *config*)

This function initializes the FlexCAN configuration structure to default value. The default value are: flexcanConfig->clkSrc = KFLEXCAN_ClkSrcOsc; flexcanConfig->baudRate = 125000U; flexcanConfig->maxMbNum = 16; flexcanConfig->enableLoopBack = false; flexcanConfig->enableSelfWakeup = false; flexcanConfig->enableIndividMask = false; flexcanConfig->enableDoze = false;

Parameters

config	Pointer to FlexCAN configuration structure.

19.2.7.4 void FLEXCAN_SetTimingConfig (CAN_Type * *base*, const flexcan_timing_config_t * *config*)

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_SetTimingConfig() overrides the baud rate set in FLEXCAN_Init().

Parameters

base	FlexCAN peripheral base address.
config	Pointer to the timing configuration structure.

19.2.7.5 void FLEXCAN_SetRxMbGlobalMask (CAN_Type * *base*, uint32_t *mask*)

This function sets the global mask for 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.

mask	Rx Message Buffer Global Mask value.

19.2.7.6 void FLEXCAN_SetRxFifoGlobalMask (CAN_Type ∗ *base*, uint32_t *mask*)

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

base	FlexCAN peripheral base address.
mask	Rx Fifo Global Mask value.

19.2.7.7 void FLEXCAN_SetRxIndividualMask (CAN_Type ∗ *base*, uint8_t *maskIdx*, uint32_t *mask*)

This function sets the individual mask for FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in FLEXCAN_Init(). If Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with same index. What calls for special attention is that only the first 32 individual masks can be used as Rx FIFO filter mask.

Parameters

base	FlexCAN peripheral base address.
maskIdx	The Index of individual Mask.
mask	Rx Individual Mask value.

19.2.7.8 void FLEXCAN_SetTxMbConfig (CAN_Type ∗ *base*, uint8_t *mbIdx*, bool *enable*)

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

base	FlexCAN peripheral base address.

mbIdx	The Message Buffer index.
enable	Enable/Disable Tx Message Buffer. • true: Enable Tx Message Buffer. • false: Disable Tx Message Buffer.

19.2.7.9 void FLEXCAN_SetRxMbConfig (CAN_Type ∗ *base,* uint8_t *mbIdx,* const flexcan_rx_mb_config_t ∗ *config,* bool *enable*)

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

base	FlexCAN peripheral base address.
mbIdx	The Message Buffer index.
config	Pointer to FlexCAN Message Buffer configuration structure.
enable	Enable/Disable Rx Message Buffer. • true: Enable Rx Message Buffer. • false: Disable Rx Message Buffer.

19.2.7.10 void FLEXCAN_SetRxFifoConfig (CAN_Type ∗ *base,* const flexcan_rx_fifo_config_t ∗ *config,* bool *enable*)

This function configures the Rx FIFO with given Rx FIFO configuration.

Parameters

base	FlexCAN peripheral base address.
config	Pointer to FlexCAN Rx FIFO configuration structure.
enable	Enable/Disable Rx FIFO. • true: Enable Rx FIFO. • false: Disable Rx FIFO.

19.2.7.11 static uint32_t FLEXCAN_GetStatusFlags (CAN_Type ∗ *base*) [inline], [static]

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators _flexcan_flags. To check the specific status, compare the return value with enumerators in _flexcan-

_flags.

Parameters

base	FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the _flexcan_flags.

19.2.7.12 static void FLEXCAN_ClearStatusFlags (CAN_Type ∗ *base,* uint32_t *mask*) [inline], [static]

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.
mask	The status flags to be cleared, it is logical OR value of _flexcan_flags.

19.2.7.13 static void FLEXCAN_GetBusErrCount (CAN_Type ∗ *base,* uint8_t ∗ *txErrBuf,* uint8_t ∗ *rxErrBuf*) [inline], [static]

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.
txErrBuf	Buffer to store Tx Error Counter value.
rxErrBuf	Buffer to store Rx Error Counter value.

19.2.7.14 static uint32_t FLEXCAN_GetMbStatusFlags (CAN_Type ∗ *base,* uint32_t *mask*) [inline], [static]

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.

19.2.7.15 static void FLEXCAN_ClearMbStatusFlags (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears the interrupt flags of a given Message Buffers.

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

19.2.7.16 static void FLEXCAN_EnableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the FlexCAN interrupts according to provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

base	FlexCAN peripheral base address.
mask	The interrupts to enable. Logical OR of _flexcan_interrupt_enable.

19.2.7.17 static void FLEXCAN_DisableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the FlexCAN interrupts according to provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

base	FlexCAN peripheral base address.
mask	The interrupts to disable. Logical OR of _flexcan_interrupt_enable.

19.2.7.18 static void FLEXCAN_EnableMbInterrupts (CAN_Type ∗ *base*, uint32_t *mask*) [inline], [static]

This function enables the interrupts of given Message Buffers

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

19.2.7.19 static void FLEXCAN_DisableMbInterrupts (CAN_Type ∗ *base*, uint32_t *mask*) [inline], [static]

This function disables the interrupts of given Message Buffers

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

19.2.7.20 static void FLEXCAN_Enable (CAN_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the FlexCAN module.

Parameters

base	FlexCAN base pointer.
enable	true to enable, false to disable.

19.2.7.21 status_t FLEXCAN_WriteTxMb (CAN_Type ∗ *base*, uint8_t *mbIdx*, const flexcan_frame_t ∗ *txFrame*)

This function writes a CAN Message 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.
mbIdx	The FlexCAN Message Buffer index.
txFrame	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.

19.2.7.22 status_t FLEXCAN_ReadRxMb (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message 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.
mbIdx	The FlexCAN Message Buffer index.
rxFrame	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.

19.2.7.23 status_t FLEXCAN_ReadRxFifo (CAN_Type * *base*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message from the FlexCAN build-in Rx FIFO.

Parameters

base	FlexCAN peripheral base address.
rxFrame	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.

19.2.7.24 status_t FLEXCAN_TransferSendBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *txFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
mbIdx	The FlexCAN Message Buffer index.
txFrame	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.

19.2.7.25 status_t FLEXCAN_TransferReceiveBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
mbIdx	The FlexCAN Message Buffer index.

rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Rx Message Buffer is full and has been read successfully.
kStatus_FLEXCAN_Rx-Overflow	- Rx Message Buffer is already overflowed and has been read successfully.
kStatus_Fail	- Rx Message Buffer is empty.

19.2.7.26 status_t FLEXCAN_TransferReceiveFifoBlocking (CAN_Type ∗ *base*, flexcan_frame_t ∗ *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
rxFrame	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.

19.2.7.27 void FLEXCAN_TransferCreateHandle (CAN_Type ∗ *base*, flexcan_handle_t ∗ *handle*, flexcan_transfer_callback_t *callback*, void ∗ *userData*)

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.
handle	FlexCAN handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

19.2.7.28 status_t FLEXCAN_TransferSendNonBlocking (CAN_Type * *base,* flexcan_handle_t * *handle,* flexcan_mb_transfer_t * *xfer*)

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

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	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.

19.2.7.29 status_t FLEXCAN_TransferReceiveNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_mb_transfer_t * *xfer*)

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

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	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.

19.2.7.30 status_t FLEXCAN_TransferReceiveFifoNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

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

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t.

Return values

kStatus_Success	- Start Rx FIFO receiving process successfully.
kStatus_FLEXCAN_Rx-FifoBusy	- Rx FIFO is currently in use.

19.2.7.31 void FLEXCAN_TransferAbortSend (CAN_Type ∗ *base,* flexcan_handle_t ∗ *handle,* uint8_t *mbIdx*)

This function aborts the interrupt driven message send process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
mbIdx	The FlexCAN Message Buffer index.

19.2.7.32 void FLEXCAN_TransferAbortReceive (CAN_Type ∗ *base,* flexcan_handle_t ∗ *handle,* uint8_t *mbIdx*)

This function aborts the interrupt driven message receive process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
mbIdx	The FlexCAN Message Buffer index.

19.2.7.33 void FLEXCAN_TransferAbortReceiveFifo (CAN_Type ∗ *base,* flexcan_handle_t ∗ *handle*)

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.

19.2.7.34 void FLEXCAN_TransferHandleIRQ (CAN_Type ∗ *base*, flexcan_handle_t ∗ *handle*)

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.

19.3 FlexCAN eDMA Driver

19.3.1 Overview

Files

- file fsl_flexcan_edma.h

Data Structures

- struct flexcan_edma_handle_t
  FlexCAN eDMA handle. More...

Typedefs

- typedef void(∗ flexcan_edma_transfer_callback_t)(CAN_Type ∗base, flexcan_edma_handle_t ∗handle, status_t status, void ∗userData)
  FlexCAN transfer callback function.

eDMA transactional

- void FLEXCAN_TransferCreateHandleEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle, flexcan_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗rxFifoEdmaHandle)
  Initializes the FlexCAN handle, which is used in transactional functions.
- status_t FLEXCAN_TransferReceiveFifoEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle, flexcan_fifo_transfer_t ∗xfer)
  Receives the CAN Message from the Rx FIFO using eDMA.
- void FLEXCAN_TransferAbortReceiveFifoEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle)
  Aborts the receive process which used eDMA.

19.3.2 Data Structure Documentation

19.3.2.1 struct _flexcan_edma_handle

Data Fields

- flexcan_edma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  FlexCAN callback function parameter.
- edma_handle_t ∗ rxFifoEdmaHandle
  The EDMA Rx FIFO channel used.

- volatile uint8_t rxFifoState
  Rx FIFO transfer state.

19.3.2.1.0.52 Field Documentation

19.3.2.1.0.52.1 flexcan_edma_transfer_callback_t flexcan_edma_handle_t::callback

19.3.2.1.0.52.2 void* flexcan_edma_handle_t::userData

19.3.2.1.0.52.3 edma_handle_t* flexcan_edma_handle_t::rxFifoEdmaHandle

19.3.2.1.0.52.4 volatile uint8_t flexcan_edma_handle_t::rxFifoState

19.3.3 Typedef Documentation

19.3.3.1 typedef void(* flexcan_edma_transfer_callback_t)(CAN_Type *base, flexcan_edma_handle_t *handle, status_t status, void *userData)

19.3.4 Function Documentation

19.3.4.1 void FLEXCAN_TransferCreateHandleEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*, flexcan_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *rxFifoEdmaHandle*)

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.
callback	The callback function.
userData	The parameter of the callback function.
rxFifoEdma-Handle	User-requested DMA handle for Rx FIFO DMA transfer.

19.3.4.2 status_t FLEXCAN_TransferReceiveFifoEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.
xfer	FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXCAN_Rx-FifoBusy	Previous transfer ongoing.

19.3.4.3 void FLEXCAN_TransferAbortReceiveFifoEDMA (CAN_Type ∗ *base*, flexcan_edma_handle_t ∗ *handle*)

This function aborts the receive process which used eDMA.

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.

Chapter 20
FTM: FlexTimer Driver

20.1 Overview

The KSDK provides a driver for the FlexTimer Module (FTM) of Kinetis devices.

20.2 Function groups

The FTM driver supports the generation of PWM signals, input capture, dual edge capture, output compare, and quadrature decoder modes. The driver also supports configuring each of the FTM fault inputs.

20.2.1 Initialization and deinitialization

The function FTM_Init() initializes the FTM with specified configurations. The function FTM_GetDefaultConfig() gets the default configurations. The initialization function configures the FTM for the requested register update mode for registers with buffers. It also sets up the FTM's fault operation mode and FTM behavior in the BDM mode.

The function FTM_Deinit() disables the FTM counter and turns off the module clock.

20.2.2 PWM Operations

The function FTM_SetupPwm() sets up FTM channels for the PWM output. The function sets up the PWM signal properties for multiple channels. Each channel has its own duty cycle and level-mode specified. However, the same PWM period and PWM mode is applied to all channels requesting the PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle).

The function FTM_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular FTM channel.

The function FTM_UpdateChnlEdgeLevelSelect() updates the level select bits of a particular FTM channel. This can be used to disable the PWM output when making changes to the PWM signal.

20.2.3 Input capture operations

The function FTM_SetupInputCapture() sets up an FTM channel for the input capture. The user can specify the capture edge and a filter value to be used when processing the input signal.

The function FTM_SetupDualEdgeCapture() can be used to measure the pulse width of a signal. A channel pair is used during capture with the input signal coming through a channel n. The user can specify whether to use one-shot or continuous capture, the capture edge for each channel, and any filter value to be used when processing the input signal.

20.2.4 Output compare operations

The function FTM_SetupOutputCompare() sets up an FTM channel for the output comparison. The user can specify the channel output on a successful comparison and a comparison value.

20.2.5 Quad decode

The function FTM_SetupQuadDecode() sets up FTM channels 0 and 1 for quad decoding. The user can specify the quad decoding mode, polarity, and filter properties for each input signal.

20.2.6 Fault operation

The function FTM_SetupFault() sets up the properties for each fault. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

20.3 Register Update

Some of the FTM registers have buffers. The driver supports various methods to update these registers with the content of the register buffer. The registers can be updated using the PWM synchronized loading or an intermediate point loading. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure.

```
uint32_t pwmSyncMode;
uint32_t reloadPoints;
```

Multiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration ftm_pwm_sync_method_t to the pwmSyncMode field.

When using an intermediate reload points, the PWM synchnronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration ftm_reload_point_t to the reloadPoints field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger.

```
FTM_SetSoftwareTrigger(FTM0, true)
```

20.4 Typical use case

20.4.1 PWM output

Output a PWM signal on two FTM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether LEDs are brighter or dimmer. */
    ftm_config_t ftmInfo;
    uint8_t updatedDutycycle = 0U;
    ftm_chnl_pwm_signal_param_t ftmParam[2];

    /* Configures the FTM parameters with frequency 24 kHZ */
    ftmParam[0].chnlNumber = (ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL;
    ftmParam[0].level = kFTM_LowTrue;
    ftmParam[0].dutyCyclePercent = 0U;
    ftmParam[0].firstEdgeDelayPercent = 0U;

    ftmParam[1].chnlNumber = (ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL;
    ftmParam[1].level = kFTM_LowTrue;
    ftmParam[1].dutyCyclePercent = 0U;
    ftmParam[1].firstEdgeDelayPercent = 0U;

    FTM_GetDefaultConfig(&ftmInfo);

    /* Initializes the FTM module. */
    FTM_Init(BOARD_FTM_BASEADDR, &ftmInfo);

    FTM_SetupPwm(BOARD_FTM_BASEADDR, ftmParam, 2U,
      kFTM_EdgeAlignedPwm, 24000U, FTM_SOURCE_CLOCK);
    FTM_StartTimer(BOARD_FTM_BASEADDR, kFTM_SystemClock);

    while (1)
    {
        /* Delays to check whether the LED brightness has changed. */
        delay();

        if (brightnessUp)
        {
            /* Increases the duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases the duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }
        /* Starts the PWM mode with an updated duty cycle. */
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
      ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
                                 updatedDutycycle);
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
      ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
                                 updatedDutycycle);
        /* Software trigger to update registers. */
        FTM_SetSoftwareTrigger(BOARD_FTM_BASEADDR, true);
    }
}
```

Files

- file fsl_ftm.h

Data Structures

- struct ftm_chnl_pwm_signal_param_t
  Options to configure a FTM channel's PWM signal. More...
- struct ftm_dual_edge_capture_param_t
  FlexTimer dual edge capture parameters. More...
- struct ftm_phase_params_t
  FlexTimer quadrature decode phase parameters. More...
- struct ftm_fault_param_t
  Structure is used to hold the parameters to configure a FTM fault. More...
- struct ftm_config_t
  FTM configuration structure. More...

Enumerations

- enum ftm_chnl_t {
  kFTM_Chnl_0 = 0U,
  kFTM_Chnl_1,
  kFTM_Chnl_2,
  kFTM_Chnl_3,
  kFTM_Chnl_4,
  kFTM_Chnl_5,
  kFTM_Chnl_6,
  kFTM_Chnl_7 }
  List of FTM channels.
- enum ftm_fault_input_t {
  kFTM_Fault_0 = 0U,
  kFTM_Fault_1,
  kFTM_Fault_2,
  kFTM_Fault_3 }
  List of FTM faults.
- enum ftm_pwm_mode_t {
  kFTM_EdgeAlignedPwm = 0U,
  kFTM_CenterAlignedPwm,
  kFTM_CombinedPwm }
  FTM PWM operation modes.
- enum ftm_pwm_level_select_t {
  kFTM_NoPwmSignal = 0U,
  kFTM_LowTrue,
  kFTM_HighTrue }
  FTM PWM output pulse mode: high-true, low-true or no output.
- enum ftm_output_compare_mode_t {
  kFTM_NoOutputSignal = (1U << FTM_CnSC_MSA_SHIFT),
  kFTM_ToggleOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (1U << FTM_CnSC_ELSA_S-

HIFT)),
kFTM_ClearOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (2U << FTM_CnSC_ELSA_SH-IFT)),
kFTM_SetOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (3U << FTM_CnSC_ELSA_SHIF-T)) }
    FlexTimer output compare mode.
- enum ftm_input_capture_edge_t {
kFTM_RisingEdge = (1U << FTM_CnSC_ELSA_SHIFT),
kFTM_FallingEdge = (2U << FTM_CnSC_ELSA_SHIFT),
kFTM_RiseAndFallEdge = (3U << FTM_CnSC_ELSA_SHIFT) }
    FlexTimer input capture edge.
- enum ftm_dual_edge_capture_mode_t {
kFTM_OneShot = 0U,
kFTM_Continuous = (1U << FTM_CnSC_MSA_SHIFT) }
    FlexTimer dual edge capture modes.
- enum ftm_quad_decode_mode_t {
kFTM_QuadPhaseEncode = 0U,
kFTM_QuadCountAndDir }
    FlexTimer quadrature decode modes.
- enum ftm_phase_polarity_t {
kFTM_QuadPhaseNormal = 0U,
kFTM_QuadPhaseInvert }
    FlexTimer quadrature phase polarities.
- enum ftm_deadtime_prescale_t {
kFTM_Deadtime_Prescale_1 = 1U,
kFTM_Deadtime_Prescale_4,
kFTM_Deadtime_Prescale_16 }
    FlexTimer pre-scaler factor for the dead time insertion.
- enum ftm_clock_source_t {
kFTM_SystemClock = 1U,
kFTM_FixedClock,
kFTM_ExternalClock }
    FlexTimer clock source selection.
- enum ftm_clock_prescale_t {
kFTM_Prescale_Divide_1 = 0U,
kFTM_Prescale_Divide_2,
kFTM_Prescale_Divide_4,
kFTM_Prescale_Divide_8,
kFTM_Prescale_Divide_16,
kFTM_Prescale_Divide_32,
kFTM_Prescale_Divide_64,
kFTM_Prescale_Divide_128 }
    FlexTimer pre-scaler factor selection for the clock source.
- enum ftm_bdm_mode_t {
kFTM_BdmMode_0 = 0U,
kFTM_BdmMode_1,
kFTM_BdmMode_2,

kFTM_BdmMode_3 }
Options for the FlexTimer behaviour in BDM Mode.
- enum ftm_fault_mode_t {
kFTM_Fault_Disable = 0U,
kFTM_Fault_EvenChnls,
kFTM_Fault_AllChnlsMan,
kFTM_Fault_AllChnlsAuto }
Options for the FTM fault control mode.
- enum ftm_external_trigger_t {
kFTM_Chnl0Trigger = (1U << 4),
kFTM_Chnl1Trigger = (1U << 5),
kFTM_Chnl2Trigger = (1U << 0),
kFTM_Chnl3Trigger = (1U << 1),
kFTM_Chnl4Trigger = (1U << 2),
kFTM_Chnl5Trigger = (1U << 3),
kFTM_Chnl6Trigger,
kFTM_Chnl7Trigger,
kFTM_InitTrigger = (1U << 6),
kFTM_ReloadInitTrigger = (1U << 7) }
FTM external trigger options.
- enum ftm_pwm_sync_method_t {
kFTM_SoftwareTrigger = FTM_SYNC_SWSYNC_MASK,
kFTM_HardwareTrigger_0 = FTM_SYNC_TRIG0_MASK,
kFTM_HardwareTrigger_1 = FTM_SYNC_TRIG1_MASK,
kFTM_HardwareTrigger_2 = FTM_SYNC_TRIG2_MASK }
FlexTimer PWM sync options to update registers with buffer.
- enum ftm_reload_point_t {
kFTM_Chnl0Match = (1U << 0),
kFTM_Chnl1Match = (1U << 1),
kFTM_Chnl2Match = (1U << 2),
kFTM_Chnl3Match = (1U << 3),
kFTM_Chnl4Match = (1U << 4),
kFTM_Chnl5Match = (1U << 5),
kFTM_Chnl6Match = (1U << 6),
kFTM_Chnl7Match = (1U << 7),
kFTM_CntMax = (1U << 8),
kFTM_CntMin = (1U << 9),
kFTM_HalfCycMatch = (1U << 10) }
FTM options available as loading point for register reload.
- enum ftm_interrupt_enable_t {

kFTM_Chnl0InterruptEnable = (1U << 0),
kFTM_Chnl1InterruptEnable = (1U << 1),
kFTM_Chnl2InterruptEnable = (1U << 2),
kFTM_Chnl3InterruptEnable = (1U << 3),
kFTM_Chnl4InterruptEnable = (1U << 4),
kFTM_Chnl5InterruptEnable = (1U << 5),
kFTM_Chnl6InterruptEnable = (1U << 6),
kFTM_Chnl7InterruptEnable = (1U << 7),
kFTM_FaultInterruptEnable = (1U << 8),
kFTM_TimeOverflowInterruptEnable = (1U << 9),
kFTM_ReloadInterruptEnable = (1U << 10) }
List of FTM interrupts.
- enum ftm_status_flags_t {
kFTM_Chnl0Flag = (1U << 0),
kFTM_Chnl1Flag = (1U << 1),
kFTM_Chnl2Flag = (1U << 2),
kFTM_Chnl3Flag = (1U << 3),
kFTM_Chnl4Flag = (1U << 4),
kFTM_Chnl5Flag = (1U << 5),
kFTM_Chnl6Flag = (1U << 6),
kFTM_Chnl7Flag = (1U << 7),
kFTM_FaultFlag = (1U << 8),
kFTM_TimeOverflowFlag = (1U << 9),
kFTM_ChnlTriggerFlag = (1U << 10),
kFTM_ReloadFlag = (1U << 11) }
List of FTM flags.
- enum _ftm_quad_decoder_flags {
kFTM_QuadDecoderCountingIncreaseFlag = FTM_QDCTRL_QUADIR_MASK,
kFTM_QuadDecoderCountingOverflowOnTopFlag = FTM_QDCTRL_TOFDIR_MASK }
List of FTM Quad Decoder flags.

Functions

- void FTM_SetupFault (FTM_Type ∗base, ftm_fault_input_t faultNumber, const ftm_fault_param_t ∗faultParams)
Sets up the working of the FTM fault protection.
- static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type ∗base, bool enable)
Enables or disables the FTM global time base signal generation to other FTMs.
- static void FTM_SetOutputMask (FTM_Type ∗base, ftm_chnl_t chnlNumber, bool mask)
Sets the FTM peripheral timer channel output mask.
- static void FTM_SetSoftwareTrigger (FTM_Type ∗base, bool enable)
Enables or disables the FTM software trigger for PWM synchronization.
- static void FTM_SetWriteProtection (FTM_Type ∗base, bool enable)
Enables or disables the FTM write protection.

Driver version

- #define FSL_FTM_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

Version 2.0.2.

Initialization and deinitialization

- status_t FTM_Init (FTM_Type ∗base, const ftm_config_t ∗config)
  Ungates the FTM clock and configures the peripheral for basic operation.
- void FTM_Deinit (FTM_Type ∗base)
  Gates the FTM clock.
- void FTM_GetDefaultConfig (ftm_config_t ∗config)
  Fills in the FTM configuration structure with the default settings.

Channel mode operations

- status_t FTM_SetupPwm (FTM_Type ∗base, const ftm_chnl_pwm_signal_param_t ∗chnlParams, uint8_t numOfChnls, ftm_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
  Configures the PWM signal parameters.
- void FTM_UpdatePwmDutycycle (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_pwm_mode_t currentPwmMode, uint8_t dutyCyclePercent)
  Updates the duty cycle of an active PWM signal.
- void FTM_UpdateChnlEdgeLevelSelect (FTM_Type ∗base, ftm_chnl_t chnlNumber, uint8_t level)
  Updates the edge level selection for a channel.
- void FTM_SetupInputCapture (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_input_capture_edge_t captureMode, uint32_t filterValue)
  Enables capturing an input signal on the channel using the function parameters.
- void FTM_SetupOutputCompare (FTM_Type ∗base, ftm_chnl_t chnlNumber, ftm_output_compare_mode_t compareMode, uint32_t compareValue)
  Configures the FTM to generate timed pulses.
- void FTM_SetupDualEdgeCapture (FTM_Type ∗base, ftm_chnl_t chnlPairNumber, const ftm_dual_edge_capture_param_t ∗edgeParam, uint32_t filterValue)
  Configures the dual edge capture mode of the FTM.

Interrupt Interface

- void FTM_EnableInterrupts (FTM_Type ∗base, uint32_t mask)
  Enables the selected FTM interrupts.
- void FTM_DisableInterrupts (FTM_Type ∗base, uint32_t mask)
  Disables the selected FTM interrupts.
- uint32_t FTM_GetEnabledInterrupts (FTM_Type ∗base)
  Gets the enabled FTM interrupts.

Status Interface

- uint32_t FTM_GetStatusFlags (FTM_Type ∗base)
  Gets the FTM status flags.
- void FTM_ClearStatusFlags (FTM_Type ∗base, uint32_t mask)
  Clears the FTM status flags.

Timer Start and Stop

- static void FTM_StartTimer (FTM_Type ∗base, ftm_clock_source_t clockSource)

Starts the FTM counter.
- static void FTM_StopTimer (FTM_Type *base)
  Stops the FTM counter.

Software output control

- static void FTM_SetSoftwareCtrlEnable (FTM_Type *base, ftm_chnl_t chnlNumber, bool value)
  Enables or disables the channel software output control.
- static void FTM_SetSoftwareCtrlVal (FTM_Type *base, ftm_chnl_t chnlNumber, bool value)
  Sets the channel software output control value.

Channel pair operations

- static void FTM_SetFaultControlEnable (FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
  This function enables/disables the fault control in a channel pair.
- static void FTM_SetDeadTimeEnable (FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
  This function enables/disables the dead time insertion in a channel pair.
- static void FTM_SetComplementaryEnable (FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
  This function enables/disables complementary mode in a channel pair.
- static void FTM_SetInvertEnable (FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
  This function enables/disables inverting control in a channel pair.

Quad Decoder

- void FTM_SetupQuadDecode (FTM_Type *base, const ftm_phase_params_t *phaseAParams, const ftm_phase_params_t *phaseBParams, ftm_quad_decode_mode_t quadMode)
  Configures the parameters and activates the quadrature decoder mode.
- static uint32_t FTM_GetQuadDecoderFlags (FTM_Type *base)
  Get the FTM Quad Decoder flags.
- static void FTM_SetQuadDecoderModuloValue (FTM_Type *base, uint32_t startValue, uint32_t overValue)
  Set the modulo values for Quad Decoder.
- static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type *base)
  Get the current Quad Decoder counter value.
- static void FTM_ClearQuadDecoderCounterValue (FTM_Type *base)
  Clear the current Quad Decoder counter value.

20.5 Data Structure Documentation

20.5.1 struct ftm_chnl_pwm_signal_param_t

Data Fields

- ftm_chnl_t chnlNumber
  The channel/channel pair number.
- ftm_pwm_level_select_t level
  PWM output active level select.

- uint8_t dutyCyclePercent
  PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)...
- uint8_t firstEdgeDelayPercent
  Used only in combined PWM mode to generate an asymmetrical PWM.

20.5.1.0.0.53 Field Documentation

20.5.1.0.0.53.1 ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber

In combined mode, this represents the channel pair number.

20.5.1.0.0.53.2 ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level

20.5.1.0.0.53.3 uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

20.5.1.0.0.53.4 uint8_t ftm_chnl_pwm_signal_param_t::firstEdgeDelayPercent

Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

20.5.2 struct ftm_dual_edge_capture_param_t

Data Fields

- ftm_dual_edge_capture_mode_t mode
  Dual Edge Capture mode.
- ftm_input_capture_edge_t currChanEdgeMode
  Input capture edge select for channel n.
- ftm_input_capture_edge_t nextChanEdgeMode
  Input capture edge select for channel n+1.

20.5.3 struct ftm_phase_params_t

Data Fields

- bool enablePhaseFilter
  True: enable phase filter; false: disable filter.
- uint32_t phaseFilterVal
  Filter value, used only if phase filter is enabled.
- ftm_phase_polarity_t phasePolarity
  Phase polarity.

20.5.4 struct ftm_fault_param_t

Data Fields

- bool enableFaultInput
  True: Fault input is enabled; false: Fault input is disabled.
- bool faultLevel
  True: Fault polarity is active low i.e., '0' indicates a fault; False: Fault polarity is active high.
- bool useFaultFilter
  True: Use the filtered fault signal; False: Use the direct path from fault input.

20.5.5 struct ftm_config_t

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

The configuration structure can be made constant so as to reside in flash.

Data Fields

- ftm_clock_prescale_t prescale
  FTM clock prescale value.
- ftm_bdm_mode_t bdmMode
  FTM behavior in BDM mode.
- uint32_t pwmSyncMode
  Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration ftm_pwm_sync_method_t.
- uint32_t reloadPoints
  FTM reload points; When using this, the PWM synchronization is not required.
- ftm_fault_mode_t faultMode
  FTM fault control mode.
- uint8_t faultFilterValue
  Fault input filter value.
- ftm_deadtime_prescale_t deadTimePrescale
  The dead time prescalar value.
- uint8_t deadTimeValue
  The dead time value.
- uint32_t extTriggers
  External triggers to enable.
- uint8_t chnlInitState
  Defines the initialization value of the channels in OUTINT register.
- uint8_t chnlPolarity
  Defines the output polarity of the channels in POL register.
- bool useGlobalTimeBase
  True: Use of an external global time base is enabled; False: disabled.

20.5.5.0.0.54 Field Documentation

20.5.5.0.0.54.1 uint32_t ftm_config_t::pwmSyncMode

20.5.5.0.0.54.2 uint32_t ftm_config_t::reloadPoints

Multiple reload points can be used by providing an OR'ed list of options available in enumeration ftm_reload_point_t.

20.5.5.0.0.54.3 uint32_t ftm_config_t::extTriggers

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

20.6 Enumeration Type Documentation

20.6.1 enum ftm_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

kFTM_Chnl_0 FTM channel number 0.
kFTM_Chnl_1 FTM channel number 1.
kFTM_Chnl_2 FTM channel number 2.
kFTM_Chnl_3 FTM channel number 3.
kFTM_Chnl_4 FTM channel number 4.
kFTM_Chnl_5 FTM channel number 5.
kFTM_Chnl_6 FTM channel number 6.
kFTM_Chnl_7 FTM channel number 7.

20.6.2 enum ftm_fault_input_t

Enumerator

kFTM_Fault_0 FTM fault 0 input pin.
kFTM_Fault_1 FTM fault 1 input pin.
kFTM_Fault_2 FTM fault 2 input pin.
kFTM_Fault_3 FTM fault 3 input pin.

20.6.3 enum ftm_pwm_mode_t

Enumerator

kFTM_EdgeAlignedPwm Edge-aligned PWM.

kFTM_CenterAlignedPwm Center-aligned PWM.
kFTM_CombinedPwm Combined PWM.

20.6.4 enum ftm_pwm_level_select_t

Enumerator

kFTM_NoPwmSignal No PWM output on pin.
kFTM_LowTrue Low true pulses.
kFTM_HighTrue High true pulses.

20.6.5 enum ftm_output_compare_mode_t

Enumerator

kFTM_NoOutputSignal No channel output when counter reaches CnV.
kFTM_ToggleOnMatch Toggle output.
kFTM_ClearOnMatch Clear output.
kFTM_SetOnMatch Set output.

20.6.6 enum ftm_input_capture_edge_t

Enumerator

kFTM_RisingEdge Capture on rising edge only.
kFTM_FallingEdge Capture on falling edge only.
kFTM_RiseAndFallEdge Capture on rising or falling edge.

20.6.7 enum ftm_dual_edge_capture_mode_t

Enumerator

kFTM_OneShot One-shot capture mode.
kFTM_Continuous Continuous capture mode.

20.6.8 enum ftm_quad_decode_mode_t

Enumerator

kFTM_QuadPhaseEncode Phase A and Phase B encoding mode.
kFTM_QuadCountAndDir Count and direction encoding mode.

20.6.9 enum ftm_phase_polarity_t

Enumerator

kFTM_QuadPhaseNormal Phase input signal is not inverted.
kFTM_QuadPhaseInvert Phase input signal is inverted.

20.6.10 enum ftm_deadtime_prescale_t

Enumerator

kFTM_Deadtime_Prescale_1 Divide by 1.
kFTM_Deadtime_Prescale_4 Divide by 4.
kFTM_Deadtime_Prescale_16 Divide by 16.

20.6.11 enum ftm_clock_source_t

Enumerator

kFTM_SystemClock System clock selected.
kFTM_FixedClock Fixed frequency clock.
kFTM_ExternalClock External clock.

20.6.12 enum ftm_clock_prescale_t

Enumerator

kFTM_Prescale_Divide_1 Divide by 1.
kFTM_Prescale_Divide_2 Divide by 2.
kFTM_Prescale_Divide_4 Divide by 4.
kFTM_Prescale_Divide_8 Divide by 8.
kFTM_Prescale_Divide_16 Divide by 16.
kFTM_Prescale_Divide_32 Divide by 32.
kFTM_Prescale_Divide_64 Divide by 64.
kFTM_Prescale_Divide_128 Divide by 128.

20.6.13 enum ftm_bdm_mode_t

Enumerator

kFTM_BdmMode_0 FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.

kFTM_BdmMode_1 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
kFTM_BdmMode_2 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
kFTM_BdmMode_3 FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

20.6.14 enum ftm_fault_mode_t

Enumerator

kFTM_Fault_Disable Fault control is disabled for all channels.
kFTM_Fault_EvenChnls Enabled for even channels only(0,2,4,6) with manual fault clearing.
kFTM_Fault_AllChnlsMan Enabled for all channels with manual fault clearing.
kFTM_Fault_AllChnlsAuto Enabled for all channels with automatic fault clearing.

20.6.15 enum ftm_external_trigger_t

Note

Actual available external trigger sources are SoC-specific

Enumerator

kFTM_Chnl0Trigger Generate trigger when counter equals chnl 0 CnV reg.
kFTM_Chnl1Trigger Generate trigger when counter equals chnl 1 CnV reg.
kFTM_Chnl2Trigger Generate trigger when counter equals chnl 2 CnV reg.
kFTM_Chnl3Trigger Generate trigger when counter equals chnl 3 CnV reg.
kFTM_Chnl4Trigger Generate trigger when counter equals chnl 4 CnV reg.
kFTM_Chnl5Trigger Generate trigger when counter equals chnl 5 CnV reg.
kFTM_Chnl6Trigger Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg.
kFTM_Chnl7Trigger Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg.
kFTM_InitTrigger Generate Trigger when counter is updated with CNTIN.
kFTM_ReloadInitTrigger Available on certain SoC's, trigger on reload point.

20.6.16 enum ftm_pwm_sync_method_t

Enumerator

kFTM_SoftwareTrigger Software triggers PWM sync.

kFTM_HardwareTrigger_0 Hardware trigger 0 causes PWM sync.
kFTM_HardwareTrigger_1 Hardware trigger 1 causes PWM sync.
kFTM_HardwareTrigger_2 Hardware trigger 2 causes PWM sync.

20.6.17 enum ftm_reload_point_t

Note

Actual available reload points are SoC-specific

Enumerator

kFTM_Chnl0Match Channel 0 match included as a reload point.
kFTM_Chnl1Match Channel 1 match included as a reload point.
kFTM_Chnl2Match Channel 2 match included as a reload point.
kFTM_Chnl3Match Channel 3 match included as a reload point.
kFTM_Chnl4Match Channel 4 match included as a reload point.
kFTM_Chnl5Match Channel 5 match included as a reload point.
kFTM_Chnl6Match Channel 6 match included as a reload point.
kFTM_Chnl7Match Channel 7 match included as a reload point.
kFTM_CntMax Use in up-down count mode only, reload when counter reaches the maximum value.

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

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

20.6.18 enum ftm_interrupt_enable_t

Note

Actual available interrupts are SoC-specific

Enumerator

kFTM_Chnl0InterruptEnable Channel 0 interrupt.
kFTM_Chnl1InterruptEnable Channel 1 interrupt.
kFTM_Chnl2InterruptEnable Channel 2 interrupt.
kFTM_Chnl3InterruptEnable Channel 3 interrupt.
kFTM_Chnl4InterruptEnable Channel 4 interrupt.
kFTM_Chnl5InterruptEnable Channel 5 interrupt.
kFTM_Chnl6InterruptEnable Channel 6 interrupt.
kFTM_Chnl7InterruptEnable Channel 7 interrupt.
kFTM_FaultInterruptEnable Fault interrupt.
kFTM_TimeOverflowInterruptEnable Time overflow interrupt.
kFTM_ReloadInterruptEnable Reload interrupt; Available only on certain SoC's.

20.6.19 enum ftm_status_flags_t

Note

Actual available flags are SoC-specific

Enumerator

kFTM_Chnl0Flag Channel 0 Flag.
kFTM_Chnl1Flag Channel 1 Flag.
kFTM_Chnl2Flag Channel 2 Flag.
kFTM_Chnl3Flag Channel 3 Flag.
kFTM_Chnl4Flag Channel 4 Flag.
kFTM_Chnl5Flag Channel 5 Flag.
kFTM_Chnl6Flag Channel 6 Flag.
kFTM_Chnl7Flag Channel 7 Flag.
kFTM_FaultFlag Fault Flag.
kFTM_TimeOverflowFlag Time overflow Flag.
kFTM_ChnlTriggerFlag Channel trigger Flag.
kFTM_ReloadFlag Reload Flag; Available only on certain SoC's.

20.6.20 enum _ftm_quad_decoder_flags

Enumerator

kFTM_QuadDecoderCountingIncreaseFlag Counting direction is increasing (FTM counter increment), or the direction is decreasing.
kFTM_QuadDecoderCountingOverflowOnTopFlag Indicates if the TOF bit was set on the top or the bottom of counting.

20.7 Function Documentation

20.7.1 status_t FTM_Init (FTM_Type * *base*, const ftm_config_t * *config*)

Note

This API should be called at the beginning of the application using the FTM driver.

Parameters

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

Returns

kStatus_Success indicates success; Else indicates failure.

20.7.2 void FTM_Deinit (FTM_Type ∗ *base*)

Parameters

base	FTM peripheral base address

20.7.3 void FTM_GetDefaultConfig (ftm_config_t ∗ *config*)

The default values are:

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

Parameters

config	Pointer to the user configuration structure.

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

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

Parameters

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

Returns

kStatus_Success if the PWM setup was successful kStatus_Error on failure

20.7.5 void FTM_UpdatePwmDutycycle (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)

Parameters

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

20.7.10 void FTM_SetupFault (FTM_Type * *base*, ftm_fault_input_t *faultNumber*, const ftm_fault_param_t * *faultParams*)

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

Parameters

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

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

Parameters

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

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

Parameters

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

20.7.13 uint32_t FTM_GetEnabledInterrupts (FTM_Type ∗ *base*)

Parameters

base	FTM peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration ftm_interrupt_enable-_t

20.7.14 uint32_t FTM_GetStatusFlags (FTM_Type ∗ *base*)

Parameters

base	FTM peripheral base address

Returns

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

20.7.15 void FTM_ClearStatusFlags (FTM_Type ∗ *base*, uint32_t *mask*)

Parameters

base	FTM peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration ftm_-status_flags_t

20.7.16 static void FTM_StartTimer (FTM_Type ∗ *base*, ftm_clock_source_t *clockSource*) [inline], [static]

Parameters

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

20.7.17 static void FTM_StopTimer (FTM_Type * *base*) [inline], [static]

Parameters

base	FTM peripheral base address

20.7.18 static void FTM_SetSoftwareCtrlEnable (FTM_Type * *base,* ftm_chnl_t *chnlNumber,* bool *value*) [inline], [static]

Parameters

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

20.7.19 static void FTM_SetSoftwareCtrlVal (FTM_Type * *base,* ftm_chnl_t *chnlNumber,* bool *value*) [inline], [static]

Parameters

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

20.7.20 static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type * *base,* bool *enable*) [inline], [static]

Parameters

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

20.7.21 static void FTM_SetOutputMask (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *mask*) [inline], [static]

Parameters

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

20.7.22 static void FTM_SetFaultControlEnable (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Parameters

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

20.7.23 static void FTM_SetDeadTimeEnable (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Parameters

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

20.7.24 static void FTM_SetComplementaryEnable (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Parameters

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

20.7.25 static void FTM_SetInvertEnable (FTM_Type ∗ *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Parameters

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

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

Parameters

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

20.7.27 static uint32_t FTM_GetQuadDecoderFlags (FTM_Type ∗ *base*) [inline], [static]

Parameters

base	FTM peripheral base address.

Returns

Flag mask of FTM Quad Decoder, see to _ftm_quad_decoder_flags.

20.7.28 static void FTM_SetQuadDecoderModuloValue (FTM_Type * *base,* uint32_t *startValue,* uint32_t *overValue*) [inline], [static]

The modulo values would configure the min and max value that the Quad decoder counter can reach. Once the counter go over, the counter value would go to the other side and decrease/increase again.

Parameters

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

20.7.29 static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type * *base*) [inline], [static]

Parameters

base	FTM peripheral base address.

Returns

Current quad Decoder counter value.

20.7.30 static void FTM_ClearQuadDecoderCounterValue (FTM_Type * *base*) [inline], [static]

The counter would be set as the initial value.

Parameters

base	FTM peripheral base address.

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

Parameters

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

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

Parameters

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

Chapter 21
GPIO: General-Purpose Input/Output Driver

21.1 Overview

Modules

- FGPIO Driver
- GPIO Driver

Files

- file fsl_gpio.h

Data Structures

- struct gpio_pin_config_t
  The GPIO pin configuration structure. More...

Enumerations

- enum gpio_pin_direction_t {
  kGPIO_DigitalInput = 0U,
  kGPIO_DigitalOutput = 1U }
  GPIO direction definition.

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  GPIO driver version 2.1.0.

21.2 Data Structure Documentation

21.2.1 struct gpio_pin_config_t

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

Data Fields

- gpio_pin_direction_t pinDirection
  GPIO direction, input or output.
- uint8_t outputLogic
  Set a default output logic, which has no use in input.

21.3 Macro Definition Documentation

21.3.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

21.4 Enumeration Type Documentation

21.4.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.
kGPIO_DigitalOutput Set current pin as digital output.

21.5 GPIO Driver

21.5.1 Overview

The KSDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of Kinetis devices.

21.5.2 Typical use case

21.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

21.5.2.2 Input Operation

```
/*  Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN,
      kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

GPIO Configuration

- void GPIO_PinInit (GPIO_Type ∗base, uint32_t pin, const gpio_pin_config_t ∗config)
  Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void GPIO_WritePinOutput (GPIO_Type ∗base, uint32_t pin, uint8_t output)
  Sets the output level of the multiple GPIO pins to the logic 1 or 0.
- static void GPIO_SetPinsOutput (GPIO_Type ∗base, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 1.
- static void GPIO_ClearPinsOutput (GPIO_Type ∗base, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 0.
- static void GPIO_TogglePinsOutput (GPIO_Type ∗base, uint32_t mask)
  Reverses the current output logic of the multiple GPIO pins.

GPIO Input Operations

- static uint32_t GPIO_ReadPinInput (GPIO_Type *base, uint32_t pin)
  Reads the current input value of the GPIO port.

GPIO Interrupt

- uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type *base)
  Reads the GPIO port interrupt status flag.
- void GPIO_ClearPinsInterruptFlags (GPIO_Type *base, uint32_t mask)
  Clears multiple GPIO pin interrupt status flags.

21.5.3 Function Documentation

21.5.3.1 void GPIO_PinInit (GPIO_Type * *base*, uint32_t *pin*, const gpio_pin_config_t * *config*)

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

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

```
* // Define a digital input pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalInput,
*   0,
* }
* //Define a digital output pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalOutput,
*   0,
* }
*
```

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO port pin number
config	GPIO pin configuration pointer

21.5.3.2 static void GPIO_WritePinOutput (GPIO_Type * *base*, uint32_t *pin*, uint8_t *output*) [inline], [static]

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO pin number
output	GPIO pin output logic level. • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

21.5.3.3 static void GPIO_SetPinsOutput (GPIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

21.5.3.4 static void GPIO_ClearPinsOutput (GPIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

21.5.3.5 static void GPIO_TogglePinsOutput (GPIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

21.5.3.6 static uint32_t GPIO_ReadPinInput (GPIO_Type ∗ *base*, uint32_t *pin*) [inline], [static]

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO pin number

Return values

GPIO	port input value • 0: corresponding pin input low-logic level. • 1: corresponding pin input high-logic level.

21.5.3.7 uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type ∗ *base*)

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)

Return values

The	current GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

21.5.3.8 void GPIO_ClearPinsInterruptFlags (GPIO_Type ∗ *base,* uint32_t *mask*)

Parameters

base	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

21.6 FGPIO Driver

This chapter describes the programming interface of the FGPIO driver. The FGPIO driver configures the FGPIO module and provides a functional interface to build the GPIO application.

Note

> FGPIO (Fast GPIO) is only available in a few MCUs. FGPIO and GPIO share the same peripheral but use different registers. FGPIO is closer to the core than the regular GPIO and it's faster to read and write.

21.6.1 Typical use case

21.6.1.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
FGPIO_PinInit(FGPIO_LED, LED_PINNUM, &led_config);
```

21.6.1.2 Input Operation

```
/*  Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_FGPIO_PIN,
      kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
FGPIO_PinInit(FGPIO_SW1, SW1_PINNUM, &sw1_config);
```

Chapter 22
HSADC: 12-bit 5MSPS Analog-to-Digital Converter

22.1 Overview

The KSDK provides a peripheral driver for the 12-bit 5MSPS Analog-to-Digital Converter (HSADC) module of Kinetis devices.

Modules

- HSADC Peripheral driver

Data Structures

- struct hsadc_config_t
  Defines the structure for configuring the HSADC's common setting. More...
- struct hsadc_converter_config_t
  Defines the structure for configuring each converter. More...
- struct hsadc_sample_config_t
  Defines the structure for configuring the sample slot. More...

Macros

- #define FSL_HSADC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  HSADC driver version.
- #define HSADC_SAMPLE_MASK(index) (1U << (index))
  Converter index to mask for sample slot.
- #define HSADC_CALIBRATION_VALUE_A_SINGLE_ENDED_MASK HSADC_CALVAL_A_CALVSING_MASK
  Bit mask of calibration value for converter A in single ended mode.
- #define HSADC_CALIBRATION_VALUE_A_SINGLE_ENDED_SHIFT HSADC_CALVAL_A_CALVSING_SHIFT
  Bit shift of calibration value for converter A in single ended mode.
- #define HSADC_CALIBRATION_VALUE_A_DIFFERENTIAL_MASK HSADC_CALVAL_A_CALVDIF_MASK
  Bit mask of calibration value for converter A in differential mode.
- #define HSADC_CALIBRATION_VALUE_A_DIFFERENTIAL_SHIFT HSADC_CALVAL_A_CALVDIF_SHIFT
  Bit shift of calibration value for converter A in differential mode.
- #define HSADC_CALIBRATION_VALUE_B_SINGLE_ENDED_MASK (HSADC_CALVAL_B_CALVSING_MASK << 16U)
  Bit mask of calibration value for converter B in single ended mode.
- #define HSADC_CALIBRATION_VALUE_B_SINGLE_ENDED_SHIFT (HSADC_CALVAL_B_CALVSING_SHIFT + 16U)
  Bit shift of calibration value for converter B in single ended mode.

- #define HSADC_CALIBRATION_VALUE_B_DIFFERENTIAL_MASK (HSADC_CALVAL_-B_CALVDIF_MASK << 16U)
  Bit mask of calibration value for converter B in differential mode.
- #define HSADC_CALIBRATION_VALUE_B_DIFFERENTIAL_SHIFT (HSADC_CALVAL_B-_CALVDIF_SHIFT + 16U)
  Bit shift of calibration value for converter B in differential mode.

Enumerations

- enum _hsadc_status_flags {
  kHSADC_ZeroCrossingFlag = (1U << 0U),
  kHSADC_HighLimitFlag = (1U << 1U),
  kHSADC_LowLimitFlag = (1U << 2U),
  kHSADC_ConverterAEndOfScanFlag = (1U << 3U),
  kHSADC_ConverterBEndOfScanFlag = (1U << 4U),
  kHSADC_ConverterAEndOfCalibrationFlag = (1U << 5U),
  kHSADC_ConverterBEndOfCalibrationFlag = (1U << 6U),
  kHSADC_ConverterAConvertingFlag = (1U << 7U),
  kHSADC_ConverterBConvertingFlag = (1U << 8U),
  kHSADC_ConverterADummyConvertingFlag = (1U << 9U),
  kHSADC_ConverterBDummyConvertingFlag = (1U << 10U),
  kHSADC_ConverterACalibratingFlag = (1U << 11U),
  kHSADC_ConverterBCalibratingFlag = (1U << 12U),
  kHSADC_ConverterAPowerDownFlag = (1U << 13U),
  kHSADC_ConverterBPowerDownFlag = (1U << 14U) }
  HSADC status flags.
- enum _hsadc_interrupt_enable {
  kHSADC_ZeroCrossingInterruptEnable = (1U << 0U),
  kHSADC_HighLimitInterruptEnable = (1U << 1U),
  kHSADC_LowLimitInterruptEnable = (1U << 2U),
  kHSADC_ConverterAEndOfScanInterruptEnable = (1U << 3U),
  kHSADC_ConverterBEndOfScanInterruptEnable = (1U << 4U),
  kHSADC_ConverterAEndOfCalibrationInterruptEnable = (1U << 5U),
  kHSADC_ConverterBEndOfCalibrationInterruptEnable = (1U << 6U) }
  HSADC Interrupts.
- enum _hsadc_converter_id {
  kHSADC_ConverterA = (1U << 0U),
  kHSADC_ConverterB = (1U << 1U) }
  HSADC Converter identifier.
- enum hsadc_dual_converter_scan_mode_t {
  kHSADC_DualConverterWorkAsOnceSequential = 0U,
  kHSADC_DualConverterWorkAsOnceParallel = 1U,
  kHSADC_DualConverterWorkAsLoopSequential = 2U,
  kHSADC_DualConverterWorkAsLoopParallel = 3U,
  kHSADC_DualConverterWorkAsTriggeredSequential = 4U,
  kHSADC_DualConverterWorkAsTriggeredParallel = 5U }

Defines the enumeration for dual converter scan mode.
- enum hsadc_resolution_t {
kHSADC_Resolution6Bit = 0U,
kHSADC_Resolution8Bit = 1U,
kHSADC_Resolution10Bit = 2U,
kHSADC_Resolution12Bit = 3U }
- enum hsadc_dma_trigger_source_t {
kHSADC_DMATriggerSourceAsEndOfScan = 0U,
kHSADC_DMATriggerSourceAsSampleReady = 1U }
Defines the enumeration for the DMA trigger source.
- enum hsadc_zero_crossing_mode_t {
kHSADC_ZeroCorssingDisabled = 0U,
kHSADC_ZeroCorssingForPtoNSign = 1U,
kHSADC_ZeroCorssingForNtoPSign = 2U,
kHSADC_ZeroCorssingForAnySignChanged = 3U }
Defines the enumeration for the sample slot's zero crossing event.
- enum hsadc_idle_work_mode_t {
kHSADC_IdleKeepNormal = 0U,
kHSADC_IdleAutoStandby = 1U,
kHSADC_IdleAutoPowerDown = 2U }
Defines the enumeration for the converter's work mode in idle mode.
- enum _hsadc_calibration_mode {
kHSADC_CalibrationModeDifferential = (1U << 0U),
kHSADC_CalibrationModeSingleEnded = (1U << 1U) }
Converter's calibration mode.

HSADC Initialization and deinitialization.

- void HSADC_Init (HSADC_Type *base, const hsadc_config_t *config)
Initializes the HSADC module.
- void HSADC_GetDefaultConfig (hsadc_config_t *config)
Gets an available pre-defined settings for module's configuration.
- void HSADC_Deinit (HSADC_Type *base)
De-initializes the HSADC module.

Converter.

- void HSADC_SetConverterConfig (HSADC_Type *base, uint16_t converterMask, const hsadc_converter_config_t *config)
Configures the converter.
- void HSADC_GetDefaultConverterConfig (hsadc_converter_config_t *config)
Gets an available pre-defined settings for each converter's configuration.
- void HSADC_EnableConverter (HSADC_Type *base, uint16_t converterMask, bool enable)
Enables the converter's conversion.
- void HSADC_EnableConverterSyncInput (HSADC_Type *base, uint16_t converterMask, bool enable)
Enables the input of an external sync signal.
- void HSADC_EnableConverterPower (HSADC_Type *base, uint16_t converterMask, bool enable)
Enables power for the converter.

- void HSADC_DoSoftwareTriggerConverter (HSADC_Type ∗base, uint16_t converterMask)
  Triggers the converter by using the software trigger.
- void HSADC_EnableConverterDMA (HSADC_Type ∗base, uint16_t converterMask, bool enable)
  Enables the DMA feature.
- void HSADC_EnableInterrupts (HSADC_Type ∗base, uint16_t mask)
  Enables the interrupts.
- void HSADC_DisableInterrupts (HSADC_Type ∗base, uint16_t mask)
  Disables the interrupts.
- uint16_t HSADC_GetStatusFlags (HSADC_Type ∗base)
  Gets the status flags.
- void HSADC_ClearStatusFlags (HSADC_Type ∗base, uint16_t mask)
  Clears the status flags.

Sample.

- void HSADC_SetSampleConfig (HSADC_Type ∗base, uint16_t sampleIndex, const hsadc_sample-_config_t ∗config)
  Configures the sample slot.
- void HSADC_GetDefaultSampleConfig (hsadc_sample_config_t ∗config)
  Gets the default sample configuration.
- static void HSADC_EnableSample (HSADC_Type ∗base, uint16_t sampleMask, bool enable)
  Enables the sample slot.
- static void HSADC_EnableSampleResultReadyInterrupts (HSADC_Type ∗base, uint16_t sample-Mask, bool enable)
  Enables the interrupt for each sample slot when its result is ready.
- static uint16_t HSADC_GetSampleReadyStatusFlags (HSADC_Type ∗base)
  Returns the sample ready flags of sample slots.
- static uint16_t HSADC_GetSampleLowLimitStatusFlags (HSADC_Type ∗base)
  Gets the low-limit flags of sample slots.
- static void HSADC_ClearSampleLowLimitStatusFlags (HSADC_Type ∗base, uint16_t sample-Mask)
  Clears low-limit flags of sample slots.
- static uint16_t HSADC_GetSampleHighLimitStatusFlags (HSADC_Type ∗base)
  Gets the high-limit flags of sample slots.
- static void HSADC_ClearSampleHighLimitStatusFlags (HSADC_Type ∗base, uint16_t sample-Mask)
  Clears high-limit flags of sample slots.
- static uint16_t HSADC_GetSampleZeroCrossingStatusFlags (HSADC_Type ∗base)
  Gets the zero crossing flags of sample slots.
- static void HSADC_ClearSampleZeroCrossingStatusFlags (HSADC_Type ∗base, uint16_t sample-Mask)
  Clears zero crossing flags of sample slots.
- static uint16_t HSADC_GetSampleResultValue (HSADC_Type ∗base, uint16_t sampleIndex)
  Gets the sample result value.

Calibration.

- void HSADC_DoAutoCalibration (HSADC_Type ∗base, uint16_t converterMask, uint16_-t calibrationModeMask)
  Starts the hardware calibration.

- uint32_t HSADC_GetCalibrationResultValue (HSADC_Type ∗base)
  Gets the calibration result value.
- void HSADC_EnableCalibrationResultValue (HSADC_Type ∗base, uint16_t converterMask, bool enable)
  Enables or disables the calibration result value.

22.2 Data Structure Documentation

22.2.1 struct hsadc_config_t

Data Fields

- hsadc_dual_converter_scan_mode_t dualConverterScanMode
  Dual converter's scan mode.
- bool enableSimultaneousMode
  Using Simultaneous mode.
- hsadc_resolution_t resolution
  Resolution mode.
- hsadc_dma_trigger_source_t DMATriggerSoruce
  DMA trigger source.
- hsadc_idle_work_mode_t idleWorkMode
  Converter's work mode when idle.
- uint16_t powerUpDelayCount
  Delay count united as 32 clocks to wait for the clock to be stable.

22.2.1.0.0.55 Field Documentation

22.2.1.0.0.55.1 hsadc_dual_converter_scan_mode_t hsadc_config_t::dualConverterScanMode

22.2.1.0.0.55.2 bool hsadc_config_t::enableSimultaneousMode

22.2.1.0.0.55.3 hsadc_resolution_t hsadc_config_t::resolution

22.2.1.0.0.55.4 hsadc_dma_trigger_source_t hsadc_config_t::DMATriggerSoruce

22.2.1.0.0.55.5 hsadc_idle_work_mode_t hsadc_config_t::idleWorkMode

22.2.1.0.0.55.6 uint16_t hsadc_config_t::powerUpDelayCount

Available range is 0-63.

22.2.2 struct hsadc_converter_config_t

Data Fields

- uint16_t clockDivisor
  Converter's clock divisor for the clock source.
- uint16_t samplingTimeCount

Sampling time count.
- uint16_t powerUpCalibrationModeMask
  Calibration mode mask in the power up period.

22.2.2.0.0.56 Field Documentation

22.2.2.0.0.56.1 uint16_t hsadc_converter_config_t::clockDivisor

Available range is 2-64.

22.2.2.0.0.56.2 uint16_t hsadc_converter_config_t::samplingTimeCount

The resultant sampling time is (1.5 + samplingTimeCount) x clock period. Available range is 0-255.

22.2.2.0.0.56.3 uint16_t hsadc_converter_config_t::powerUpCalibrationModeMask

See the "_hsadc_calibration_mode". If this field isn't zero, call the function HSADC_GetStatusFlags() to check whether the End of Calibration flag is set to wait for sthe calibration process to complete. If this is zero, it indicates no calibration is executed in power up period.

22.2.3 struct hsadc_sample_config_t

channelNumber, channel67MuxNumber, and enableDifferentialPair have following relationship: channelNumber equals 0∼7 represents channel 0∼7 of converter A. channelNumber equals 8∼15 represents channel 0∼7 of converter B. 1) When channelNumber = 6 and enableDifferentialPair = false, channel67MuxNumber represents converter A's channel 6's sub multiplex channel number. 2) When channelNumber = 6 and enableDifferentialPair = true, channel67MuxNumber represents converter A's channel 6 and channel 7's sub multiplex channel number. 3) When channelNumber = 7 and enableDifferentialPair = false, channel67MuxNumber represents converter A's channel 7's sub multiplex channel number. 4) When channelNumber = 7 and enableDifferentialPair = true, channel67MuxNumber represents converter A's channel 6 and channel 7's sub multiplex channel number. 5) When channelNumber = 14 and enableDifferentialPair = false, channel67MuxNumber represents converter B's channel 6's sub multiplex channel number. 6) When channelNumber = 14 and enableDifferentialPair = true, channel67MuxNumber represents converter B's channel 6 and channel 7's sub multiplex channel number. 7) When channelNumber = 15 and enableDifferentialPair = false, channel67MuxNumber represents converter B's channel 7's sub multiplex channel number. 8) When channelNumber = 15 and enableDifferentialPair = true, channel67MuxNumber represents converter B's channel 6 and channel 7's sub multiplex channel number. 9) In other cases, channel67MuxNumber won't be functional.

Data Fields

- uint16_t channelNumber
  Channel number.
- uint16_t channel67MuxNumber
  Channel 6/7's sub multiplex channel number.
- bool enableDifferentialPair

Use differential sample input or not.
- hsadc_zero_crossing_mode_t zeroCrossingMode
  Zero crossing mode.
- uint16_t highLimitValue
  High-limit value.
- uint16_t lowLimitValue
  Low-limit value.
- uint16_t offsetValue
  Offset value.
- bool enableWaitSync
  Wait for sync input to launch this sample's conversion or not.

22.2.3.0.0.57 Field Documentation

22.2.3.0.0.57.1 uint16_t hsadc_sample_config_t::channelNumber

Available range is 0-15.

22.2.3.0.0.57.2 uint16_t hsadc_sample_config_t::channel67MuxNumber

When channelNumber = 6 or 14, its available range is 0∼6. When channelNumber = 7 or 15, its available range is 0∼5.

22.2.3.0.0.57.3 bool hsadc_sample_config_t::enableDifferentialPair

In differential mode, the sub multiplex channel number of channel 6 and channel 7 must be configured to be same.

22.2.3.0.0.57.4 hsadc_zero_crossing_mode_t hsadc_sample_config_t::zeroCrossingMode

22.2.3.0.0.57.5 uint16_t hsadc_sample_config_t::highLimitValue

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

22.2.3.0.0.57.6 uint16_t hsadc_sample_config_t::lowLimitValue

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

22.2.3.0.0.57.7 uint16_t hsadc_sample_config_t::offsetValue

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

22.2.3.0.0.57.8 bool hsadc_sample_config_t::enableWaitSync

22.3 Macro Definition Documentation

22.3.1 #define FSL_HSADC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

22.3.2 #define HSADC_SAMPLE_MASK(*index*) (1U << (index))

22.3.3 #define HSADC_CALIBRATION_VALUE_A_SINGLE_ENDED_MASK HSADC-_CALVAL_A_CALVSING_MASK

22.3.4 #define HSADC_CALIBRATION_VALUE_A_SINGLE_ENDED_SHIFT HSADC-_CALVAL_A_CALVSING_SHIFT

22.3.5 #define HSADC_CALIBRATION_VALUE_A_DIFFERENTIAL_MASK HSADC_-CALVAL_A_CALVDIF_MASK

22.3.6 #define HSADC_CALIBRATION_VALUE_A_DIFFERENTIAL_SHIFT HSADC_-CALVAL_A_CALVDIF_SHIFT

22.3.7 #define HSADC_CALIBRATION_VALUE_B_SINGLE_ENDED_MASK (HSADC-_CALVAL_B_CALVSING_MASK << 16U)

22.3.8 #define HSADC_CALIBRATION_VALUE_B_SINGLE_ENDED_SHIFT (HSADC-_CALVAL_B_CALVSING_SHIFT + 16U)

22.3.9 #define HSADC_CALIBRATION_VALUE_B_DIFFERENTIAL_MASK (HSADC_-CALVAL_B_CALVDIF_MASK << 16U)

22.3.10 #define HSADC_CALIBRATION_VALUE_B_DIFFERENTIAL_SHIFT (HSADC-_CALVAL_B_CALVDIF_SHIFT + 16U)

22.4 Enumeration Type Documentation

22.4.1 enum _hsadc_status_flags

Enumerator

kHSADC_ZeroCrossingFlag Zero crossing.
kHSADC_HighLimitFlag High-limit.
kHSADC_LowLimitFlag Low-limit.
kHSADC_ConverterAEndOfScanFlag End of Scan, converter A.
kHSADC_ConverterBEndOfScanFlag End of Scan, converter B.
kHSADC_ConverterAEndOfCalibrationFlag End of Calibration, converter A.

kHSADC_ConverterBEndOfCalibrationFlag End of Calibration, converter B.
kHSADC_ConverterAConvertingFlag Conversion in progress, converter A.
kHSADC_ConverterBConvertingFlag Conversion in progress, converter B.
kHSADC_ConverterADummyConvertingFlag Dummy conversion in progress, converter A.
kHSADC_ConverterBDummyConvertingFlag Dummy conversion in progress, converter B.
kHSADC_ConverterACalibratingFlag Calibration in progress, converter A.
kHSADC_ConverterBCalibratingFlag Calibration in progress, converter B.
kHSADC_ConverterAPowerDownFlag The converter is powered down, converter A.
kHSADC_ConverterBPowerDownFlag The converter is powered down, converter B.

22.4.2 enum _hsadc_interrupt_enable

Enumerator

kHSADC_ZeroCrossingInterruptEnable Zero crossing interrupt.
kHSADC_HighLimitInterruptEnable High-limit interrupt.
kHSADC_LowLimitInterruptEnable Low-limit interrupt.
kHSADC_ConverterAEndOfScanInterruptEnable End of Scan interrupt, converter A.
kHSADC_ConverterBEndOfScanInterruptEnable End of Scan interrupt, converter B.
kHSADC_ConverterAEndOfCalibrationInterruptEnable End of Calibration, converter A.
kHSADC_ConverterBEndOfCalibrationInterruptEnable End of Calibration, converter B.

22.4.3 enum _hsadc_converter_id

Enumerator

kHSADC_ConverterA Converter A.
kHSADC_ConverterB Converter B.

22.4.4 enum hsadc_dual_converter_scan_mode_t

Enumerator

kHSADC_DualConverterWorkAsOnceSequential Once (single) sequential.
kHSADC_DualConverterWorkAsOnceParallel Once parallel.
kHSADC_DualConverterWorkAsLoopSequential Loop sequential.
kHSADC_DualConverterWorkAsLoopParallel Loop parallel.
kHSADC_DualConverterWorkAsTriggeredSequential Triggered sequential.
kHSADC_DualConverterWorkAsTriggeredParallel Triggered parallel.

22.4.5 enum hsadc_resolution_t

Enumerator

kHSADC_Resolution6Bit 6 bit resolution mode.
kHSADC_Resolution8Bit 8 bit resolution mode.
kHSADC_Resolution10Bit 10 bit resolution mode.
kHSADC_Resolution12Bit 12 bit resolution mode.

22.4.6 enum hsadc_dma_trigger_source_t

Enumerator

kHSADC_DMATriggerSourceAsEndOfScan DMA trigger source is end of scan interrupt.
kHSADC_DMATriggerSourceAsSampleReady DMA trigger source is RDY bits.

22.4.7 enum hsadc_zero_crossing_mode_t

Enumerator

kHSADC_ZeroCorssingDisabled Zero Crossing disabled.
kHSADC_ZeroCorssingForPtoNSign Zero Crossing enabled for positive to negative sign change.
kHSADC_ZeroCorssingForNtoPSign Zero Crossing enabled for negative to positive sign change.
kHSADC_ZeroCorssingForAnySignChanged Zero Crossing enabled for any sign change.

22.4.8 enum hsadc_idle_work_mode_t

Enumerator

kHSADC_IdleKeepNormal Keep normal.
kHSADC_IdleAutoStandby Fall into standby mode automatically.
kHSADC_IdleAutoPowerDown Fall into power down mode automatically.

22.4.9 enum _hsadc_calibration_mode

Enumerator

kHSADC_CalibrationModeDifferential Calibration request for differential mode.
kHSADC_CalibrationModeSingleEnded Calibration request for single ended mode.

22.5 Function Documentation

22.5.1 void HSADC_Init (HSADC_Type * *base*, const hsadc_config_t * *config*)

This function initializes the HSADC module. The operations are:

- Enable the clock for HSADC.
- Set the global settings for HSADC converter.

Parameters

base	HSADC peripheral base address.
config	Pointer to configuration structure. See the "hsadc_config_t".

22.5.2 void HSADC_GetDefaultConfig (hsadc_config_t * *config*)

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

```
*   config->dualConverterScanMode = kHSADC_DualConverterWorkAsTriggeredParallel
      ;
*   config->enableSimultaneousMode = true;
*   config->resolution = kHSADC_Resolution12Bit;
*   config->DMATriggerSoruce = kHSADC_DMATriggerSourceAsEndOfScan;
*   config->idleWorkMode = kHSADC_IdleKeepNormal;
*   config->powerUpDelay = 18U;
*
```

Parameters

config	Pointer to configuration structure. See the "hsadc_config_t"

22.5.3 void HSADC_Deinit (HSADC_Type * *base*)

This function de-initializes the HSADC module. The operations are:

- Power down both converters.
- Disable the clock for HSADC.

Parameters

base	HSADC peripheral base address.

22.5.4 void HSADC_SetConverterConfig (HSADC_Type * *base*, uint16_t *converterMask*, const hsadc_converter_config_t * *config*)

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be configured. See the "_hsadc_converter_id".
config	Pointer to configuration structure. See the "hsadc_converter_config_t".

22.5.5 void HSADC_GetDefaultConverterConfig (hsadc_converter_config_t * *config*)

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

```
*   config->clockDivisor = 4U;
*   config->samplingTimeCount = 0U;
*   config->enablePowerUpCalibration = false;
*   config->powerUpCalibrationModeMask = kHSADC_CalibrationModeSingleEnded
      ;
*
```

Parameters

config	Pointer to configuration structure. See the "hsadc_converter_config_t"

22.5.6 void HSADC_EnableConverter (HSADC_Type * *base*, uint16_t *converterMask*, bool *enable*)

This function enables the converter's conversion by making the converter exit stop mode. The conversion should only be launched after the converter is enabled. When this feature is asserted to be "false", the current scan is stopped and no further scans can start. All the software and hardware triggers are ignored.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
enable	Enable or disable the feature.

22.5.7 void HSADC_EnableConverterSyncInput (HSADC_Type * *base*, uint16_t *converterMask*, bool *enable*)

This function enables the input of the external sync signal. The external sync signal could be used to trigger the conversion if the hardware trigger-related setting is used. Note: When in "Once" scan mode, this gate is off automatically after an available sync is received. Enable the input again manually if another sync signal is needed.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
enable	Enable or disable the feature.

22.5.8 void HSADC_EnableConverterPower (HSADC_Type * *base*, uint16_t *converterMask*, bool *enable*)

This function enables the power for the converter. The converter should be powered on before conversion. Once this API is called, the converter is powered on after a few moments (so-called power up delay) to make the power stable.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
enable	Enable or disable the feature.

22.5.9 void HSADC_DoSoftwareTriggerConverter (HSADC_Type * *base*, uint16_t *converterMask*)

This function triggers the converter using a software trigger. The software trigger can be used to start a conversion sequence.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".

22.5.10 void HSADC_EnableConverterDMA (HSADC_Type ∗ *base,* uint16_t *converterMask,* bool *enable*)

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
enable	Enable or disable the feature.

22.5.11 void HSADC_EnableInterrupts (HSADC_Type ∗ *base,* uint16_t *mask*)

Parameters

base	HSADC peripheral base address.
mask	Mask value for interrupt events. See the "_hsadc_interrupt_enable".

22.5.12 void HSADC_DisableInterrupts (HSADC_Type ∗ *base,* uint16_t *mask*)

Parameters

base	HSADC peripheral base address.
mask	Mask value for interrupt events. See the "_hsadc_interrupt_enable".

22.5.13 uint16_t HSADC_GetStatusFlags (HSADC_Type ∗ *base*)

Parameters

base	HSADC peripheral base address.

Returns

Mask value for the event flags. See the "_hsadc_status_flags".

22.5.14 void HSADC_ClearStatusFlags (HSADC_Type ∗ *base*, uint16_t *mask*)

Parameters

base	HSADC peripheral base address.
flags	Mask value for the event flags to be cleared. See the "_hsadc_status_flags".

22.5.15 void HSADC_SetSampleConfig (HSADC_Type ∗ *base*, uint16_t *sampleIndex*, const hsadc_sample_config_t ∗ *config*)

A sample list in this module works like a conversion sequence. Each sample slot can be used to designate to sample which channel is in converter A and converter B. The detail mapping relationship between sample slot and converter's channel can be found in the SoC reference manual.

Parameters

base	HSADC peripheral base address.
sampleIndex	Index of sample slot in conversion sequence. Available range is 0-15.
config	Pointer to configuration structure. See the "hsadc_sample_config_t".

22.5.16 void HSADC_GetDefaultSampleConfig (hsadc_sample_config_t ∗ *config*)

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

```
*   config->channelNumber = 0U;
*   config->channel6MuxNumber = 0U;
*   config->channel7MuxNumber = 0U;
*   config->enableDifferentialPair = false;
*   config->zeroCrossingMode = kHSADC_ZeroCorssingDisabled;
*   config->highLimitValue = 0x7FF8U;
*   config->lowLimitValue = 0U;
*   config->offsetValue = 0U;
*   config->enableWaitSync = false;
*
```

Parameters

config	Pointer to configuration structure. See the "hsadc_sample_config_t".

22.5.17 static void HSADC_EnableSample (HSADC_Type ∗ *base,* uint16_t *sampleMask,* bool *enable*) [inline], [static]

This function enables the sample slot. Only the enabled sample slot can join the conversion sequence.

Parameters

base	HSADC peripheral base address.
sampleMask	Mask value of sample slots in conversion sequence. Each bit corresponds to a sample slot.
enable	Enable or disable the feature.

22.5.18 static void HSADC_EnableSampleResultReadyInterrupts (HSADC_Type ∗ *base,* uint16_t *sampleMask,* bool *enable*) [inline], [static]

Parameters

base	HSADC peripheral base address.
sampleMask	Mask value of sample slots in conversion sequence. Each bit is corresponding to a sample slot.
enable	Enable or disable the feature.

22.5.19 static uint16_t HSADC_GetSampleReadyStatusFlags (HSADC_Type ∗ *base*) [inline], [static]

Parameters

base	HSADC peripheral base address.

Returns

Mask value for the sample slots if their result are ready.

22.5.20 static uint16_t HSADC_GetSampleLowLimitStatusFlags (HSADC_Type * *base*) [inline], [static]

Parameters

base	HSADC peripheral base address.

Returns

Mask value for the sample slots if their results exceed the low limit.

22.5.21 static void HSADC_ClearSampleLowLimitStatusFlags (HSADC_Type * *base*, uint16_t *sampleMask*) [inline], [static]

Parameters

base	HSADC peripheral base address.
sampleMask	Mask value for the sample slots' flags to be cleared.

22.5.22 static uint16_t HSADC_GetSampleHighLimitStatusFlags (HSADC_Type * *base*) [inline], [static]

Parameters

base	HSADC peripheral base address.

Returns

Mask value for the sample slots if their results exceed the high limit.

22.5.23 static void HSADC_ClearSampleHighLimitStatusFlags (HSADC_Type * *base*, uint16_t *sampleMask*) [inline], [static]

Parameters

base	HSADC peripheral base address.

sampleMask	Mask value for the sample slots to be cleared flags.

22.5.24 static uint16_t HSADC_GetSampleZeroCrossingStatusFlags (HSADC_Type * *base*) [inline], [static]

Parameters

base	HSADC peripheral base address.

Returns

Mask value for the sample slots if their results cause the zero crossing event.

22.5.25 static void HSADC_ClearSampleZeroCrossingStatusFlags (HSADC_Type * *base,* uint16_t *sampleMask*) [inline], [static]

Parameters

base	HSADC peripheral base address.
sampleMask	Mask value for the sample slots to be cleared flags.

22.5.26 static uint16_t HSADC_GetSampleResultValue (HSADC_Type * *base,* uint16_t *sampleIndex*) [inline], [static]

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

Parameters

base	HSADC peripheral base address.
sampleIndex	Index of sample slot.

Returns

Sample's conversion value.

22.5.27 void HSADC_DoAutoCalibration (HSADC_Type * *base*, uint16_t *converterMask*, uint16_t *calibrationModeMask*)

This function starts the single ended calibration and differential calibration for converter A and converter B at the same time. Note that this is a non blocking function. End of Scan flag and End of Calibration flag are both be set after the calibration process. As a result, the user should check these two flags by using the function HSADC_GetStatusFlags() to wait for the calibration process to complete.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
calibration-ModeMask	Mask for calibration mode to be operated. See the "_hsadc_calibration_mode". Shouldn't be zero.

22.5.28 uint32_t HSADC_GetCalibrationResultValue (HSADC_Type * *base*)

This function returns the single ended calibration value and differential calibration value for converter A and converter B. The calibration value of each calibration mode for each converter can be received from this function's return value by using the mask and shift definition from HSADC_CALIBRATION_VALUE_A_SINGLE_ENDED_MASK to HSADC_CALIBRATION_VALUE_B_DIFFERENTIAL_SHIFT.

Parameters

base	HSADC peripheral base address.

Returns

Calibration value for converter A and converter B.

22.5.29 void HSADC_EnableCalibrationResultValue (HSADC_Type * *base*, uint16_t *converterMask*, bool *enable*)

This function enables or disables converter A and converter B to use the calibration values to obtain the final conversion result by calibration sum operation.

Parameters

base	HSADC peripheral base address.
converterMask	Mask for converters to be operated. See the "_hsadc_converter_id".
enable	Enable or disable the feature.

22.6 HSADC Peripheral driver

This chapter describes the programming interface of the HSADC Peripheral driver. The HSADC driver configures the HSADC module.

The HSADC consists of two separate analog-to-digital converters, each with eight analog inputs and its own sample and hold circuit. A common digital control module configures and controls the functioning of the converters.

To match the hardware feature, the HSADC driver is designed with 4 parts: APIs for configuring common digital control module, APIs for configuring each converter, APIs for operating sample slots and APIs for calibration.

The common digital control configuration is set when initializing the HSADC module in the application and deciding how the two converters work together. The converter configuration APIs set each converter's attributes and operate them. Finally, the sample slot API configures the sample slot with the input channel and gather them to be a conversion sequence. After triggering (using a software trigger or an external hardware trigger), the sequence is started and the conversion is executed.

22.6.1 Function groups

22.6.1.1 Initialization and deinitialization

This function group initializes/de-initializes the HSADC. The initialization should be done first before any operation to the HSADC module in the application. It enables the clock and sets the configuration for the common digital control. An API is provided to fill the configuration with available default settings.

22.6.1.2 Each converter

This function group configures each of the two converters in the HSADC module.

22.6.1.3 Each sample

This function group is for the operations to sample slot.

22.6.1.4 Calibration

This function group calibrates to get more accurate result.

22.6.2 Typical use case

22.6.2.1 Triggered parallel

```
hsadc_config_t hsadcConfigStruct;
```

```
hsadc_converter_config_t hsadcConverterConfigStruct;
hsadc_sample_config_t hsadcSampleConfigStruct;
uint16_t sampleMask;

//...

// Initialization for HSADC.
HSADC_GetDefaultConfig(&hsadcConfigStruct);
HSADC_Init(ADC, &hsadcConfigStruct);

// Configures each converter.
HSADC_GetDefaultConverterConfig(&hsadcConverterConfigStruct);
HSADC_SetConverterConfig(ADC, kHSADC_ConverterA |
      kHSADC_ConverterB, &hsadcConverterConfigStruct);
// Enable the power for each converter.
HSADC_EnableConverterPower(ADC, kHSADC_ConverterA |
      kHSADC_ConverterB, true);
while ( (kHSADC_ConverterAPowerDownFlag |
      kHSADC_ConverterBPowerDownFlag)
        == ((kHSADC_ConverterAPowerDownFlag |
      kHSADC_ConverterBPowerDownFlag) &
      HSADC_GetStatusFlags(ADC))  )
{}
// Opens the clock to each converter.
HSADC_EnableConverter(ADC, kHSADC_ConverterA |
      kHSADC_ConverterB, true);

// Configures the samples.
HSADC_GetDefaultSampleConfig(&hsadcSampleConfigStruct);

/* For converter A. */
hsadcSampleConfigStruct.channelNumber = DEMO_HSADC_CONVA_CHN_NUM1;
hsadcSampleConfigStruct.channel6MuxNumber = DEMO_HSADC_CONVA_CHN6_MUX_NUM1;
hsadcSampleConfigStruct.channel7MuxNumber = DEMO_HSADC_CONVA_CHN7_MUX_NUM1;
HSADC_SetSampleConfig(DEMO_HSADC_INSTANCE, 0U, &hsadcSampleConfigStruct);
HSADC_SetSampleConfig(DEMO_HSADC_INSTANCE, 1U, &hsadcSampleConfigStruct);

/* For converter B. */
hsadcSampleConfigStruct.channelNumber = DEMO_HSADC_CONVA_CHN_NUM2;
hsadcSampleConfigStruct.channel6MuxNumber = DEMO_HSADC_CONVA_CHN6_MUX_NUM2;
hsadcSampleConfigStruct.channel7MuxNumber = DEMO_HSADC_CONVA_CHN7_MUX_NUM2;
HSADC_SetSampleConfig(DEMO_HSADC_INSTANCE, 8U, &hsadcSampleConfigStruct);
HSADC_SetSampleConfig(DEMO_HSADC_INSTANCE, 9U, &hsadcSampleConfigStruct);

// Enable the sample slot.
sampleMask = HSADC_SAMPLE_MASK(0U) // For Converter A.
           | HSADC_SAMPLE_MASK(1U) // For Converter A.
           | HSADC_SAMPLE_MASK(8U) // For Converter B.
           | HSADC_SAMPLE_MASK(9U);// For Converter B.
HSADC_EnableSample(ADC, sampleMask, true);
HSADC_EnableSample(ADC, (uint16_t)(~sampleMask), false);// Disable other sample slot.

// Triggers the converter.
// Triggering the converter A executes both converter conversions when in
// "kHSADC_DualConverterWorkAsTriggeredParallel" work mode.
HSADC_DoSoftwareTriggerConverter(ADC,
      kHSADC_ConverterA);


// Waits for the conversion to be done.
while (kHSADC_ConverterAEndOfScanFlag != (
      kHSADC_ConverterAEndOfScanFlag &
      HSADC_GetStatusFlags(ADC)))
{}

if (sampleMask == (sampleMask & HSADC_GetSampleReadyStatusFlags(ADC)) )
{
    PRINTF("HSADC Value1: %d\r\n", (int16_t)HSADC_GetSampleResultValue(ADC, 0U));
```

```
    PRINTF("HSADC Value2: %d\r\n", (int16_t)HSADC_GetSampleResultValue(ADC, 1U));
    PRINTF("HSADC Value3: %d\r\n", (int16_t)HSADC_GetSampleResultValue(ADC, 8U));
    PRINTF("HSADC Value4: %d\r\n", (int16_t)HSADC_GetSampleResultValue(ADC, 9U));
}
HSADC_ClearStatusFlags(ADC, kHSADC_ConverterAEndOfScanFlag
      );
```

Chapter 23
I2C: Inter-Integrated Circuit Driver

23.1 Overview

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver
- I2C µCOS/II Driver
- I2C µCOS/III Driver

23.2 I2C Driver

23.2.1 Overview

The KSDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of Kinetis devices.

The I2C driver includes functional APIs and transactional APIs.

Functional APIs are target the low-level APIs. Functional APIs can be used for the I2C master/slave initialization/configuration/operation for optimization/customization purpose. Using the functional APIs requires the knowledge of the I2C master peripheral and how to organize functional APIs to meet the application requirements. The I2C functional operation groups provide the functional APIs set.

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions I2C_MasterTransfer-NonBlocking() set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

23.2.2 Typical use case

23.2.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Gets the default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Inititializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Sends a start and a slave address. */
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address,
      kI2C_Write/kI2C_Read);

/* Waits for the sent out address. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR)) & kI2C_IntPendingFlag))
{

}

if(status & kI2C_ReceiveNakFlag)
{
    return kStatus_I2C_Nak;
}

result = I2C_MasterWriteBlocking(EXAMPLE_I2C_MASTER_BASEADDR, txBuff, BUFFER_SIZE);

if(result)
{
    /* If an error occours, send STOP. */
```

```
    I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
    return result;
}

while(!(I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR) & kI2C_IntPendingFlag))
{

}

/* Wait for all data to be sent out and sends STOP. */
I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
```

23.2.2.2 Master Operation in interrupt transactional method

```
i2c_master_handle_t g_m_handle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *
      userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Gets a default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

I2C_MasterTransferCreateHandle(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle,
      i2c_master_callback, NULL);
I2C_MasterTransferNonBlocking(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle, &
      masterXfer);

/*  Waits for a transfer to be completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

23.2.2.3 Master Operation in DMA transactional method

```
i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
```

```
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *
      userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Gets the default configuration for the master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

DMAMGR_RequestChannel((dma_request_source_t)DMA_REQUEST_SRC, 0, &dmaHandle);

I2C_MasterTransferCreateHandleDMA(EXAMPLE_I2C_MASTER_BASEADDR, &
      g_m_dma_handle, i2c_master_callback, NULL, &dmaHandle);
I2C_MasterTransferDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, &masterXfer);

/*  Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

23.2.2.4 Slave Operation in functional method

```
i2c_slave_config_t slaveConfig;
uint8_t status;
status_t result = kStatus_Success;

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
       addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
      kI2C_RangeMatch;
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

/* Waits for an address match. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag))
{

}

/* A slave transmits; master is reading from the slave. */
if (status & kI2C_TransferDirectionFlag)
{
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}
else
{
```

```
    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}

return result;
```

23.2.2.5 Slave Operation in interrupt transactional method

```
i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
      userData)
{
    switch (xfer->event)
    {
        /*  Transmit request */
        case kI2C_SlaveTransmitEvent:
            /*  Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /*  Receives request */
        case kI2C_SlaveReceiveEvent:
            /*  Update information for received process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /*  Transfer is done */
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
       addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
      kI2C_RangeMatch;

I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

I2C_SlaveTransferCreateHandle(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
      i2c_slave_callback, NULL);

I2C_SlaveTransferNonBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
      kI2C_SlaveCompletionEvent);

/*  Waits for a transfer to be completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;
```

Files

- file fsl_i2c.h

Data Structures

- struct i2c_master_config_t
  I2C master user configuration. More...
- struct i2c_slave_config_t
  I2C slave user configuration. More...
- struct i2c_master_transfer_t
  I2C master transfer structure. More...
- struct i2c_master_handle_t
  I2C master handle structure. More...
- struct i2c_slave_transfer_t
  I2C slave transfer structure. More...
- struct i2c_slave_handle_t
  I2C slave handle structure. More...

Typedefs

- typedef void(∗ i2c_master_transfer_callback_t)(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t status, void ∗userData)
  I2C master transfer callback typedef.
- typedef void(∗ i2c_slave_transfer_callback_t)(I2C_Type ∗base, i2c_slave_transfer_t ∗xfer, void ∗userData)
  I2C slave transfer callback typedef.

Enumerations

- enum _i2c_status {
  kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_I2C, 0),
  kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_I2C, 1),
  kStatus_I2C_Nak = MAKE_STATUS(kStatusGroup_I2C, 2),
  kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_I2C, 3),
  kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_I2C, 4) }
  I2C status return codes.
- enum _i2c_flags {
  kI2C_ReceiveNakFlag = I2C_S_RXAK_MASK,
  kI2C_IntPendingFlag = I2C_S_IICIF_MASK,
  kI2C_TransferDirectionFlag = I2C_S_SRW_MASK,
  kI2C_RangeAddressMatchFlag = I2C_S_RAM_MASK,
  kI2C_ArbitrationLostFlag = I2C_S_ARBL_MASK,
  kI2C_BusBusyFlag = I2C_S_BUSY_MASK,
  kI2C_AddressMatchFlag = I2C_S_IAAS_MASK,

kI2C_TransferCompleteFlag = I2C_S_TCF_MASK }
I2C peripheral flags.
- enum _i2c_interrupt_enable { kI2C_GlobalInterruptEnable = I2C_C1_IICIE_MASK }
I2C feature interrupt source.
- enum i2c_direction_t {
kI2C_Write = 0x0U,
kI2C_Read = 0x1U }
The direction of master and slave transfers.
- enum i2c_slave_address_mode_t {
kI2C_Address7bit = 0x0U,
kI2C_RangeMatch = 0X2U }
Addressing mode.
- enum _i2c_master_transfer_flags {
kI2C_TransferDefaultFlag = 0x0U,
kI2C_TransferNoStartFlag = 0x1U,
kI2C_TransferRepeatedStartFlag = 0x2U,
kI2C_TransferNoStopFlag = 0x4U }
I2C transfer control flag.
- enum i2c_slave_transfer_event_t {
kI2C_SlaveAddressMatchEvent = 0x01U,
kI2C_SlaveTransmitEvent = 0x02U,
kI2C_SlaveReceiveEvent = 0x04U,
kI2C_SlaveTransmitAckEvent = 0x08U,
kI2C_SlaveCompletionEvent = 0x20U,
kI2C_SlaveAllEvents }
Set of events sent to the callback for nonblocking slave transfers.

Driver version

- #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
I2C driver version 2.0.2.

Initialization and deinitialization

- void I2C_MasterInit (I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
Initializes the I2C peripheral.
- void I2C_SlaveInit (I2C_Type ∗base, const i2c_slave_config_t ∗slaveConfig)
Initializes the I2C peripheral.
- void I2C_MasterDeinit (I2C_Type ∗base)
De-initializes the I2C master peripheral.
- void I2C_SlaveDeinit (I2C_Type ∗base)
De-initializes the I2C slave peripheral.
- void I2C_MasterGetDefaultConfig (i2c_master_config_t ∗masterConfig)
Sets the I2C master configuration structure to default values.
- void I2C_SlaveGetDefaultConfig (i2c_slave_config_t ∗slaveConfig)

Sets the I2C slave configuration structure to default values.
- static void I2C_Enable (I2C_Type *base, bool enable)
  Enables or disabless the I2C peripheral operation.

Status

- uint32_t I2C_MasterGetStatusFlags (I2C_Type *base)
  Gets the I2C status flags.
- static uint32_t I2C_SlaveGetStatusFlags (I2C_Type *base)
  Gets the I2C status flags.
- static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C status flag state.
- static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C status flag state.

Interrupts

- void I2C_EnableInterrupts (I2C_Type *base, uint32_t mask)
  Enables I2C interrupt requests.
- void I2C_DisableInterrupts (I2C_Type *base, uint32_t mask)
  Disables I2C interrupt requests.

DMA Control

- static uint32_t I2C_GetDataRegAddr (I2C_Type *base)
  Gets the I2C tx/rx data register address.

Bus Operations

- void I2C_MasterSetBaudRate (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the I2C master transfer baud rate.
- status_t I2C_MasterStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)
  Sends a START on the I2C bus.
- status_t I2C_MasterStop (I2C_Type *base)
  Sends a STOP signal on the I2C bus.
- status_t I2C_MasterRepeatedStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)
  Sends a REPEATED START on the I2C bus.
- status_t I2C_MasterWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
  Performs a polling send transaction on the I2C bus without a STOP signal.
- status_t I2C_MasterReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
  Performs a polling receive transaction on the I2C bus with a STOP signal.
- status_t I2C_SlaveWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
  Performs a polling send transaction on the I2C bus.
- void I2C_SlaveReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
  Performs a polling receive transaction on the I2C bus.
- status_t I2C_MasterTransferBlocking (I2C_Type *base, i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Transactional

- void I2C_MasterTransferCreateHandle (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_callback_t callback, void *userData)
  Initializes the I2C handle which is used in transactional functions.
- status_t I2C_MasterTransferNonBlocking (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_t *xfer)
  Performs a master interrupt non-blocking transfer on the I2C bus.
- status_t I2C_MasterTransferGetCount (I2C_Type *base, i2c_master_handle_t *handle, size_t *count)
  Gets the master transfer status during a interrupt non-blocking transfer.
- void I2C_MasterTransferAbort (I2C_Type *base, i2c_master_handle_t *handle)
  Aborts an interrupt non-blocking transfer early.
- void I2C_MasterTransferHandleIRQ (I2C_Type *base, void *i2cHandle)
  Master interrupt handler.
- void I2C_SlaveTransferCreateHandle (I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)
  Initializes the I2C handle which is used in transactional functions.
- status_t I2C_SlaveTransferNonBlocking (I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)
  Starts accepting slave transfers.
- void I2C_SlaveTransferAbort (I2C_Type *base, i2c_slave_handle_t *handle)
  Aborts the slave transfer.
- status_t I2C_SlaveTransferGetCount (I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
  Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.
- void I2C_SlaveTransferHandleIRQ (I2C_Type *base, void *i2cHandle)
  Slave interrupt handler.

23.2.3 Data Structure Documentation

23.2.3.1 struct i2c_master_config_t

Data Fields

- bool enableMaster
  Enables the I2C peripheral at initialization time.
- uint32_t baudRate_Bps
  Baud rate configuration of I2C peripheral.
- uint8_t glitchFilterWidth
  Controls the width of the glitch.

23.2.3.1.0.58 Field Documentation

23.2.3.1.0.58.1 bool i2c_master_config_t::enableMaster

23.2.3.1.0.58.2 uint32_t i2c_master_config_t::baudRate_Bps

23.2.3.1.0.58.3 uint8_t i2c_master_config_t::glitchFilterWidth

23.2.3.2 struct i2c_slave_config_t

Data Fields

- bool enableSlave
  Enables the I2C peripheral at initialization time.
- bool enableGeneralCall
  Enables the general call addressing mode.
- bool enableWakeUp
  Enables/disables waking up MCU from low-power mode.
- bool enableBaudRateCtl
  Enables/disables independent slave baud rate on SCL in very fast I2C modes.
- uint16_t slaveAddress
  A slave address configuration.
- uint16_t upperAddress
  A maximum boundary slave address used in a range matching mode.
- i2c_slave_address_mode_t addressingMode
  An addressing mode configuration of i2c_slave_address_mode_config_t.

23.2.3.2.0.59 Field Documentation

23.2.3.2.0.59.1 bool i2c_slave_config_t::enableSlave

23.2.3.2.0.59.2 bool i2c_slave_config_t::enableGeneralCall

23.2.3.2.0.59.3 bool i2c_slave_config_t::enableWakeUp

23.2.3.2.0.59.4 bool i2c_slave_config_t::enableBaudRateCtl

23.2.3.2.0.59.5 uint16_t i2c_slave_config_t::slaveAddress

23.2.3.2.0.59.6 uint16_t i2c_slave_config_t::upperAddress

23.2.3.2.0.59.7 i2c_slave_address_mode_t i2c_slave_config_t::addressingMode

23.2.3.3 struct i2c_master_transfer_t

Data Fields

- uint32_t flags
  A transfer flag which controls the transfer.
- uint8_t slaveAddress
  7-bit slave address.

- i2c_direction_t direction
  A transfer direction, read or write.
- uint32_t subaddress
  A sub address.
- uint8_t subaddressSize
  A size of the command buffer.
- uint8_t *volatile data
  A transfer buffer.
- volatile size_t dataSize
  A transfer size.

23.2.3.3.0.60 Field Documentation

23.2.3.3.0.60.1 uint32_t i2c_master_transfer_t::flags

23.2.3.3.0.60.2 uint8_t i2c_master_transfer_t::slaveAddress

23.2.3.3.0.60.3 i2c_direction_t i2c_master_transfer_t::direction

23.2.3.3.0.60.4 uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

23.2.3.3.0.60.5 uint8_t i2c_master_transfer_t::subaddressSize

23.2.3.3.0.60.6 uint8_t* volatile i2c_master_transfer_t::data

23.2.3.3.0.60.7 volatile size_t i2c_master_transfer_t::dataSize

23.2.3.4 struct _i2c_master_handle

I2C master handle typedef.

Data Fields

- i2c_master_transfer_t transfer
  I2C master transfer copy.
- size_t transferSize
  Total bytes to be transferred.
- uint8_t state
  A transfer state maintained during transfer.
- i2c_master_transfer_callback_t completionCallback
  A callback function called when the transfer is finished.
- void * userData
  A callback parameter passed to the callback function.

23.2.3.4.0.61 Field Documentation

23.2.3.4.0.61.1 i2c_master_transfer_t i2c_master_handle_t::transfer

23.2.3.4.0.61.2 size_t i2c_master_handle_t::transferSize

23.2.3.4.0.61.3 uint8_t i2c_master_handle_t::state

23.2.3.4.0.61.4 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

23.2.3.4.0.61.5 void* i2c_master_handle_t::userData

23.2.3.5 struct i2c_slave_transfer_t

Data Fields

- i2c_slave_transfer_event_t event
  A reason that the callback is invoked.
- uint8_t *volatile data
  A transfer buffer.
- volatile size_t dataSize
  A transfer size.
- status_t completionStatus
  Success or error code describing how the transfer completed.
- size_t transferredCount
  A number of bytes actually transferred since the start or since the last repeated start.

23.2.3.5.0.62 Field Documentation

23.2.3.5.0.62.1 i2c_slave_transfer_event_t i2c_slave_transfer_t::event

23.2.3.5.0.62.2 uint8_t* volatile i2c_slave_transfer_t::data

23.2.3.5.0.62.3 volatile size_t i2c_slave_transfer_t::dataSize

23.2.3.5.0.62.4 status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

23.2.3.5.0.62.5 size_t i2c_slave_transfer_t::transferredCount

23.2.3.6 struct _i2c_slave_handle

I2C slave handle typedef.

Data Fields

- bool isBusy
  Indicates whether a transfer is busy.
- i2c_slave_transfer_t transfer

I2C slave transfer copy.
- uint32_t eventMask
  A mask of enabled events.
- i2c_slave_transfer_callback_t callback
  A callback function called at the transfer event.
- void * userData
  A callback parameter passed to the callback.

23.2.3.6.0.63 Field Documentation

23.2.3.6.0.63.1 bool i2c_slave_handle_t::isBusy

23.2.3.6.0.63.2 i2c_slave_transfer_t i2c_slave_handle_t::transfer

23.2.3.6.0.63.3 uint32_t i2c_slave_handle_t::eventMask

23.2.3.6.0.63.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

23.2.3.6.0.63.5 void* i2c_slave_handle_t::userData

23.2.4 Macro Definition Documentation

23.2.4.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

23.2.5 Typedef Documentation

23.2.5.1 typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)

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

23.2.6 Enumeration Type Documentation

23.2.6.1 enum _i2c_status

Enumerator

kStatus_I2C_Busy I2C is busy with current transfer.
kStatus_I2C_Idle Bus is Idle.
kStatus_I2C_Nak NAK received during transfer.
kStatus_I2C_ArbitrationLost Arbitration lost during transfer.
kStatus_I2C_Timeout Wait event timeout.

23.2.6.2 enum _i2c_flags

The following status register flags can be cleared:

- kI2C_ArbitrationLostFlag
- kI2C_IntPendingFlag
- #kI2C_StartDetectFlag
- #kI2C_StopDetectFlag

Note

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

Enumerator

kI2C_ReceiveNakFlag I2C receive NAK flag.
kI2C_IntPendingFlag I2C interrupt pending flag.
kI2C_TransferDirectionFlag I2C transfer direction flag.
kI2C_RangeAddressMatchFlag I2C range address match flag.
kI2C_ArbitrationLostFlag I2C arbitration lost flag.
kI2C_BusBusyFlag I2C bus busy flag.
kI2C_AddressMatchFlag I2C address match flag.
kI2C_TransferCompleteFlag I2C transfer complete flag.

23.2.6.3 enum _i2c_interrupt_enable

Enumerator

kI2C_GlobalInterruptEnable I2C global interrupt.

23.2.6.4 enum i2c_direction_t

Enumerator

kI2C_Write Master transmits to the slave.
kI2C_Read Master receives from the slave.

23.2.6.5 enum i2c_slave_address_mode_t

Enumerator

kI2C_Address7bit 7-bit addressing mode.
kI2C_RangeMatch Range address match addressing mode.

23.2.6.6 enum _i2c_master_transfer_flags

Enumerator

kI2C_TransferDefaultFlag A transfer starts with a start signal, stops with a stop signal.
kI2C_TransferNoStartFlag A transfer starts without a start signal.
kI2C_TransferRepeatedStartFlag A transfer starts with a repeated start signal.
kI2C_TransferNoStopFlag A transfer ends without a stop signal.

23.2.6.7 enum i2c_slave_transfer_event_t

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

Note

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

Enumerator

kI2C_SlaveAddressMatchEvent Received the slave address after a start or repeated start.
kI2C_SlaveTransmitEvent A callback is requested to provide data to transmit (slave-transmitter role).
kI2C_SlaveReceiveEvent A callback is requested to provide a buffer in which to place received data (slave-receiver role).
kI2C_SlaveTransmitAckEvent A callback needs to either transmit an ACK or NACK.
kI2C_SlaveCompletionEvent A stop was detected or finished transfer, completing the transfer.
kI2C_SlaveAllEvents A bit mask of all available events.

23.2.7 Function Documentation

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

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

Note

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

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

Parameters

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

23.2.7.2 void I2C_SlaveInit (I2C_Type ∗ *base*, const i2c_slave_config_t ∗ *slaveConfig*)

Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note

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

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

Parameters

base	I2C base pointer
slaveConfig	A pointer to the slave configuration structure

23.2.7.3 void I2C_MasterDeinit (I2C_Type ∗ *base*)

Call this API to gate the I2C clock. The I2C master module can't work unless the I2C_MasterInit is called.

Parameters

base	I2C base pointer

23.2.7.4 void I2C_SlaveDeinit (I2C_Type ∗ *base*)

Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C_SlaveInit is called to enable the clock.

Parameters

base	I2C base pointer

23.2.7.5 void I2C_MasterGetDefaultConfig (i2c_master_config_t ∗ *masterConfig*)

The purpose of this API is to get the configuration structure initialized for use in the I2C_Master-Configure(). Use the initialized structure unchanged in the I2C_MasterConfigure() or modify the structure before calling the I2C_MasterConfigure(). Example:

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

Parameters

masterConfig	A pointer to the master configuration structure.

23.2.7.6 void I2C_SlaveGetDefaultConfig (i2c_slave_config_t ∗ *slaveConfig*)

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

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

Parameters

slaveConfig	A pointer to the slave configuration structure.

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

Parameters

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

23.2.7.8 uint32_t I2C_MasterGetStatusFlags (I2C_Type * *base*)

Parameters

base	I2C base pointer

Returns

status flag; use the status flag to AND _i2c_flags and get the related status.

23.2.7.9 static uint32_t I2C_SlaveGetStatusFlags (I2C_Type * *base*) [inline], [static]

Parameters

base	I2C base pointer

Returns

status flag; use the status flag to AND _i2c_flags and get the related status.

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

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

Parameters

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

23.2.7.11 static void I2C_SlaveClearStatusFlags (I2C_Type ∗ *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared: kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag

Parameters

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

23.2.7.12 void I2C_EnableInterrupts (I2C_Type ∗ *base*, uint32_t *mask*)

Parameters

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

23.2.7.13 void I2C_DisableInterrupts (I2C_Type ∗ *base*, uint32_t *mask*)

Parameters

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

23.2.7.14 static uint32_t I2C_GetDataRegAddr (I2C_Type ∗ *base*) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

base	I2C base pointer

Returns

data register address

23.2.7.15 void I2C_MasterSetBaudRate (I2C_Type ∗ *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz*)

Parameters

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

23.2.7.16 status_t I2C_MasterStart (I2C_Type ∗ *base,* uint8_t *address,* i2c_direction_t *direction*)

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

base	I2C peripheral base pointer
address	7-bit slave device address.
direction	Master transfer directions(transmit/receive).

Return values

kStatus_Success	Successfully send the start signal.
kStatus_I2C_Busy	Current bus is busy.

23.2.7.17 status_t I2C_MasterStop (I2C_Type ∗ *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

23.2.7.18 status_t I2C_MasterRepeatedStart (I2C_Type ∗ *base,* uint8_t *address,* i2c_direction_t *direction*)

Parameters

base	I2C peripheral base pointer
address	7-bit slave device address.
direction	Master transfer directions(transmit/receive).

Return values

kStatus_Success	Successfully send the start signal.
kStatus_I2C_Busy	Current bus is busy but not occupied by current I2C master.

23.2.7.19 status_t I2C_MasterWriteBlocking (I2C_Type ∗ *base,* const uint8_t ∗ *txBuff,* size_t *txSize*)

Parameters

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

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

23.2.7.20 status_t I2C_MasterReadBlocking (I2C_Type ∗ *base,* uint8_t ∗ *rxBuff,* size_t *rxSize*)

Note

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

Parameters

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

Return values

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

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

Parameters

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

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

23.2.7.22 void I2C_SlaveReadBlocking (I2C_Type ∗ *base,* uint8_t ∗ *rxBuff,* size_t *rxSize*)

Parameters

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

23.2.7.23 status_t I2C_MasterTransferBlocking (I2C_Type ∗ *base,* i2c_master_transfer_t ∗ *xfer*)

Note

The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

base	I2C peripheral base address.
xfer	Pointer to the transfer structure.

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Busy	Previous transmission still not finished.
kStatus_I2C_Timeout	Transfer error, wait signal timeout.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

23.2.7.24 void I2C_MasterTransferCreateHandle (I2C_Type * *base,* i2c_master_handle_t * *handle,* i2c_master_transfer_callback_t *callback,* void * *userData*)

Parameters

base	I2C base pointer.
handle	pointer to i2c_master_handle_t structure to store the transfer state.
callback	pointer to user callback function.
userData	user paramater passed to the callback function.

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

Note

Calling the API returns immediately after transfer initiates, user needs to call I2C_MasterGet-TransferCount to poll the transfer status to check whether the transfer is finished, if the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

base	I2C base pointer.
handle	pointer to i2c_master_handle_t structure which stores the transfer state.
xfer	pointer to i2c_master_transfer_t structure.

Return values

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

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

Parameters

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

Return values

kStatus_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

23.2.7.27 void I2C_MasterTransferAbort (I2C_Type * *base*, i2c_master_handle_t * *handle*)

Note

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

Parameters

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

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

Parameters

base	I2C base pointer.
i2cHandle	pointer to i2c_master_handle_t structure.

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

Parameters

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

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

Call this API after calling the I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The k-I2C_SlaveTransmitEvent and #kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
handle	Pointer to #i2c_slave_handle_t structure which stores the transfer state.
eventMask	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

#kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

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

Note

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

Parameters

base	I2C base pointer.
handle	pointer to i2c_slave_handle_t structure which stores the transfer state.

23.2.7.32 status_t I2C_SlaveTransferGetCount (I2C_Type * *base*, i2c_slave_handle_t * *handle*, size_t * *count*)

Parameters

base	I2C base pointer.
handle	pointer to i2c_slave_handle_t structure.
count	Number of bytes transferred so far by the non-blocking transaction.

Return values

kStatus_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

23.2.7.33 void I2C_SlaveTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

base	I2C base pointer.
i2cHandle	pointer to i2c_slave_handle_t structure which stores the transfer state

23.3 I2C eDMA Driver

23.3.1 Overview

Files

- file fsl_i2c_edma.h

Data Structures

- struct i2c_master_edma_handle_t
  I2C master eDMA transfer structure. More...

Typedefs

- typedef void(∗ i2c_master_edma_transfer_callback_t)(I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, status_t status, void ∗userData)
  I2C master eDMA transfer callback typedef.

I2C Block eDMA Transfer Operation

- void I2C_MasterCreateEDMAHandle (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, i2c_master_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗edmaHandle)
  Initializes the I2C handle which is used in transcational functions.
- status_t I2C_MasterTransferEDMA (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, i2c_master_transfer_t ∗xfer)
  Performs a master eDMA non-blocking transfer on the I2C bus.
- status_t I2C_MasterTransferGetCountEDMA (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, size_t ∗count)
  Gets a master transfer status during the eDMA non-blocking transfer.
- void I2C_MasterTransferAbortEDMA (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle)
  Aborts a master eDMA non-blocking transfer early.

23.3.2 Data Structure Documentation

23.3.2.1 struct _i2c_master_edma_handle

I2C master eDMA handle typedef.

Data Fields

- i2c_master_transfer_t transfer
  I2C master transfer structure.

- size_t transferSize
  Total bytes to be transferred.
- uint8_t state
  I2C master transfer status.
- edma_handle_t * dmaHandle
  The eDMA handler used.
- i2c_master_edma_transfer_callback_t completionCallback
  A callback function called after the eDMA transfer is finished.
- void * userData
  A callback parameter passed to the callback function.

23.3.2.1.0.64 Field Documentation

23.3.2.1.0.64.1 i2c_master_transfer_t i2c_master_edma_handle_t::transfer

23.3.2.1.0.64.2 size_t i2c_master_edma_handle_t::transferSize

23.3.2.1.0.64.3 uint8_t i2c_master_edma_handle_t::state

23.3.2.1.0.64.4 edma_handle_t* i2c_master_edma_handle_t::dmaHandle

23.3.2.1.0.64.5 i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completionCallback

23.3.2.1.0.64.6 void* i2c_master_edma_handle_t::userData

23.3.3 Typedef Documentation

23.3.3.1 typedef void(* i2c_master_edma_transfer_callback_t)(I2C_Type *base, i2c_master_edma_handle_t *handle, status_t status, void *userData)

23.3.4 Function Documentation

23.3.4.1 void I2C_MasterCreateEDMAHandle (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*, i2c_master_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *edmaHandle*)

Parameters

base	I2C peripheral base address.
handle	A pointer to the i2c_master_edma_handle_t structure.

callback	A pointer to the user callback function.
userData	A user parameter passed to the callback function.
edmaHandle	eDMA handle pointer.

23.3.4.2 status_t I2C_MasterTransferEDMA (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*, i2c_master_transfer_t * *xfer*)

Parameters

base	I2C peripheral base address.
handle	A pointer to the i2c_master_edma_handle_t structure.
xfer	A pointer to the transfer structure of i2c_master_transfer_t.

Return values

kStatus_Success	Sucessfully 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.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

23.3.4.3 status_t I2C_MasterTransferGetCountEDMA (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	I2C peripheral base address.
handle	A pointer to the i2c_master_edma_handle_t structure.
count	A number of bytes transferred by the non-blocking transaction.

23.3.4.4 void I2C_MasterTransferAbortEDMA (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*)

Parameters

base	I2C peripheral base address.
handle	A pointer to the i2c_master_edma_handle_t structure.

23.4 I2C DMA Driver

23.4.1 Overview

Files

- file fsl_i2c_dma.h

Data Structures

- struct i2c_master_dma_handle_t
  I2C master DMA transfer structure. More...

Typedefs

- typedef void(∗ i2c_master_dma_transfer_callback_t)(I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, status_t status, void ∗userData)
  I2C master DMA transfer callback typedef.

I2C Block DMA Transfer Operation

- void I2C_MasterTransferCreateHandleDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, i2c_master_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Initializes the I2C handle which is used in transcational functions.
- status_t I2C_MasterTransferDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, i2c_master_transfer_t ∗xfer)
  Performs a master DMA non-blocking transfer on the I2C bus.
- status_t I2C_MasterTransferGetCountDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, size_t ∗count)
  Gets a master transfer status during a DMA non-blocking transfer.
- void I2C_MasterTransferAbortDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle)
  Aborts a master DMA non-blocking transfer early.

23.4.2 Data Structure Documentation

23.4.2.1 struct _i2c_master_dma_handle

I2C master DMA handle typedef.

Data Fields

- i2c_master_transfer_t transfer
  I2C master transfer struct.

- size_t transferSize
  Total bytes to be transferred.
- uint8_t state
  I2C master transfer status.
- dma_handle_t * dmaHandle
  The DMA handler used.
- i2c_master_dma_transfer_callback_t completionCallback
  A callback function called after the DMA transfer finished.
- void * userData
  A callback parameter passed to the callback function.

23.4.2.1.0.65 Field Documentation

23.4.2.1.0.65.1 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

23.4.2.1.0.65.2 size_t i2c_master_dma_handle_t::transferSize

23.4.2.1.0.65.3 uint8_t i2c_master_dma_handle_t::state

23.4.2.1.0.65.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

23.4.2.1.0.65.5 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

23.4.2.1.0.65.6 void* i2c_master_dma_handle_t::userData

23.4.3 Typedef Documentation

23.4.3.1 typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)

23.4.4 Function Documentation

23.4.4.1 void I2C_MasterTransferCreateHandleDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*, i2c_master_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

Parameters

base	I2C peripheral base address
handle	Pointer to the i2c_master_dma_handle_t structure

callback	Pointer to the user callback function
userData	A user parameter passed to the callback function
dmaHandle	DMA handle pointer

23.4.4.2 status_t I2C_MasterTransferDMA (I2C_Type ∗ *base*, i2c_master_dma_handle_t ∗ *handle*, i2c_master_transfer_t ∗ *xfer*)

Parameters

base	I2C peripheral base address
handle	A pointer to the i2c_master_dma_handle_t structure
xfer	A pointer to the transfer structure of the i2c_master_transfer_t

Return values

kStatus_Success	Sucessfully completes the data transmission.
kStatus_I2C_Busy	A previous transmission is still not finished.
kStatus_I2C_Timeout	A transfer error, waits for the signal timeout.
kStatus_I2C_Arbitration-Lost	A transfer error, arbitration lost.
kStataus_I2C_Nak	A transfer error, receives NAK during transfer.

23.4.4.3 status_t I2C_MasterTransferGetCountDMA (I2C_Type ∗ *base*, i2c_master_dma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	I2C peripheral base address
handle	A pointer to the i2c_master_dma_handle_t structure
count	A number of bytes transferred so far by the non-blocking transaction.

23.4.4.4 void I2C_MasterTransferAbortDMA (I2C_Type ∗ *base*, i2c_master_dma_handle_t ∗ *handle*)

Parameters

base	I2C peripheral base address
handle	A pointer to the i2c_master_dma_handle_t structure.

23.5 I2C FreeRTOS Driver

23.5.1 Overview

Files

- file fsl_i2c_freertos.h

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t ∗handle, I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t ∗handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t ∗handle, i2c_master_transfer_t ∗transfer)
  Performs I2C transfer.

23.5.2 Data Structure Documentation

23.5.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type ∗ base
  I2C base address.
- i2c_master_handle_t drv_handle
  A handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  A mutex to lock the handle during a transfer.
- SemaphoreHandle_t sem
  A semaphore to notify and unblock task when the transfer ends.
- OS_EVENT ∗ mutex
  A mutex to lock the handle during a trasfer.
- OS_FLAG_GRP ∗ event
  A semaphore to notify and unblock a task when the transfer ends.
- OS_SEM mutex
  A mutex to lock the handle during a trasfer.
- OS_FLAG_GRP event
  A semaphore to notify and unblock a task when the transfer ends.

23.5.3 Function Documentation

23.5.3.1 status_t I2C_RTOS_Init (i2c_rtos_handle_t * *handle*, I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

Initializes the I2C.

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	The configuration structure to set-up I2C in master mode.
srcClock_Hz	The frequency of an input clock of the I2C module.

Returns

status of the operation.

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle; the pointer to an allocated space for the RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	A configuration structure to set-up the I2C in master mode.
srcClock_Hz	A frequency of the input clock of the I2C module.

Returns

status of the operation.

23.5.3.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle.

23.5.3.3 status_t I2C_RTOS_Transfer (i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)

Performs the I2C transfer.

This function performs an I2C transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

This function performs the I2C transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

23.6 I2C µCOS/II Driver

23.6.1 Overview

Files

- file fsl_i2c_ucosii.h
- file fsl_i2c_ucosiii.h

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t ∗handle, I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Initializes the I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t ∗handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t ∗handle, i2c_master_transfer_t ∗transfer)
  Performs the I2C transfer.

23.6.2 Data Structure Documentation

23.6.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type ∗ base
  I2C base address.
- i2c_master_handle_t drv_handle
  A handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  A mutex to lock the handle during a transfer.
- SemaphoreHandle_t sem
  A semaphore to notify and unblock task when the transfer ends.
- OS_EVENT ∗ mutex
  A mutex to lock the handle during a trasfer.
- OS_FLAG_GRP ∗ event
  A semaphore to notify and unblock a task when the transfer ends.
- OS_SEM mutex
  A mutex to lock the handle during a trasfer.
- OS_FLAG_GRP event
  A semaphore to notify and unblock a task when the transfer ends.

23.6.3 Function Documentation

23.6.3.1 status_t I2C_RTOS_Init (i2c_rtos_handle_t ∗ *handle*, I2C_Type ∗ *base*, const i2c_master_config_t ∗ *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle; the pointer to an allocated space for RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	A configuration structure to set-up the I2C in master mode.
srcClock_Hz	A frequency of the input clock of the I2C module.

Returns

status of the operation.

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle; the pointer to an allocated space for the RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	A configuration structure to set-up the I2C in master mode.
srcClock_Hz	A frequency of the input clock of the I2C module.

Returns

status of the operation.

23.6.3.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t ∗ *handle*)

This function deinitializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle.

23.6.3.3 status_t I2C_RTOS_Transfer (i2c_rtos_handle_t ∗ *handle,* i2c_master_transfer_t ∗ *transfer*)

This function performs the I2C transfer according to the data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	A structure specifying the transfer parameters.

Returns

status of the operation.

23.7 I2C µCOS/III Driver

Chapter 24
LLWU: Low-Leakage Wakeup Unit Driver

24.1 Overview

The KSDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of Kinetis devices. The LLWU module allows the user to select external pin sources and internal modules as a wake-up source from low-leakage power modes.

24.2 External wakeup pins configurations

Configures the external wakeup pins' working modes, gets, and clears the wake pin flags. External wakeup pins are accessed by the `pinIndex`, which is started from 1. Numbers of the external pins depend on the SoC configuration.

24.3 Internal wakeup modules configurations

Enables/disables the internal wakeup modules and gets the module flags. Internal modules are accessed by the `moduleIndex`, which is started from 1. Numbers of external pins depend the on SoC configuration.

24.4 Digital pin filter for external wakeup pin configurations

Configures the digital pin filter of the external wakeup pins' working modes, gets, and clears the pin filter flags. Digital pin filters are accessed by the `filterIndex`, which is started from 1. Numbers of external pins depend on the SoC configuration.

Files

- file fsl_llwu.h

Enumerations

- enum llwu_external_pin_mode_t {
  kLLWU_ExternalPinDisable = 0U,
  kLLWU_ExternalPinRisingEdge = 1U,
  kLLWU_ExternalPinFallingEdge = 2U,
  kLLWU_ExternalPinAnyEdge = 3U }
  External input pin control modes.
- enum llwu_pin_filter_mode_t {
  kLLWU_PinFilterDisable = 0U,
  kLLWU_PinFilterRisingEdge = 1U,
  kLLWU_PinFilterFallingEdge = 2U,
  kLLWU_PinFilterAnyEdge = 3U }
  Digital filter control modes.

Driver version

- #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  LLWU driver version 2.0.1.

24.5 Macro Definition Documentation

24.5.1 #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

24.6 Enumeration Type Documentation

24.6.1 enum llwu_external_pin_mode_t

Enumerator

kLLWU_ExternalPinDisable Pin disabled as a wakeup input.
kLLWU_ExternalPinRisingEdge Pin enabled with the rising edge detection.
kLLWU_ExternalPinFallingEdge Pin enabled with the falling edge detection.
kLLWU_ExternalPinAnyEdge Pin enabled with any change detection.

24.6.2 enum llwu_pin_filter_mode_t

Enumerator

kLLWU_PinFilterDisable Filter disabled.
kLLWU_PinFilterRisingEdge Filter positive edge detection.
kLLWU_PinFilterFallingEdge Filter negative edge detection.
kLLWU_PinFilterAnyEdge Filter any edge detection.

Chapter 25
LPTMR: Low-Power Timer

25.1 Overview

The KSDK provides a driver for the Low-Power Timer (LPTMR) of Kinetis devices.

25.2 Function groups

The LPTMR driver supports operating the module as a time counter or as a pulse counter.

25.2.1 Initialization and deinitialization

The function LPTMR_Init() initializes the LPTMR with specified configurations. The function LPTMR_-GetDefaultConfig() gets the default configurations. The initialization function configures the LPTMR for a timer or a pulse counter mode mode. It also sets up the LPTMR's free running mode operation and a clock source.

The function LPTMR_DeInit() disables the LPTMR module and gates the module clock.

25.2.2 Timer period Operations

The function LPTMR_SetTimerPeriod() sets the timer period in units of count. Timers counts from 0 to the count value set here.

The function LPTMR_GetCurrentTimerCount() reads the current timer counting value. This function returns the real-time timer counting value ranging from 0 to a timer period.

The timer period operation function takes the count value in ticks. Call the utility macros provided in the fsl_common.h file to convert to microseconds or milliseconds.

25.2.3 Start and Stop timer operations

The function LPTMR_StartTimer() starts the timer counting. After calling this function, the timer counts up to the counter value set earlier by using the LPTMR_SetPeriod() function. Each time the timer reaches the count value and increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

The function LPTMR_StopTimer() stops the timer counting and resets the timer's counter register.

25.2.4 Status

Provides functions to get and clear the LPTMR status.

25.2.5 Interrupt

Provides functions to enable/disable LPTMR interrupts and get the currently enabled interrupts.

25.3 Typical use case

25.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically.

```
int main(void)
{
    uint32_t currentCounter = 0U;
    lptmr_config_t lptmrConfig;

    LED_INIT();

    /* Board pin, clock, debug console initialization */
    BOARD_InitHardware();

    /* Configures the LPTMR */
    LPTMR_GetDefaultConfig(&lptmrConfig);

    /* Initializes the LPTMR */
    LPTMR_Init(LPTMR0, &lptmrConfig);

    /* Sets the timer period */
    LPTMR_SetTimerPeriod(LPTMR0, USEC_TO_COUNT(1000000U, LPTMR_SOURCE_CLOCK));

    /* Enables a timer interrupt */
    LPTMR_EnableInterrupts(LPTMR0,
      kLPTMR_TimerInterruptEnable);

    /* Enables the NVIC */
    EnableIRQ(LPTMR0_IRQn);

    PRINTF("Low Power Timer Example\r\n");

    /* Starts counting */
    LPTMR_StartTimer(LPTMR0);
    while (1)
    {
        if (currentCounter != lptmrCounter)
        {
            currentCounter = lptmrCounter;
            PRINTF("LPTMR interrupt No.%d \r\n", currentCounter);
        }
    }
}
```

Files

- file fsl_lptmr.h

Data Structures

- struct lptmr_config_t
  LPTMR config structure. More...

Enumerations

- enum lptmr_pin_select_t {
  kLPTMR_PinSelectInput_0 = 0x0U,
  kLPTMR_PinSelectInput_1 = 0x1U,
  kLPTMR_PinSelectInput_2 = 0x2U,
  kLPTMR_PinSelectInput_3 = 0x3U }
  LPTMR pin selection used in pulse counter mode.
- enum lptmr_pin_polarity_t {
  kLPTMR_PinPolarityActiveHigh = 0x0U,
  kLPTMR_PinPolarityActiveLow = 0x1U }
  LPTMR pin polarity used in pulse counter mode.
- enum lptmr_timer_mode_t {
  kLPTMR_TimerModeTimeCounter = 0x0U,
  kLPTMR_TimerModePulseCounter = 0x1U }
  LPTMR timer mode selection.
- enum lptmr_prescaler_glitch_value_t {
  kLPTMR_Prescale_Glitch_0 = 0x0U,
  kLPTMR_Prescale_Glitch_1 = 0x1U,
  kLPTMR_Prescale_Glitch_2 = 0x2U,
  kLPTMR_Prescale_Glitch_3 = 0x3U,
  kLPTMR_Prescale_Glitch_4 = 0x4U,
  kLPTMR_Prescale_Glitch_5 = 0x5U,
  kLPTMR_Prescale_Glitch_6 = 0x6U,
  kLPTMR_Prescale_Glitch_7 = 0x7U,
  kLPTMR_Prescale_Glitch_8 = 0x8U,
  kLPTMR_Prescale_Glitch_9 = 0x9U,
  kLPTMR_Prescale_Glitch_10 = 0xAU,
  kLPTMR_Prescale_Glitch_11 = 0xBU,
  kLPTMR_Prescale_Glitch_12 = 0xCU,
  kLPTMR_Prescale_Glitch_13 = 0xDU,
  kLPTMR_Prescale_Glitch_14 = 0xEU,
  kLPTMR_Prescale_Glitch_15 = 0xFU }
  LPTMR prescaler/glitch filter values.
- enum lptmr_prescaler_clock_select_t {
  kLPTMR_PrescalerClock_0 = 0x0U,
  kLPTMR_PrescalerClock_1 = 0x1U,
  kLPTMR_PrescalerClock_2 = 0x2U,
  kLPTMR_PrescalerClock_3 = 0x3U }
  LPTMR prescaler/glitch filter clock select.
- enum lptmr_interrupt_enable_t { kLPTMR_TimerInterruptEnable = LPTMR_CSR_TIE_MASK }
  List of the LPTMR interrupts.

- enum lptmr_status_flags_t { kLPTMR_TimerCompareFlag = LPTMR_CSR_TCF_MASK }
  List of the LPTMR status flags.

Driver version

- #define FSL_LPTMR_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

Initialization and deinitialization

- void LPTMR_Init (LPTMR_Type ∗base, const lptmr_config_t ∗config)
  Ungates the LPTMR clock and configures the peripheral for a basic operation.
- void LPTMR_Deinit (LPTMR_Type ∗base)
  Gates the LPTMR clock.
- void LPTMR_GetDefaultConfig (lptmr_config_t ∗config)
  Fills in the LPTMR configuration structure with default settings.

Interrupt Interface

- static void LPTMR_EnableInterrupts (LPTMR_Type ∗base, uint32_t mask)
  Enables the selected LPTMR interrupts.
- static void LPTMR_DisableInterrupts (LPTMR_Type ∗base, uint32_t mask)
  Disables the selected LPTMR interrupts.
- static uint32_t LPTMR_GetEnabledInterrupts (LPTMR_Type ∗base)
  Gets the enabled LPTMR interrupts.

Status Interface

- static uint32_t LPTMR_GetStatusFlags (LPTMR_Type ∗base)
  Gets the LPTMR status flags.
- static void LPTMR_ClearStatusFlags (LPTMR_Type ∗base, uint32_t mask)
  Clears the LPTMR status flags.

Read and write the timer period

- static void LPTMR_SetTimerPeriod (LPTMR_Type ∗base, uint16_t ticks)
  Sets the timer period in units of count.
- static uint16_t LPTMR_GetCurrentTimerCount (LPTMR_Type ∗base)
  Reads the current timer counting value.

Timer Start and Stop

- static void LPTMR_StartTimer (LPTMR_Type ∗base)
  Starts the timer.
- static void LPTMR_StopTimer (LPTMR_Type ∗base)
  Stops the timer.

25.4 Data Structure Documentation

25.4.1 struct lptmr_config_t

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

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

Data Fields

- lptmr_timer_mode_t timerMode
  Time counter mode or pulse counter mode.
- lptmr_pin_select_t pinSelect
  LPTMR pulse input pin select; used only in pulse counter mode.
- lptmr_pin_polarity_t pinPolarity
  LPTMR pulse input pin polarity; used only in pulse counter mode.
- bool enableFreeRunning
  True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set.
- bool bypassPrescaler
  True: bypass prescaler; false: use clock from prescaler.
- lptmr_prescaler_clock_select_t prescalerClockSource
  LPTMR clock source.
- lptmr_prescaler_glitch_value_t value
  Prescaler or glitch filter value.

25.5 Enumeration Type Documentation

25.5.1 enum lptmr_pin_select_t

Enumerator

kLPTMR_PinSelectInput_0 Pulse counter input 0 is selected.
kLPTMR_PinSelectInput_1 Pulse counter input 1 is selected.
kLPTMR_PinSelectInput_2 Pulse counter input 2 is selected.
kLPTMR_PinSelectInput_3 Pulse counter input 3 is selected.

25.5.2 enum lptmr_pin_polarity_t

Enumerator

kLPTMR_PinPolarityActiveHigh Pulse Counter input source is active-high.
kLPTMR_PinPolarityActiveLow Pulse Counter input source is active-low.

25.5.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.
kLPTMR_TimerModePulseCounter Pulse Counter mode.

25.5.4 enum lptmr_prescaler_glitch_value_t

Enumerator

kLPTMR_Prescale_Glitch_0 Prescaler divide 2, glitch filter does not support this setting.
kLPTMR_Prescale_Glitch_1 Prescaler divide 4, glitch filter 2.
kLPTMR_Prescale_Glitch_2 Prescaler divide 8, glitch filter 4.
kLPTMR_Prescale_Glitch_3 Prescaler divide 16, glitch filter 8.
kLPTMR_Prescale_Glitch_4 Prescaler divide 32, glitch filter 16.
kLPTMR_Prescale_Glitch_5 Prescaler divide 64, glitch filter 32.
kLPTMR_Prescale_Glitch_6 Prescaler divide 128, glitch filter 64.
kLPTMR_Prescale_Glitch_7 Prescaler divide 256, glitch filter 128.
kLPTMR_Prescale_Glitch_8 Prescaler divide 512, glitch filter 256.
kLPTMR_Prescale_Glitch_9 Prescaler divide 1024, glitch filter 512.
kLPTMR_Prescale_Glitch_10 Prescaler divide 2048 glitch filter 1024.
kLPTMR_Prescale_Glitch_11 Prescaler divide 4096, glitch filter 2048.
kLPTMR_Prescale_Glitch_12 Prescaler divide 8192, glitch filter 4096.
kLPTMR_Prescale_Glitch_13 Prescaler divide 16384, glitch filter 8192.
kLPTMR_Prescale_Glitch_14 Prescaler divide 32768, glitch filter 16384.
kLPTMR_Prescale_Glitch_15 Prescaler divide 65536, glitch filter 32768.

25.5.5 enum lptmr_prescaler_clock_select_t

Note

Clock connections are SoC-specific

Enumerator

kLPTMR_PrescalerClock_0 Prescaler/glitch filter clock 0 selected.
kLPTMR_PrescalerClock_1 Prescaler/glitch filter clock 1 selected.
kLPTMR_PrescalerClock_2 Prescaler/glitch filter clock 2 selected.
kLPTMR_PrescalerClock_3 Prescaler/glitch filter clock 3 selected.

25.5.6 enum lptmr_interrupt_enable_t

Enumerator

kLPTMR_TimerInterruptEnable Timer interrupt enable.

25.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

25.6 Function Documentation

25.6.1 void LPTMR_Init (LPTMR_Type ∗ *base*, const lptmr_config_t ∗ *config*)

Note

This API should be called at the beginning of the application using the LPTMR driver.

Parameters

base	LPTMR peripheral base address
config	A pointer to the LPTMR configuration structure.

25.6.2 void LPTMR_Deinit (LPTMR_Type ∗ *base*)

Parameters

base	LPTMR peripheral base address

25.6.3 void LPTMR_GetDefaultConfig (lptmr_config_t ∗ *config*)

The default values are:

```
*    config->timerMode = kLPTMR_TimerModeTimeCounter;
*    config->pinSelect = kLPTMR_PinSelectInput_0;
*    config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
*    config->enableFreeRunning = false;
*    config->bypassPrescaler = true;
*    config->prescalerClockSource = kLPTMR_PrescalerClock_1;
*    config->value = kLPTMR_Prescale_Glitch_0;
*
```

Parameters

config	A pointer to the LPTMR configuration structure.

25.6.4 static void LPTMR_EnableInterrupts (LPTMR_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	LPTMR peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration lptmr-_interrupt_enable_t

25.6.5 static void LPTMR_DisableInterrupts (LPTMR_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	LPTMR peripheral base address
mask	The interrupts to disable. This is a logical OR of members of the enumeration lptmr-_interrupt_enable_t

25.6.6 static uint32_t LPTMR_GetEnabledInterrupts (LPTMR_Type * *base*) [inline], [static]

Parameters

base	LPTMR peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration lptmr_interrupt_-enable_t

25.6.7 static uint32_t LPTMR_GetStatusFlags (LPTMR_Type * *base*) [inline], [static]

Parameters

base	LPTMR peripheral base address

Returns

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

25.6.8 static void LPTMR_ClearStatusFlags (LPTMR_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	LPTMR peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration lptmr_-status_flags_t

25.6.9 static void LPTMR_SetTimerPeriod (LPTMR_Type * *base*, uint16_t *ticks*) [inline], [static]

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

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the fsl_common.h to convert to ticks.

Parameters

base	LPTMR peripheral base address
ticks	A timer period in units of ticks

25.6.10 static uint16_t LPTMR_GetCurrentTimerCount (LPTMR_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

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

Parameters

base	LPTMR peripheral base address

Returns

The current counter value in ticks

25.6.11 static void LPTMR_StartTimer (LPTMR_Type * *base*) [inline], [static]

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

base	LPTMR peripheral base address

25.6.12 static void LPTMR_StopTimer (LPTMR_Type * *base*) [inline], [static]

This function stops the timer and resets the timer's counter register.

Parameters

base	LPTMR peripheral base address

Chapter 26
MPU: Memory Protection Unit

26.1 Overview

The MPU driver provides hardware access control for all memory references generated in the device. Use the MPU driver to program the region descriptors that define memory spaces and their access rights. After initialization, the MPU concurrently monitors the system bus transactions and evaluates the appropriateness.

26.2 Initialization and Deinitialize

To initialize the MPU module, call the MPU_Init() function and provide the user configuration data structure. This function sets the configuration of the MPU module automatically and enables the MPU module.

Note that the configuration start address, end address, the region valid value, and the debugger's access permission for the MPU region 0 cannot be changed.

This is example code to configure the MPU driver:

```
 // Defines the MPU memory access permission configuration structure . //
  mpu_rwxrights_master_access_control_t mpuRwxAccessRightsMasters =
 {
      kMPU_SupervisorReadWriteExecute,
      kMPU_UserNoAccessRights,
      kMPU_IdentifierDisable,
      kMPU_SupervisorEqualToUsermode,
      kMPU_UserNoAccessRights,
      kMPU_IdentifierDisable,
      kMPU_SupervisorEqualToUsermode,
      kMPU_UserNoAccessRights,
      kMPU_IdentifierDisable,
      kMPU_SupervisorEqualToUsermode,
      kMPU_UserNoAccessRights,
      kMPU_IdentifierDisable
 }
  mpu_rwrights_master_access_control_t mpuRwAccessRightsMasters =
 {
       false,
       false,
       false,
       false,
       false,
       false,
       false,
       false
};

 // Defines the MPU region configuration structure. //
  mpu_region_config_t mpuRegionConfig =
 {
      0,
      0x0,
      0xffffffff,
      mpuRwxAccessRightsMasters,
      mpuRwAccessRightsMasters,
```

```
        0,
        0
};

 // Defines the MPU user configuration structure. //
 mpu_config_t mpuUserConfig =
 {
        mpuRegionConfig,
        NULL
 };

 // Initializes the MPU region 0. //
 MPU_Init(MPU, &mpuUserConfig);
```

26.3 Basic Control Operations

MPU can be enabled/disabled for the entire memory protection region by calling the MPU_Enable(). To save the power for any unused special regions when the entire memory protection region is disabled, call the MPU_RegionEnable().

After MPU initialization, the MPU_SetRegionLowMasterAccessRights() and MPU_SetRegionHighMasterAccessRights() can be used to change the access rights for special master ports and for special region numbers. The MPU_SetRegionConfig can be used to set the whole region with the start/end address with access rights.

The MPU_GetHardwareInfo() API is provided to get the hardware information for the device. The MPU_GetSlavePortErrorStatus() API is provided to get the error status of a special slave port. When an error happens in this port, the MPU_GetDetailErrorAccessInfo() API is provided to get the detailed error information.

Files

- file fsl_mpu.h

Data Structures

- struct mpu_hardware_info_t
  MPU hardware basic information. More...
- struct mpu_access_err_info_t
  MPU detail error access information. More...
- struct mpu_rwxrights_master_access_control_t
  MPU read/write/execute rights control for bus master 0 ~ 3. More...
- struct mpu_rwrights_master_access_control_t
  MPU read/write access control for bus master 4 ~ 7. More...
- struct mpu_region_config_t
  MPU region configuration structure. More...
- struct mpu_config_t
  The configuration structure for the MPU initialization. More...

Macros

- #define MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n) (n ∗ 6)
  MPU the bit shift for masters with privilege rights: read write and execute.

- #define MPU_REGION_RWXRIGHTS_MASTER_MASK(n) (0x1Fu << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))
  MPU masters with read, write and execute rights bit mask.
- #define MPU_REGION_RWXRIGHTS_MASTER_WIDTH 5
  MPU masters with read, write and execute rights bit width.
- #define MPU_REGION_RWXRIGHTS_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_MASK(n))
  MPU masters with read, write and execute rights priority setting.
- #define MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n) (n ∗ 6 + MPU_REGION_RWXRIGHTS_MASTER_WIDTH)
  MPU masters with read, write and execute rights process enable bit shift.
- #define MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n) (0x1u << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))
  MPU masters with read, write and execute rights process enable bit mask.
- #define MPU_REGION_RWXRIGHTS_MASTER_PE(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n))
  MPU masters with read, write and execute rights process enable setting.
- #define MPU_REGION_RWRIGHTS_MASTER_SHIFT(n) ((n - FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_COUNT) ∗ 2 + 24)
  MPU masters with normal read write permission bit shift.
- #define MPU_REGION_RWRIGHTS_MASTER_MASK(n) (0x3u << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))
  MPU masters with normal read write rights bit mask.
- #define MPU_REGION_RWRIGHTS_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWRIGHTS_MASTER_MASK(n))
  MPU masters with normal read write rights priority setting.
- #define MPU_SLAVE_PORT_NUM (4u)
  the Slave port numbers.

Enumerations

- enum mpu_region_total_num_t {
  kMPU_8Regions = 0x0U,
  kMPU_12Regions = 0x1U,
  kMPU_16Regions = 0x2U }
  Describes the number of MPU regions.
- enum mpu_slave_t {
  kMPU_Slave0 = 0U,
  kMPU_Slave1 = 1U,
  kMPU_Slave2 = 2U,
  kMPU_Slave3 = 3U,
  kMPU_Slave4 = 4U }
  MPU slave port number.
- enum mpu_err_access_control_t {

kMPU_NoRegionHit = 0U,
kMPU_NoneOverlappRegion = 1U,
kMPU_OverlappRegion = 2U }
MPU error access control detail.
- enum mpu_err_access_type_t {
kMPU_ErrTypeRead = 0U,
kMPU_ErrTypeWrite = 1U }
MPU error access type.
- enum mpu_err_attributes_t {
kMPU_InstructionAccessInUserMode = 0U,
kMPU_DataAccessInUserMode = 1U,
kMPU_InstructionAccessInSupervisorMode = 2U,
kMPU_DataAccessInSupervisorMode = 3U }
MPU access error attributes.
- enum mpu_supervisor_access_rights_t {
kMPU_SupervisorReadWriteExecute = 0U,
kMPU_SupervisorReadExecute = 1U,
kMPU_SupervisorReadWrite = 2U,
kMPU_SupervisorEqualToUsermode = 3U }
MPU access rights in supervisor mode for bus master 0 ∼ 3.
- enum mpu_user_access_rights_t {
kMPU_UserNoAccessRights = 0U,
kMPU_UserExecute = 1U,
kMPU_UserWrite = 2U,
kMPU_UserWriteExecute = 3U,
kMPU_UserRead = 4U,
kMPU_UserReadExecute = 5U,
kMPU_UserReadWrite = 6U,
kMPU_UserReadWriteExecute = 7U }
MPU access rights in user mode for bus master 0 ∼ 3.

Driver version

- #define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
MPU driver version 2.1.0.

Initialization and deinitialization

- void MPU_Init (MPU_Type ∗base, const mpu_config_t ∗config)
Initializes the MPU with the user configuration structure.
- void MPU_Deinit (MPU_Type ∗base)
Deinitializes the MPU regions.

Basic Control Operations

- static void MPU_Enable (MPU_Type ∗base, bool enable)
Enables/disables the MPU globally.
- static void MPU_RegionEnable (MPU_Type ∗base, uint32_t number, bool enable)

Enables/disables the MPU for a special region.
- void MPU_GetHardwareInfo (MPU_Type ∗base, mpu_hardware_info_t ∗hardwareInform)
  Gets the MPU basic hardware information.
- void MPU_SetRegionConfig (MPU_Type ∗base, const mpu_region_config_t ∗regionConfig)
  Sets the MPU region.
- void MPU_SetRegionAddr (MPU_Type ∗base, uint32_t regionNum, uint32_t startAddr, uint32_t endAddr)
  Sets the region start and end address.
- void MPU_SetRegionRwxMasterAccessRights (MPU_Type ∗base, uint32_t regionNum, uint32_t masterNum, const mpu_rwxrights_master_access_control_t ∗accessRights)
  Sets the MPU region access rights for masters with read, write and execute rights.
- void MPU_SetRegionRwMasterAccessRights (MPU_Type ∗base, uint32_t regionNum, uint32_t masterNum, const mpu_rwrights_master_access_control_t ∗accessRights)
  Sets the MPU region access rights for masters with read and write rights.
- bool MPU_GetSlavePortErrorStatus (MPU_Type ∗base, mpu_slave_t slaveNum)
  Gets the numbers of slave ports where errors occur.
- void MPU_GetDetailErrorAccessInfo (MPU_Type ∗base, mpu_slave_t slaveNum, mpu_access_err_info_t ∗errInform)
  Gets the MPU detailed error access information.

26.4 Data Structure Documentation

26.4.1 struct mpu_hardware_info_t

Data Fields

- uint8_t hardwareRevisionLevel
  Specifies the MPU's hardware and definition reversion level.
- uint8_t slavePortsNumbers
  Specifies the number of slave ports connected to MPU.
- mpu_region_total_num_t regionsNumbers
  Indicates the number of region descriptors implemented.

26.4.1.0.0.66 Field Documentation

26.4.1.0.0.66.1 uint8_t mpu_hardware_info_t::hardwareRevisionLevel

26.4.1.0.0.66.2 uint8_t mpu_hardware_info_t::slavePortsNumbers

26.4.1.0.0.66.3 mpu_region_total_num_t mpu_hardware_info_t::regionsNumbers

26.4.2 struct mpu_access_err_info_t

Data Fields

- uint32_t master
  Access error master.
- mpu_err_attributes_t attributes
  Access error attributes.

- mpu_err_access_type_t accessType
  Access error type.
- mpu_err_access_control_t accessControl
  Access error control.
- uint32_t address
  Access error address.

26.4.2.0.0.67 Field Documentation

26.4.2.0.0.67.1 uint32_t mpu_access_err_info_t::master

26.4.2.0.0.67.2 mpu_err_attributes_t mpu_access_err_info_t::attributes

26.4.2.0.0.67.3 mpu_err_access_type_t mpu_access_err_info_t::accessType

26.4.2.0.0.67.4 mpu_err_access_control_t mpu_access_err_info_t::accessControl

26.4.2.0.0.67.5 uint32_t mpu_access_err_info_t::address

26.4.3 struct mpu_rwxrights_master_access_control_t

Data Fields

- mpu_supervisor_access_rights_t superAccessRights
  Master access rights in supervisor mode.
- mpu_user_access_rights_t userAccessRights
  Master access rights in user mode.

26.4.3.0.0.68 Field Documentation

26.4.3.0.0.68.1 mpu_supervisor_access_rights_t mpu_rwxrights_master_access_control_t::superAccessRights

26.4.3.0.0.68.2 mpu_user_access_rights_t mpu_rwxrights_master_access_control_t::userAccessRights

26.4.4 struct mpu_rwrights_master_access_control_t

Data Fields

- bool writeEnable
  Enables or disables write permission.
- bool readEnable
  Enables or disables read permission.

26.4.4.0.0.69 Field Documentation

26.4.4.0.0.69.1 bool mpu_rwrights_master_access_control_t::writeEnable

26.4.4.0.0.69.2 bool mpu_rwrights_master_access_control_t::readEnable

26.4.5 struct mpu_region_config_t

This structure is used to configure the regionNum region. The accessRights1[0] ∼ accessRights1[3] are used to configure the bus master 0 ∼ 3 with the privilege rights setting. The accessRights2[0] ∼ accessRights2[3] are used to configure the high master 4 ∼ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note: MPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. MPU protection does not allow writes from the core to affect the "regionNum 0" start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantee the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

Data Fields

- uint32_t regionNum
  MPU region number, range form 0 ∼ FSL_FEATURE_MPU_DESCRIPTOR_COUNT - 1.
- uint32_t startAddress
  Memory region start address.
- uint32_t endAddress
  Memory region end address.
- mpu_rwxrights_master_access_control_t accessRights1 [4]
  Masters with read, write and execute rights setting.
- mpu_rwrights_master_access_control_t accessRights2 [4]
  Masters with normal read write rights setting.

26.4.5.0.0.70 Field Documentation

26.4.5.0.0.70.1 uint32_t mpu_region_config_t::regionNum

26.4.5.0.0.70.2 uint32_t mpu_region_config_t::startAddress

Note: bit0 ∼ bit4 always be marked as 0 by MPU. The actual start address is 0-modulo-32 byte address.

26.4.5.0.0.70.3 uint32_t mpu_region_config_t::endAddress

Note: bit0 ∼ bit4 always be marked as 1 by MPU. The actual end address is 31-modulo-32 byte address.

26.4.5.0.0.70.4 mpu_rwxrights_master_access_control_t mpu_region_config_t::accessRights1[4]

26.4.5.0.0.70.5 mpu_rwrights_master_access_control_t mpu_region_config_t::accessRights2[4]

26.4.6 struct mpu_config_t

This structure is used when calling the MPU_Init function.

Data Fields

- mpu_region_config_t regionConfig
  region access permission.
- struct _mpu_config ∗ next
  pointer to the next structure.

26.4.6.0.0.71 Field Documentation

26.4.6.0.0.71.1 mpu_region_config_t mpu_config_t::regionConfig

26.4.6.0.0.71.2 struct _mpu_config∗ mpu_config_t::next

26.5 Macro Definition Documentation

26.5.1 #define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

26.5.2 #define MPU_REGION_RWXRIGHTS_MASTER_SHIFT(*n*) (n ∗ 6)

26.5.3 #define MPU_REGION_RWXRIGHTS_MASTER_MASK(*n*) (0x1Fu << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))

26.5.4 #define MPU_REGION_RWXRIGHTS_MASTER_WIDTH 5

26.5.5 #define MPU_REGION_RWXRIGHTS_MASTER(*n, x*) (((uint32_t)(((uint32-_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_MASK(n))

26.5.6 #define MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(*n*) (n ∗ 6 + MPU_REGION_RWXRIGHTS_MASTER_WIDTH)

26.5.7 #define MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(*n*) (0x1u << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))

26.5.8 #define MPU_REGION_RWXRIGHTS_MASTER_PE(*n, x*) (((uint32_-t)(((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIF-T(n))) & MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n))

26.5.9 #define MPU_REGION_RWRIGHTS_MASTER_SHIFT(*n*) ((n - FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_COUNT) ∗ 2 + 24)

26.5.10 #define MPU_REGION_RWRIGHTS_MASTER_MASK(*n*) (0x3u << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))

26.5.11 #define MPU_REGION_RWRIGHTS_MASTER(*n, x*) (((uint32_t)(((uint32-_t)(x)) << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWRIGHTS_MASTER_MASK(n))

26.5.12 #define MPU_SLAVE_PORT_NUM (4u)

26.6 Enumeration Type Documentation

26.6.1 enum mpu_region_total_num_t

Enumerator

kMPU_8Regions MPU supports 8 regions.
kMPU_12Regions MPU supports 12 regions.
kMPU_16Regions MPU supports 16 regions.

26.6.2 enum mpu_slave_t

Enumerator

kMPU_Slave0 MPU slave port 0.
kMPU_Slave1 MPU slave port 1.
kMPU_Slave2 MPU slave port 2.
kMPU_Slave3 MPU slave port 3.
kMPU_Slave4 MPU slave port 4.

26.6.3 enum mpu_err_access_control_t

Enumerator

kMPU_NoRegionHit No region hit error.
kMPU_NoneOverlappRegion Access single region error.
kMPU_OverlappRegion Access overlapping region error.

26.6.4 enum mpu_err_access_type_t

Enumerator

kMPU_ErrTypeRead MPU error access type — read.
kMPU_ErrTypeWrite MPU error access type — write.

26.6.5 enum mpu_err_attributes_t

Enumerator

kMPU_InstructionAccessInUserMode Access instruction error in user mode.
kMPU_DataAccessInUserMode Access data error in user mode.
kMPU_InstructionAccessInSupervisorMode Access instruction error in supervisor mode.
kMPU_DataAccessInSupervisorMode Access data error in supervisor mode.

26.6.6 enum mpu_supervisor_access_rights_t

Enumerator

kMPU_SupervisorReadWriteExecute Read write and execute operations are allowed in supervisor mode.
kMPU_SupervisorReadExecute Read and execute operations are allowed in supervisor mode.
kMPU_SupervisorReadWrite Read write operations are allowed in supervisor mode.
kMPU_SupervisorEqualToUsermode Access permission equal to user mode.

26.6.7 enum mpu_user_access_rights_t

Enumerator

kMPU_UserNoAccessRights No access allowed in user mode.
kMPU_UserExecute Execute operation is allowed in user mode.
kMPU_UserWrite Write operation is allowed in user mode.
kMPU_UserWriteExecute Write and execute operations are allowed in user mode.
kMPU_UserRead Read is allowed in user mode.
kMPU_UserReadExecute Read and execute operations are allowed in user mode.
kMPU_UserReadWrite Read and write operations are allowed in user mode.
kMPU_UserReadWriteExecute Read write and execute operations are allowed in user mode.

26.7 Function Documentation

26.7.1 void MPU_Init (MPU_Type ∗ *base*, const mpu_config_t ∗ *config*)

This function configures the MPU module with the user-defined configuration.

Parameters

base	MPU peripheral base address.
config	The pointer to the configuration structure.

26.7.2 void MPU_Deinit (MPU_Type ∗ *base*)

Parameters

base	MPU peripheral base address.

26.7.3 static void MPU_Enable (MPU_Type ∗ *base,* bool *enable*) [inline], [static]

Call this API to enable or disable the MPU module.

Parameters

base	MPU peripheral base address.
enable	True enable MPU, false disable MPU.

26.7.4 static void MPU_RegionEnable (MPU_Type ∗ *base,* uint32_t *number,* bool *enable*) [inline], [static]

When MPU is enabled, call this API to disable an unused region of an enabled MPU. Call this API to minimize the power dissipation.

Parameters

base	MPU peripheral base address.
number	MPU region number.
enable	True enable the special region MPU, false disable the special region MPU.

26.7.5 void MPU_GetHardwareInfo (MPU_Type ∗ *base,* mpu_hardware_info_t ∗ *hardwareInform*)

Parameters

base	MPU peripheral base address.
hardware-Inform	The pointer to the MPU hardware information structure. See "mpu_hardware_info_-t".

26.7.6 void MPU_SetRegionConfig (MPU_Type ∗ *base*, const mpu_region_config_t ∗ *regionConfig*)

Note: Due to the MPU protection, the Region number 0 does not allow writes from core to affect the start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters.

Parameters

base	MPU peripheral base address.
regionConfig	The pointer to the MPU user configuration structure. See "mpu_region_config_t".

26.7.7 void MPU_SetRegionAddr (MPU_Type ∗ *base*, uint32_t *regionNum*, uint32_t *startAddr*, uint32_t *endAddr*)

Memory region start address. Note: bit0 $\sim$ bit4 is always marked as 0 by MPU. The actual start address by MPU is 0-modulo-32 byte address. Memory region end address. Note: bit0 $\sim$ bit4 always be marked as 1 by MPU. The actual end address used by MPU is 31-modulo-32 byte address. Note: Due to the MPU protection, the startAddr and endAddr can't be changed by the core when regionNum is 0.

Parameters

base	MPU peripheral base address.
regionNum	MPU region number. The range is from 0 to FSL_FEATURE_MPU_DESCRIPTO-R_COUNT - 1.
startAddr	Region start address.
endAddr	Region end address.

26.7.8 void MPU_SetRegionRwxMasterAccessRights (MPU_Type ∗ *base*, uint32_t *regionNum*, uint32_t *masterNum*, const mpu_rwxrights_master_access_control_t ∗ *accessRights*)

The MPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The privilege rights masters have the read, write and execute access rights. So except the normal read and write rights, the execute rights is also allowed for these masters. The privilege rights masters are normally range from bus masters 0 - 3. However, the maximum master number is device-specific. See the "FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDE-X". The normal rights masters access rights control see "MPU_SetRegionRwMasterAccessRights()".

Parameters

base	MPU peripheral base address.
regionNum	MPU region number. Should range from 0 to FSL_FEATURE_MPU_DESCRIPTO-R_COUNT - 1.
masterNum	MPU bus master number. Should range from 0 to FSL_FEATURE_MPU_PRIVIL-EGED_RIGHTS_MASTER_MAX_INDEX.
accessRights	The pointer to the MPU access rights configuration. See "mpu_rwxrights_master_-access_control_t".

26.7.9 void MPU_SetRegionRwMasterAccessRights (MPU_Type * *base*, uint32_t *regionNum*, uint32_t *masterNum*, const mpu_rwrights_master_access_control_t * *accessRights*)

The MPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The normal rights masters only have the read and write access permissions. The privilege rights access control see "MPU_SetRegionRwxMasterAccessRights".

Parameters

base	MPU peripheral base address.
regionNum	MPU region number. The range is from 0 to FSL_FEATURE_MPU_DESCRIPTO-R_COUNT - 1.
masterNum	MPU bus master number. Should range from FSL_FEATURE_MPU_PRIVILEG-ED_RIGHTS_MASTER_COUNT to ~ FSL_FEATURE_MPU_MASTER_MAX_-INDEX.
accessRights	The pointer to the MPU access rights configuration. See "mpu_rwrights_master_-access_control_t".

26.7.10 bool MPU_GetSlavePortErrorStatus (MPU_Type * *base*, mpu_slave_t *slaveNum*)

Parameters

base	MPU peripheral base address.
slaveNum	MPU slave port number.

Returns

The slave ports error status. true - error happens in this slave port. false - error didn't happen in this slave port.

26.7.11 void MPU_GetDetailErrorAccessInfo (MPU_Type ∗ *base*, mpu_slave_t *slaveNum*, mpu_access_err_info_t ∗ *errInform*)

Parameters

base	MPU peripheral base address.
slaveNum	MPU slave port number.
errInform	The pointer to the MPU access error information. See "mpu_access_err_info_t".

Chapter 27
PDB: Programmable Delay Block

27.1 Overview

The KSDK provides a peripheral driver for the Programmable Delay Block (PDB) module of Kinetis devices.

The PDB driver includes a basic PDB counter, trigger generators for ADC, DAC, and pulse-out.

The basic PDB counter can be used as a general programmable timer with an interrupt. The counter increases automatically with the divided clock signal after it is triggered to start by an external trigger input or the software trigger. There are "milestones" for output trigger event. When the counter is equal to any of these "milestones", the corresponding trigger is generated and sent out to other modules. These "milestones" are for the following:

- Counter delay interrupt, which is the interrupt for the PDB module
- ADC pre-trigger to trigger the ADC conversion
- DAC interval trigger to trigger the DAC buffer and move the buffer read pointer
- Pulse-out triggers to generate a single of rising and falling edges, which can be assembled to a window.

The "milestone" values have a flexible load mode. To call the APIs to set these value is equivalent to writing data to their buffer. The loading event occurs as the load mode describes. This design ensures that all "milestones" can be updated at the same time.

27.2 Typical use case

27.2.1 Working as basic DPB counter with a PDB interrupt.

```
int main(void)
{
    // ...
    EnableIRQ(DEMO_PDB_IRQ_ID);

    // ...
    // Configures the PDB counter.
    PDB_GetDefaultConfig(&pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
       value is less than or equal to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
      kPDB_DelayInterruptEnable);
    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    while (1)
    {
        // ...
        g_PdbDelayInterruptFlag = false;
```

```
        PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
        while (!g_PdbDelayInterruptFlag)
        {
        }
    }
 }

void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    // ...
    g_PdbDelayInterruptFlag = true;
    PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
      kPDB_DelayEventFlag);
}
```

27.2.2 Working with an additional trigger. The ADC trigger is used as an example.

```
void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
      kPDB_DelayEventFlag);
    g_PdbDelayInterruptCounter++;
    g_PdbDelayInterruptFlag = true;
}

void DEMO_PDB_InitADC(void)
{
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Init(DEMO_PDB_ADC_INSTANCE, &adc16ConfigStruct);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, false);
    ADC16_DoAutoCalibration(DEMO_PDB_ADC_INSTANCE);
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, true);

    adc16ChannelConfigStruct.channelNumber = DEMO_PDB_ADC_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
      true; /* Enable the interrupt. */
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */
    ADC16_SetChannelConfig(DEMO_PDB_ADC_INSTANCE, DEMO_PDB_ADC_CHANNEL_GROUP, &
      adc16ChannelConfigStruct);
}

void DEMO_PDB_ADC_IRQ_HANDLER_FUNCTION(void)
{
    uint32_t tmp32;

    tmp32 = ADC16_GetChannelConversionValue(DEMO_PDB_ADC_INSTANCE,
      DEMO_PDB_ADC_CHANNEL_GROUP); /* Read to clear COCO flag. */
    g_AdcInterruptCounter++;
    g_AdcInterruptFlag = true;
}

int main(void)
{
    // ...

    EnableIRQ(DEMO_PDB_IRQ_ID);
    EnableIRQ(DEMO_PDB_ADC_IRQ_ID);
```

```
    // ...

    // Configures the PDB counter.
    PDB_GetDefaultConfig(&pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
       value is less than or equal to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
      kPDB_DelayInterruptEnable);

    // Configures the ADC pre-trigger.
    pdbAdcPreTriggerConfigStruct.enablePreTriggerMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
    pdbAdcPreTriggerConfigStruct.enableOutputMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
    pdbAdcPreTriggerConfigStruct.enableBackToBackOperationMask = 0U;
    PDB_SetADCPreTriggerConfig(DEMO_PDB_INSTANCE, DEMO_PDB_ADC_TRIGGER_CHANNEL, &
      pdbAdcPreTriggerConfigStruct);
    PDB_SetADCPreTriggerDelayValue(DEMO_PDB_INSTANCE,
                                   DEMO_PDB_ADC_TRIGGER_CHANNEL, DEMO_PDB_ADC_PRETRIGGER_CHANNEL, 200U);
                        // The available pre-trigger delay value is less than or equal to the modulus
       value.

    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    // Configures the ADC.
    DEMO_PDB_InitADC();

    while (1)
    {
        g_PdbDelayInterruptFlag = false;
        g_AdcInterruptFlag = false;
        PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
        while ((!g_PdbDelayInterruptFlag) || (!g_AdcInterruptFlag))
        {
        }
        // ...
    }
}
```

Files

- file fsl_pdb.h

Data Structures

- struct pdb_config_t
  PDB module configuration. More...
- struct pdb_adc_pretrigger_config_t
  PDB ADC Pre-Trigger configuration. More...
- struct pdb_dac_trigger_config_t
  PDB DAC trigger configuration. More...

Enumerations

- enum _pdb_status_flags {
  kPDB_LoadOKFlag = PDB_SC_LDOK_MASK,
  kPDB_DelayEventFlag = PDB_SC_PDBIF_MASK }
  PDB flags.

- enum _pdb_adc_pretrigger_flags {
  kPDB_ADCPreTriggerChannel0Flag = PDB_S_CF(1U << 0),
  kPDB_ADCPreTriggerChannel1Flag = PDB_S_CF(1U << 1),
  kPDB_ADCPreTriggerChannel0ErrorFlag = PDB_S_ERR(1U << 0),
  kPDB_ADCPreTriggerChannel1ErrorFlag = PDB_S_ERR(1U << 1) }
  PDB ADC PreTrigger channel flags.
- enum _pdb_interrupt_enable {
  kPDB_SequenceErrorInterruptEnable = PDB_SC_PDBEIE_MASK,
  kPDB_DelayInterruptEnable = PDB_SC_PDBIE_MASK }
  PDB buffer interrupts.
- enum pdb_load_value_mode_t {
  kPDB_LoadValueImmediately = 0U,
  kPDB_LoadValueOnCounterOverflow = 1U,
  kPDB_LoadValueOnTriggerInput = 2U,
  kPDB_LoadValueOnCounterOverflowOrTriggerInput = 3U }
  PDB load value mode.
- enum pdb_prescaler_divider_t {
  kPDB_PrescalerDivider1 = 0U,
  kPDB_PrescalerDivider2 = 1U,
  kPDB_PrescalerDivider4 = 2U,
  kPDB_PrescalerDivider8 = 3U,
  kPDB_PrescalerDivider16 = 4U,
  kPDB_PrescalerDivider32 = 5U,
  kPDB_PrescalerDivider64 = 6U,
  kPDB_PrescalerDivider128 = 7U }
  Prescaler divider.
- enum pdb_divider_multiplication_factor_t {
  kPDB_DividerMultiplicationFactor1 = 0U,
  kPDB_DividerMultiplicationFactor10 = 1U,
  kPDB_DividerMultiplicationFactor20 = 2U,
  kPDB_DividerMultiplicationFactor40 = 3U }
  Multiplication factor select for prescaler.
- enum pdb_trigger_input_source_t {

kPDB_TriggerInput0 = 0U,
kPDB_TriggerInput1 = 1U,
kPDB_TriggerInput2 = 2U,
kPDB_TriggerInput3 = 3U,
kPDB_TriggerInput4 = 4U,
kPDB_TriggerInput5 = 5U,
kPDB_TriggerInput6 = 6U,
kPDB_TriggerInput7 = 7U,
kPDB_TriggerInput8 = 8U,
kPDB_TriggerInput9 = 9U,
kPDB_TriggerInput10 = 10U,
kPDB_TriggerInput11 = 11U,
kPDB_TriggerInput12 = 12U,
kPDB_TriggerInput13 = 13U,
kPDB_TriggerInput14 = 14U,
kPDB_TriggerSoftware = 15U }
Trigger input source.

Driver version

- #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  PDB driver version 2.0.1.

Initialization

- void PDB_Init (PDB_Type ∗base, const pdb_config_t ∗config)
  Initializes the PDB module.
- void PDB_Deinit (PDB_Type ∗base)
  De-initializes the PDB module.
- void PDB_GetDefaultConfig (pdb_config_t ∗config)
  Initializes the PDB user configuration structure.
- static void PDB_Enable (PDB_Type ∗base, bool enable)
  Enables the PDB module.

Basic Counter

- static void PDB_DoSoftwareTrigger (PDB_Type ∗base)
  Triggers the PDB counter by software.
- static void PDB_DoLoadValues (PDB_Type ∗base)
  Loads the counter values.
- static void PDB_EnableDMA (PDB_Type ∗base, bool enable)
  Enables the DMA for the PDB module.
- static void PDB_EnableInterrupts (PDB_Type ∗base, uint32_t mask)
  Enables the interrupts for the PDB module.
- static void PDB_DisableInterrupts (PDB_Type ∗base, uint32_t mask)
  Disables the interrupts for the PDB module.
- static uint32_t PDB_GetStatusFlags (PDB_Type ∗base)
  Gets the status flags of the PDB module.

- static void PDB_ClearStatusFlags (PDB_Type ∗base, uint32_t mask)
  Clears the status flags of the PDB module.
- static void PDB_SetModulusValue (PDB_Type ∗base, uint32_t value)
  Specifies the period of the counter.
- static uint32_t PDB_GetCounterValue (PDB_Type ∗base)
  Gets the PDB counter's current value.
- static void PDB_SetCounterDelayValue (PDB_Type ∗base, uint32_t value)
  Sets the value for PDB counter delay event.

ADC Pre-Trigger

- static void PDB_SetADCPreTriggerConfig (PDB_Type ∗base, uint32_t channel, pdb_adc_pretrigger_config_t ∗config)
  Configures the ADC PreTrigger in PDB module.
- static void PDB_SetADCPreTriggerDelayValue (PDB_Type ∗base, uint32_t channel, uint32_t preChannel, uint32_t value)
  Sets the value for ADC Pre-Trigger delay event.
- static uint32_t PDB_GetADCPreTriggerStatusFlags (PDB_Type ∗base, uint32_t channel)
  Gets the ADC Pre-Trigger's status flags.
- static void PDB_ClearADCPreTriggerStatusFlags (PDB_Type ∗base, uint32_t channel, uint32_t mask)
  Clears the ADC Pre-Trigger's status flags.

Pulse-Out Trigger

- static void PDB_EnablePulseOutTrigger (PDB_Type ∗base, uint32_t channelMask, bool enable)
  Enables the pulse out trigger channels.
- static void PDB_SetPulseOutTriggerDelayValue (PDB_Type ∗base, uint32_t channel, uint32_t value1, uint32_t value2)
  Sets event values for pulse out trigger.

27.3 Data Structure Documentation

27.3.1 struct pdb_config_t

Data Fields

- pdb_load_value_mode_t loadValueMode
  Select the load value mode.
- pdb_prescaler_divider_t prescalerDivider
  Select the prescaler divider.
- pdb_divider_multiplication_factor_t dividerMultiplicationFactor
  Multiplication factor select for prescaler.
- pdb_trigger_input_source_t triggerInputSource
  Select the trigger input source.
- bool enableContinuousMode
  Enable the PDB operation in Continuous mode.

27.3.1.0.0.72 Field Documentation

27.3.1.0.0.72.1 pdb_load_value_mode_t pdb_config_t::loadValueMode

27.3.1.0.0.72.2 pdb_prescaler_divider_t pdb_config_t::prescalerDivider

27.3.1.0.0.72.3 pdb_divider_multiplication_factor_t pdb_config_t::dividerMultiplicationFactor

27.3.1.0.0.72.4 pdb_trigger_input_source_t pdb_config_t::triggerInputSource

27.3.1.0.0.72.5 bool pdb_config_t::enableContinuousMode

27.3.2 struct pdb_adc_pretrigger_config_t

Data Fields

- uint32_t enablePreTriggerMask
  PDB Channel Pre-Trigger Enable.
- uint32_t enableOutputMask
  PDB Channel Pre-Trigger Output Select.
- uint32_t enableBackToBackOperationMask
  PDB Channel Pre-Trigger Back-to-Back Operation Enable.

27.3.2.0.0.73 Field Documentation

27.3.2.0.0.73.1 uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask

27.3.2.0.0.73.2 uint32_t pdb_adc_pretrigger_config_t::enableOutputMask

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

27.3.2.0.0.73.3 uint32_t pdb_adc_pretrigger_config_t::enableBackToBackOperationMask

Back-to-back operation enables the ADC conversions complete to trigger the next PDB channel pre-trigger and trigger output, so that the ADC conversions can be triggered on next set of configuration and results registers.

27.3.3 struct pdb_dac_trigger_config_t

Data Fields

- bool enableExternalTriggerInput
  Enables the external trigger for DAC interval counter.
- bool enableIntervalTrigger
  Enables the DAC interval trigger.

27.3.3.0.0.74 Field Documentation

27.3.3.0.0.74.1 bool pdb_dac_trigger_config_t::enableExternalTriggerInput

27.3.3.0.0.74.2 bool pdb_dac_trigger_config_t::enableIntervalTrigger

27.4 Macro Definition Documentation

27.4.1 #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

27.5 Enumeration Type Documentation

27.5.1 enum _pdb_status_flags

Enumerator

kPDB_LoadOKFlag This flag is automatically cleared when the values in buffers are loaded into the internal registers after the LDOK bit is set or the PDBEN is cleared.
kPDB_DelayEventFlag PDB timer delay event flag.

27.5.2 enum _pdb_adc_pretrigger_flags

Enumerator

kPDB_ADCPreTriggerChannel0Flag Pre-Trigger 0 flag.
kPDB_ADCPreTriggerChannel1Flag Pre-Trigger 1 flag.
kPDB_ADCPreTriggerChannel0ErrorFlag Pre-Trigger 0 Error.
kPDB_ADCPreTriggerChannel1ErrorFlag Pre-Trigger 1 Error.

27.5.3 enum _pdb_interrupt_enable

Enumerator

kPDB_SequenceErrorInterruptEnable PDB sequence error interrupt enable.
kPDB_DelayInterruptEnable PDB delay interrupt enable.

27.5.4 enum pdb_load_value_mode_t

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx_SC[LDOK]). These values are for:

- PDB counter (PDBx_MOD, PDBx_IDLY)
- ADC trigger (PDBx_CHnDLYm)

- DAC trigger (PDBx_DACINTx)
- CMP trigger (PDBx_POyDLY)

Enumerator

kPDB_LoadValueImmediately Load immediately after 1 is written to LDOK.
kPDB_LoadValueOnCounterOverflow Load when the PDB counter overflows (reaches the MOD register value).
kPDB_LoadValueOnTriggerInput Load a trigger input event is detected.
kPDB_LoadValueOnCounterOverflowOrTriggerInput Load either when the PDB counter overflows or a trigger input is detected.

27.5.5 enum pdb_prescaler_divider_t

Counting uses the peripheral clock divided by multiplication factor selected by times of MULT.

Enumerator

kPDB_PrescalerDivider1 Divider x1.
kPDB_PrescalerDivider2 Divider x2.
kPDB_PrescalerDivider4 Divider x4.
kPDB_PrescalerDivider8 Divider x8.
kPDB_PrescalerDivider16 Divider x16.
kPDB_PrescalerDivider32 Divider x32.
kPDB_PrescalerDivider64 Divider x64.
kPDB_PrescalerDivider128 Divider x128.

27.5.6 enum pdb_divider_multiplication_factor_t

Selects the multiplication factor of the prescaler divider for the counter clock.

Enumerator

kPDB_DividerMultiplicationFactor1 Multiplication factor is 1.
kPDB_DividerMultiplicationFactor10 Multiplication factor is 10.
kPDB_DividerMultiplicationFactor20 Multiplication factor is 20.
kPDB_DividerMultiplicationFactor40 Multiplication factor is 40.

27.5.7 enum pdb_trigger_input_source_t

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

Enumerator

kPDB_TriggerInput0 Trigger-In 0.
kPDB_TriggerInput1 Trigger-In 1.
kPDB_TriggerInput2 Trigger-In 2.
kPDB_TriggerInput3 Trigger-In 3.
kPDB_TriggerInput4 Trigger-In 4.
kPDB_TriggerInput5 Trigger-In 5.
kPDB_TriggerInput6 Trigger-In 6.
kPDB_TriggerInput7 Trigger-In 7.
kPDB_TriggerInput8 Trigger-In 8.
kPDB_TriggerInput9 Trigger-In 9.
kPDB_TriggerInput10 Trigger-In 10.
kPDB_TriggerInput11 Trigger-In 11.
kPDB_TriggerInput12 Trigger-In 12.
kPDB_TriggerInput13 Trigger-In 13.
kPDB_TriggerInput14 Trigger-In 14.
kPDB_TriggerSoftware Trigger-In 15, software trigger.

27.6 Function Documentation

27.6.1 void PDB_Init (PDB_Type ∗ *base*, const pdb_config_t ∗ *config*)

This function initializes for PDB module. The operations included are:

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

base	PDB peripheral base address.
config	Pointer to configuration structure. See "pdb_config_t".

27.6.2 void PDB_Deinit (PDB_Type ∗ *base*)

Parameters

base	PDB peripheral base address.

27.6.3 void PDB_GetDefaultConfig (pdb_config_t ∗ *config*)

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

```
*   config->loadValueMode = kPDB_LoadValueImmediately;
*   config->prescalerDivider = kPDB_PrescalerDivider1;
*   config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1
      ;
*   config->triggerInputSource = kPDB_TriggerSoftware;
*   config->enableContinuousMode = false;
*
```

Parameters

config	Pointer to configuration structure. See "pdb_config_t".

27.6.4 static void PDB_Enable (PDB_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the module or not.

27.6.5 static void PDB_DoSoftwareTrigger (PDB_Type ∗ *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

27.6.6 static void PDB_DoLoadValues (PDB_Type ∗ *base*) [inline], [static]

This function loads the counter values from their internal buffer. See "pdb_load_value_mode_t" about PDB's load mode.

Parameters

base	PDB peripheral base address.

27.6.7 static void PDB_EnableDMA (PDB_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the feature or not.

27.6.8 static void PDB_EnableInterrupts (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	PDB peripheral base address.
mask	Mask value for interrupts. See "_pdb_interrupt_enable".

27.6.9 static void PDB_DisableInterrupts (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	PDB peripheral base address.
mask	Mask value for interrupts. See "_pdb_interrupt_enable".

27.6.10 static uint32_t PDB_GetStatusFlags (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

Returns

Mask value for asserted flags. See "_pdb_status_flags".

27.6.11 static void PDB_ClearStatusFlags (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	PDB peripheral base address.
mask	Mask value of flags. See "_pdb_status_flags".

27.6.12 static void PDB_SetModulusValue (PDB_Type * *base*, uint32_t *value*) [inline], [static]

Parameters

base	PDB peripheral base address.
value	Setting value for the modulus. 16-bit is available.

27.6.13 static uint32_t PDB_GetCounterValue (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

Returns

PDB counter's current value.

27.6.14 static void PDB_SetCounterDelayValue (PDB_Type * *base*, uint32_t *value*) [inline], [static]

Parameters

base	PDB peripheral base address.
value	Setting value for PDB counter delay event. 16-bit is available.

27.6.15 static void PDB_SetADCPreTriggerConfig (PDB_Type * *base*, uint32_t *channel*, pdb_adc_pretrigger_config_t * *config*) [inline], [static]

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.
config	Pointer to configuration structure. See "pdb_adc_pretrigger_config_t".

27.6.16 static void PDB_SetADCPreTriggerDelayValue (PDB_Type * *base,* uint32_t *channel,* uint32_t *preChannel,* uint32_t *value*) [inline], [static]

This function sets the value for ADC Pre-Trigger delay event. IT Specifies the delay value for the channel's corresponding pre-trigger. The pre-trigger asserts when the PDB counter is equal to the setting value here.

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.
preChannel	Channel group index for ADC instance.
value	Setting value for ADC Pre-Trigger delay event. 16-bit is available.

27.6.17 static uint32_t PDB_GetADCPreTriggerStatusFlags (PDB_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.

Returns

Mask value for asserted flags. See "_pdb_adc_pretrigger_flags".

27.6.18 static void PDB_ClearADCPreTriggerStatusFlags (PDB_Type * *base,* uint32_t *channel,* uint32_t *mask*) [inline], [static]

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.
mask	Mask value for flags. See "_pdb_adc_pretrigger_flags".

27.6.19 static void PDB_EnablePulseOutTrigger (PDB_Type ∗ *base,* uint32_t *channelMask,* bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
channelMask	Channel mask value for multiple pulse out trigger channel.
enable	Enable the feature or not.

27.6.20 static void PDB_SetPulseOutTriggerDelayValue (PDB_Type ∗ *base,* uint32_t *channel,* uint32_t *value1,* uint32_t *value2*) [inline], [static]

This function is used to set event values for pulse output trigger. These pulse output trigger delay values specify the delay for the PDB Pulse-Out. Pulse-Out goes high when the PDB counter is equal to the pulse output high value (value1). Pulse-Out goes low when the PDB counter is equal to the pulse output low value (value2).

Parameters

base	PDB peripheral base address.
channel	Channel index for pulse out trigger channel.
value1	Setting value for pulse out high.
value2	Setting value for pulse out low.

Chapter 28
PIT: Periodic Interrupt Timer

28.1 Overview

The KSDK provides a driver for the Periodic Interrupt Timer (PIT) of Kinetis devices.

28.2 Function groups

The PIT driver supports operating the module as a time counter.

28.2.1 Initialization and deinitialization

The function PIT_Init() initializes the PIT with specified configurations. The function PIT_GetDefault-Config() gets the default configurations. The initialization function configures the PIT operation in debug mode.

The function PIT_SetTimerChainMode() configures the chain mode operation of each PIT channel.

The function PIT_Deinit() disables the PIT timers and disables the module clock.

28.2.2 Timer period Operations

The function PITR_SetTimerPeriod() sets the timer period in units of count. Timers begin counting down from the value set by this function until it reaches 0.

The function PIT_GetCurrentTimerCount() reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. User can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds

28.2.3 Start and Stop timer operations

The function PIT_StartTimer() starts the timer counting. After calling this function, the timer loads the period value set earlier via the PIT_SetPeriod() function and starts counting down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function PIT_StopTimer() stops the timer counting.

28.2.4 Status

Provides functions to get and clear the PIT status.

28.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

28.3 Typical use case

28.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically.

```
int main(void)
{
    /* Structure of initialize PIT */
    pit_config_t pitConfig;

    /* Initialize and enable LED */
    LED_INIT();

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    PIT_GetDefaultConfig(&pitConfig);

    /* Init pit module */
    PIT_Init(PIT, &pitConfig);

    /* Set timer period for channel 0 */
    PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000000U,
      PIT_SOURCE_CLOCK));

    /* Enable timer interrupts for channel 0 */
    PIT_EnableInterrupts(PIT, kPIT_Chnl_0,
      kPIT_TimerInterruptEnable);

    /* Enable at the NVIC */
    EnableIRQ(PIT_IRQ_ID);

    /* Start channel 0 */
    PRINTF("\r\nStarting channel No.0 ...");
    PIT_StartTimer(PIT, kPIT_Chnl_0);

    while (true)
    {
        /* Check whether occur interupt and toggle LED */
        if (true == pitIsrFlag)
        {
            PRINTF("\r\n Channel No.0 interrupt is occured !");
            LED_TOGGLE();
            pitIsrFlag = false;
        }
    }
}
```

Files

- file fsl_pit.h

Data Structures

- struct pit_config_t
  PIT config structure. More...

Enumerations

- enum pit_chnl_t {
  kPIT_Chnl_0 = 0U,
  kPIT_Chnl_1,
  kPIT_Chnl_2,
  kPIT_Chnl_3 }
  List of PIT channels.
- enum pit_interrupt_enable_t { kPIT_TimerInterruptEnable = PIT_TCTRL_TIE_MASK }
  List of PIT interrupts.
- enum pit_status_flags_t { kPIT_TimerFlag = PIT_TFLG_TIF_MASK }
  List of PIT status flags.

Driver version

- #define FSL_PIT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

Initialization and deinitialization

- void PIT_Init (PIT_Type ∗base, const pit_config_t ∗config)
  Ungates the PIT clock, enables the PIT module and configures the peripheral for basic operation.
- void PIT_Deinit (PIT_Type ∗base)
  Gate the PIT clock and disable the PIT module.
- static void PIT_GetDefaultConfig (pit_config_t ∗config)
  Fill in the PIT config struct with the default settings.

Interrupt Interface

- static void PIT_EnableInterrupts (PIT_Type ∗base, pit_chnl_t channel, uint32_t mask)
  Enables the selected PIT interrupts.
- static void PIT_DisableInterrupts (PIT_Type ∗base, pit_chnl_t channel, uint32_t mask)
  Disables the selected PIT interrupts.
- static uint32_t PIT_GetEnabledInterrupts (PIT_Type ∗base, pit_chnl_t channel)
  Gets the enabled PIT interrupts.

Status Interface

- static uint32_t PIT_GetStatusFlags (PIT_Type ∗base, pit_chnl_t channel)
  Gets the PIT status flags.
- static void PIT_ClearStatusFlags (PIT_Type ∗base, pit_chnl_t channel, uint32_t mask)
  Clears the PIT status flags.

Read and Write the timer period

- static void PIT_SetTimerPeriod (PIT_Type ∗base, pit_chnl_t channel, uint32_t count)
  Sets the timer period in units of count.
- static uint32_t PIT_GetCurrentTimerCount (PIT_Type ∗base, pit_chnl_t channel)
  Reads the current timer counting value.

Timer Start and Stop

- static void PIT_StartTimer (PIT_Type ∗base, pit_chnl_t channel)
  Starts the timer counting.
- static void PIT_StopTimer (PIT_Type ∗base, pit_chnl_t channel)
  Stops the timer counting.

28.4 Data Structure Documentation

28.4.1 struct pit_config_t

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 config struct can be made const so it resides in flash

Data Fields

- bool enableRunInDebug
  true: Timers run in debug mode; false: Timers stop in debug mode

28.5 Enumeration Type Documentation

28.5.1 enum pit_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

kPIT_Chnl_0 PIT channel number 0.
kPIT_Chnl_1 PIT channel number 1.
kPIT_Chnl_2 PIT channel number 2.
kPIT_Chnl_3 PIT channel number 3.

28.5.2 enum pit_interrupt_enable_t

Enumerator

kPIT_TimerInterruptEnable Timer interrupt enable.

28.5.3 enum pit_status_flags_t

Enumerator

kPIT_TimerFlag Timer flag.

28.6 Function Documentation

28.6.1 void PIT_Init (PIT_Type ∗ *base,* const pit_config_t ∗ *config*)

Note

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

Parameters

base	PIT peripheral base address
config	Pointer to user's PIT config structure

28.6.2 void PIT_Deinit (PIT_Type ∗ *base*)

Parameters

base	PIT peripheral base address

28.6.3 static void PIT_GetDefaultConfig (pit_config_t ∗ *config*) [inline], [static]

The default values are:

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

Parameters

config	Pointer to user's PIT config structure.

28.6.4 static void PIT_EnableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

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

28.6.5 static void PIT_DisableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

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

28.6.6 static uint32_t PIT_GetEnabledInterrupts (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

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

28.6.7 static uint32_t PIT_GetStatusFlags (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

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

28.6.8 static void PIT_ClearStatusFlags (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

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

28.6.9 static void PIT_SetTimerPeriod (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *count*) [inline], [static]

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

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	PIT peripheral base address
channel	Timer channel number

count	Timer period in units of ticks

28.6.10 static uint32_t PIT_GetCurrentTimerCount (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

User can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

Current timer counting value in ticks

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

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

Parameters

base	PIT peripheral base address
channel	Timer channel number.

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

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

Parameters

base	PIT peripheral base address
channel	Timer channel number.

Chapter 29
PMC: Power Management Controller

29.1 Overview

The KSDK provides a Peripheral driver for the Power Management Controller (PMC) module of Kinetis devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

Files

- file fsl_pmc.h

Data Structures

- struct pmc_low_volt_detect_config_t
  Low-Voltage Detect Configuration Structure. More...
- struct pmc_low_volt_warning_config_t
  Low-Voltage Warning Configuration Structure. More...

Driver version

- #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  PMC driver version.

Power Management Controller Control APIs

- void PMC_ConfigureLowVoltDetect (PMC_Type ∗base, const pmc_low_volt_detect_config_t ∗config)
  Configure the low-voltage detect setting.
- static bool PMC_GetLowVoltDetectFlag (PMC_Type ∗base)
  Get Low-Voltage Detect Flag status.
- static void PMC_ClearLowVoltDetectFlag (PMC_Type ∗base)
  Acknowledge to clear the Low-voltage Detect flag.
- void PMC_ConfigureLowVoltWarning (PMC_Type ∗base, const pmc_low_volt_warning_config_t ∗config)
  Configure the low-voltage warning setting.
- static bool PMC_GetLowVoltWarningFlag (PMC_Type ∗base)
  Get Low-Voltage Warning Flag status.
- static void PMC_ClearLowVoltWarningFlag (PMC_Type ∗base)
  Acknowledge to Low-Voltage Warning flag.

29.2 Data Structure Documentation

29.2.1 struct pmc_low_volt_detect_config_t

Data Fields

- bool enableInt
  Enable interrupt when low-voltage detect.
- bool enableReset
  Enable system reset when low-voltage detect.

29.2.2 struct pmc_low_volt_warning_config_t

Data Fields

- bool enableInt
  Enable interrupt when low-voltage warning.

29.3 Macro Definition Documentation

29.3.1 #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

29.4 Function Documentation

29.4.1 void PMC_ConfigureLowVoltDetect (PMC_Type * *base,* const pmc_low_volt_detect_config_t * *config*)

This function configures the low-voltage detect setting, including the trip point voltage setting, enable interrupt or not, enable system reset or not.

Parameters

base	PMC peripheral base address.
config	Low-Voltage detect configuration structure.

29.4.2 static bool PMC_GetLowVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

Parameters

base	PMC peripheral base address.

Returns

Current low-voltage detect flag
- true: Low-voltage detected
- false: Low-voltage not detected

29.4.3 static void PMC_ClearLowVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

Parameters

base	PMC peripheral base address.

29.4.4 void PMC_ConfigureLowVoltWarning (PMC_Type * *base,* const pmc_low_volt_warning_config_t * *config*)

This function configures the low-voltage warning setting, including the trip point voltage setting and enable interrupt or not.

Parameters

base	PMC peripheral base address.
config	Low-Voltage warning configuration structure.

29.4.5 static bool PMC_GetLowVoltWarningFlag (PMC_Type * *base*) [inline], [static]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

base	PMC peripheral base address.

Returns

Current LVWF status
- true: Low-Voltage Warning Flag is set.
- false: the Low-Voltage Warning does not happen.

29.4.6 static void PMC_ClearLowVoltWarningFlag (PMC_Type * *base*) [inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

base	PMC peripheral base address.

Chapter 30
PORT: Port Control and Interrupts

30.1 Overview

The KSDK provides a driver for the Port Control and Interrupts (PORT) module of Kinetis devices.

30.2 Typical configuration use case

30.2.1 Input PORT configuration

```
/* Input pin PORT configuration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
/*  Sets the configuration */
PORT_SetPinConfig(PORTA, 4, &config);
```

30.2.2 I2C PORT Configuration

```
/*  I2C pin PORTconfiguration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainEnable,
    kPORT_LowDriveStrength,
    kPORT_MuxAlt5,
    kPORT_UnLockRegister,
};
PORT_SetPinConfig(PORTE,24u,&config);
PORT_SetPinConfig(PORTE,25u,&config);
```

Files

- file fsl_port.h

Enumerations

- enum port_interrupt_t {
  kPORT_InterruptOrDMADisabled = 0x0U,
  kPORT_InterruptLogicZero = 0x8U,
  kPORT_InterruptRisingEdge = 0x9U,
  kPORT_InterruptFallingEdge = 0xAU,
  kPORT_InterruptEitherEdge = 0xBU,

kPORT_InterruptLogicOne = 0xCU }
Configures the interrupt generation condition.

Driver version

- #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
  Version 2.0.2.

30.3 Macro Definition Documentation

30.3.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

30.4 Enumeration Type Documentation

30.4.1 enum port_interrupt_t

Enumerator

kPORT_InterruptOrDMADisabled Interrupt/DMA request is disabled.
kPORT_InterruptLogicZero Interrupt when logic zero.
kPORT_InterruptRisingEdge Interrupt on rising edge.
kPORT_InterruptFallingEdge Interrupt on falling edge.
kPORT_InterruptEitherEdge Interrupt on either edge.
kPORT_InterruptLogicOne Interrupt when logic one.

Chapter 31
PWM: Pulse Width Modulator

31.1 Overview

The SDK provides a driver for the Pulse Width Modulator (PWM) of Kinetis devices.

The function PWM_Init() initializes the PWM sub module with specified configurations, the function PWM_GetDefaultConfig() could help to get the default configurations. The initialization function configures the sub module for the requested register update mode for registers with buffers. It also sets up the sub module operation in debug and wait modes.

The function PWM_SetupPwm() sets up PWM channels for PWM output, the function can set up PWM signal properties for multiple channels. The PWM has 2 channels: A & B. Each channel has its own duty cycle and level-mode specified, however the same PWM period and PWM mode is applied to all channels requesting PWM output. The signal duty cycle is provided as a percentage of the PWM period, its value should be between 0 and 100; 0=inactive signal(0% duty cycle) and 100=always active signal (100% duty cycle). The function also sets up the channel dead time value which is used when the user selects complementary mode of operation.

The function PWM_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular PWM channel.

The function PWM_SetupInputCapture() sets up a PWM channel for input capture. The user can specify the capture edge and the mode; one-shot capture or free-running capture.

The function PWM_SetupFault() sets up the properties for each fault.

The function PWM_StartTimer() can be used to start one or multiple sub modules. The function PWM_StopTimer() can be used to stop one or multiple sub modules.

Provide functions to get and clear the PWM status.

Provide functions to enable/disable PWM interrupts and get current enabled interrupts.

31.2 Register Update

Some of the PWM registers have buffers, the driver support various methods to update these registers with the content of the register buffer. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure.

```
pwm_register_reload_t reloadLogic;
pwm_reload_source_select_t reloadSelect;
pwm_load_frequency_t reloadFrequency;
```

The user can select one of the reload options provided in enumeration pwm_register_reload_t. When using immediate reload, the reloadFrequency field is not used.

The driver initialization function sets up the appropriate bits in the PWM module based on the register update options selected.

The below function should be used to initiate a register reload. The example shows register reload initiated on PWM sub modules 0, 1, and 2.

```
PWM_SetPwmLdok(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 |
                                   kPWM_Control_Module_1 |
                                   kPWM_Control_Module_2, true);
```

31.3 Typical use case

31.3.1 PWM output

Output PWM signal on 3 PWM sub module with different dutycycles. Periodically update the PWM signal duty cycle. Each sub module runs in Complementary output mode with PWM A used to generate the complementary PWM pair.

```
static void PWM_DRV_Init3PhPwm(void)
{
    uint16_t deadTimeVal;
    pwm_signal_param_t pwmSignal[2];
    uint32_t pwmSourceClockInHz;
    uint32_t pwmFrequencyInHz = 1000;

    pwmSourceClockInHz = CLOCK_GetFreq(kCLOCK_FastPeriphClk);

    /* Set deadtime count, we set this to about 650ns */
    deadTimeVal = ((uint64_t)pwmSourceClockInHz * 650) / 1000000000;

    pwmSignal[0].pwmChannel = kPWM_PwmA;
    pwmSignal[0].level = kPWM_HighTrue;
    pwmSignal[0].dutyCyclePercent = 50; /* 1 percent dutycycle */
    pwmSignal[0].deadtimeValue = deadTimeVal;

    pwmSignal[1].pwmChannel = kPWM_PwmB;
    pwmSignal[1].level = kPWM_HighTrue;
    /* Dutycycle field of PWM B does not matter as we are running in PWM A complementary mode */
    pwmSignal[1].dutyCyclePercent = 50;
    pwmSignal[1].deadtimeValue = deadTimeVal;

    /*********** PWMA_SM0 - phase A, configuration, setup 2 channel as an example ************/
    PWM_SetupPwm(BOARD_PWM_BASEADDR,
                 kPWM_Module_0,
                 pwmSignal,
                 2,
                 kPWM_SignedCenterAligned,
                 pwmFrequencyInHz,
                 pwmSourceClockInHz);

    /*********** PWMA_SM1 - phase B configuration, setup PWM A channel only ************/
    PWM_SetupPwm(BOARD_PWM_BASEADDR,
                 kPWM_Module_1,
                 pwmSignal,
                 1,
                 kPWM_SignedCenterAligned,
                 pwmFrequencyInHz,
                 pwmSourceClockInHz);

    /*********** PWMA_SM2 - phase C configuration, setup PWM A channel only ************/
    PWM_SetupPwm(BOARD_PWM_BASEADDR,
```

```
                 kPWM_Module_2,
                 pwmSignal,
                 1,
                 kPWM_SignedCenterAligned,
                 pwmFrequencyInHz,
                 pwmSourceClockInHz);
}

int main(void)
{
    /* Structure of initialize PWM */
    pwm_config_t pwmConfig;
    static uint16_t delay;
    uint32_t pwmVal = 4;
    uint16_t i;

    /* Board pin, clock, debug console initialization */
    BOARD_InitHardware();

    PRINTF("FlexPWM driver example\n");

    PWM_GetDefaultConfig(&pwmConfig);

    /* Use full cycle reload */
    pwmConfig.reloadLogic = kPWM_ReloadPwmFullCycle;
    /* PWM A & PWM B form a complementary PWM pair */
    pwmConfig.pairOperation = kPWM_ComplementaryPwmA;
    pwmConfig.enableDebugMode = true;

    /* Initialize sub module 0 */
    if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_0, &pwmConfig) == kStatus_Fail)
    {
        PRINTF("PWM initialization failed\n");
        return 1;
    }

    /* Initialize sub module 1 */
    pwmConfig.clockSource = kPWM_Submodule0Clock;
    pwmConfig.initializationControl =
      kPWM_Initialize_MasterSync;
    if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_1, &pwmConfig) == kStatus_Fail)
    {
        PRINTF("PWM initialization failed\n");
        return 1;
    }

    /* Initialize sub module 2 the same way as sub module 1 */
    if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_2, &pwmConfig) == kStatus_Fail)
    {
        PRINTF("PWM initialization failed\n");
        return 1;
    }

    /* Call the initialization function with demo configuration */
    PWM_DRV_Init3PhPwm();

    /* Set the load okay bit for all sub modules to load registers from their buffer */
    PWM_SetPwmLdok(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 |
      kPWM_Control_Module_1 | kPWM_Control_Module_2, true);

    /* Start the PWM generation from sub modules 0, 1 and 2 */
    PWM_StartTimer(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 |
      kPWM_Control_Module_1 | kPWM_Control_Module_2);

    delay = 0x0fffU;

    while (1U)
    {
```

```
        for (i = 0U; i < delay; i++)
        {
            __ASM volatile("nop");
        }
        pwmVal = pwmVal + 4;

        /* Reset the duty cycle percentage */
        if (pwmVal > 100)
        {
            pwmVal = 4;
        }

        /* Update duty cycles for all 3 PWM signals */
        PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR,
      kPWM_Module_0, kPWM_PwmA, kPWM_SignedCenterAligned, pwmVal);
        PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR,
      kPWM_Module_1, kPWM_PwmA, kPWM_SignedCenterAligned, (pwmVal >> 1));
        PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR,
      kPWM_Module_2, kPWM_PwmA, kPWM_SignedCenterAligned, (pwmVal >> 2));

        /* Set the load okay bit for all submodules to load registers from their buffer */
        PWM_SetPwmLdok(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 |
      kPWM_Control_Module_1 | kPWM_Control_Module_2, true);
    }
}
```

Files

- file fsl_pwm.h

Data Structures

- struct pwm_signal_param_t
  Structure for the user to define the PWM signal characteristics. More...
- struct pwm_config_t
  PWM config structure. More...
- struct pwm_fault_param_t
  Structure is used to hold the parameters to configure a PWM fault. More...
- struct pwm_input_capture_param_t
  Structure is used to hold parameters to configure the capture capability of a signal pin. More...

Macros

- #define PWM_SUBMODULE_SWCONTROL_WIDTH 2
  Number of bits per submodule for software output control.

Enumerations

- enum pwm_submodule_t {
  kPWM_Module_0 = 0U,
  kPWM_Module_1,
  kPWM_Module_2,
  kPWM_Module_3 }
  List of PWM submodules.
- enum pwm_channels_t
  List of PWM channels in each module.

- enum pwm_value_register_t {
  kPWM_ValueRegister_0 = 0U,
  kPWM_ValueRegister_1,
  kPWM_ValueRegister_2,
  kPWM_ValueRegister_3,
  kPWM_ValueRegister_4,
  kPWM_ValueRegister_5 }

  List of PWM value registers.
- enum pwm_clock_source_t {
  kPWM_BusClock = 0U,
  kPWM_ExternalClock,
  kPWM_Submodule0Clock }

  PWM clock source selection.
- enum pwm_clock_prescale_t {
  kPWM_Prescale_Divide_1 = 0U,
  kPWM_Prescale_Divide_2,
  kPWM_Prescale_Divide_4,
  kPWM_Prescale_Divide_8,
  kPWM_Prescale_Divide_16,
  kPWM_Prescale_Divide_32,
  kPWM_Prescale_Divide_64,
  kPWM_Prescale_Divide_128 }

  PWM prescaler factor selection for clock source.
- enum pwm_force_output_trigger_t {
  kPWM_Force_Local = 0U,
  kPWM_Force_Master,
  kPWM_Force_LocalReload,
  kPWM_Force_MasterReload,
  kPWM_Force_LocalSync,
  kPWM_Force_MasterSync,
  kPWM_Force_External,
  kPWM_Force_ExternalSync }

  Options that can trigger a PWM FORCE_OUT.
- enum pwm_init_source_t {
  kPWM_Initialize_LocalSync = 0U,
  kPWM_Initialize_MasterReload,
  kPWM_Initialize_MasterSync,
  kPWM_Initialize_ExtSync }

  PWM counter initialization options.
- enum pwm_load_frequency_t {

kPWM_LoadEveryOportunity = 0U,
kPWM_LoadEvery2Oportunity,
kPWM_LoadEvery3Oportunity,
kPWM_LoadEvery4Oportunity,
kPWM_LoadEvery5Oportunity,
kPWM_LoadEvery6Oportunity,
kPWM_LoadEvery7Oportunity,
kPWM_LoadEvery8Oportunity,
kPWM_LoadEvery9Oportunity,
kPWM_LoadEvery10Oportunity,
kPWM_LoadEvery11Oportunity,
kPWM_LoadEvery12Oportunity,
kPWM_LoadEvery13Oportunity,
kPWM_LoadEvery14Oportunity,
kPWM_LoadEvery15Oportunity,
kPWM_LoadEvery16Oportunity }
PWM load frequency selection.

- enum pwm_fault_input_t {
kPWM_Fault_0 = 0U,
kPWM_Fault_1,
kPWM_Fault_2,
kPWM_Fault_3 }
List of PWM fault selections.
- enum pwm_input_capture_edge_t {
kPWM_Disable = 0U,
kPWM_FallingEdge,
kPWM_RisingEdge,
kPWM_RiseAndFallEdge }
PWM capture edge select.
- enum pwm_force_signal_t {
kPWM_UsePwm = 0U,
kPWM_InvertedPwm,
kPWM_SoftwareControl,
kPWM_UseExternal }
PWM output options when a FORCE_OUT signal is asserted.
- enum pwm_chnl_pair_operation_t {
kPWM_Independent = 0U,
kPWM_ComplementaryPwmA,
kPWM_ComplementaryPwmB }
Options available for the PWM A & B pair operation.
- enum pwm_register_reload_t {
kPWM_ReloadImmediate = 0U,
kPWM_ReloadPwmHalfCycle,
kPWM_ReloadPwmFullCycle,
kPWM_ReloadPwmHalfAndFullCycle }
Options available on how to load the buffered-registers with new values.

- enum pwm_fault_recovery_mode_t {
  kPWM_NoRecovery = 0U,
  kPWM_RecoverHalfCycle,
  kPWM_RecoverFullCycle,
  kPWM_RecoverHalfAndFullCycle }
  Options available on how to re-enable the PWM output when recovering from a fault.
- enum pwm_interrupt_enable_t {
  kPWM_CompareVal0InterruptEnable = (1U << 0),
  kPWM_CompareVal1InterruptEnable = (1U << 1),
  kPWM_CompareVal2InterruptEnable = (1U << 2),
  kPWM_CompareVal3InterruptEnable = (1U << 3),
  kPWM_CompareVal4InterruptEnable = (1U << 4),
  kPWM_CompareVal5InterruptEnable = (1U << 5),
  kPWM_CaptureX0InterruptEnable = (1U << 6),
  kPWM_CaptureX1InterruptEnable = (1U << 7),
  kPWM_CaptureB0InterruptEnable = (1U << 8),
  kPWM_CaptureB1InterruptEnable = (1U << 9),
  kPWM_CaptureA0InterruptEnable = (1U << 10),
  kPWM_CaptureA1InterruptEnable = (1U << 11),
  kPWM_ReloadInterruptEnable = (1U << 12),
  kPWM_ReloadErrorInterruptEnable = (1U << 13),
  kPWM_Fault0InterruptEnable = (1U << 16),
  kPWM_Fault1InterruptEnable = (1U << 17),
  kPWM_Fault2InterruptEnable = (1U << 18),
  kPWM_Fault3InterruptEnable = (1U << 19) }
  List of PWM interrupt options.
- enum pwm_status_flags_t {
  kPWM_CompareVal0Flag = (1U << 0),
  kPWM_CompareVal1Flag = (1U << 1),
  kPWM_CompareVal2Flag = (1U << 2),
  kPWM_CompareVal3Flag = (1U << 3),
  kPWM_CompareVal4Flag = (1U << 4),
  kPWM_CompareVal5Flag = (1U << 5),
  kPWM_CaptureX0Flag = (1U << 6),
  kPWM_CaptureX1Flag = (1U << 7),
  kPWM_CaptureB0Flag = (1U << 8),
  kPWM_CaptureB1Flag = (1U << 9),
  kPWM_CaptureA0Flag = (1U << 10),
  kPWM_CaptureA1Flag = (1U << 11),
  kPWM_ReloadFlag = (1U << 12),
  kPWM_ReloadErrorFlag = (1U << 13),
  kPWM_RegUpdatedFlag = (1U << 14),
  kPWM_Fault0Flag = (1U << 16),
  kPWM_Fault1Flag = (1U << 17),
  kPWM_Fault2Flag = (1U << 18),

kPWM_Fault3Flag = (1U << 19) }
List of PWM status flags.
- enum pwm_mode_t {
kPWM_SignedCenterAligned = 0U,
kPWM_CenterAligned,
kPWM_SignedEdgeAligned,
kPWM_EdgeAligned }
PWM operation mode.
- enum pwm_level_select_t {
kPWM_HighTrue = 0U,
kPWM_LowTrue }
PWM output pulse mode, high-true or low-true.
- enum pwm_reload_source_select_t {
kPWM_LocalReload = 0U,
kPWM_MasterReload }
PWM reload source select.
- enum pwm_fault_clear_t {
kPWM_Automatic = 0U,
kPWM_ManualNormal,
kPWM_ManualSafety }
PWM fault clearing options.
- enum pwm_module_control_t {
kPWM_Control_Module_0 = (1U << 0),
kPWM_Control_Module_1 = (1U << 1),
kPWM_Control_Module_2 = (1U << 2),
kPWM_Control_Module_3 = (1U << 3) }
Options for submodule master control operation.

Functions

- void PWM_SetupInputCapture (PWM_Type ∗base, pwm_submodule_t subModule, pwm_channels_t pwmChannel, const pwm_input_capture_param_t ∗inputCaptureParams)
Sets up the PWM input capture.
- void PWM_SetupFaults (PWM_Type ∗base, pwm_fault_input_t faultNum, const pwm_fault_param_t ∗faultParams)
Sets up the PWM fault protection.
- void PWM_SetupForceSignal (PWM_Type ∗base, pwm_submodule_t subModule, pwm_channels_t pwmChannel, pwm_force_signal_t mode)
Selects the signal to output on a PWM pin when a FORCE_OUT signal is asserted.
- static void PWM_OutputTriggerEnable (PWM_Type ∗base, pwm_submodule_t subModule, pwm_value_register_t valueRegister, bool activate)
Enables or disables the PWM output trigger.
- static void PWM_SetupSwCtrlOut (PWM_Type ∗base, pwm_submodule_t subModule, pwm_channels_t pwmChannel, bool value)
Sets the software control output for a pin to high or low.
- static void PWM_SetPwmLdok (PWM_Type ∗base, uint8_t subModulesToUpdate, bool value)
Sets or clears the PWM LDOK bit on a single or multiple submodules.

Driver version

- #define FSL_PWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Version 2.0.0.

Initialization and deinitialization

- status_t PWM_Init (PWM_Type ∗base, pwm_submodule_t subModule, const pwm_config_t ∗config)
  Ungates the PWM submodule clock and configures the peripheral for basic operation.
- void PWM_Deinit (PWM_Type ∗base, pwm_submodule_t subModule)
  Gate the PWM submodule clock.
- void PWM_GetDefaultConfig (pwm_config_t ∗config)
  Fill in the PWM config struct with the default settings.

Module PWM output

- status_t PWM_SetupPwm (PWM_Type ∗base, pwm_submodule_t subModule, const pwm_signal_param_t ∗chnlParams, uint8_t numOfChnls, pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
  Sets up the PWM signals for a PWM submodule.
- void PWM_UpdatePwmDutycycle (PWM_Type ∗base, pwm_submodule_t subModule, pwm_channels_t pwmSignal, pwm_mode_t currPwmMode, uint8_t dutyCyclePercent)
  Updates the PWM signal's dutycycle.

Interrupts Interface

- void PWM_EnableInterrupts (PWM_Type ∗base, pwm_submodule_t subModule, uint32_t mask)
  Enables the selected PWM interrupts.
- void PWM_DisableInterrupts (PWM_Type ∗base, pwm_submodule_t subModule, uint32_t mask)
  Disables the selected PWM interrupts.
- uint32_t PWM_GetEnabledInterrupts (PWM_Type ∗base, pwm_submodule_t subModule)
  Gets the enabled PWM interrupts.

Status Interface

- uint32_t PWM_GetStatusFlags (PWM_Type ∗base, pwm_submodule_t subModule)
  Gets the PWM status flags.
- void PWM_ClearStatusFlags (PWM_Type ∗base, pwm_submodule_t subModule, uint32_t mask)
  Clears the PWM status flags.

Timer Start and Stop

- static void PWM_StartTimer (PWM_Type ∗base, uint8_t subModulesToStart)
  Starts the PWM counter for a single or multiple submodules.
- static void PWM_StopTimer (PWM_Type ∗base, uint8_t subModulesToStop)
  Stops the PWM counter for a single or multiple submodules.

31.4 Data Structure Documentation

31.4.1 struct pwm_signal_param_t

Data Fields

- pwm_channels_t pwmChannel
  PWM channel being configured; PWM A or PWM B.
- uint8_t dutyCyclePercent
  PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...
- pwm_level_select_t level
  PWM output active level select.
- uint16_t deadtimeValue
  The deadtime value; only used if channel pair is operating in complementary mode.

31.4.1.0.0.75 Field Documentation

31.4.1.0.0.75.1 uint8_t pwm_signal_param_t::dutyCyclePercent

100=always active signal (100% duty cycle)

31.4.2 struct pwm_config_t

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.

The config struct can be made const so it resides in flash

Data Fields

- bool enableDebugMode
  true: PWM continues to run in debug mode; false: PWM is paused in debug mode
- bool enableWait
  true: PWM continues to run in WAIT mode; false: PWM is paused in WAIT mode
- uint8_t faultFilterCount
  Fault filter count.
- uint8_t faultFilterPeriod
  Fault filter period;value of 0 will bypass the filter.
- pwm_init_source_t initializationControl
  Option to initialize the counter.
- pwm_clock_source_t clockSource
  Clock source for the counter.
- pwm_clock_prescale_t prescale
  Pre-scaler to divide down the clock.
- pwm_chnl_pair_operation_t pairOperation
  Channel pair in indepedent or complementary mode.
- pwm_register_reload_t reloadLogic

PWM Reload logic setup.
- pwm_reload_source_select_t reloadSelect
  Reload source select.
- pwm_load_frequency_t reloadFrequency
  Specifies when to reload, used when user's choice is not immediate reload.
- pwm_force_output_trigger_t forceTrigger
  Specify which signal will trigger a FORCE_OUT.

31.4.3 struct pwm_fault_param_t

Data Fields

- pwm_fault_clear_t faultClearingMode
  Fault clearing mode to use.
- bool faultLevel
  true: Logic 1 indicates fault; false: Logic 0 indicates fault
- bool enableCombinationalPath
  true: Combinational Path from fault input is enabled; false: No combination path is available
- pwm_fault_recovery_mode_t recoverMode
  Specify when to re-enable the PWM output.

31.4.4 struct pwm_input_capture_param_t

Data Fields

- bool captureInputSel
  true: Use the edge counter signal as source false: Use the raw input signal from the pin as source
- uint8_t edgeCompareValue
  Compare value, used only if edge counter is used as source.
- pwm_input_capture_edge_t edge0
  Specify which edge causes a capture for input circuitry 0.
- pwm_input_capture_edge_t edge1
  Specify which edge causes a capture for input circuitry 1.
- bool enableOneShotCapture
  true: Use one-shot capture mode; false: Use free-running capture mode
- uint8_t fifoWatermark
  Watermark level for capture FIFO.

31.4.4.0.0.76 Field Documentation

31.4.4.0.0.76.1 uint8_t pwm_input_capture_param_t::fifoWatermark

The capture flags in the status register will set if the word count in the FIFO is greater than this watermark level

31.5 Enumeration Type Documentation

31.5.1 enum pwm_submodule_t

Enumerator

kPWM_Module_0 Submodule 0.
kPWM_Module_1 Submodule 1.
kPWM_Module_2 Submodule 2.
kPWM_Module_3 Submodule 3.

31.5.2 enum pwm_value_register_t

Enumerator

kPWM_ValueRegister_0 PWM Value0 register.
kPWM_ValueRegister_1 PWM Value1 register.
kPWM_ValueRegister_2 PWM Value2 register.
kPWM_ValueRegister_3 PWM Value3 register.
kPWM_ValueRegister_4 PWM Value4 register.
kPWM_ValueRegister_5 PWM Value5 register.

31.5.3 enum pwm_clock_source_t

Enumerator

kPWM_BusClock The IPBus clock is used as the clock.
kPWM_ExternalClock EXT_CLK is used as the clock.
kPWM_Submodule0Clock Clock of the submodule 0 (AUX_CLK) is used as the source clock.

31.5.4 enum pwm_clock_prescale_t

Enumerator

kPWM_Prescale_Divide_1 PWM clock frequency = fclk/1.
kPWM_Prescale_Divide_2 PWM clock frequency = fclk/2.
kPWM_Prescale_Divide_4 PWM clock frequency = fclk/4.
kPWM_Prescale_Divide_8 PWM clock frequency = fclk/8.
kPWM_Prescale_Divide_16 PWM clock frequency = fclk/16.
kPWM_Prescale_Divide_32 PWM clock frequency = fclk/32.
kPWM_Prescale_Divide_64 PWM clock frequency = fclk/64.
kPWM_Prescale_Divide_128 PWM clock frequency = fclk/128.

31.5.5 enum pwm_force_output_trigger_t

Enumerator

kPWM_Force_Local The local force signal, CTRL2[FORCE], from the submodule is used to force updates.
kPWM_Force_Master The master force signal from submodule 0 is used to force updates.
kPWM_Force_LocalReload The local reload signal from this submodule is used to force updates without regard to the state of LDOK.
kPWM_Force_MasterReload The master reload signal from submodule 0 is used to force updates if LDOK is set.
kPWM_Force_LocalSync The local sync signal from this submodule is used to force updates.
kPWM_Force_MasterSync The master sync signal from submodule0 is used to force updates.
kPWM_Force_External The external force signal, EXT_FORCE, from outside the PWM module causes updates.
kPWM_Force_ExternalSync The external sync signal, EXT_SYNC, from outside the PWM module causes updates.

31.5.6 enum pwm_init_source_t

Enumerator

kPWM_Initialize_LocalSync Local sync causes initialization.
kPWM_Initialize_MasterReload Master reload from submodule 0 causes initialization.
kPWM_Initialize_MasterSync Master sync from submodule 0 causes initialization.
kPWM_Initialize_ExtSync EXT_SYNC causes initialization.

31.5.7 enum pwm_load_frequency_t

Enumerator

kPWM_LoadEveryOportunity Every PWM opportunity.
kPWM_LoadEvery2Oportunity Every 2 PWM opportunities.
kPWM_LoadEvery3Oportunity Every 3 PWM opportunities.
kPWM_LoadEvery4Oportunity Every 4 PWM opportunities.
kPWM_LoadEvery5Oportunity Every 5 PWM opportunities.
kPWM_LoadEvery6Oportunity Every 6 PWM opportunities.
kPWM_LoadEvery7Oportunity Every 7 PWM opportunities.
kPWM_LoadEvery8Oportunity Every 8 PWM opportunities.
kPWM_LoadEvery9Oportunity Every 9 PWM opportunities.
kPWM_LoadEvery10Oportunity Every 10 PWM opportunities.
kPWM_LoadEvery11Oportunity Every 11 PWM opportunities.
kPWM_LoadEvery12Oportunity Every 12 PWM opportunities.

kPWM_LoadEvery13Oportunity Every 13 PWM opportunities.
kPWM_LoadEvery14Oportunity Every 14 PWM opportunities.
kPWM_LoadEvery15Oportunity Every 15 PWM opportunities.
kPWM_LoadEvery16Oportunity Every 16 PWM opportunities.

31.5.8 enum pwm_fault_input_t

Enumerator

kPWM_Fault_0 Fault 0 input pin.
kPWM_Fault_1 Fault 1 input pin.
kPWM_Fault_2 Fault 2 input pin.
kPWM_Fault_3 Fault 3 input pin.

31.5.9 enum pwm_input_capture_edge_t

Enumerator

kPWM_Disable Disabled.
kPWM_FallingEdge Capture on falling edge only.
kPWM_RisingEdge Capture on rising edge only.
kPWM_RiseAndFallEdge Capture on rising or falling edge.

31.5.10 enum pwm_force_signal_t

Enumerator

kPWM_UsePwm Generated PWM signal is used by the deadtime logic.
kPWM_InvertedPwm Inverted PWM signal is used by the deadtime logic.
kPWM_SoftwareControl Software controlled value is used by the deadtime logic.
kPWM_UseExternal PWM_EXTA signal is used by the deadtime logic.

31.5.11 enum pwm_chnl_pair_operation_t

Enumerator

kPWM_Independent PWM A & PWM B operate as 2 independent channels.
kPWM_ComplementaryPwmA PWM A & PWM B are complementary channels, PWM A generates the signal.
kPWM_ComplementaryPwmB PWM A & PWM B are complementary channels, PWM B generates the signal.

31.5.12 enum pwm_register_reload_t

Enumerator

kPWM_ReloadImmediate Buffered-registers get loaded with new values as soon as LDOK bit is set.
kPWM_ReloadPwmHalfCycle Registers loaded on a PWM half cycle.
kPWM_ReloadPwmFullCycle Registers loaded on a PWM full cycle.
kPWM_ReloadPwmHalfAndFullCycle Registers loaded on a PWM half & full cycle.

31.5.13 enum pwm_fault_recovery_mode_t

Enumerator

kPWM_NoRecovery PWM output will stay inactive.
kPWM_RecoverHalfCycle PWM output re-enabled at the first half cycle.
kPWM_RecoverFullCycle PWM output re-enabled at the first full cycle.
kPWM_RecoverHalfAndFullCycle PWM output re-enabled at the first half or full cycle.

31.5.14 enum pwm_interrupt_enable_t

Enumerator

kPWM_CompareVal0InterruptEnable PWM VAL0 compare interrupt.
kPWM_CompareVal1InterruptEnable PWM VAL1 compare interrupt.
kPWM_CompareVal2InterruptEnable PWM VAL2 compare interrupt.
kPWM_CompareVal3InterruptEnable PWM VAL3 compare interrupt.
kPWM_CompareVal4InterruptEnable PWM VAL4 compare interrupt.
kPWM_CompareVal5InterruptEnable PWM VAL5 compare interrupt.
kPWM_CaptureX0InterruptEnable PWM capture X0 interrupt.
kPWM_CaptureX1InterruptEnable PWM capture X1 interrupt.
kPWM_CaptureB0InterruptEnable PWM capture B0 interrupt.
kPWM_CaptureB1InterruptEnable PWM capture B1 interrupt.
kPWM_CaptureA0InterruptEnable PWM capture A0 interrupt.
kPWM_CaptureA1InterruptEnable PWM capture A1 interrupt.
kPWM_ReloadInterruptEnable PWM reload interrupt.
kPWM_ReloadErrorInterruptEnable PWM reload error interrupt.
kPWM_Fault0InterruptEnable PWM fault 0 interrupt.
kPWM_Fault1InterruptEnable PWM fault 1 interrupt.
kPWM_Fault2InterruptEnable PWM fault 2 interrupt.
kPWM_Fault3InterruptEnable PWM fault 3 interrupt.

31.5.15 enum pwm_status_flags_t

Enumerator

kPWM_CompareVal0Flag PWM VAL0 compare flag.
kPWM_CompareVal1Flag PWM VAL1 compare flag.
kPWM_CompareVal2Flag PWM VAL2 compare flag.
kPWM_CompareVal3Flag PWM VAL3 compare flag.
kPWM_CompareVal4Flag PWM VAL4 compare flag.
kPWM_CompareVal5Flag PWM VAL5 compare flag.
kPWM_CaptureX0Flag PWM capture X0 flag.
kPWM_CaptureX1Flag PWM capture X1 flag.
kPWM_CaptureB0Flag PWM capture B0 flag.
kPWM_CaptureB1Flag PWM capture B1 flag.
kPWM_CaptureA0Flag PWM capture A0 flag.
kPWM_CaptureA1Flag PWM capture A1 flag.
kPWM_ReloadFlag PWM reload flag.
kPWM_ReloadErrorFlag PWM reload error flag.
kPWM_RegUpdatedFlag PWM registers updated flag.
kPWM_Fault0Flag PWM fault 0 flag.
kPWM_Fault1Flag PWM fault 1 flag.
kPWM_Fault2Flag PWM fault 2 flag.
kPWM_Fault3Flag PWM fault 3 flag.

31.5.16 enum pwm_mode_t

Enumerator

kPWM_SignedCenterAligned Signed center-aligned.
kPWM_CenterAligned Unsigned cente-aligned.
kPWM_SignedEdgeAligned Signed edge-aligned.
kPWM_EdgeAligned Unsigned edge-aligned.

31.5.17 enum pwm_level_select_t

Enumerator

kPWM_HighTrue High level represents "on" or "active" state.
kPWM_LowTrue Low level represents "on" or "active" state.

31.5.18 enum pwm_reload_source_select_t

Enumerator

kPWM_LocalReload The local reload signal is used to reload registers.
kPWM_MasterReload The master reload signal (from submodule 0) is used to reload.

31.5.19 enum pwm_fault_clear_t

Enumerator

kPWM_Automatic Automatic fault clearing.
kPWM_ManualNormal Manual fault clearing with no fault safety mode.
kPWM_ManualSafety Manual fault clearing with fault safety mode.

31.5.20 enum pwm_module_control_t

Enumerator

kPWM_Control_Module_0 Control submodule 0's start/stop,buffer reload operation.
kPWM_Control_Module_1 Control submodule 1's start/stop,buffer reload operation.
kPWM_Control_Module_2 Control submodule 2's start/stop,buffer reload operation.
kPWM_Control_Module_3 Control submodule 3's start/stop,buffer reload operation.

31.6 Function Documentation

31.6.1 status_t PWM_Init (PWM_Type ∗ *base,* pwm_submodule_t *subModule,* const pwm_config_t ∗ *config*)

Note

This API should be called at the beginning of the application using the PWM driver.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure

config	Pointer to user's PWM config structure.

Returns

kStatus_Success means success; else failed.

31.6.2 void PWM_Deinit (PWM_Type ∗ *base*, pwm_submodule_t *subModule*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to deinitialize

31.6.3 void PWM_GetDefaultConfig (pwm_config_t ∗ *config*)

The default values are:

```
*   config->enableDebugMode = false;
*   config->enableWait = false;
*   config->reloadSelect = kPWM_LocalReload;
*   config->faultFilterCount = 0;
*   config->faultFilterPeriod = 0;
*   config->clockSource = kPWM_BusClock;
*   config->prescale = kPWM_Prescale_Divide_1;
*   config->initializationControl = kPWM_Initialize_LocalSync;
*   config->forceTrigger = kPWM_Force_Local;
*   config->reloadFrequency = kPWM_LoadEveryOportunity;
*   config->reloadLogic = kPWM_ReloadImmediate;
*   config->pairOperation = kPWM_Independent;
*
```

Parameters

config	Pointer to user's PWM config structure.

31.6.4 status_t PWM_SetupPwm (PWM_Type ∗ *base*, pwm_submodule_t *subModule*, const pwm_signal_param_t ∗ *chnlParams*, uint8_t *numOfChnls*, pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*)

The function initializes the submodule according to the parameters passed in by the user. The function also sets up the value compare registers to match the PWM signal requirements. If the dead time insertion logic is enabled, the pulse period is reduced by the dead time period specified by the user.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
chnlParams	Array of PWM channel parameters to configure the channel(s)
numOfChnls	Number of channels to configure, this should be the size of the array passed in. Array size should not be more than 2 as each submodule has 2 pins to output PWM
mode	PWM operation mode, options available in enumeration pwm_mode_t
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	PWM main counter clock in Hz.

Returns

Returns kStatusFail if there was error setting up the signal; kStatusSuccess otherwise

31.6.5 void PWM_UpdatePwmDutycycle (PWM_Type * *base*, pwm_submodule_t *subModule*, pwm_channels_t *pwmSignal*, pwm_mode_t *currPwmMode*, uint8_t *dutyCyclePercent*)

The function updates the PWM dutycyle to the new value that is passed in. If the dead time insertion logic is enabled then the pulse period is reduced by the dead time period specified by the user.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
pwmSignal	Signal (PWM A or PWM B) to update
currPwmMode	The current PWM mode set during PWM setup
dutyCycle-Percent	New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

31.6.6 void PWM_SetupInputCapture (PWM_Type * *base*, pwm_submodule_t *subModule*, pwm_channels_t *pwmChannel*, const pwm_input_capture_param_t * *inputCaptureParams*)

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 pin for input capture operation.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
pwmChannel	Channel in the submodule to setup
inputCapture-Params	Parameters passed in to set up the input pin

31.6.7 void PWM_SetupFaults (PWM_Type * *base,* pwm_fault_input_t *faultNum,* const pwm_fault_param_t * *faultParams*)

PWM has 4 fault inputs.

Parameters

base	PWM peripheral base address
faultNum	PWM fault to configure.
faultParams	Pointer to the PWM fault config structure

31.6.8 void PWM_SetupForceSignal (PWM_Type * *base,* pwm_submodule_t *subModule,* pwm_channels_t *pwmChannel,* pwm_force_signal_t *mode*)

The user specifies which channel to configure by supplying the submodule number and whether to modify PWM A or PWM B within that submodule.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
pwmChannel	Channel to configure
mode	Signal to output when a FORCE_OUT is triggered

31.6.9 void PWM_EnableInterrupts (PWM_Type * *base,* pwm_submodule_t *subModule,* uint32_t *mask*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
mask	The interrupts to enable. This is a logical OR of members of the enumeration pwm_-interrupt_enable_t

31.6.10 void PWM_DisableInterrupts (PWM_Type ∗ *base*, pwm_submodule_t *subModule*, uint32_t *mask*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
mask	The interrupts to enable. This is a logical OR of members of the enumeration pwm_-interrupt_enable_t

31.6.11 uint32_t PWM_GetEnabledInterrupts (PWM_Type ∗ *base*, pwm_submodule_t *subModule*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure

Returns

The enabled interrupts. This is the logical OR of members of the enumeration pwm_interrupt_-enable_t

31.6.12 uint32_t PWM_GetStatusFlags (PWM_Type ∗ *base*, pwm_submodule_t *subModule*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure

Returns

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

31.6.13 void PWM_ClearStatusFlags (PWM_Type * *base*, pwm_submodule_t *subModule*, uint32_t *mask*)

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
mask	The status flags to clear. This is a logical OR of members of the enumeration pwm_-status_flags_t

31.6.14 static void PWM_StartTimer (PWM_Type * *base*, uint8_t *subModulesToStart*) [inline], [static]

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
subModulesTo-Start	PWM submodules to start. This is a logical OR of members of the enumeration pwm-_module_control_t

31.6.15 static void PWM_StopTimer (PWM_Type * *base*, uint8_t *subModulesToStop*) [inline], [static]

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
subModulesTo-Stop	PWM submodules to stop. This is a logical OR of members of the enumeration pwm-_module_control_t

31.6.16 static void PWM_OutputTriggerEnable (PWM_Type * *base*, pwm_submodule_t *subModule*, pwm_value_register_t *valueRegister*, bool *activate*) [inline], [static]

This function allows the user to enable or disable the PWM trigger. The PWM has 2 triggers. Trigger 0 is activated when the counter matches VAL 0, VAL 2, or VAL 4 register. Trigger 1 is activated when the counter matches VAL 1, VAL 3, or VAL 5 register.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
valueRegister	Value register that will activate the trigger
activate	true: Enable the trigger; false: Disable the trigger

31.6.17 static void PWM_SetupSwCtrlOut (PWM_Type * *base*, pwm_submodule_t *subModule*, pwm_channels_t *pwmChannel*, bool *value*) [inline], [static]

The user specifies which channel to modify by supplying the submodule number and whether to modify PWM A or PWM B within that submodule.

Parameters

base	PWM peripheral base address
subModule	PWM submodule to configure
pwmChannel	Channel to configure
value	true: Supply a logic 1, false: Supply a logic 0.

31.6.18 static void PWM_SetPwmLdok (PWM_Type * *base*, uint8_t *subModulesToUpdate*, bool *value*) [inline], [static]

Set LDOK bit to load buffered values into CTRL[PRSC] and the INIT, FRACVAL and VAL registers. The values are loaded immediately if kPWM_ReloadImmediate option was choosen during config. Else the values are loaded at the next PWM reload point. This function can issue the load command to multiple submodules at the same time.

Parameters

base	PWM peripheral base address
subModulesTo-Update	PWM submodules to update with buffered values. This is a logical OR of members of the enumeration pwm_module_control_t
value	true: Set LDOK bit for the submodule list; false: Clear LDOK bit

Chapter 32
RCM: Reset Control Module Driver

32.1 Overview

The KSDK provides a Peripheral driver for the Reset Control Module (RCM) module of Kinetis devices.

Files

- file fsl_rcm.h

Data Structures

- struct rcm_reset_pin_filter_config_t
  Reset pin filter configuration. More...

Enumerations

- enum rcm_reset_source_t {
  kRCM_SourceLvd = RCM_SRS0_LVD_MASK,
  kRCM_SourceWdog = RCM_SRS0_WDOG_MASK,
  kRCM_SourcePin = RCM_SRS0_PIN_MASK,
  kRCM_SourcePor = RCM_SRS0_POR_MASK,
  kRCM_SourceLockup = RCM_SRS1_LOCKUP_MASK << 8U,
  kRCM_SourceSw = RCM_SRS1_SW_MASK << 8U,
  kRCM_SourceSackerr = RCM_SRS1_SACKERR_MASK << 8U }
  System Reset Source Name definitions.
- enum rcm_run_wait_filter_mode_t {
  kRCM_FilterDisable = 0U,
  kRCM_FilterBusClock = 1U,
  kRCM_FilterLpoClock = 2U }
  Reset pin filter select in Run and Wait modes.

Driver version

- #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  RCM driver version 2.0.1.

Reset Control Module APIs

- static uint32_t RCM_GetPreviousResetSources (RCM_Type ∗base)
  Gets the reset source status which caused a previous reset.
- void RCM_ConfigureResetPinFilter (RCM_Type ∗base, const rcm_reset_pin_filter_config_t ∗config)
  Configures the reset pin filter.

32.2 Data Structure Documentation

32.2.1 struct rcm_reset_pin_filter_config_t

Data Fields

- bool enableFilterInStop
  Reset pin filter select in stop mode.
- rcm_run_wait_filter_mode_t filterInRunWait
  Reset pin filter in run/wait mode.
- uint8_t busClockFilterCount
  Reset pin bus clock filter width.

32.2.1.0.0.77 Field Documentation

32.2.1.0.0.77.1 bool rcm_reset_pin_filter_config_t::enableFilterInStop

32.2.1.0.0.77.2 rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait

32.2.1.0.0.77.3 uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount

32.3 Macro Definition Documentation

32.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

32.4 Enumeration Type Documentation

32.4.1 enum rcm_reset_source_t

Enumerator

kRCM_SourceLvd Low-voltage detect reset.
kRCM_SourceWdog Watchdog reset.
kRCM_SourcePin External pin reset.
kRCM_SourcePor Power on reset.
kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceSackerr Parameter could get all reset flags.

32.4.2 enum rcm_run_wait_filter_mode_t

Enumerator

kRCM_FilterDisable All filtering disabled.
kRCM_FilterBusClock Bus clock filter enabled.
kRCM_FilterLpoClock LPO clock filter enabled.

32.5 Function Documentation

32.5.1 static uint32_t RCM_GetPreviousResetSources (RCM_Type * *base*) [inline], [static]

This function gets the current reset source status. Use source masks defined in the rcm_reset_source_t to get the desired source status.

Example:

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetPreviousResetSources(RCM) &
      kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (
      kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

base	RCM peripheral base address.

Returns

All reset source status bit map.

32.5.2 void RCM_ConfigureResetPinFilter (RCM_Type * *base*, const rcm_reset_pin_filter_config_t * *config*)

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

base	RCM peripheral base address.
config	Pointer to the configuration structure.

Chapter 33
SIM: System Integration Module Driver

33.1 Overview

The KSDK provides a peripheral driver for the System Integration Module (SIM) of Kinetis devices.

Files

- file fsl_sim.h

Data Structures

- struct sim_uid_t
  Unique ID. More...

Enumerations

- enum _sim_flash_mode {
  kSIM_FlashDisableInWait = SIM_FCFG1_FLASHDOZE_MASK,
  kSIM_FlashDisable = SIM_FCFG1_FLASHDIS_MASK }
  Flash enable mode.

Functions

- void SIM_GetUniqueId (sim_uid_t ∗uid)
  Get the unique identification register value.
- static void SIM_SetFlashMode (uint8_t mode)
  Set the flash enable mode.

Driver version

- #define FSL_SIM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Driver version 2.0.0.

33.2 Data Structure Documentation

33.2.1 struct sim_uid_t

Data Fields

- uint32_t MH
  UIDMH.
- uint32_t ML
  UIDML.

- uint32_t L
  UIDL.

33.2.1.0.0.78 Field Documentation

33.2.1.0.0.78.1 uint32_t sim_uid_t::MH

33.2.1.0.0.78.2 uint32_t sim_uid_t::ML

33.2.1.0.0.78.3 uint32_t sim_uid_t::L

33.3 Enumeration Type Documentation

33.3.1 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.
kSIM_FlashDisable Disable flash in normal mode.

33.4 Function Documentation

33.4.1 void SIM_GetUniqueId (sim_uid_t * *uid*)

Parameters

uid	Pointer to the structure to save the UID value.

33.4.2 static void SIM_SetFlashMode (uint8_t *mode*) [inline], [static]

Parameters

mode	The mode to set, see _sim_flash_mode for mode details.

Chapter 34
SMC: System Mode Controller Driver

34.1 Overview

The KSDK provides a peripheral driver for the System Mode Controller (SMC) module of Kinetis devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, SMC_SetPowerModexxx() function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

Files

- file fsl_smc.h

Enumerations

- enum smc_power_mode_protection_t {
  kSMC_AllowPowerModeVlp = SMC_PMPROT_AVLP_MASK,
  kSMC_AllowPowerModeAll }
  Power Modes Protection.
- enum smc_power_state_t {
  kSMC_PowerStateRun = 0x01U << 0U,
  kSMC_PowerStateStop = 0x01U << 1U,
  kSMC_PowerStateVlpr = 0x01U << 2U,
  kSMC_PowerStateVlpw = 0x01U << 3U,
  kSMC_PowerStateVlps = 0x01U << 4U }
  Power Modes in PMSTAT.
- enum smc_run_mode_t {
  kSMC_RunNormal = 0U,
  kSMC_RunVlpr = 2U }
  Run mode definition.
- enum smc_stop_mode_t {
  kSMC_StopNormal = 0U,
  kSMC_StopVlps = 2U }
  Stop mode definition.
- enum smc_partial_stop_option_t {
  kSMC_PartialStop = 0U,
  kSMC_PartialStop1 = 1U,
  kSMC_PartialStop2 = 2U }
  Partial STOP option.
- enum _smc_status { kStatus_SMC_StopAbort = MAKE_STATUS(kStatusGroup_POWER, 0) }
  SMC configuration status.

Driver version

- #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
  SMC driver version 2.0.2.

System mode controller APIs

- static void SMC_SetPowerModeProtection (SMC_Type *base, uint8_t allowedModes)
  Configures all power mode protection settings.
- static smc_power_state_t SMC_GetPowerModeState (SMC_Type *base)
  Gets the current power mode status.
- status_t SMC_SetPowerModeRun (SMC_Type *base)
  Configures the system to RUN power mode.
- status_t SMC_SetPowerModeWait (SMC_Type *base)
  Configures the system to WAIT power mode.
- status_t SMC_SetPowerModeStop (SMC_Type *base, smc_partial_stop_option_t option)
  Configures the system to Stop power mode.
- status_t SMC_SetPowerModeVlpr (SMC_Type *base)
  Configures the system to VLPR power mode.
- status_t SMC_SetPowerModeVlpw (SMC_Type *base)
  Configures the system to VLPW power mode.
- status_t SMC_SetPowerModeVlps (SMC_Type *base)
  Configures the system to VLPS power mode.

34.2 Macro Definition Documentation

34.2.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

34.3 Enumeration Type Documentation

34.3.1 enum smc_power_mode_protection_t

Enumerator

kSMC_AllowPowerModeVlp Allow Very-Low-Power Mode.
kSMC_AllowPowerModeAll Allow all power mode.

34.3.2 enum smc_power_state_t

Enumerator

kSMC_PowerStateRun 0000_0001 - Current power mode is RUN
kSMC_PowerStateStop 0000_0010 - Current power mode is STOP
kSMC_PowerStateVlpr 0000_0100 - Current power mode is VLPR
kSMC_PowerStateVlpw 0000_1000 - Current power mode is VLPW
kSMC_PowerStateVlps 0001_0000 - Current power mode is VLPS

34.3.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal normal RUN mode.
kSMC_RunVlpr Very-Low-Power RUN mode.

34.3.4 enum smc_stop_mode_t

Enumerator

kSMC_StopNormal Normal STOP mode.
kSMC_StopVlps Very-Low-Power STOP mode.

34.3.5 enum smc_partial_stop_option_t

Enumerator

kSMC_PartialStop STOP - Normal Stop mode.
kSMC_PartialStop1 Partial Stop with both system and bus clocks disabled.
kSMC_PartialStop2 Partial Stop with system clock disabled and bus clock enabled.

34.3.6 enum _smc_status

Enumerator

kStatus_SMC_StopAbort Entering Stop mode is abort.

34.4 Function Documentation

34.4.1 static void SMC_SetPowerModeProtection (SMC_Type * *base*, uint8_t *allowedModes*) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the smc_power_mode_protection_t. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map, for example, to allow LLS and VLLS, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps). To allow all modes, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll).

Parameters

base	SMC peripheral base address.
allowedModes	Bitmap of the allowed power modes.

34.4.2 static smc_power_state_t SMC_GetPowerModeState (SMC_Type ∗ *base*) [inline], [static]

This function returns the current power mode stat. Once application switches the power mode, it should always check the stat to check whether it runs into the specified mode or not. An application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc_power_state_t for information about the power stat.

Parameters

base	SMC peripheral base address.

Returns

Current power mode status.

34.4.3 status_t SMC_SetPowerModeRun (SMC_Type ∗ *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

34.4.4 status_t SMC_SetPowerModeWait (SMC_Type ∗ *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

34.4.5 status_t SMC_SetPowerModeStop (SMC_Type * *base*, smc_partial_stop_option_t *option*)

Parameters

base	SMC peripheral base address.
option	Partial Stop mode option.

Returns

SMC configuration error code.

34.4.6 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

34.4.7 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

34.4.8 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

Chapter 35
TRNG: True Random Number Generator

35.1 Overview

The KSDK provides the peripheral driver for the True Random Number Generator (TRNG) module of Kinetis devices.

The True Random Number Generator is hardware accelerator module that generates a 512-bit entropy as needed by an entropy consuming module or by other post processing functions. A typical entropy consumer is a pseudo random number generator (PRNG) which can be implemented to achieve both true randomness and cryptographic strength random numbers using the TRNG output as its entropy seed. The entropy generated by a TRNG is intended for direct use by functions that generate secret keys, per-message secrets, random challenges, and other similar quantities used in cryptographic algorithms.

35.2 TRNG Initialization

1. Define the TRNG user configuration structure. Use TRNG_InitUserConfigDefault() function to set it to default TRNG configuration values.
2. Initialize the TRNG module, call the TRNG_Init() function and pass the user configuration structure. This function automatically enables the TRNG module and its clock. After that, the TRNG is enabled and the entropy generation starts working.
3. To disable the TRNG module, call the TRNG_Deinit() function.

35.3 Get random data from TRNG

1. TRNG_GetRandomData() function gets random data from the TRNG module.

This example code shows how to initialize and get random data from the TRNG driver:

```
{
    trng_user_config_t  trngConfig;
    status_t            status;
    uint32_t            data;

    /* Initialize TRNG configuration structure to default.*/
    TRNG_InitUserConfigDefault(&trngConfig);

    /* Initialize TRNG */
    status = TRNG_Init(TRNG0, &trngConfig);

    if (status == kStatus_Success)
    {
        /* Read Random data*/
        if((status = TRNG_GetRandomData(TRNG0, data, sizeof(data))) ==
      kStatus_TRNG_Success)
        {
            /* Print data*/
            PRINTF("Random = 0x%X\r\n", i, data );

            PRINTF("Succeed.\r\n");
```

```
        }
        else
        {
            PRINTF("TRNG failed! (0x%x)\r\n", status);
        }

        /* Deinitialize TRNG*/
        TRNG_Deinit(TRNG0);
    }
    else
    {
        PRINTF("TRNG initialization failed!\r\n");
    }
}
```

Files

- file fsl_trng.h

Data Structures

- struct trng_statistical_check_limit_t
  Data structure for definition of statistical check limits. More...
- struct trng_config_t
  Data structure for the TRNG initialization. More...

Enumerations

- enum trng_sample_mode_t {
  kTRNG_SampleModeVonNeumann = 0U,
  kTRNG_SampleModeRaw = 1U,
  kTRNG_SampleModeVonNeumannRaw }
  TRNG sample mode.
- enum trng_clock_mode_t {
  kTRNG_ClockModeRingOscillator = 0U,
  kTRNG_ClockModeSystem = 1U }
  TRNG clock mode.
- enum trng_ring_osc_div_t {
  kTRNG_RingOscDiv0 = 0U,
  kTRNG_RingOscDiv2 = 1U,
  kTRNG_RingOscDiv4 = 2U,
  kTRNG_RingOscDiv8 = 3U }
  TRNG ring oscillator divide.

Functions

- status_t TRNG_GetDefaultConfig (trng_config_t ∗userConfig)
  Initializes the user configuration structure to default values.
- status_t TRNG_Init (TRNG_Type ∗base, const trng_config_t ∗userConfig)
  Initializes the TRNG.
- void TRNG_Deinit (TRNG_Type ∗base)
  Shuts down the TRNG.
- status_t TRNG_GetRandomData (TRNG_Type ∗base, void ∗data, size_t dataSize)
  Gets random data.

Driver version

- #define FSL_TRNG_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  TRNG driver version 2.0.1.

35.4 Data Structure Documentation

35.4.1 struct trng_statistical_check_limit_t

Used by trng_config_t.

Data Fields

- uint32_t maximum
  Maximum limit.
- uint32_t minimum
  Minimum limit.

35.4.1.0.0.79 Field Documentation

35.4.1.0.0.79.1 uint32_t trng_statistical_check_limit_t::maximum

35.4.1.0.0.79.2 uint32_t trng_statistical_check_limit_t::minimum

35.4.2 struct trng_config_t

This structure initializes the TRNG by calling the the TRNG_Init() function. It contains all TRNG configurations.

Data Fields

- bool lock
  Disable programmability of TRNG registers.
- trng_clock_mode_t clockMode
  Clock mode used to operate TRNG.
- trng_ring_osc_div_t ringOscDiv
  Ring oscillator divide used by TRNG.
- trng_sample_mode_t sampleMode
  Sample mode of the TRNG ring oscillator.
- uint16_t entropyDelay
  Entropy Delay.
- uint16_t sampleSize
  Sample Size.
- uint16_t sparseBitLimit
  Sparse Bit Limit which defines the maximum number of consecutive samples that may be discarded before an error is generated.
- uint8_t retryCount

Retry count.
- uint8_t longRunMaxLimit
  Largest allowable number of consecutive samples of all 1, or all 0, that is allowed during the Entropy generation.
- trng_statistical_check_limit_t monobitLimit
  Maximum and minimum limits for statistical check of number of ones/zero detected during entropy generation.
- trng_statistical_check_limit_t runBit1Limit
  Maximum and minimum limits for statistical check of number of runs of length 1 detected during entropy generation.
- trng_statistical_check_limit_t runBit2Limit
  Maximum and minimum limits for statistical check of number of runs of length 2 detected during entropy generation.
- trng_statistical_check_limit_t runBit3Limit
  Maximum and minimum limits for statistical check of number of runs of length 3 detected during entropy generation.
- trng_statistical_check_limit_t runBit4Limit
  Maximum and minimum limits for statistical check of number of runs of length 4 detected during entropy generation.
- trng_statistical_check_limit_t runBit5Limit
  Maximum and minimum limits for statistical check of number of runs of length 5 detected during entropy generation.
- trng_statistical_check_limit_t runBit6PlusLimit
  Maximum and minimum limits for statistical check of number of runs of length 6 or more detected during entropy generation.
- trng_statistical_check_limit_t pokerLimit
  Maximum and minimum limits for statistical check of "Poker Test".
- trng_statistical_check_limit_t frequencyCountLimit
  Maximum and minimum limits for statistical check of entropy sample frequency count.

35.4.2.0.0.80 Field Documentation

35.4.2.0.0.80.1 bool trng_config_t::lock

35.4.2.0.0.80.2 trng_clock_mode_t trng_config_t::clockMode

35.4.2.0.0.80.3 trng_ring_osc_div_t trng_config_t::ringOscDiv

35.4.2.0.0.80.4 trng_sample_mode_t trng_config_t::sampleMode

35.4.2.0.0.80.5 uint16_t trng_config_t::entropyDelay

Defines the length (in system clocks) of each Entropy sample taken.

35.4.2.0.0.80.6 uint16_t trng_config_t::sampleSize

Defines the total number of Entropy samples that will be taken during Entropy generation.

35.4.2.0.0.80.7 uint16_t trng_config_t::sparseBitLimit

This limit is used only for During Von Neumann sampling (enabled by TRNG_HAL_SetSampleMode()). Samples are discarded if two consecutive raw samples are both 0 or both 1. If this discarding occurs for a long period of time, it indicates that there is insufficient Entropy.

35.4.2.0.0.80.8 uint8_t trng_config_t::retryCount

It defines the number of times a statistical check may fails during the TRNG Entropy Generation before generating an error.

35.4.2.0.0.80.9 uint8_t trng_config_t::longRunMaxLimit

35.4.2.0.0.80.10 trng_statistical_check_limit_t trng_config_t::monobitLimit

35.4.2.0.0.80.11 trng_statistical_check_limit_t trng_config_t::runBit1Limit

35.4.2.0.0.80.12 trng_statistical_check_limit_t trng_config_t::runBit2Limit

35.4.2.0.0.80.13 trng_statistical_check_limit_t trng_config_t::runBit3Limit

35.4.2.0.0.80.14 trng_statistical_check_limit_t trng_config_t::runBit4Limit

35.4.2.0.0.80.15 trng_statistical_check_limit_t trng_config_t::runBit5Limit

35.4.2.0.0.80.16 trng_statistical_check_limit_t trng_config_t::runBit6PlusLimit

35.4.2.0.0.80.17 trng_statistical_check_limit_t trng_config_t::pokerLimit

35.4.2.0.0.80.18 trng_statistical_check_limit_t trng_config_t::frequencyCountLimit

35.5 Macro Definition Documentation

35.5.1 #define FSL_TRNG_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Current version: 2.0.1

Change log:

- Version 2.0.1
  - add support for KL8x and KL28Z
  - update default OSCDIV for K81 to divide by 2

35.6 Enumeration Type Documentation

35.6.1 enum trng_sample_mode_t

Used by trng_config_t.

Enumerator

kTRNG_SampleModeVonNeumann Use Von Neumann data into both Entropy shifter and Statistical Checker.
kTRNG_SampleModeRaw Use raw data into both Entropy shifter and Statistical Checker.
kTRNG_SampleModeVonNeumannRaw Use Von Neumann data into Entropy shifter. Use raw data into Statistical Checker.

35.6.2 enum trng_clock_mode_t

Used by trng_config_t.

Enumerator

kTRNG_ClockModeRingOscillator Ring oscillator is used to operate the TRNG (default).
kTRNG_ClockModeSystem System clock is used to operate the TRNG. This is for test use only, and indeterminate results may occur.

35.6.3 enum trng_ring_osc_div_t

Used by trng_config_t.

Enumerator

kTRNG_RingOscDiv0 Ring oscillator with no divide.
kTRNG_RingOscDiv2 Ring oscillator divided-by-2.
kTRNG_RingOscDiv4 Ring oscillator divided-by-4.
kTRNG_RingOscDiv8 Ring oscillator divided-by-8.

35.7 Function Documentation

35.7.1 status_t TRNG_GetDefaultConfig (trng_config_t ∗ *userConfig*)

This function initializes the configuration structure to default values. The default values are:

```
*     user_config->lock = 0;
*     user_config->clockMode = kTRNG_ClockModeRingOscillator;
*     user_config->ringOscDiv = kTRNG_RingOscDiv0;  Or  to other kTRNG_RingOscDiv[2|8]
      depending on the platform.
*     user_config->sampleMode = kTRNG_SampleModeRaw;
*     user_config->entropyDelay = 3200;
*     user_config->sampleSize = 2500;
*     user_config->sparseBitLimit = TRNG_USER_CONFIG_DEFAULT_SPARSE_BIT_LIMIT;
*     user_config->retryCount = 63;
*     user_config->longRunMaxLimit = 34;
*     user_config->monobitLimit.maximum = 1384;
*     user_config->monobitLimit.minimum = 1116;
*     user_config->runBit1Limit.maximum = 405;
```

```
*     user_config->runBit1Limit.minimum = 227;
*     user_config->runBit2Limit.maximum = 220;
*     user_config->runBit2Limit.minimum = 98;
*     user_config->runBit3Limit.maximum = 125;
*     user_config->runBit3Limit.minimum = 37;
*     user_config->runBit4Limit.maximum = 75;
*     user_config->runBit4Limit.minimum = 11;
*     user_config->runBit5Limit.maximum = 47;
*     user_config->runBit5Limit.minimum = 1;
*     user_config->runBit6PlusLimit.maximum = 47;
*     user_config->runBit6PlusLimit.minimum = 1;
*     user_config->pokerLimit.maximum = 26912;
*     user_config->pokerLimit.minimum = 24445;
*     user_config->frequencyCountLimit.maximum = 25600;
*     user_config->frequencyCountLimit.minimum = 1600;
*
```

Parameters

user_config	User configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

35.7.2 status_t TRNG_Init (TRNG_Type ∗ *base,* const trng_config_t ∗ *userConfig*)

This function initializes the TRNG. When called, the TRNG entropy generation starts immediately.

Parameters

base	TRNG base address
userConfig	Pointer to initialize configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

35.7.3 void TRNG_Deinit (TRNG_Type ∗ *base*)

This function shuts down the TRNG.

Parameters

base	TRNG base address

35.7.4 status_t TRNG_GetRandomData (TRNG_Type ∗ *base*, void ∗ *data*, size_t *dataSize*)

This function gets random data from the TRNG.

Parameters

base	TRNG base address
data	Pointer address used to store random data
dataSize	Size of the buffer pointed by the data parameter

Returns

random data

Chapter 36
UART: Universal Asynchronous Receiver/Transmitter Driver

36.1 Overview

Modules

- UART DMA Driver
- UART Driver
- UART FreeRTOS Driver
- UART eDMA Driver
- UART μCOS/II Driver
- UART μCOS/III Driver

36.2 UART Driver

36.2.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of Kinetis devices.

The UART driver includes two parts: functional APIs and transactional APIs.

Functional APIs are used for UART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the UART peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. UART functional operation groups provide the functional API set.

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the uart_handle_t as the second parameter. Initialize the handle by calling the UART_TransferCreateHandle() API.

Transactional APIs support asynchronous transfer, which means that the functions UART_TransferSendNonBlocking() and UART_TransferReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_UART_TxIdle and kStatus_UART_RxIdle.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the UART_TransferCreateHandle(). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The UART_TransferReceiveNonBlocking() function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the kStatus_UART_RxIdle.

If the receive ring buffer is full, the upper layer is informed through a callback with the kStatus_UART_RxRingBufferOverrun. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code:

```
UART_TransferCreateHandle(UART0, &handle, UART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

36.2.2 Typical use case

36.2.2.1 UART Send/receive using a polling method

```
uint8_t ch;
```

```
UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

UART_Init(UART1,&user_config,120000000U);

while(1)
{
    UART_ReadBlocking(UART1, &ch, 1);
    UART_WriteBlocking(UART1, &ch, 1);
}
```

36.2.2.2 UART Send/receive using an interrupt method

```
uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Send out.
    UART_TransferSendNonBlocking(&g_uartHandle, &g_uartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }

    // Prepare to receive.
```

```
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receive.
    UART_TransferReceiveNonBlocking(&g_uartHandle, &g_uartHandle, &
      receiveXfer);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

36.2.2.3 UART Receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE     32

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = RX_DATA_SIZE;
    rxFinished = false;

    // Receive.
    UART_TransferReceiveNonBlocking(UART1, &g_uartHandle, &receiveXfer);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data. */
    {
```

```
        ;
    }
    else
    {
        if (bytesRead) /* Received some data, process first. */
        {
            ;
        }

        // Wait receive finished.
        while (!rxFinished)
        {
        }
    }

    // ...
}
```

36.2.2.4 UART Send/Receive using the DMA method

```
uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);

    // Set up the DMA
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, UART_TX_DMA_CHANNEL, UART_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, UART_RX_DMA_CHANNEL, UART_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_RX_DMA_CHANNEL);

    DMA_Init(DMA0);
```

```
    /* Create DMA handle. */
    DMA_CreateHandle(&g_uartTxDmaHandle, DMA0, UART_TX_DMA_CHANNEL);
    DMA_CreateHandle(&g_uartRxDmaHandle, DMA0, UART_RX_DMA_CHANNEL);

    UART_TransferCreateHandleDMA(UART1, &g_uartHandle, UART_UserCallback, NULL,
       &g_uartTxDmaHandle, &g_uartRxDmaHandle);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Send out.
    UART_TransferSendDMA(UART1, &g_uartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receive.
    UART_TransferReceiveDMA(UART1, &g_uartHandle, &receiveXfer);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

Files

- file fsl_uart.h

Data Structures

- struct uart_config_t
  UART configuration structure. More...
- struct uart_transfer_t
  UART transfer structure. More...
- struct uart_handle_t
  UART handle structure. More...

Typedefs

- typedef void(∗ uart_transfer_callback_t)(UART_Type ∗base, uart_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

Enumerations

- enum _uart_status {
kStatus_UART_TxBusy = MAKE_STATUS(kStatusGroup_UART, 0),
kStatus_UART_RxBusy = MAKE_STATUS(kStatusGroup_UART, 1),
kStatus_UART_TxIdle = MAKE_STATUS(kStatusGroup_UART, 2),
kStatus_UART_RxIdle = MAKE_STATUS(kStatusGroup_UART, 3),
kStatus_UART_TxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 4),
kStatus_UART_RxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 5),
kStatus_UART_FlagCannotClearManually,
kStatus_UART_Error = MAKE_STATUS(kStatusGroup_UART, 7),
kStatus_UART_RxRingBufferOverrun = MAKE_STATUS(kStatusGroup_UART, 8),
kStatus_UART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_UART, 9),
kStatus_UART_NoiseError = MAKE_STATUS(kStatusGroup_UART, 10),
kStatus_UART_FramingError = MAKE_STATUS(kStatusGroup_UART, 11),
kStatus_UART_ParityError = MAKE_STATUS(kStatusGroup_UART, 12),
kStatus_UART_BaudrateNotSupport = MAKE_STATUS(kStatusGroup_UART, 13) }
Error codes for the UART driver.
- enum uart_parity_mode_t {
kUART_ParityDisabled = 0x0U,
kUART_ParityEven = 0x2U,
kUART_ParityOdd = 0x3U }
UART parity mode.
- enum uart_stop_bit_count_t {
kUART_OneStopBit = 0U,
kUART_TwoStopBit = 1U }
UART stop bit count.
- enum _uart_interrupt_enable {
kUART_RxActiveEdgeInterruptEnable = (UART_BDH_RXEDGIE_MASK),
kUART_TxDataRegEmptyInterruptEnable = (UART_C2_TIE_MASK << 8),
kUART_TransmissionCompleteInterruptEnable = (UART_C2_TCIE_MASK << 8),
kUART_RxDataRegFullInterruptEnable = (UART_C2_RIE_MASK << 8),
kUART_IdleLineInterruptEnable = (UART_C2_ILIE_MASK << 8),
kUART_RxOverrunInterruptEnable = (UART_C3_ORIE_MASK << 16),
kUART_NoiseErrorInterruptEnable = (UART_C3_NEIE_MASK << 16),
kUART_FramingErrorInterruptEnable = (UART_C3_FEIE_MASK << 16),
kUART_ParityErrorInterruptEnable = (UART_C3_PEIE_MASK << 16) }
UART interrupt configuration structure, default settings all disabled.
- enum _uart_flags {

kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),
kUART_RxOverrunFlag = (UART_S1_OR_MASK),
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),
kUART_FramingErrorFlag = (UART_S1_FE_MASK),
kUART_ParityErrorFlag = (UART_S1_PF_MASK),
kUART_RxActiveEdgeFlag = (UART_S2_RXEDGIF_MASK << 8),
kUART_RxActiveFlag = (UART_S2_RAF_MASK << 8) }
UART status flags.

Driver version

- #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))
  UART driver version 2.1.1.

Initialization and deinitialization

- status_t UART_Init (UART_Type ∗base, const uart_config_t ∗config, uint32_t srcClock_Hz)
  Initializes a UART instance with user configuration structure and peripheral clock.
- void UART_Deinit (UART_Type ∗base)
  Deinitializes a UART instance.
- void UART_GetDefaultConfig (uart_config_t ∗config)
  Gets the default configuration structure.
- status_t UART_SetBaudRate (UART_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the UART instance baud rate.

Status

- uint32_t UART_GetStatusFlags (UART_Type ∗base)
  Get UART status flags.
- status_t UART_ClearStatusFlags (UART_Type ∗base, uint32_t mask)
  Clears status flags with the provided mask.

Interrupts

- void UART_EnableInterrupts (UART_Type ∗base, uint32_t mask)
  Enables UART interrupts according to the provided mask.
- void UART_DisableInterrupts (UART_Type ∗base, uint32_t mask)
  Disables the UART interrupts according to the provided mask.
- uint32_t UART_GetEnabledInterrupts (UART_Type ∗base)
  Gets the enabled UART interrupts.

Bus Operations

- static void UART_EnableTx (UART_Type *base, bool enable)
  Enables or disables the UART transmitter.
- static void UART_EnableRx (UART_Type *base, bool enable)
  Enables or disables the UART receiver.
- static void UART_WriteByte (UART_Type *base, uint8_t data)
  Writes to the TX register.
- static uint8_t UART_ReadByte (UART_Type *base)
  Reads the RX register directly.
- void UART_WriteBlocking (UART_Type *base, const uint8_t *data, size_t length)
  Writes to the TX register using a blocking method.
- status_t UART_ReadBlocking (UART_Type *base, uint8_t *data, size_t length)
  Read RX data register using a blocking method.

Transactional

- void UART_TransferCreateHandle (UART_Type *base, uart_handle_t *handle, uart_transfer_callback_t callback, void *userData)
  Initializes the UART handle.
- void UART_TransferStartRingBuffer (UART_Type *base, uart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void UART_TransferStopRingBuffer (UART_Type *base, uart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- status_t UART_TransferSendNonBlocking (UART_Type *base, uart_handle_t *handle, uart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void UART_TransferAbortSend (UART_Type *base, uart_handle_t *handle)
  Aborts the interrupt driven data transmit.
- status_t UART_TransferGetSendCount (UART_Type *base, uart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been written to UART TX register.
- status_t UART_TransferReceiveNonBlocking (UART_Type *base, uart_handle_t *handle, uart_transfer_t *xfer, size_t *receivedBytes)
  Receives a buffer of data using an interrupt method.
- void UART_TransferAbortReceive (UART_Type *base, uart_handle_t *handle)
  Aborts the interrupt-driven data receiving.
- status_t UART_TransferGetReceiveCount (UART_Type *base, uart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been received.
- void UART_TransferHandleIRQ (UART_Type *base, uart_handle_t *handle)
  UART IRQ handle function.
- void UART_TransferHandleErrorIRQ (UART_Type *base, uart_handle_t *handle)
  UART Error IRQ handle function.

36.2.3 Data Structure Documentation

36.2.3.1 struct uart_config_t

Data Fields

- uint32_t baudRate_Bps
  UART baud rate.
- uart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- bool enableTx
  Enable TX.
- bool enableRx
  Enable RX.

36.2.3.2 struct uart_transfer_t

Data Fields

- uint8_t ∗ data
  The buffer of data to be transfer.
- size_t dataSize
  The byte count to be transfer.

36.2.3.2.0.81 Field Documentation

36.2.3.2.0.81.1 uint8_t∗ uart_transfer_t::data

36.2.3.2.0.81.2 size_t uart_transfer_t::dataSize

36.2.3.3 struct _uart_handle

Data Fields

- uint8_t ∗volatile txData
  Address of remaining data to send.
- volatile size_t txDataSize
  Size of the remaining data to send.
- size_t txDataSizeAll
  Size of the data to send out.
- uint8_t ∗volatile rxData
  Address of remaining data to receive.
- volatile size_t rxDataSize
  Size of the remaining data to receive.
- size_t rxDataSizeAll
  Size of the data to receive.
- uint8_t ∗ rxRingBuffer
  Start address of the receiver ring buffer.
- size_t rxRingBufferSize

Size of the ring buffer.

- volatile uint16_t rxRingBufferHead

  Index for the driver to store received data into ring buffer.
- volatile uint16_t rxRingBufferTail

  Index for the user to get data from the ring buffer.
- uart_transfer_callback_t callback

  Callback function.
- void ∗ userData

  UART callback function parameter.
- volatile uint8_t txState

  TX transfer state.
- volatile uint8_t rxState

  RX transfer state.

36.2.3.3.0.82 Field Documentation

36.2.3.3.0.82.1 uint8_t* volatile uart_handle_t::txData

36.2.3.3.0.82.2 volatile size_t uart_handle_t::txDataSize

36.2.3.3.0.82.3 size_t uart_handle_t::txDataSizeAll

36.2.3.3.0.82.4 uint8_t* volatile uart_handle_t::rxData

36.2.3.3.0.82.5 volatile size_t uart_handle_t::rxDataSize

36.2.3.3.0.82.6 size_t uart_handle_t::rxDataSizeAll

36.2.3.3.0.82.7 uint8_t* uart_handle_t::rxRingBuffer

36.2.3.3.0.82.8 size_t uart_handle_t::rxRingBufferSize

36.2.3.3.0.82.9 volatile uint16_t uart_handle_t::rxRingBufferHead

36.2.3.3.0.82.10 volatile uint16_t uart_handle_t::rxRingBufferTail

36.2.3.3.0.82.11 uart_transfer_callback_t uart_handle_t::callback

36.2.3.3.0.82.12 void* uart_handle_t::userData

36.2.3.3.0.82.13 volatile uint8_t uart_handle_t::txState

36.2.4 Macro Definition Documentation

36.2.4.1 #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

36.2.5 Typedef Documentation

36.2.5.1 typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)

36.2.6 Enumeration Type Documentation

36.2.6.1 enum _uart_status

Enumerator

kStatus_UART_TxBusy Transmitter is busy.
kStatus_UART_RxBusy Receiver is busy.
kStatus_UART_TxIdle UART transmitter is idle.
kStatus_UART_RxIdle UART receiver is idle.
kStatus_UART_TxWatermarkTooLarge TX FIFO watermark too large.

kStatus_UART_RxWatermarkTooLarge RX FIFO watermark too large.
kStatus_UART_FlagCannotClearManually UART flag can't be manually cleared.
kStatus_UART_Error Error happens on UART.
kStatus_UART_RxRingBufferOverrun UART RX software ring buffer overrun.
kStatus_UART_RxHardwareOverrun UART RX receiver overrun.
kStatus_UART_NoiseError UART noise error.
kStatus_UART_FramingError UART framing error.
kStatus_UART_ParityError UART parity error.
kStatus_UART_BaudrateNotSupport Baudrate is not support in current clock source.

36.2.6.2 enum uart_parity_mode_t

Enumerator

kUART_ParityDisabled Parity disabled.
kUART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kUART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

36.2.6.3 enum uart_stop_bit_count_t

Enumerator

kUART_OneStopBit One stop bit.
kUART_TwoStopBit Two stop bits.

36.2.6.4 enum _uart_interrupt_enable

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

Enumerator

kUART_RxActiveEdgeInterruptEnable RX active edge interrupt.
kUART_TxDataRegEmptyInterruptEnable Transmit data register empty interrupt.
kUART_TransmissionCompleteInterruptEnable Transmission complete interrupt.
kUART_RxDataRegFullInterruptEnable Receiver data register full interrupt.
kUART_IdleLineInterruptEnable Idle line interrupt.
kUART_RxOverrunInterruptEnable Receiver overrun interrupt.
kUART_NoiseErrorInterruptEnable Noise error flag interrupt.
kUART_FramingErrorInterruptEnable Framing error flag interrupt.
kUART_ParityErrorInterruptEnable Parity error flag interrupt.

36.2.6.5 enum _uart_flags

This provides constants for the UART status flags for use in the UART functions.

Enumerator

kUART_TxDataRegEmptyFlag TX data register empty flag.
kUART_TransmissionCompleteFlag Transmission complete flag.
kUART_RxDataRegFullFlag RX data register full flag.
kUART_IdleLineFlag Idle line detect flag.
kUART_RxOverrunFlag RX overrun flag.
kUART_NoiseErrorFlag RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kUART_ParityErrorFlag If parity enabled, sets upon parity error detection.
kUART_RxActiveEdgeFlag RX pin active edge interrupt flag, sets when active edge detected.
kUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.

36.2.7 Function Documentation

36.2.7.1 status_t UART_Init (UART_Type ∗ *base*, const uart_config_t ∗ *config*, uint32_t *srcClock_Hz*)

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the UART_GetDefaultConfig() function. Example below shows how to use this API to configure UART.

```
*  uart_config_t uartConfig;
*  uartConfig.baudRate_Bps = 115200U;
*  uartConfig.parityMode = kUART_ParityDisabled;
*  uartConfig.stopBitCount = kUART_OneStopBit;
*  uartConfig.txFifoWatermark = 0;
*  uartConfig.rxFifoWatermark = 1;
*  UART_Init(UART1, &uartConfig, 20000000U);
*
```

Parameters

base	UART peripheral base address.
config	Pointer to user-defined configuration structure.

srcClock_Hz	UART clock source frequency in HZ.

Return values

kStatus_UART_BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_Success	Status UART initialize succeed

36.2.7.2 void UART_Deinit (UART_Type * *base*)

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

base	UART peripheral base address.

36.2.7.3 void UART_GetDefaultConfig (uart_config_t * *config*)

This function initializes the UART configuration structure to a default value. The default values are: uartConfig->baudRate_Bps = 115200U; uartConfig->bitCountPerChar = kUART_8BitsPerChar; uartConfig->parityMode = kUART_ParityDisabled; uartConfig->stopBitCount = kUART_OneStopBit; uartConfig->txFifoWatermark = 0; uartConfig->rxFifoWatermark = 1; uartConfig->enableTx = false; uartConfig->enableRx = false;

Parameters

config	Pointer to configuration structure.

36.2.7.4 status_t UART_SetBaudRate (UART_Type * *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz*)

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the UART_Init.

```
*  UART_SetBaudRate(UART1, 115200U, 20000000U);
*
```

Parameters

base	UART peripheral base address.
baudRate_Bps	UART baudrate to be set.
srcClock_Hz	UART clock source freqency in HZ.

Return values

kStatus_UART_Baudrate-NotSupport	Baudrate is not support in current clock source.
kStatus_Success	Set baudrate succeed

36.2.7.5 uint32_t UART_GetStatusFlags (UART_Type * *base*)

This function get all UART status flags, the flags are returned as the logical OR value of the enumerators _uart_flags. To check a specific status, compare the return value with enumerators in _uart_flags. For example, to check whether the TX is empty:

```
*     if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
*     {
*         ...
*     }
*
```

Parameters

base	UART peripheral base address.

Returns

UART status flags which are ORed by the enumerators in the _uart_flags.

36.2.7.6 status_t UART_ClearStatusFlags (UART_Type * *base*, uint32_t *mask*)

This function clears UART status flags with a provided mask. Automatically cleared flag can't be cleared by this function. Some flags can only be cleared or set by hardware itself. These flags are: kUART-T_TxDataRegEmptyFlag, kUART_TransmissionCompleteFlag, kUART_RxDataRegFullFlag, kUART_-RxActiveFlag, kUART_NoiseErrorInRxDataRegFlag, kUART_ParityErrorInRxDataRegFlag, kUART_-TxFifoEmptyFlag,kUART_RxFifoEmptyFlag Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

base	UART peripheral base address.
mask	The status flags to be cleared, it is logical OR value of _uart_flags.

Return values

kStatus_UART_Flag-CannotClearManually	The flag can't be cleared by this function but it is cleared automatically by hardware.
kStatus_Success	Status in the mask are cleared.

36.2.7.7 void UART_EnableInterrupts (UART_Type * *base,* uint32_t *mask*)

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to enable TX empty interrupt and RX full interrupt:

```
*     UART_EnableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
*
```

Parameters

base	UART peripheral base address.
mask	The interrupts to enable. Logical OR of _uart_interrupt_enable.

36.2.7.8 void UART_DisableInterrupts (UART_Type * *base,* uint32_t *mask*)

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to disable TX empty interrupt and RX full interrupt:

```
*     UART_DisableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
*
```

Parameters

base	UART peripheral base address.
mask	The interrupts to disable. Logical OR of _uart_interrupt_enable.

36.2.7.9 uint32_t UART_GetEnabledInterrupts (UART_Type ∗ *base*)

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators _uart_interrupt_enable. To check a specific interrupts enable status, compare the return value with enumerators in _uart_interrupt_enable. For example, to check whether TX empty interrupt is enabled:

```
*     uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);
*
*     if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*     {
*         ...
*     }
*
```

Parameters

base	UART peripheral base address.

Returns

UART interrupt flags which are logical OR of the enumerators in _uart_interrupt_enable.

36.2.7.10 static void UART_EnableTx (UART_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the UART transmitter.

Parameters

base	UART peripheral base address.
enable	True to enable, false to disable.

36.2.7.11 static void UART_EnableRx (UART_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the UART receiver.

Parameters

base	UART peripheral base address.
enable	True to enable, false to disable.

36.2.7.12 static void UART_WriteByte (UART_Type * *base*, uint8_t *data*) [inline], [static]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

base	UART peripheral base address.
data	The byte to write.

36.2.7.13 static uint8_t UART_ReadByte (UART_Type * *base*) [inline], [static]

This function reads data from the TX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

base	UART peripheral base address.

Returns

The byte read from UART data register.

36.2.7.14 void UART_WriteBlocking (UART_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Note

This function does not check whether all the data has been sent out to the bus. Before disabling the TX, check kUART_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	UART peripheral base address.
data	Start address of the data to write.
length	Size of the data to write.

36.2.7.15 status_t UART_ReadBlocking (UART_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data and read data from the TX register.

Parameters

base	UART peripheral base address.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_UART_Rx-HardwareOverrun	Receiver overrun happened while receiving data.
kStatus_UART_Noise-Error	Noise error happened while receiving data.
kStatus_UART_Framing-Error	Framing error happened while receiving data.
kStatus_UART_Parity-Error	Parity error happened while receiving data.
kStatus_Success	Successfully received all data.

36.2.7.16 void UART_TransferCreateHandle (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_callback_t *callback*, void * *userData*)

This function initializes the UART handle which can be used for other UART transactional APIs. Usually, for a specified UART instance, call this API once to get the initialized handle.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

36.2.7.17 void UART_TransferStartRingBuffer (UART_Type * *base*, uart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note

> When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
ringBuffer	Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
ringBufferSize	size of the ring buffer.

36.2.7.18 void UART_TransferStopRingBuffer (UART_Type * *base*, uart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

36.2.7.19 status_t UART_TransferSendNonBlocking (UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*)

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the kStatus_UART_TxIdle as status parameter.

Note

> The kStatus_UART_TxIdle is passed to the upper layer when all data is written to the TX register. However it does not ensure that all data are sent out. Before disabling the TX, check the kUART_-TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART transfer structure. See uart_transfer_t.

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_UART_TxBusy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	Invalid argument.

36.2.7.20 void UART_TransferAbortSend (UART_Type ∗ *base*, uart_handle_t ∗ *handle*)

This function aborts the interrupt driven data sending. The user can get the remainBytes to find out how many bytes are still not sent out.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

36.2.7.21 status_t UART_TransferGetSendCount (UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes that have been written to UART TX register by interrupt method.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.2.7.22 status_t UART_TransferReceiveNonBlocking (UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*, size_t ∗ *receivedBytes*)

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter kStatus_UART_RxIdle. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the xfer->data and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the xfer->data[5]. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the xfer->data. When all data is received, the upper layer is notified.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART transfer structure, see uart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into transmit queue.
kStatus_UART_RxBusy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

36.2.7.23 void UART_TransferAbortReceive (UART_Type ∗ *base*, uart_handle_t ∗ *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to know how many bytes not received yet.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

36.2.7.24 status_t UART_TransferGetReceiveCount (UART_Type ∗ *base*, uart_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.2.7.25 void UART_TransferHandleIRQ (UART_Type ∗ *base*, uart_handle_t ∗ *handle*)

This function handles the UART transmit and receive IRQ request.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

36.2.7.26 void UART_TransferHandleErrorIRQ (UART_Type ∗ *base*, uart_handle_t ∗ *handle*)

This function handle the UART error IRQ request.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

36.3 UART DMA Driver

36.3.1 Overview

Files

- file fsl_uart_dma.h

Data Structures

- struct uart_dma_handle_t
  UART DMA handle. More...

Typedefs

- typedef void(∗ uart_dma_transfer_callback_t)(UART_Type ∗base, uart_dma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

EDMA transactional

- void UART_TransferCreateHandleDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txDmaHandle, dma_handle_t ∗rxDmaHandle)
  Initializes the UART handle which is used in transactional functions and sets the callback.
- status_t UART_TransferSendDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Sends data using DMA.
- status_t UART_TransferReceiveDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Receives data using DMA.
- void UART_TransferAbortSendDMA (UART_Type ∗base, uart_dma_handle_t ∗handle)
  Aborts the send data using DMA.
- void UART_TransferAbortReceiveDMA (UART_Type ∗base, uart_dma_handle_t ∗handle)
  Aborts the received data using DMA.
- status_t UART_TransferGetSendCountDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been written to UART TX register.
- status_t UART_TransferGetReceiveCountDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

36.3.2 Data Structure Documentation

36.3.2.1 struct _uart_dma_handle

Data Fields

- UART_Type ∗ base
  UART peripheral base address.
- uart_dma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  UART callback function parameter.
- size_t rxDataSizeAll
  Size of the data to receive.
- size_t txDataSizeAll
  Size of the data to send out.
- dma_handle_t ∗ txDmaHandle
  The DMA TX channel used.
- dma_handle_t ∗ rxDmaHandle
  The DMA RX channel used.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

36.3.2.1.0.83 Field Documentation

36.3.2.1.0.83.1 UART_Type* uart_dma_handle_t::base

36.3.2.1.0.83.2 uart_dma_transfer_callback_t uart_dma_handle_t::callback

36.3.2.1.0.83.3 void* uart_dma_handle_t::userData

36.3.2.1.0.83.4 size_t uart_dma_handle_t::rxDataSizeAll

36.3.2.1.0.83.5 size_t uart_dma_handle_t::txDataSizeAll

36.3.2.1.0.83.6 dma_handle_t* uart_dma_handle_t::txDmaHandle

36.3.2.1.0.83.7 dma_handle_t* uart_dma_handle_t::rxDmaHandle

36.3.2.1.0.83.8 volatile uint8_t uart_dma_handle_t::txState

36.3.3 Typedef Documentation

36.3.3.1 typedef void(* uart_dma_transfer_callback_t)(UART_Type *base, uart_dma_handle_t *handle, status_t status, void *userData)

36.3.4 Function Documentation

36.3.4.1 void UART_TransferCreateHandleDMA (UART_Type * *base,* uart_dma_handle_t * *handle,* uart_dma_transfer_callback_t *callback,* void * *userData,* dma_handle_t * *txDmaHandle,* dma_handle_t * *rxDmaHandle*)

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.
callback	UART callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested DMA handle for RX DMA transfer.
txDmaHandle	User requested DMA handle for TX DMA transfer.

36.3.4.2 status_t UART_TransferSendDMA (UART_Type ∗ *base*, uart_dma_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*)

This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART DMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

36.3.4.3 status_t UART_TransferReceiveDMA (UART_Type ∗ *base*, uart_dma_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*)

This function receives data using DMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.
xfer	UART DMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

36.3.4.4 void UART_TransferAbortSendDMA (UART_Type ∗ *base*, uart_dma_handle_t ∗ *handle*)

This function aborts the sent data using DMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.

36.3.4.5 void UART_TransferAbortReceiveDMA (UART_Type ∗ *base*, uart_dma_handle_t ∗ *handle*)

This function abort receive data which using DMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.

36.3.4.6 status_t UART_TransferGetSendCountDMA (UART_Type ∗ *base*, uart_dma_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes that have been written to UART TX register by DMA.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferIn-Progress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.3.4.7 status_t UART_TransferGetReceiveCountDMA (UART_Type * *base*, uart_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferIn-Progress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.4 UART eDMA Driver

36.4.1 Overview

Files

- file fsl_uart_edma.h

Data Structures

- struct uart_edma_handle_t
  UART eDMA handle. More...

Typedefs

- typedef void(∗ uart_edma_transfer_callback_t)(UART_Type ∗base, uart_edma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

eDMA transactional

- void UART_TransferCreateHandleEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗txEdmaHandle, edma_handle_t ∗rxEdmaHandle)
  Initializes the UART handle which is used in transactional functions.
- status_t UART_SendEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Sends data using eDMA.
- status_t UART_ReceiveEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Receive data using eDMA.
- void UART_TransferAbortSendEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle)
  Aborts the sent data using eDMA.
- void UART_TransferAbortReceiveEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle)
  Aborts the receive data using eDMA.
- status_t UART_TransferGetSendCountEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes that have been written to UART TX register.
- status_t UART_TransferGetReceiveCountEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes that have been received.

36.4.2 Data Structure Documentation

36.4.2.1 struct _uart_edma_handle

Data Fields

- uart_edma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  UART callback function parameter.
- size_t rxDataSizeAll
  Size of the data to receive.
- size_t txDataSizeAll
  Size of the data to send out.
- edma_handle_t ∗ txEdmaHandle
  The eDMA TX channel used.
- edma_handle_t ∗ rxEdmaHandle
  The eDMA RX channel used.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

36.4.2.1.0.84 Field Documentation

36.4.2.1.0.84.1 uart_edma_transfer_callback_t uart_edma_handle_t::callback

36.4.2.1.0.84.2 void* uart_edma_handle_t::userData

36.4.2.1.0.84.3 size_t uart_edma_handle_t::rxDataSizeAll

36.4.2.1.0.84.4 size_t uart_edma_handle_t::txDataSizeAll

36.4.2.1.0.84.5 edma_handle_t* uart_edma_handle_t::txEdmaHandle

36.4.2.1.0.84.6 edma_handle_t* uart_edma_handle_t::rxEdmaHandle

36.4.2.1.0.84.7 volatile uint8_t uart_edma_handle_t::txState

36.4.3 Typedef Documentation

36.4.3.1 typedef void(* uart_edma_transfer_callback_t)(UART_Type *base, uart_edma_handle_t *handle, status_t status, void *userData)

36.4.4 Function Documentation

36.4.4.1 void UART_TransferCreateHandleEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *txEdmaHandle*, edma_handle_t * *rxEdmaHandle*)

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.
callback	UART callback, NULL means no callback.
userData	User callback function data.
rxEdmaHandle	User requested DMA handle for RX DMA transfer.
txEdmaHandle	User requested DMA handle for TX DMA transfer.

36.4.4.2 status_t UART_SendEDMA (UART_Type ∗ *base*, uart_edma_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*)

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART eDMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

36.4.4.3 status_t UART_ReceiveEDMA (UART_Type ∗ *base*, uart_edma_handle_t ∗ *handle*, uart_transfer_t ∗ *xfer*)

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.
xfer	UART eDMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

36.4.4.4 void UART_TransferAbortSendEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*)

This function aborts sent data using eDMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.

36.4.4.5 void UART_TransferAbortReceiveEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*)

This function aborts receive data using eDMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.

36.4.4.6 status_t UART_TransferGetSendCountEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to UART TX register by DMA.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.4.4.7 status_t UART_TransferGetReceiveCountEDMA (UART_Type ∗ *base,* uart_edma_handle_t ∗ *handle,* uint32_t ∗ *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

36.5 UART FreeRTOS Driver

36.5.1 Overview

Files

- file fsl_uart_freertos.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t ∗handle, uart_handle_t ∗t_handle, const struct rtos_uart_config ∗cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t ∗handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

36.5.2 Data Structure Documentation

36.5.2.1 struct rtos_uart_config

Data Fields

- UART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity

Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

36.5.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

36.5.3 Function Documentation

36.5.3.1 int UART_RTOS_Init (uart_rtos_handle_t * *handle*, uart_handle_t * *t_handle*, const struct rtos_uart_config * *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
t_handle	The pointer to allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 succeed, others fail.

36.5.3.2 int UART_RTOS_Deinit (uart_rtos_handle_t * *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

36.5.3.3 int UART_RTOS_Send (uart_rtos_handle_t * *handle*, const uint8_t * *buffer*, uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

36.5.3.4 int UART_RTOS_Receive (uart_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

This function receives data from UART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to a variable of size_t where the number of received data is filled.

36.6 UART µCOS/II Driver

36.6.1 Overview

Files

- file fsl_uart_ucosii.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t ∗handle, uart_handle_t ∗t_handle, const struct rtos_uart_config ∗cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t ∗handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

36.6.2 Data Structure Documentation

36.6.2.1 struct rtos_uart_config

Data Fields

- UART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity

Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

36.6.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

36.6.3 Function Documentation

36.6.3.1 int UART_RTOS_Init (uart_rtos_handle_t * *handle,* uart_handle_t * *t_handle,* const struct rtos_uart_config * *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
uart_t_handle	The pointer to allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 Succeed, others fail.

36.6.3.2 int UART_RTOS_Deinit (uart_rtos_handle_t ∗ *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

36.6.3.3 int UART_RTOS_Send (uart_rtos_handle_t ∗ *handle,* const uint8_t ∗ *buffer,* uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

36.6.3.4 int UART_RTOS_Receive (uart_rtos_handle_t ∗ *handle,* uint8_t ∗ *buffer,* uint32_t *length,* size_t ∗ *received*)

This function receives data from UART. It is a synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to a variable of size_t where the number of received data is filled.

36.7 UART µCOS/III Driver

36.7.1 Overview

Files

- file fsl_uart_ucosiii.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t ∗handle, uart_handle_t ∗t_handle, const struct rtos_uart_config ∗cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t ∗handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

36.7.2 Data Structure Documentation

36.7.2.1 struct rtos_uart_config

Data Fields

- UART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity

Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

36.7.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

36.7.3 Function Documentation

36.7.3.1 int UART_RTOS_Init (uart_rtos_handle_t ∗ *handle*, uart_handle_t ∗ *t_handle*, const struct rtos_uart_config ∗ *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
uart_t_handle	The pointer to an allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 Succeed, others fail.

36.7.3.2 int UART_RTOS_Deinit (uart_rtos_handle_t ∗ *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

36.7.3.3 int UART_RTOS_Send (uart_rtos_handle_t ∗ *handle,* const uint8_t ∗ *buffer,* uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

36.7.3.4 int UART_RTOS_Receive (uart_rtos_handle_t ∗ *handle,* uint8_t ∗ *buffer,* uint32_t *length,* size_t ∗ *received*)

This function receives data from UART. It is a synchronous API. If any data is immediately available, it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to variable of a size_t where the number of received data is filled.

Chapter 37
WDOG: Watchdog Timer Driver

37.1 Overview

The KSDK provides a peripheral driver for the Watchdog module (WDOG) of Kinetis devices.

37.2 Typical use case

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG_Init(wdog_base,&config);
```

Files

- file fsl_wdog.h

Data Structures

- struct wdog_work_mode_t
  Defines WDOG work mode. More...
- struct wdog_config_t
  Describes WDOG configuration structure. More...
- struct wdog_test_config_t
  Describes WDOG test mode configuration structure. More...

Enumerations

- enum wdog_clock_source_t {
  kWDOG_LpoClockSource = 0U,
  kWDOG_AlternateClockSource = 1U }
  Describes WDOG clock source.
- enum wdog_clock_prescaler_t {
  kWDOG_ClockPrescalerDivide1 = 0x0U,
  kWDOG_ClockPrescalerDivide2 = 0x1U,
  kWDOG_ClockPrescalerDivide3 = 0x2U,
  kWDOG_ClockPrescalerDivide4 = 0x3U,
  kWDOG_ClockPrescalerDivide5 = 0x4U,
  kWDOG_ClockPrescalerDivide6 = 0x5U,
  kWDOG_ClockPrescalerDivide7 = 0x6U,
  kWDOG_ClockPrescalerDivide8 = 0x7U }
  Describes the selection of the clock prescaler.

- enum wdog_test_mode_t {
  kWDOG_QuickTest = 0U,
  kWDOG_ByteTest = 1U }
  Describes WDOG test mode.
- enum wdog_tested_byte_t {
  kWDOG_TestByte0 = 0U,
  kWDOG_TestByte1 = 1U,
  kWDOG_TestByte2 = 2U,
  kWDOG_TestByte3 = 3U }
  Describes WDOG tested byte selection in byte test mode.
- enum _wdog_interrupt_enable_t { kWDOG_InterruptEnable = WDOG_STCTRLH_IRQRSTEN_-MASK }
  WDOG interrupt configuration structure, default settings all disabled.
- enum _wdog_status_flags_t {
  kWDOG_RunningFlag = WDOG_STCTRLH_WDOGEN_MASK,
  kWDOG_TimeoutFlag = WDOG_STCTRLL_INTFLG_MASK }
  WDOG status flags.

Driver version

- #define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Defines WDOG driver version 2.0.0.

Unlock sequence

- #define WDOG_FIRST_WORD_OF_UNLOCK (0xC520U)
  First word of unlock sequence.
- #define WDOG_SECOND_WORD_OF_UNLOCK (0xD928U)
  Second word of unlock sequence.

Refresh sequence

- #define WDOG_FIRST_WORD_OF_REFRESH (0xA602U)
  First word of refresh sequence.
- #define WDOG_SECOND_WORD_OF_REFRESH (0xB480U)
  Second word of refresh sequence.

WDOG Initialization and De-initialization

- void WDOG_GetDefaultConfig (wdog_config_t ∗config)
  Initializes WDOG configure sturcture.
- void WDOG_Init (WDOG_Type ∗base, const wdog_config_t ∗config)
  Initializes the WDOG.
- void WDOG_Deinit (WDOG_Type ∗base)
  Shuts down the WDOG.
- void WDOG_SetTestModeConfig (WDOG_Type ∗base, wdog_test_config_t ∗config)
  Configures WDOG functional test.

WDOG Functional Operation

- static void WDOG_Enable (WDOG_Type *base)
  Enables the WDOG module.
- static void WDOG_Disable (WDOG_Type *base)
  Disables the WDOG module.
- static void WDOG_EnableInterrupts (WDOG_Type *base, uint32_t mask)
  Enable WDOG interrupt.
- static void WDOG_DisableInterrupts (WDOG_Type *base, uint32_t mask)
  Disable WDOG interrupt.
- uint32_t WDOG_GetStatusFlags (WDOG_Type *base)
  Gets WDOG all status flags.
- void WDOG_ClearStatusFlags (WDOG_Type *base, uint32_t mask)
  Clear WDOG flag.
- static void WDOG_SetTimeoutValue (WDOG_Type *base, uint32_t timeoutCount)
  Set the WDOG timeout value.
- static void WDOG_SetWindowValue (WDOG_Type *base, uint32_t windowValue)
  Sets the WDOG window value.
- static void WDOG_Unlock (WDOG_Type *base)
  Unlocks the WDOG register written.
- void WDOG_Refresh (WDOG_Type *base)
  Refreshes the WDOG timer.
- static uint16_t WDOG_GetResetCount (WDOG_Type *base)
  Gets the WDOG reset count.
- static void WDOG_ClearResetCount (WDOG_Type *base)
  Clears the WDOG reset count.

37.3 Data Structure Documentation

37.3.1 struct wdog_work_mode_t

Data Fields

- bool enableStop
  Enables or disables WDOG in stop mode.
- bool enableDebug
  Enables or disables WDOG in debug mode.

37.3.2 struct wdog_config_t

Data Fields

- bool enableWdog
  Enables or disables WDOG.
- wdog_clock_source_t clockSource
  Clock source select.
- wdog_clock_prescaler_t prescaler
  Clock prescaler value.
- wdog_work_mode_t workMode

Configures WDOG work mode in debug stop and wait mode.
- bool enableUpdate
  Update write-once register enable.
- bool enableInterrupt
  Enables or disables WDOG interrupt.
- bool enableWindowMode
  Enables or disables WDOG window mode.
- uint32_t windowValue
  Window value.
- uint32_t timeoutValue
  Timeout value.

37.3.3 struct wdog_test_config_t

Data Fields

- wdog_test_mode_t testMode
  Selects test mode.
- wdog_tested_byte_t testedByte
  Selects tested byte in byte test mode.
- uint32_t timeoutValue
  Timeout value.

37.4 Macro Definition Documentation

37.4.1 #define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

37.5 Enumeration Type Documentation

37.5.1 enum wdog_clock_source_t

Enumerator

kWDOG_LpoClockSource WDOG clock sourced from LPO.
kWDOG_AlternateClockSource WDOG clock sourced from alternate clock source.

37.5.2 enum wdog_clock_prescaler_t

Enumerator

kWDOG_ClockPrescalerDivide1 Divided by 1.
kWDOG_ClockPrescalerDivide2 Divided by 2.
kWDOG_ClockPrescalerDivide3 Divided by 3.
kWDOG_ClockPrescalerDivide4 Divided by 4.
kWDOG_ClockPrescalerDivide5 Divided by 5.

kWDOG_ClockPrescalerDivide6 Divided by 6.
kWDOG_ClockPrescalerDivide7 Divided by 7.
kWDOG_ClockPrescalerDivide8 Divided by 8.

37.5.3 enum wdog_test_mode_t

Enumerator

kWDOG_QuickTest Selects quick test.
kWDOG_ByteTest Selects byte test.

37.5.4 enum wdog_tested_byte_t

Enumerator

kWDOG_TestByte0 Byte 0 selected in byte test mode.
kWDOG_TestByte1 Byte 1 selected in byte test mode.
kWDOG_TestByte2 Byte 2 selected in byte test mode.
kWDOG_TestByte3 Byte 3 selected in byte test mode.

37.5.5 enum _wdog_interrupt_enable_t

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

Enumerator

kWDOG_InterruptEnable WDOG timeout generates an interrupt before reset.

37.5.6 enum _wdog_status_flags_t

This structure contains the WDOG status flags for use in the WDOG functions.

Enumerator

kWDOG_RunningFlag Running flag, set when WDOG is enabled.
kWDOG_TimeoutFlag Interrupt flag, set when an exception occurs.

37.6 Function Documentation

37.6.1 void WDOG_GetDefaultConfig (wdog_config_t * *config*)

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

```
*    wdogConfig->enableWdog = true;
*    wdogConfig->clockSource = kWDOG_LpoClockSource;
*    wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
*    wdogConfig->workMode.enableWait = true;
*    wdogConfig->workMode.enableStop = false;
*    wdogConfig->workMode.enableDebug = false;
*    wdogConfig->enableUpdate = true;
*    wdogConfig->enableInterrupt = false;
*    wdogConfig->enableWindowMode = false;
*    wdogConfig->windowValue = 0;
*    wdogConfig->timeoutValue = 0xFFFFU;
*
```

Parameters

config	Pointer to WDOG config structure.

See Also

wdog_config_t

37.6.2 void WDOG_Init (WDOG_Type * *base,* const wdog_config_t * *config*)

This function initializes the WDOG. When called, the WDOG runs according to the configuration. If user wants to reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in configuration.

Example:

```
*    wdog_config_t config;
*    WDOG_GetDefaultConfig(&config);
*    config.timeoutValue = 0x7ffU;
*    config.enableUpdate = true;
*    WDOG_Init(wdog_base,&config);
*
```

Parameters

base	WDOG peripheral base address
config	The configuration of WDOG

37.6.3 void WDOG_Deinit (WDOG_Type * *base*)

This function shuts down the WDOG. Make sure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

37.6.4 void WDOG_SetTestModeConfig (WDOG_Type * *base*, wdog_test_config_t * *config*)

This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Make sure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

Example:

```
*   wdog_test_config_t test_config;
*   test_config.testMode = kWDOG_QuickTest;
*   test_config.timeoutValue = 0xfffffu;
*   WDOG_SetTestModeConfig(wdog_base, &test_config);
*
```

Parameters

base	WDOG peripheral base address
config	The functional test configuration of WDOG

37.6.5 static void WDOG_Enable (WDOG_Type * *base*) [inline], [static]

This function write value into WDOG_STCTRLH register to enable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address

37.6.6 static void WDOG_Disable (WDOG_Type * *base*) [inline], [static]

This function write value into WDOG_STCTRLH register to disable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address

37.6.7 static void WDOG_EnableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function write value into WDOG_STCTRLH register to enable WDOG interrupt, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to enable The parameter can be combination of the following source if defined: • kWDOG_InterruptEnable

37.6.8 static void WDOG_DisableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function write value into WDOG_STCTRLH register to disable WDOG interrupt, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to disable The parameter can be combination of the following source if defined: • kWDOG_InterruptEnable

37.6.9 uint32_t WDOG_GetStatusFlags (WDOG_Type ∗ *base*)

This function gets all status flags.

Example for getting Running Flag:

```
*   uint32_t status;
*   status = WDOG_GetStatusFlags(wdog_base) &
      kWDOG_RunningFlag;
*
```

Parameters

base	WDOG peripheral base address

Returns

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

See Also

_wdog_status_flags_t

- true: a related status flag has been set.
- false: a related status flag is not set.

37.6.10 void WDOG_ClearStatusFlags (WDOG_Type ∗ *base,* uint32_t *mask*)

This function clears WDOG status flag.

Example for clearing timeout(interrupt) flag:

```
*   WDOG_ClearStatusFlags(wdog_base,kWDOG_TimeoutFlag);
*
```

Parameters

base	WDOG peripheral base address
mask	The status flags to clear. The parameter could be any combination of the following values: kWDOG_TimeoutFlag

37.6.11 static void WDOG_SetTimeoutValue (WDOG_Type * *base,* uint32_t *timeoutCount*) [inline], [static]

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than 2xWCT time + 20 bus clock cycles. This function write value into WDOG_TOVALH and WDOG_TOVALL registers which are wirte-once. Make sure the WCT window is still open and these two registers have not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
timeoutCount	WDOG timeout value, count of WDOG clock tick.

37.6.12 static void WDOG_SetWindowValue (WDOG_Type * *base,* uint32_t *windowValue*) [inline], [static]

This function sets the WDOG window value. This function write value into WDOG_WINH and WDOG-_WINL registers which are wirte-once. Make sure the WCT window is still open and these two registers have not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
windowValue	WDOG window value.

37.6.13 static void WDOG_Unlock (WDOG_Type * *base*) [inline], [static]

This function unlocks the WDOG register written. Before starting the unlock sequence and following congfiguration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire, After the configuration finishes, re-enable the global interrupts.

Parameters

base	WDOG peripheral base address

37.6.14 void WDOG_Refresh (WDOG_Type * *base*)

This function feeds the WDOG. This function should be called before WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

base	WDOG peripheral base address

37.6.15 static uint16_t WDOG_GetResetCount (WDOG_Type * *base*) `[inline]`, `[static]`

This function gets the WDOG reset count value.

Parameters

base	WDOG peripheral base address

Returns

WDOG reset count value.

37.6.16 static void WDOG_ClearResetCount (WDOG_Type * *base*) `[inline]`, `[static]`

This function clears the WDOG reset count value.

Parameters

base	WDOG peripheral base address

Chapter 38
XBARA: Inter-Peripheral Crossbar Switch

38.1 Overview

The KSDK provides a peripheral driver for the Inter-Peripheral Crossbar Switch (XBARA) block of Kinetis devices.

The XBARA peripheral driver configures the XBARA (Inter-Peripheral Crossbar Switch) and handles initialization and configuration of the XBARA module.

XBARA driver has two parts:

- Signal connection interconnects input and output signals.
- Active edge feature - Some of the outputs provide an active edge detection. If an active edge occurs, an interrupt or a DMA request can be called. APIs handle user callbacks for the interrupts. The driver also includes API for clearing and reading status bits.

38.2 Function

38.2.1 XBARA Initialization

To initialize the XBARA driver, a state structure has to be passed into the initialization function. This block of memory keeps pointers to user's callback functions and parameters to these functions. The XBARA module is initialized by calling the XBARA_Init() function.

38.2.2 Call diagram

1. Call the "XBARA_Init()" function to initialize the XBARA module.
2. Optionally, call the "XBARA_SetSignalsConnection()" function to Set connection between the selected XBARA_IN[∗] input and the XBARA_OUT[∗] output signal. It connects the XBARA input to the selected XBARA output. A configuration structure of the "xbara_input_signal_t" type and "xbara_output_signal_t" type is required.
3. Call the "XBARA_SetOutputSignalConfig" function to set the active edge features, such interrupts or DMA requests. A configuration structure of the "xbara_control_config_t" type is required to point to structure that keeps configuration of control register.
4. Finally, the XBARA works properly.

38.3 Typical use case

Data Structures

- struct xbara_control_config_t
  Defines the configuration structure of the XBARA control register. More...

Macros

- #define FSL_XBARA_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))
  Version 2.0.3.

Enumerations

- enum xbara_active_edge_t {
  kXBARA_EdgeNone = 0U,
  kXBARA_EdgeRising = 1U,
  kXBARA_EdgeFalling = 2U,
  kXBARA_EdgeRisingAndFalling = 3U }
  XBARA active edge for detection.
- enum xbara_request_t {
  kXBARA_RequestDisable = 0U,
  kXBARA_RequestDMAEnable = 1U,
  kXBARA_RequestInterruptEnalbe = 2U }
  Defines the XBARA DMA and interrupt configurations.
- enum xbara_status_flag_t {
  kXBARA_EdgeDetectionOut0,
  kXBARA_EdgeDetectionOut1,
  kXBARA_EdgeDetectionOut2,
  kXBARA_EdgeDetectionOut3 }
  XBARA status flags.

XBARA functional Operation.

- void XBARA_Init (XBARA_Type ∗base)
  Initializes the XBARA module.
- void XBARA_Deinit (XBARA_Type ∗base)
  Shuts down the XBARA module.
- void XBARA_SetSignalsConnection (XBARA_Type ∗base, xbar_input_signal_t input, xbar_output_signal_t output)
  Sets a connection between the selected XBARA_IN[∗] input and the XBARA_OUT[∗] output signal.
- uint32_t XBARA_GetStatusFlags (XBARA_Type ∗base)
  Gets the active edge detection status.
- void XBARA_ClearStatusFlags (XBARA_Type ∗base, uint32_t mask)
  Clears the the edge detection status flags of relative mask.
- void XBARA_SetOutputSignalConfig (XBARA_Type ∗base, xbar_output_signal_t output, const xbara_control_config_t ∗controlConfig)
  Configures the XBARA control register.

38.4 Data Structure Documentation

38.4.1 struct xbara_control_config_t

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.

Data Fields

- xbara_active_edge_t activeEdge
  Active edge to be detected.
- xbara_request_t requestType
  Selects DMA/Interrupt request.

38.4.1.0.0.85 Field Documentation

38.4.1.0.0.85.1 xbara_active_edge_t xbara_control_config_t::activeEdge

38.4.1.0.0.85.2 xbara_request_t xbara_control_config_t::requestType

38.5 Macro Definition Documentation

38.5.1 #define FSL_XBARA_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

38.6 Enumeration Type Documentation

38.6.1 enum xbara_active_edge_t

Enumerator

kXBARA_EdgeNone Edge detection status bit never asserts.
kXBARA_EdgeRising Edge detection status bit asserts on rising edges.
kXBARA_EdgeFalling Edge detection status bit asserts on falling edges.
kXBARA_EdgeRisingAndFalling Edge detection status bit asserts on rising and falling edges.

38.6.2 enum xbara_request_t

Enumerator

kXBARA_RequestDisable Interrupt and DMA are disabled.
kXBARA_RequestDMAEnable DMA enabled, interrupt disabled.
kXBARA_RequestInterruptEnalbe Interrupt enabled, DMA disabled.

38.6.3 enum xbara_status_flag_t

This provides constants for the XBARA status flags for use in the XBARA functions.

Enumerator

kXBARA_EdgeDetectionOut0 XBAR_OUT0 active edge interrupt flag, sets when active edge detected.

kXBARA_EdgeDetectionOut1 XBAR_OUT1 active edge interrupt flag, sets when active edge detected.
kXBARA_EdgeDetectionOut2 XBAR_OUT2 active edge interrupt flag, sets when active edge detected.
kXBARA_EdgeDetectionOut3 XBAR_OUT3 active edge interrupt flag, sets when active edge detected.

38.7 Function Documentation

38.7.1 void XBARA_Init (XBARA_Type * *base*)

This function un-gates the XBARA clock.

Parameters

base	XBARA peripheral address.

38.7.2 void XBARA_Deinit (XBARA_Type * *base*)

This function disables XBARA clock.

Parameters

base	XBARA peripheral address.

38.7.3 void XBARA_SetSignalsConnection (XBARA_Type * *base*, xbar_input_signal_t *input*, xbar_output_signal_t *output*)

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.
input	XBARA input signal.
output	XBARA output signal.

38.7.4 uint32_t XBARA_GetStatusFlags (XBARA_Type * *base*)

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

the mask of these status flag bits.

38.7.5 void XBARA_ClearStatusFlags (XBARA_Type * *base,* uint32_t *mask*)

Parameters

base	XBARA peripheral address.
mask	the status flags to clear.

38.7.6 void XBARA_SetOutputSignalConfig (XBARA_Type * *base,* xbar_output_signal_t *output,* const xbara_control_config_t * *controlConfig*)

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_RequestInterruptEnalbe;
XBARA_SetOutputSignalConfig(XBARA, kXBARA_OutputDMAMUX18, &userConfig);
```

Parameters

base	XBARA peripheral address.
output	XBARA output number.
controlConfig	Pointer to structure that keeps configuration of control register.

Chapter 39
XBARB: Inter-Peripheral Crossbar Switch

39.1 Overview

The KSDK provides a peripheral driver for the Inter-Peripheral Crossbar Switch (XBARB) block of Kinetis devices.

The XBARB peripheral driver configures the XBARB (Inter-Peripheral Crossbar Switch) and handles initialization and configuration of the XBARB module.

XBARB driver has two parts:

- Signal connection interconnects input and output signals.

39.2 Function groups

39.2.1 XBARB Initialization

To initialize the XBARB driver, a state structure has to be passed into the initialization function. This block of memory keeps pointers to user's callback functions and parameters to these functions. The XBARB module is initialized by calling the XBARB_Init() function.

39.2.2 Call diagram

1. Call the "XBARB_Init()" function to initialize the XBARB module.
2. Optionally, call the "XBARB_SetSignalsConnection()" function to Set connection between the selected XBARB_IN[∗] input and the XBARB_OUT[∗] output signal. It connects the XBARB input to the selected XBARB output. A configuration structure of the "xbarb_input_signal_t" type and "xbarb_output_signal_t" type is required.
3. Finally, the XBARB works properly.

39.3 Typical use case

Macros

- #define FSL_XBARB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  Version 2.0.1.

XBARB functional Operation.

- void XBARB_Init (XBARB_Type ∗base)
  Initializes the XBARB module.
- void XBARB_Deinit (XBARB_Type ∗base)

Shuts down the XBARB module.
- void XBARB_SetSignalsConnection (XBARB_Type *base, xbar_input_signal_t input, xbar_output_signal_t output)
  Configures a connection between the selected XBARB_IN[*] input and the XBARB_OUT[*] output signal.

39.4 Macro Definition Documentation

39.4.1 #define FSL_XBARB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

39.5 Function Documentation

39.5.1 void XBARB_Init (XBARB_Type * *base*)

This function un-gates the XBARB clock.

Parameters

base	XBARB peripheral address.

39.5.2 void XBARB_Deinit (XBARB_Type * *base*)

This function disables XBARB clock.

Parameters

base	XBARB peripheral address.

39.5.3 void XBARB_SetSignalsConnection (XBARB_Type * *base*, xbar_input_signal_t *input*, xbar_output_signal_t *output*)

This function configures which XBARB input is connected to the selected XBARB output. If more than one XBARB module is available, only the inputs and outputs from the same module can be connected.

Parameters

base	XBARB peripheral address.
input	XBARB input signal.

output	XBARB output signal.

Chapter 40
Debug Console

40.1 Overview

This part describes the programming interface of the debug console driver. The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

40.2 Function groups

40.2.1 Initialization

To initialize the debug console, call the DbgConsole_Init() function with these parameters. This function automatically enables the module and the clock.

```
/*
 * @brief Initializes the the peripheral used to debug messages.
 *
 * @param baseAddr      Indicates which address of the peripheral is used to send debug messages.
 * @param baudRate      The desired baud rate in bits per second.
 * @param device        Low level device type for the debug console, can be one of:
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_UART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPSCI,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_USBCDC.
 * @param clkSrcFreq    Frequency of peripheral source clock.
 *
 * @return              Whether initialization was successful or not.
 */
status_t DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device, uint32_t clkSrcFreq)
```

Selects the supported debug console hardware device type, such as

```
DEBUG_CONSOLE_DEVICE_TYPE_NONE
DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
DEBUG_CONSOLE_DEVICE_TYPE_UART
DEBUG_CONSOLE_DEVICE_TYPE_LPUART
DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral. The debug console state is stored in the debug_console_state_t structure, such as shown here:

```
typedef struct DebugConsoleState
{
    uint8_t                     type;
    void*                       base;
    debug_console_ops_t         ops;
} debug_console_state_t;
```

This example shows how to call the DbgConsole_Init() given the user configuration structure:

```
uint32_t uartClkSrcFreq = CLOCK_GetFreq(BOARD_DEBUG_UART_CLKSRC);

DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE, DEBUG_CONSOLE_DEVICE_TYPE_UART,
      uartClkSrcFreq);
```

40.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width][.precision][length]specifier", which is explained below

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
Do not support	

specifier	Description
d or i	Signed decimal integer
f	Decimal floating point
F	Decimal floating point capital letters
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer capital letters
o	Signed octal
b	Binary value
p	Pointer address
u	Unsigned decimal integer
c	Character
s	String of characters
n	Nothing printed

- Support a format specifier for SCANF following this prototype " %[∗][width][length]specifier", which is explained below

∗	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e., it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X), and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char ∗

specifier	Qualifying Input	Type of argument
i	Integer: : Number optionally preceded with a + or - sign	int *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char *
u	Unsigned decimal integer.	unsigned int *

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file:

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the KSDK printf/scanf:

```
#if SDK_DEBUGCONSOLE    /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF                 DbgConsole_Printf
#define SCANF                  DbgConsole_Scanf
#define PUTCHAR                DbgConsole_Putchar
#define GETCHAR                DbgConsole_Getchar
#else                   /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF                 printf
#define SCANF                  scanf
#define PUTCHAR                putchar
#define GETCHAR                getchar
#endif /* SDK_DEBUGCONSOLE */
```

40.3 Typical use case

Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

Print out failure messages using KSDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \" %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file ,
      line, func);
    for (;;)
    {}
}
```

Note:

To use 'printf' and 'scanf' for GNUC Base, add file **'fsl_sbrk.c'** in path: **..\{package}\devices\{subset}\utilities\fsl-_sbrk.c** to your project.

Modules

- Semihosting

40.4 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism could be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system

40.4.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging

Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

40.4.2 Guide Semihosting for Keil uVision

NOTE: Keil supports Semihosting only for M3/M4 cores.

Step 1: Prepare code

Remove function fputc and fgetc is used to support KEIL in "fsl_debug_console.c" then add the following code to project:

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;       /* used for Debug Input */
```

```
struct __FILE
{
    int handle;
};
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{ /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}
```

Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click
2. Next, select "Target" tab and not select "Use MicroLIB".
3. Next, select “Debug” tab, select “J-Link/J-Trace Cortex” and click “Setting button”.
4. Next, select “Debug” tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7

Step 4: Building the project

1. Choose “Debug” on menu bar or Ctrl F5
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer"
3. Run line by line to see result in Console Window.

40.4.3 Guide Semihosting for KDS

NOTE: After the setting we can use "printf" for debugging

Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select “Libraries” on “Cross ARM C Linker” and delete “nosys”.
3. Select “Miscellaneous” on "Cross ARM C Linker”, add “-specs=rdimon.specs” to “Other link flages” and tick “Use newlib-nano” and click OK.

Step 2: Building the project

1. In menu bar, choose Project>Build Project.

Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick “Enable semihosting and Telnet”. Press “Apply” and “Debug”.
2. After click Debug, the Window same as below, run line by line to see result in Console Window.

40.4.4 Guide Semihosting for ATL

NOTE: Hardware jlink have to be used to enable semihosting

Step 1: Prepare code

Add the following code to project:

```
int _write(int file, char *ptr, int len)
{
  /* Implement your write code here, this is used by puts and printf for example */
  int i=0;
  for(i=0 ; i<len ; i++)
    ITM_SendChar((*ptr++));
  return len;
}
```

Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Aplication" choose "-Semihosting_ATL_xxx debug jlink".
2. In tab "Debugger" setup as follows:
    - JTAG mode must be selected
    - SWV tracing must be enabled

  - Enter the Core Clock frequency. This is H/W board specific.
  - Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.
3. Click "Apply" and "Debug".

Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view “SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. Recommend not enabling other SWV trace functionalities at the same time, as this may over-use the SWO pin causing packet loss due to limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high speed). Save the SWV configuration by clicking the OK button. The configuration is saved together with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board and enable SWV trace recoding. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be resent, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

40.4.5 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-Link GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "Putty". Setup like this :
   - "Host Name (or IP address)" : localhost
   - "Port" :2333
   - "Connection type" : Telet.
   - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__heap_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}

--defsym=__heap_size__=0x2000")

Step 2: Building the project

1. Change "CMakeLists.txt":
Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=nano.specs")"
to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=rdimon.specs")"
Replace paragraph
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
To
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --specs=rdimon.specs ")
Remove

target_link_libraries(semihosting_ARMGCC.elf debug nosys)

2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

(a) Download the image and set as follows:

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

(b) Press "enter". The Putty window now shows the printf() output.

Chapter 41
Notification Framework

41.1 Overview

This section describes the programming interface of the Notifier driver.

41.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

The configuration transition includes 3 steps:

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.
   The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system changes to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This is an example to use the Notifier in the Power Manager application:

```
#include "fsl_notifier.h"

/* Definition of the Power Manager callback */
status_t callback0(notifier_notification_block_t *notify, void *data)
{

    status_t ret = kStatus_Success;

    ...
    ...
    ...

    return ret;
}
/* Definition of the Power Manager user function */
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *userData)
{
```

```
    ...
    ...
    ...
}
...
...
...
...
...
/* Main function */
int main(void)
{
    /* Define a notifier handle */
    notifier_handle_t powerModeHandle;

    /* Callback configuration */
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
                kNOTIFIER_CallbackBeforeAfter,
                (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    /* Power mode configurations */
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    /* Definition of a transition to and out the power modes */
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    /* Create Notifier handle */
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
      APP_PowerModeSwitch, NULL);
    ...
    ...
    /* Power mode switch */
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
      kNOTIFIER_PolicyAgreement);
}
```

Data Structures

- struct notifier_notification_block_t
  notification block passed to the registered callback function. More...
- struct notifier_callback_config_t
  Callback configuration structure. More...
- struct notifier_handle_t
  Notifier handle structure. More...

Typedefs

- typedef void notifier_user_config_t
  Notifier user configuration type.
- typedef status_t(∗ notifier_user_function_t)(notifier_user_config_t ∗targetConfig, void ∗userData)
  Notifier user function prototype Use this function to execute specific operations in configuration switch.

- typedef status_t(* notifier_callback_t)(notifier_notification_block_t *notify, void *data)
  Callback prototype.

Enumerations

- enum _notifier_status {
  kStatus_NOTIFIER_ErrorNotificationBefore,
  kStatus_NOTIFIER_ErrorNotificationAfter }
  Notifier error codes.
- enum notifier_policy_t {
  kNOTIFIER_PolicyAgreement,
  kNOTIFIER_PolicyForcible }
  Notifier policies.
- enum notifier_notification_type_t {
  kNOTIFIER_NotifyRecover = 0x00U,
  kNOTIFIER_NotifyBefore = 0x01U,
  kNOTIFIER_NotifyAfter = 0x02U }
  Notification type.
- enum notifier_callback_type_t {
  kNOTIFIER_CallbackBefore = 0x01U,
  kNOTIFIER_CallbackAfter = 0x02U,
  kNOTIFIER_CallbackBeforeAfter = 0x03U }
  The callback type, indicates what kinds of notification the callback handles.

Functions

- status_t NOTIFIER_CreateHandle (notifier_handle_t *notifierHandle, notifier_user_config_t **configs, uint8_t configsNumber, notifier_callback_config_t *callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void *userData)
  Create Notifier handle.
- status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)
  Switch configuration according to a pre-defined structure.
- uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)
  This function returns the last failed notification callback.

41.3 Data Structure Documentation

41.3.1 struct notifier_notification_block_t

Data Fields

- notifier_user_config_t * targetConfig
  Pointer to target configuration.
- notifier_policy_t policy
  Configure transition policy.
- notifier_notification_type_t notifyType
  Configure notification type.

41.3.1.0.0.86 Field Documentation

41.3.1.0.0.86.1 notifier_user_config_t* notifier_notification_block_t::targetConfig

41.3.1.0.0.86.2 notifier_policy_t notifier_notification_block_t::policy

41.3.1.0.0.86.3 notifier_notification_type_t notifier_notification_block_t::notifyType

41.3.2 struct notifier_callback_config_t

This structure holds configuration of callbacks. Callbacks of this type are expected to be statically allocated. This structure contains following application-defined data: callback - pointer to the callback function callbackType - specifies when the callback is called callbackData - pointer to the data passed to the callback.

Data Fields

- notifier_callback_t callback
  Pointer to the callback function.
- notifier_callback_type_t callbackType
  Callback type.
- void * callbackData
  Pointer to the data passed to the callback.

41.3.2.0.0.87 Field Documentation

41.3.2.0.0.87.1 notifier_callback_t notifier_callback_config_t::callback

41.3.2.0.0.87.2 notifier_callback_type_t notifier_callback_config_t::callbackType

41.3.2.0.0.87.3 void* notifier_callback_config_t::callbackData

41.3.3 struct notifier_handle_t

Notifier handle structure. Contains data necessary for Notifier proper function. Stores references to registered configurations, callbacks, information about their numbers, user function, user data and other internal data. NOTIFIER_CreateHandle() must be called to initialize this handle.

Data Fields

- notifier_user_config_t ** configsTable
  Pointer to configure table.
- uint8_t configsNumber
  Number of configurations.
- notifier_callback_config_t * callbacksTable
  Pointer to callback table.

- uint8_t callbacksNumber
  Maximum number of callback configurations.
- uint8_t errorCallbackIndex
  Index of callback returns error.
- uint8_t currentConfigIndex
  Index of current configuration.
- notifier_user_function_t userFunction
  user function.
- void ∗ userData
  user data passed to user function.

41.3.3.0.0.88 Field Documentation

41.3.3.0.0.88.1 notifier_user_config_t∗∗ notifier_handle_t::configsTable

41.3.3.0.0.88.2 uint8_t notifier_handle_t::configsNumber

41.3.3.0.0.88.3 notifier_callback_config_t∗ notifier_handle_t::callbacksTable

41.3.3.0.0.88.4 uint8_t notifier_handle_t::callbacksNumber

41.3.3.0.0.88.5 uint8_t notifier_handle_t::errorCallbackIndex

41.3.3.0.0.88.6 uint8_t notifier_handle_t::currentConfigIndex

41.3.3.0.0.88.7 notifier_user_function_t notifier_handle_t::userFunction

41.3.3.0.0.88.8 void∗ notifier_handle_t::userData

41.4 Typedef Documentation

41.4.1 typedef void notifier_user_config_t

Reference of user defined configuration is stored in an array, notifier switch between these configurations based on this array.

41.4.2 typedef status_t(∗ notifier_user_function_t)(notifier_user_config_t ∗targetConfig, void ∗userData)

Before and after this function execution, different notification is sent to registered callbacks. If this function returns any error code, NOTIFIER_SwitchConfig() exits.

Parameters

targetConfig	target Configuration.
userData	Refers to other specific data passed to user function.

Returns

An error code or kStatus_Success.

41.4.3 typedef status_t(∗ notifier_callback_t)(notifier_notification_block_t ∗notify, void ∗data)

Declaration of callback. It is common for registered callbacks. Reference to function of this type is part of notifier_callback_config_t callback configuration structure. Depending on callback type, function of this prototype is called (see NOTIFIER_SwitchConfig()) before configuration switch, after it or in both use cases to notify about the switch progress (see notifier_callback_type_t). When called, type of the notification is passed as parameter along with reference to the target configuration structure (see notifier_notification_block_t) and any data passed during the callback registration. When notified before configuration switch, depending on the configuration switch policy (see notifier_policy_t) the callback may deny the execution of user function by returning any error code different from kStatus_Success (see NOTIFIER_SwitchConfig()).

Parameters

notify	Notification block.
data	Callback data. Refers to the data passed during callback registration. Intended to pass any driver or application data such as internal state information.

Returns

An error code or kStatus_Success.

41.5 Enumeration Type Documentation

41.5.1 enum _notifier_status

Used as return value of Notifier functions.

Enumerator

kStatus_NOTIFIER_ErrorNotificationBefore Error occurs during send "BEFORE" notification.
kStatus_NOTIFIER_ErrorNotificationAfter Error occurs during send "AFTER" notification.

41.5.2 enum notifier_policy_t

Defines whether user function execution is forced or not. For kNOTIFIER_PolicyForcible, the user function is executed regardless of the callback results, while kNOTIFIER_PolicyAgreement policy is used to exit NOTIFIER_SwitchConfig() when any of the callbacks returns error code. See also NOTIFIER_-SwitchConfig() description.

Enumerator

kNOTIFIER_PolicyAgreement NOTIFIER_SwitchConfig() method is exited when any of the callbacks returns error code.
kNOTIFIER_PolicyForcible user function is executed regardless of the results.

41.5.3 enum notifier_notification_type_t

Used to notify registered callbacks

Enumerator

kNOTIFIER_NotifyRecover Notify IP to recover to previous work state.
kNOTIFIER_NotifyBefore Notify IP that configuration setting is going to change.
kNOTIFIER_NotifyAfter Notify IP that configuration setting has been changed.

41.5.4 enum notifier_callback_type_t

Used in the callback configuration structure (notifier_callback_config_t) to specify when the registered callback is called during configuration switch initiated by NOTIFIER_SwitchConfig(). Callback can be invoked in following situations:

- before the configuration switch (Callback return value can affect NOTIFIER_SwitchConfig() execution. See the NOTIFIER_SwitchConfig() and notifier_policy_t documentation).
- after unsuccessful attempt to switch configuration
- after successful configuration switch

Enumerator

kNOTIFIER_CallbackBefore Callback handles BEFORE notification.
kNOTIFIER_CallbackAfter Callback handles AFTER notification.
kNOTIFIER_CallbackBeforeAfter Callback handles BEFORE and AFTER notification.

41.6 Function Documentation

41.6.1 status_t NOTIFIER_CreateHandle (notifier_handle_t * *notifierHandle,* notifier_user_config_t ** *configs,* uint8_t *configsNumber,* notifier_callback-_config_t * *callbacks,* uint8_t *callbacksNumber,* notifier_user_function_t *userFunction,* void * *userData*)

Parameters

notifierHandle	A pointer to notifier handle
configs	A pointer to an array with references to all configurations which is handled by the Notifier.
configsNumber	Number of configurations. Size of the configuration array.
callbacks	A pointer to an array of callback configurations. If there are no callbacks to register during Notifier initialization, use NULL value.
callbacks-Number	Number of registered callbacks. Size of callbacks array.
userFunction	user function.
userData	user data passed to user function.

Returns

An error code or kStatus_Success.

41.6.2 status_t NOTIFIER_SwitchConfig (notifier_handle_t * *notifierHandle*, uint8_t *configIndex*, notifier_policy_t *policy*)

This function sets the system to the target configuration. Before transition, the Notifier sends notifications to all callbacks registered to the callback table. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array. The same order is used for before and after switch notifications. The notifications before the configuration switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the configuration change, further execution of this function depends on the notifier policy: the configuration change is either forced (kNOTIFIER_PolicyForcible) or exited (kNOTIFIER_PolicyAgreement). When configuration change is forced, the result of the before switch notifications are ignored. If agreement is required, if any callback returns an error code then further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked The index of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and NOTIFIER_Get-ErrorCallback() can be used to get it. Regardless of the policies, if any callback returned an error code, an error code denoting in which phase the error occurred is returned when NOTIFIER_SwitchConfig() exits.

Parameters

notifierHandle	pointer to notifier handle
configIndex	Index of the target configuration.
policy	Transaction policy, kNOTIFIER_PolicyAgreement or kNOTIFIER_PolicyForcible.

Returns

An error code or kStatus_Success.

41.6.3 uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t * *notifierHandle*)

This function returns index of the last callback that failed during the configuration switch while the last N-OTIFIER_SwitchConfig() was called. If the last NOTIFIER_SwitchConfig() call ended successfully value equal to callbacks number is returned. Returned value represents index in the array of static call-backs.

Parameters

notifierHandle	pointer to notifier handle

Returns

Callback index of last failed callback or value equal to callbacks count.

Chapter 42
Shell

42.1 Overview

This part describes the programming interface of the Shell middleware. Shell controls MCUs by commands via the specified communication peripheral based on the debug console driver.

42.2 Function groups

42.2.1 Initialization

To initialize the Shell middleware, call the SHELL_Init() function with these parameters. This function automatically enables the middleware.

```
void SHELL_Init(p_shell_context_t context, send_data_cb_t send_cb,
      recv_data_cb_t recv_cb, char *prompt);
```

Then, after the initialization was successful, call a command to control MCUs.

This example shows how to call the SHELL_Init() given the user configuration structure.

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
```

42.2.2 Advanced Feature

- Support to get a character from standard input devices.

```
static uint8_t GetChar(p_shell_context_t context);
```

Commands	Description
Help	Lists all commands which are supported by Shell.
Exit	Exits the Shell program.
strCompare	Compares the two input strings.

Input character	Description
A	Gets the latest command in the history.
B	Gets the first command in the history.
C	Replaces one character at the right of the pointer.

Input character	Description
D	Replaces one character at the left of the pointer.
	Run AutoComplete function
	Run cmdProcess function
	Clears a command.

42.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
SHELL_Main(&user_context);
```

Data Structures

- struct p_shell_context_t
  Data structure for Shell environment. More...
- struct shell_command_context_t
  User command data structure. More...
- struct shell_command_context_list_t
  Structure list command. More...

Macros

- #define SHELL_USE_HISTORY (0U)
  Macro to set on/off history feature.
- #define SHELL_SEARCH_IN_HIST (1U)
  Macro to set on/off history feature.
- #define SHELL_USE_FILE_STREAM (0U)
  Macro to select method stream.
- #define SHELL_AUTO_COMPLETE (1U)
  Macro to set on/off auto-complete feature.
- #define SHELL_BUFFER_SIZE (64U)
  Macro to set console buffer size.
- #define SHELL_MAX_ARGS (8U)
  Macro to set maximum arguments in command.
- #define SHELL_HIST_MAX (3U)
  Macro to set maximum count of history commands.
- #define SHELL_MAX_CMD (6U)
  Macro to set maximum count of commands.

Typedefs

- typedef void(∗ send_data_cb_t)(uint8_t ∗buf, uint32_t len)
  Shell user send data callback prototype.
- typedef void(∗ recv_data_cb_t)(uint8_t ∗buf, uint32_t len)
  Shell user receiver data callback prototype.
- typedef int(∗ printf_data_t)(const char ∗format,...)

Shell user printf data prototype.
- typedef int32_t(∗ cmd_function_t)(p_shell_context_t context, int32_t argc, char ∗∗argv)
  User command function prototype.

Enumerations

- enum fun_key_status_t {
  kSHELL_Normal = 0U,
  kSHELL_Special = 1U,
  kSHELL_Function = 2U }
  A type for the handle special key.

Shell functional Operation

- void SHELL_Init (p_shell_context_t context, send_data_cb_t send_cb, recv_data_cb_t recv_cb, printf_data_t shell_printf, char ∗prompt)
  Enables the clock gate and configure the Shell module according to the configuration structure.
- int32_t SHELL_RegisterCommand (const shell_command_context_t ∗command_context)
  Shell register command.
- int32_t SHELL_Main (p_shell_context_t context)
  Main loop for Shell.

42.3 Data Structure Documentation

42.3.1 struct shell_context_struct

Data Fields

- char ∗ prompt
  Prompt string.
- enum _fun_key_status stat
  Special key status.
- char line [SHELL_BUFFER_SIZE]
  Consult buffer.
- uint8_t cmd_num
  Number of user commands.
- uint8_t l_pos
  Total line position.
- uint8_t c_pos
  Current line position.
- send_data_cb_t send_data_func
  Send data interface operation.
- recv_data_cb_t recv_data_func
  Receive data interface operation.
- uint16_t hist_current
  Current history command in hist buff.
- uint16_t hist_count
  Total history command in hist buff.
- char hist_buf [SHELL_HIST_MAX][SHELL_BUFFER_SIZE]

History buffer.
- bool exit
  Exit Flag.

42.3.2 struct shell_command_context_t

Data Fields

- const char * pcCommand
  The command that is executed.
- char * pcHelpString
  String that describes how to use the command.
- const cmd_function_t pFuncCallBack
  A pointer to the callback function that returns the output generated by the command.
- uint8_t cExpectedNumberOfParameters
  Commands expect a fixed number of parameters, which may be zero.

42.3.2.0.0.89 Field Documentation

42.3.2.0.0.89.1 const char* shell_command_context_t::pcCommand

For example "help". It must be all lower case.

42.3.2.0.0.89.2 char* shell_command_context_t::pcHelpString

It should start with the command itself, and end with "\r\n". For example "help: Returns a list of all the commands\r\n".

42.3.2.0.0.89.3 const cmd_function_t shell_command_context_t::pFuncCallBack

42.3.2.0.0.89.4 uint8_t shell_command_context_t::cExpectedNumberOfParameters

42.3.3 struct shell_command_context_list_t

Data Fields

- const shell_command_context_t * CommandList [SHELL_MAX_CMD]
  The command table list.
- uint8_t numberOfCommandInList
  The total command in list.

42.4 Macro Definition Documentation

42.4.1 #define SHELL_USE_HISTORY (0U)

42.4.2 #define SHELL_SEARCH_IN_HIST (1U)

42.4.3 #define SHELL_USE_FILE_STREAM (0U)

42.4.4 #define SHELL_AUTO_COMPLETE (1U)

42.4.5 #define SHELL_BUFFER_SIZE (64U)

42.4.6 #define SHELL_MAX_ARGS (8U)

42.4.7 #define SHELL_HIST_MAX (3U)

42.4.8 #define SHELL_MAX_CMD (6U)

42.5 Typedef Documentation

42.5.1 typedef void(∗ send_data_cb_t)(uint8_t ∗buf, uint32_t len)

42.5.2 typedef void(∗ recv_data_cb_t)(uint8_t ∗buf, uint32_t len)

42.5.3 typedef int(∗ printf_data_t)(const char ∗format,...)

42.5.4 typedef int32_t(∗ cmd_function_t)(p_shell_context_t context, int32_t argc, char ∗∗argv)

42.6 Enumeration Type Documentation

42.6.1 enum fun_key_status_t

Enumerator

kSHELL_Normal Normal key.
kSHELL_Special Special key.
kSHELL_Function Function key.

42.7 Function Documentation

42.7.1 void SHELL_Init (p_shell_context_t *context,* send_data_cb_t *send_cb,* recv_data_cb_t *recv_cb,* printf_data_t *shell_printf,* char ∗ *prompt*)

This function must be called before calling all other Shell functions. Call operation the Shell commands with user-defined settings. The example below shows how to set up the middleware Shell and how to call the SHELL_Init function by passing in these parameters: Example:

```
*   shell_context_struct user_context;
*   SHELL_Init(&user_context, SendDataFunc, ReceiveDataFunc, "SHELL>> ");
*
```

Parameters

context	The pointer to the Shell environment and runtime states.
send_cb	The pointer to call back send data function.
recv_cb	The pointer to call back receive data function.
prompt	The string prompt of Shell

42.7.2 int32_t SHELL_RegisterCommand (const shell_command_context_t ∗ *command_context*)

Parameters

command_-context	The pointer to the command data structure.

Returns

-1 if error or 0 if success

42.7.3 int32_t SHELL_Main (p_shell_context_t *context*)

Main loop for Shell; After this function is called, Shell begins to initialize the basic variables and starts to work.

Parameters

context	The pointer to the Shell environment and runtime states.

Returns

This function does not return until Shell command exit was called.

Chapter 43
DMA Manager

43.1 Overview

DMA Manager provides a series of functions to manage the DMAMUX channels.

43.2 Function groups

43.2.1 DMAMGR Initialization and De-initialization

This function group initializes and deinitializes the DMA Manager.

43.2.2 DMAMGR Operation

This function group requests/releases the DMAMUX channel and configures the channel request source.

43.3 Typical use case

43.3.1 DMAMGR static channel allocate

```
DMAMUX_Type *dmamux_base;
uint8_t channel;

/* Initialize DMAMGR */
DMAMGR_Init();
/* Request a DMAMUX channel by static allocate mechanism */
dmamux_base = DMAMUX0;
channel = 0;
DMAMGR_RequestChannel(kDmaRequestMux0AlwaysOn63, &dmamux_base, &channel,
      kDMAMGR_STATIC_ALLOCATE);
```

43.3.2 DMAMGR dynamic channel allocate

```
DMAMUX_Type *dmamux_base;
uint8_t channel;

/* Initialize DMAMGR */
DMAMGR_Init();
/* Request a DMAMUX channel by static allocate mechanism */
dmamux_base = DMAMUX0;
channel = 0;
DMAMGR_RequestChannel(kDmaRequestMux0AlwaysOn63, &dmamux_base, &channel,
      kDMAMGR_DYNAMIC_ALLOCATE);
```

Macros

- #define DMAMGR_DYNAMIC_ALLOCATE 0xFFU

Dynamic channel allocate mechanism.

Enumerations

- enum _dma_manager_status {
kStatus_DMAMGR_ChannelOccupied = MAKE_STATUS(kStatusGroup_DMAMGR, 0),
kStatus_DMAMGR_ChannelNotUsed = MAKE_STATUS(kStatusGroup_DMAMGR, 1),
kStatus_DMAMGR_NoFreeChannel = MAKE_STATUS(kStatusGroup_DMAMGR, 2),
kStatus_DMAMGR_ChannelNotMatchSource = MAKE_STATUS(kStatusGroup_DMAMGR, 3)
}
  DMA manager status.

DMAMGR Initialize and De-initialize

- void DMAMGR_Init (void)
  Initializes the DAM manager.
- void DMAMGR_Deinit (void)
  Deinitializes the DMA manager.

DMAMGR Operation

- status_t DMAMGR_RequestChannel (dma_request_source_t requestSource, uint8_t virtualChannel, void ∗handle)
  Requests a DMA channel.
- status_t DMAMGR_ReleaseChannel (void ∗handle)
  Releases a DMA channel.

43.4 Macro Definition Documentation

43.4.1 #define DMAMGR_DYNAMIC_ALLOCATE 0xFFU

43.5 Enumeration Type Documentation

43.5.1 enum _dma_manager_status

Enumerator

kStatus_DMAMGR_ChannelOccupied Channel has been occupied.
kStatus_DMAMGR_ChannelNotUsed Channel has not been used.
kStatus_DMAMGR_NoFreeChannel All channels have been occupied.
kStatus_DMAMGR_ChannelNotMatchSource Channels do not match the request source.

43.6 Function Documentation

43.6.1 void DMAMGR_Init (void)

This function initializes the DMA manager, ungates all DMAMUX clocks, and initializes the eDMA or DMA peripheral.

43.6.2 void DMAMGR_Deinit (void)

This function deinitializes the DMA manager, disables all DMAMUX channels, gates all DMAMUX clocks, and deinitializes the eDMA or DMA peripheral.

43.6.3 status_t DMAMGR_RequestChannel (dma_request_source_t *requestSource,* uint8_t *virtualChannel,* void ∗ *handle*)

This function requests a DMA channel which is not occupied. The two channels to allocate the mechanism are dynamic and static channels. For the dynamic allocation mechanism (virtualChannel = DMAMGR_-DYNAMIC_ALLOCATE), DMAMGR allocates a DMA channel according to the given request source and then configure it. For static allocation mechanism, DMAMGR configures the given channel according to the given request source and channel number.

Parameters

requestSource	DMA channel request source number. See the soc.h.
virtualChannel	The channel number user wants to occupy. If using the dynamic channel allocate mechanism, set the virtualChannel equal to DMAMGR_DYNAMIC_ALLOCATE.
handle	DMA or EDMA handle pointer.

Return values

kStatus_Success	In dynamic/static channel allocate mechanism, allocate DMAMUX channel successfully.
kStatus_DMAMGR_No-FreeChannel	In dynamic channel allocate mechanism, all DMAMUX channels has been occupied.
kStatus_DMAMGR_-ChannelNotMatchSource	In static channel allocate mechanism, the given channel do not match the given request.
kStatus_DMAMGR_-ChannelOccupied	In static channel allocate mechanism, the given channel has been occupied.

43.6.4 status_t DMAMGR_ReleaseChannel (void ∗ *handle*)

This function releases an occupied DMA channel.

Parameters

handle	DMA or eDMA handle pointer.

Return values

kStatus_Success	Release the given channel successfully.
kStatus_DMAMGR_-ChannelNotUsed	The given channel which to be released is not been used before.

Chapter 44
Memory-Mapped Cryptographic Acceleration Unit (MMCAU)

44.1 Overview

The Kinetis mmCAU software library uses the mmCAU co-processor that is connected to the Kinetis ARM Cortex-M4/M0+ Private Peripheral Bus (PPB). In this chapter, CAU refers to both CAU and mmCAU unless explicitly noted.

44.2 Purpose

The following chapter describes how to use the mmCAU software library in any application to integrate a cryptographic algorithm or hashing function supported by the software library. Freescale products supported by the software library are Kinetis MCU/MPUs. Check the specific Freescale product for CAU availability.

44.3 Library Features

The library is as compact and generic as possible to simplify the integration with existing cryptographic software. The library has a standard header file with ANSI C prototypes for all functions: "cau_api.h". This software library is thread safe only if CAU registers are saved on a context switch. The Kinetis mmCAU software library is also compatible to ARM C compiler conventions (EABI). All pointers passed to mmCAU API functions (input and output data blocks, keys, key schedules, and so on) are aligned to 0-modulo-4 addresses.

For applications that don't need to deal with the aligned addresses, a simple wrapper layer is provided. The wrapper layer consists of the "fsl_mmcau.h" header file and "fsl_mmcau.c" source code file. The only function of the wrapper layer is that it supports unaligned addresses

. The CAU library supports the following encryption/decryption algorithms and hashing functions:

- AES128
- AES192
- AES256
- DES
- MD5
- SHA1
- SHA256

Note: 3DES crypto algorithms are supported by calling the corresponding DES crypto function three times. Hardware support for SHA256 is only present in the CAU version 2. See the appropriate MCU/MPU reference manual for details about availability. Additionally, the cau_sha256_initialize_output() function checks the hardware revision and returns a (-1) value if the CAU lacks SHA256 support.

44.4 CAU and mmCAU software library overview

Table 1 shows the crypto algorithms and hashing functions included in the software library:

Crypto Algorithms	AES128 AES192 AES256	cau_aes_set_key
		cau_aes_encrypt
		cau_aes_decrypt
	DES/3DES	cau_des_chk_parity
		cau_des_encrypt
		cau_des_decrypt
Hashing Functions	MD5	cau_md5_initialize_output
		cau_md5_hash_n
		cau_md5_update
		cau_md5_hash
	SHA1	cau_sha1_initialize_output
		cau_sha1_hash_n
		cau_sha1_update
		cau_sha1_hash
	SHA256	cau_sha256_initialize_output
		cau_sha256_hash_n
		cau_sha256_update
		cau_sha256_hash

Table 1: Library Overview

44.5 mmCAU software library usage

The software library contains the following files:

File	Description
cau_api.h	CAU and mmCAU header file
lib_mmcau.a	mmCAU library: Kinetis

Table 2: File Description

The header file and lib_mmcau.a must always be included in the project.

Functions

- void cau_aes_set_key (const unsigned char ∗key, const int key_size, unsigned char ∗key_sch)

AES: Performs an AES key expansion.
- void cau_aes_encrypt (const unsigned char ∗in, const unsigned char ∗key_sch, const int nr, unsigned char ∗out)
  AES: Encrypts a single 16 byte block.
- void cau_aes_decrypt (const unsigned char ∗in, const unsigned char ∗key_sch, const int nr, unsigned char ∗out)
  AES: Decrypts a single 16-byte block.
- int cau_des_chk_parity (const unsigned char ∗key)
  DES: Checks key parity.
- void cau_des_encrypt (const unsigned char ∗in, const unsigned char ∗key, unsigned char ∗out)
  DES: Encrypts a single 8-byte block.
- void cau_des_decrypt (const unsigned char ∗in, const unsigned char ∗key, unsigned char ∗out)
  DES: Decrypts a single 8-byte block.
- void cau_md5_initialize_output (const unsigned char ∗md5_state)
  MD5: Initializes the MD5 state variables.
- void cau_md5_hash_n (const unsigned char ∗msg_data, const int num_blks, unsigned char ∗md5_-state)
  MD5: Updates MD5 state variables with n message blocks.
- void cau_md5_update (const unsigned char ∗msg_data, const int num_blks, unsigned char ∗md5_-state)
  MD5: Updates MD5 state variables.
- void cau_md5_hash (const unsigned char ∗msg_data, unsigned char ∗md5_state)
  MD5: Updates MD5 state variables with one message block.
- void cau_sha1_initialize_output (const unsigned int ∗sha1_state)
  SHA1: Initializes the SHA1 state variables.
- void cau_sha1_hash_n (const unsigned char ∗msg_data, const int num_blks, unsigned int ∗sha1_-state)
  SHA1: Updates SHA1 state variables with n message blocks.
- void cau_sha1_update (const unsigned char ∗msg_data, const int num_blks, unsigned int ∗sha1_-state)
  SHA1: Updates SHA1 state variables.
- void cau_sha1_hash (const unsigned char ∗msg_data, unsigned int ∗sha1_state)
  SHA1: Updates SHA1 state variables with one message block.
- int cau_sha256_initialize_output (const unsigned int ∗output)
  SHA256: Initializes the SHA256 state variables.
- void cau_sha256_hash_n (const unsigned char ∗input, const int num_blks, unsigned int ∗output)
  SHA256: Updates SHA256 state variables with n message blocks.
- void cau_sha256_update (const unsigned char ∗input, const int num_blks, unsigned int ∗output)
  SHA256: Updates SHA256 state variables.
- void cau_sha256_hash (const unsigned char ∗input, unsigned int ∗output)
  SHA256: Updates SHA256 state variables with one message block.
- status_t MMCAU_AES_SetKey (const uint8_t ∗key, const size_t keySize, uint8_t ∗keySch)
  AES: Performs an AES key expansion.
- status_t MMCAU_AES_EncryptEcb (const uint8_t ∗in, const uint8_t ∗keySch, uint32_t aesRounds, uint8_t ∗out)
  AES: Encrypts a single 16 byte block.
- status_t MMCAU_AES_DecryptEcb (const uint8_t ∗in, const uint8_t ∗keySch, uint32_t aesRounds, uint8_t ∗out)
  AES: Decrypts a single 16-byte block.
- status_t MMCAU_DES_ChkParity (const uint8_t ∗key)

DES: Checks the key parity.
- status_t MMCAU_DES_EncryptEcb (const uint8_t ∗in, const uint8_t ∗key, uint8_t ∗out)
  DES: Encrypts a single 8-byte block.
- status_t MMCAU_DES_DecryptEcb (const uint8_t ∗in, const uint8_t ∗key, uint8_t ∗out)
  DES: Decrypts a single 8-byte block.
- status_t MMCAU_MD5_InitializeOutput (uint32_t ∗md5State)
  MD5: Initializes the MD5 state variables.
- status_t MMCAU_MD5_HashN (const uint8_t ∗msgData, uint32_t numBlocks, uint32_t ∗md5-State)
  MD5: Updates the MD5 state variables with n message blocks.
- status_t MMCAU_MD5_Update (const uint8_t ∗msgData, uint32_t numBlocks, uint32_t ∗md5-State)
  MD5: Updates the MD5 state variables.
- status_t MMCAU_SHA1_InitializeOutput (uint32_t ∗sha1State)
  SHA1: Initializes the SHA1 state variables.
- status_t MMCAU_SHA1_HashN (const uint8_t ∗msgData, uint32_t numBlocks, uint32_t ∗sha1-State)
  SHA1: Updates the SHA1 state variables with n message blocks.
- status_t MMCAU_SHA1_Update (const uint8_t ∗msgData, uint32_t numBlocks, uint32_t ∗sha1-State)
  SHA1: Updates the SHA1 state variables.
- status_t MMCAU_SHA256_InitializeOutput (uint32_t ∗sha256State)
  SHA256: Initializes the SHA256 state variables.
- status_t MMCAU_SHA256_HashN (const uint8_t ∗input, uint32_t numBlocks, uint32_t ∗sha256-State)
  SHA256: Updates the SHA256 state variables with n message blocks.
- status_t MMCAU_SHA256_Update (const uint8_t ∗input, uint32_t numBlocks, uint32_t ∗sha256-State)
  SHA256: Updates SHA256 state variables.

44.6 Function Documentation

44.6.1 void cau_aes_set_key (const unsigned char ∗ *key*, const int *key_size*, unsigned char ∗ *key_sch*)

This function performs an AES key expansion

Parameters

	key	Pointer to input key (128, 192, 256 bits in length).
	key_size	Key size in bits (128, 192, 256)
out	*key_sch*	Pointer to key schedule output (44, 52, 60 longwords)

Note

All pointers must have word (4 bytes) alignment
Table below shows the requirements for the cau_aes_set_key() function when using AES128, AE-S192 or AES256.

[in] Key Size (bits)	[out] Key Schedule Size (32 bit data values)
128	44
192	52
256	60

44.6.2 void cau_aes_encrypt (const unsigned char ∗ *in*, const unsigned char ∗ *key_sch*, const int *nr*, unsigned char ∗ *out*)

This function encrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

	in	Pointer to 16-byte block of input plaintext
	key_sch	Pointer to key schedule (44, 52, 60 longwords)
	nr	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	*out*	Pointer to 16-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.
Table below shows the requirements for the cau_aes_encrypt()/cau_aes_decrypt() function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

44.6.3 void cau_aes_decrypt (const unsigned char ∗ *in*, const unsigned char ∗ *key_sch*, const int *nr*, unsigned char ∗ *out*)

This function decrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

	in	Pointer to 16-byte block of input ciphertext
	key_sch	Pointer to key schedule (44, 52, 60 longwords)
	nr	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	*out*	Pointer to 16-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.
Table below shows the requirements for the cau_aes_encrypt()/cau_aes_decrypt() function when using AES128, AES192 or AES256.
Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:--------—:	:------------------------------—:	:----------------------—:
AES128	44	10
AES192	52	12
AES256	60	14

44.6.4 int cau_des_chk_parity (const unsigned char ∗ *key*)

This function checks the parity of a DES key

Parameters

key	64-bit DES key with parity bits. Must have word (4 bytes) alignment.

Returns

0 no error
-1 parity error

44.6.5 void cau_des_encrypt (const unsigned char ∗ *in,* const unsigned char ∗ *key,* unsigned char ∗ *out*)

This function encrypts a single 8-byte block with DES algorithm.

Parameters

	in	Pointer to 8-byte block of input plaintext
	key	Pointer to 64-bit DES key with parity bits
out	*out*	Pointer to 8-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.

44.6.6 void cau_des_decrypt (const unsigned char * *in,* const unsigned char * *key,* unsigned char * *out*)

This function decrypts a single 8-byte block with DES algorithm.

Parameters

	in	Pointer to 8-byte block of input ciphertext
	key	Pointer to 64-bit DES key with parity bits
out	*out*	Pointer to 8-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.

44.6.7 void cau_md5_initialize_output (const unsigned char * *md5_state*)

This function initializes the MD5 state variables. The output can be used as input to cau_md5_hash() and cau_md5_hash_n().

Parameters

out	*md5_state*	Pointer to 128-bit block of md5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

44.6.8 void cau_md5_hash_n (const unsigned char * *msg_data*, const int *num_blks*, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
`in,out`	*md5_state*	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_md5_initialize_output() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.9 void cau_md5_update (const unsigned char * *msg_data*, const int *num_blks*, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls cau_md5_initialize_output() first.

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
`out`	*md5_state*	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_md5_initialize_output() function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

44.6.10 void cau_md5_hash (const unsigned char * *msg_data*, unsigned char * *md5_state*)

This function updates MD5 state variables for one input message block

Parameters

	msg_data	Pointer to start of 512-bits of input message data
in,out	*md5_state*	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_md5_initialize_output() function must be called when starting a new hash.

44.6.11 void cau_sha1_initialize_output (const unsigned int * *sha1_state*)

This function initializes the SHA1 state variables. The output can be used as input to cau_sha1_hash() and cau_sha1_hash_n().

Parameters

out	*sha1_state*	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

44.6.12 void cau_sha1_hash_n (const unsigned char * *msg_data,* const int *num_blks,* unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
in,out	*sha1_state*	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha1_initialize_output() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.13 void cau_sha1_update (const unsigned char * *msg_data*, const int *num_blks*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls cau_sha1_initialize_output() first.

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
`out`	*sha1_state*	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha1_initialize_output() function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

44.6.14 void cau_sha1_hash (const unsigned char * *msg_data*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one input message block

Parameters

	msg_data	Pointer to start of 512-bits of input message data
`in,out`	*sha1_state*	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha1_initialize_output() function must be called when starting a new hash.

44.6.15 int cau_sha256_initialize_output (const unsigned int * *output*)

This function initializes the SHA256 state variables. The output can be used as input to cau_sha256_hash() and cau_sha256_hash_n().

Parameters

`out`	*sha256_state*	Pointer to 256-bit block of SHA2 state variables a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Returns

0 No error. CAU hardware support for SHA256 is present.
-1 Error. CAU hardware support for SHA256 is not present.

44.6.16 void cau_sha256_hash_n (const unsigned char * *input*, const int *num_blks*, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
`in,out`	*sha256_state*	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha256_initialize_output() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.17 void cau_sha256_update (const unsigned char * *input*, const int *num_blks*, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls cau_sha256_initialize_output() first.

Parameters

	msg_data	Pointer to start of input message data
	num_blks	Number of 512-bit blocks to process
`out`	*sha256_state*	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha256_initialize_output() function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

44.6.18 void cau_sha256_hash (const unsigned char * *input,* unsigned int * *output*)

This function updates SHA256 state variables for one input message block

Parameters

	msg_data	Pointer to start of 512-bits of input message data
in,out	*sha256_state*	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment
Input message and digest output blocks must not overlap. The cau_sha256_initialize_output() function must be called when starting a new hash.

44.6.19 status_t MMCAU_AES_SetKey (const uint8_t * *key*, const size_t *keySize*, uint8_t * *keySch*)

This function performs an AES key expansion.

Parameters

	key	Pointer to input key (128, 192, 256 bits in length).
	keySize	Key size in bytes (16, 24, 32)
out	*keySch*	Pointer to key schedule output (44, 52, 60 longwords)

Note

Table below shows the requirements for the MMCAU_AES_SetKey() function when using AES128, AES192, or AES256.
[in] Key Size (bits)	[out] Key Schedule Size (32 bit data values)
:-------------------—:	:--—:
128	44
192	52
256	60

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

44.6.20 status_t MMCAU_AES_EncryptEcb (const uint8_t * *in*, const uint8_t * *keySch*, uint32_t *aesRounds*, uint8_t * *out*)

This function encrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	in	Pointer to 16-byte block of input plaintext.
	keySch	Pointer to key schedule (44, 52, 60 longwords).
	aesRounds	Number of AES rounds (10, 12, 14 = f(key_schedule)).
`out`	*out*	Pointer to 16-byte block of output ciphertext.

Note

Input and output blocks may overlap.
Table below shows the requirements for the MMCAU_AES_EncryptEcb()/MMCAU_AES_-DecryptEcb() function when using AES128, AES192 or AES256.
Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:--------—:	:------------------------------—:	:---------------------—:
AES128	44	10
AES192	52	12
AES256	60	14

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

44.6.21 status_t MMCAU_AES_DecryptEcb (const uint8_t ∗ *in*, const uint8_t ∗ *keySch*, uint32_t *aesRounds*, uint8_t ∗ *out*)

This function decrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	in	Pointer to 16-byte block of input ciphertext.
	keySch	Pointer to key schedule (44, 52, 60 longwords).
	aesRounds	Number of AES rounds (10, 12, 14 = f(key_schedule)).
`out`	*out*	Pointer to 16-byte block of output plaintext.

Note

Input and output blocks may overlap.
Table below shows the requirements for the cau_aes_encrypt()/cau_aes_decrypt(). function when using AES128, AES192 or AES256.
| Block Cipher | [in] Key Schedule Size (longwords) | [in] Number of AES rounds |
| :--------—: | :------------------------------—: | :---------------------—: |

AES128	44	10
AES192	52	12
AES256	60	14

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

44.6.22 status_t MMCAU_DES_ChkParity (const uint8_t * *key*)

This function checks the parity of a DES key.

Parameters

key	64-bit DES key with parity bits.

Returns

kStatus_Success No error.
kStatus_Fail Parity error.
kStatus_InvalidArgument Key argument is NULL.

44.6.23 status_t MMCAU_DES_EncryptEcb (const uint8_t * *in,* const uint8_t * *key,* uint8_t * *out*)

This function encrypts a single 8-byte block with the DES algorithm.

Parameters

	in	Pointer to 8-byte block of input plaintext.
	key	Pointer to 64-bit DES key with parity bits.
`out`	*out*	Pointer to 8-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

44.6.24 status_t MMCAU_DES_DecryptEcb (const uint8_t * *in*, const uint8_t * *key*, uint8_t * *out*)

This function decrypts a single 8-byte block with the DES algorithm.

Parameters

	in	Pointer to 8-byte block of input ciphertext.
	key	Pointer to 64-bit DES key with parity bits.
out	*out*	Pointer to 8-byte block of output plaintext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

44.6.25 status_t MMCAU_MD5_InitializeOutput (uint32_t ∗ *md5State*)

This function initializes the MD5 state variables. The output can be used as input to MMCAU_MD5_-HashN().

Parameters

out	*md5State*	Pointer to 128-bit block of md5 state variables: a,b,c,d

44.6.26 status_t MMCAU_MD5_HashN (const uint8_t ∗ *msgData,* uint32_t *numBlocks,* uint32_t ∗ *md5State*)

This function updates the MD5 state variables for one or more input message blocks.

Parameters

	msgData	Pointer to start of input message data.
	numBlocks	Number of 512-bit blocks to process.
in,out	*md5State*	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The MMCAU_MD5_InitializeOutput() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.27 status_t MMCAU_MD5_Update (const uint8_t * *msgData*, uint32_t *numBlocks*, uint32_t * *md5State*)

This function updates the MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls MMCAU_MD5_InitializeOutput() first.

Parameters

	msgData	Pointer to start of input message data.
	numBlocks	Number of 512-bit blocks to process.
out	*md5State*	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The MMCAU_MD5_InitializeOutput() function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

44.6.28 status_t MMCAU_SHA1_InitializeOutput (uint32_t ∗ *sha1State*)

This function initializes the SHA1 state variables. The output can be used as input to MMCAU_SHA1_-HashN().

Parameters

out	*sha1State*	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

44.6.29 status_t MMCAU_SHA1_HashN (const uint8_t ∗ *msgData*, uint32_t *numBlocks*, uint32_t ∗ *sha1State*)

This function updates the SHA1 state variables for one or more input message blocks.

Parameters

	msgData	Pointer to start of input message data.
	numBlocks	Number of 512-bit blocks to process.
in,out	*sha1State*	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The MMCAU_SHA1_InitializeOutput() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.30 status_t MMCAU_SHA1_Update (const uint8_t * *msgData*, uint32_t *numBlocks*, uint32_t * *sha1State*)

This function updates the SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls MMCAU_SHA1_InitializeOutput() first.

Parameters

	msgData	Pointer to start of input message data.
	numBlocks	Number of 512-bit blocks to process.
out	*sha1State*	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The MMCAU_SHA1_InitializeOutput() function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

44.6.31 status_t MMCAU_SHA256_InitializeOutput (uint32_t * *sha256State*)

This function initializes the SHA256 state variables. The output can be used as input to MMCAU_SHA256_HashN().

Parameters

out	*sha256State*	Pointer to 256-bit block of SHA2 state variables a, b, c, d, e, f, g, h.

Returns

kStatus_Success No error. CAU hardware support for SHA256 is present.
kStatus_Fail Error. CAU hardware support for SHA256 is not present.
kStatus_InvalidArgument Error. sha256State is NULL.

44.6.32 status_t MMCAU_SHA256_HashN (const uint8_t * *input,* uint32_t *numBlocks,* uint32_t * *sha256State*)

This function updates SHA256 state variables for one or more input message blocks.

Parameters

	msgData	Pointer to start of input message data.

	numBlocks	Number of 512-bit blocks to process.
in,out	*sha256State*	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Note

Input message and digest output blocks must not overlap. The MMCAU_SHA256_InitializeOutput() function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

44.6.33 status_t MMCAU_SHA256_Update (const uint8_t ∗ *input*, uint32_t *numBlocks*, uint32_t ∗ *sha256State*)

This function updates the SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls cau_sha256_initialize_output() first.

Parameters

	msgData	Pointer to start of input message data.
	numBlocks	Number of 512-bit blocks to process.
out	*sha256State*	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Note

Input message and digest output blocks must not overlap. The MMCAU_SHA256_InitializeOutput() function is not required to be called. as it is called internally to start a new hash. All input message blocks must be contiguous.

Chapter 45
Secured Digital Card/Embedded MultiMedia Card (CARD)

45.1 Overview

The Kinetis SDK provides a driver to access the Secured Digital Card and Embedded MultiMedia Card based on the SDHC driver.

Function groups

This function group implements the SD card functional API.

This function group implements the MMC card functional API.

Typical use case

```
/* Initialize SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

while (true)
{
    if (kStatus_Success != SD_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
      DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != SD_ReadBlocks(card, g_dataRead, DATA_BLOCK_START, DATA_BLOCK_COUNT)
      )
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }

    if (kStatus_Success != SD_EraseBlocks(card, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Erase multiple data blocks failed.\r\n");
    }
}

SD_Deinit(card);


/* Initialize SDHC. */
```

```
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (MMC_Init(card))
{
    PRINTF("\n MMC card init failed \n");
}

while (true)
{
    if (kStatus_Success != MMC_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
      DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != MMC_ReadBlocks(card, g_dataRead, DATA_BLOCK_START,
      DATA_BLOCK_COUNT))
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }
}

MMC_Deinit(card);
```

Data Structures

- struct sd_card_t
  SD card state. More...
- struct mmc_card_t
  SD card state. More...
- struct mmc_boot_config_t
  MMC card boot configuration definition. More...

Macros

- #define FSL_SDMMC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 1U)) /*2.1.1*/
  Driver version.
- #define FSL_SDMMC_DEFAULT_BLOCK_SIZE (512U)
  Default block size.

Enumerations

- enum _sdmmc_status {
  kStatus_SDMMC_NotSupportYet = MAKE_STATUS(kStatusGroup_SDMMC, 0U),
  kStatus_SDMMC_TransferFailed = MAKE_STATUS(kStatusGroup_SDMMC, 1U),
  kStatus_SDMMC_SetCardBlockSizeFailed = MAKE_STATUS(kStatusGroup_SDMMC, 2U),
  kStatus_SDMMC_HostNotSupport = MAKE_STATUS(kStatusGroup_SDMMC, 3U),
  kStatus_SDMMC_CardNotSupport = MAKE_STATUS(kStatusGroup_SDMMC, 4U),
  kStatus_SDMMC_AllSendCidFailed = MAKE_STATUS(kStatusGroup_SDMMC, 5U),
  kStatus_SDMMC_SendRelativeAddressFailed = MAKE_STATUS(kStatusGroup_SDMMC, 6U),
  kStatus_SDMMC_SendCsdFailed = MAKE_STATUS(kStatusGroup_SDMMC, 7U),
  kStatus_SDMMC_SelectCardFailed = MAKE_STATUS(kStatusGroup_SDMMC, 8U),
  kStatus_SDMMC_SendScrFailed = MAKE_STATUS(kStatusGroup_SDMMC, 9U),
  kStatus_SDMMC_SetDataBusWidthFailed = MAKE_STATUS(kStatusGroup_SDMMC, 10U),
  kStatus_SDMMC_GoIdleFailed = MAKE_STATUS(kStatusGroup_SDMMC, 11U),
  kStatus_SDMMC_HandShakeOperationConditionFailed,
  kStatus_SDMMC_SendApplicationCommandFailed,
  kStatus_SDMMC_SwitchFailed = MAKE_STATUS(kStatusGroup_SDMMC, 14U),
  kStatus_SDMMC_StopTransmissionFailed = MAKE_STATUS(kStatusGroup_SDMMC, 15U),
  kStatus_SDMMC_WaitWriteCompleteFailed = MAKE_STATUS(kStatusGroup_SDMMC, 16U),
  kStatus_SDMMC_SetBlockCountFailed = MAKE_STATUS(kStatusGroup_SDMMC, 17U),
  kStatus_SDMMC_SetRelativeAddressFailed = MAKE_STATUS(kStatusGroup_SDMMC, 18U),
  kStatus_SDMMC_SwitchHighSpeedFailed = MAKE_STATUS(kStatusGroup_SDMMC, 19U),
  kStatus_SDMMC_SendExtendedCsdFailed = MAKE_STATUS(kStatusGroup_SDMMC, 20U),
  kStatus_SDMMC_ConfigureBootFailed = MAKE_STATUS(kStatusGroup_SDMMC, 21U),
  kStatus_SDMMC_ConfigureExtendedCsdFailed = MAKE_STATUS(kStatusGroup_SDMMC, 22-U),
  kStatus_SDMMC_EnableHighCapacityEraseFailed,
  kStatus_SDMMC_SendTestPatternFailed = MAKE_STATUS(kStatusGroup_SDMMC, 24U),
  kStatus_SDMMC_ReceiveTestPatternFailed = MAKE_STATUS(kStatusGroup_SDMMC, 25U) }

  SD/MMC card API's running status.
- enum _sd_card_flag {
  kSD_SupportHighCapacityFlag = (1U << 1U),
  kSD_Support4BitWidthFlag = (1U << 2U),
  kSD_SupportSdhcFlag = (1U << 3U),
  kSD_SupportSdxcFlag = (1U << 4U) }

  SD card flags.
- enum _mmc_card_flag {
  kMMC_SupportHighCapacityFlag = (1U << 0U),
  kMMC_SupportHighSpeedFlag = (1U << 1U),
  kMMC_SupportHighSpeed52MHZFlag = (1U << 2U),
  kMMC_SupportHighSpeed26MHZFlag = (1U << 3U),
  kMMC_SupportAlternateBootFlag = (1U << 4U) }

  MMC card flags.

SDCARD Function

- status_t SD_Init (sd_card_t ∗card)
  Initialize the card on a specific host controller.
- void SD_Deinit (sd_card_t ∗card)
  Deinitialize the card.
- bool SD_CheckReadOnly (sd_card_t ∗card)
  Check whether the card is write-protected.
- status_t SD_ReadBlocks (sd_card_t ∗card, uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)
  Read blocks from the specific card.
- status_t SD_WriteBlocks (sd_card_t ∗card, const uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)
  Write blocks of data to the specific card.
- status_t SD_EraseBlocks (sd_card_t ∗card, uint32_t startBlock, uint32_t blockCount)
  Erase blocks of the specific card.

MMCCARD Function

- status_t MMC_Init (mmc_card_t ∗card)
  Initialize the MMC card.
- void MMC_Deinit (mmc_card_t ∗card)
  Deinitialize the card.
- bool MMC_CheckReadOnly (mmc_card_t ∗card)
  Check if the card is read only.
- status_t MMC_ReadBlocks (mmc_card_t ∗card, uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)
  Read data blocks from the card.
- status_t MMC_WriteBlocks (mmc_card_t ∗card, const uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)
  Write data blocks to the card.
- status_t MMC_EraseGroups (mmc_card_t ∗card, uint32_t startGroup, uint32_t endGroup)
  Erase groups of the card.
- status_t MMC_SelectPartition (mmc_card_t ∗card, mmc_access_partition_t partitionNumber)
  Select the partition to access.
- status_t MMC_SetBootConfig (mmc_card_t ∗card, const mmc_boot_config_t ∗config)
  Configure boot activity of the card.

45.2 Data Structure Documentation

45.2.1 struct sd_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- sdhc_host_t host
  Host information.
- uint32_t busClock_Hz

SD bus clock frequency united in Hz.
- uint32_t relativeAddress
  Relative address of the card.
- uint32_t version
  Card version.
- uint32_t flags
  Flags in _sd_card_flag.
- uint32_t rawCid [4U]
  Raw CID content.
- uint32_t rawCsd [4U]
  Raw CSD content.
- uint32_t rawScr [2U]
  Raw CSD content.
- uint32_t ocr
  Raw OCR content.
- sd_cid_t cid
  CID.
- sd_csd_t csd
  CSD.
- sd_scr_t scr
  SCR.
- uint32_t blockCount
  Card total block number.
- uint32_t blockSize
  Card block size.

45.2.2 struct mmc_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- sdhc_host_t host
  Host information.
- uint32_t busClock_Hz
  MMC bus clock united in Hz.
- uint32_t relativeAddress
  Relative address of the card.
- bool enablePreDefinedBlockCount
  Enable PRE-DEFINED block count when read/write.
- uint32_t flags
  Capability flag in _mmc_card_flag.
- uint32_t rawCid [4U]
  Raw CID content.
- uint32_t rawCsd [4U]
  Raw CSD content.
- uint32_t rawExtendedCsd [MMC_EXTENDED_CSD_BYTES/4U]
  Raw MMC Extended CSD content.

- uint32_t ocr
  Raw OCR content.
- mmc_cid_t cid
  CID.
- mmc_csd_t csd
  CSD.
- mmc_extended_csd_t extendedCsd
  Extended CSD.
- uint32_t blockSize
  Card block size.
- uint32_t userPartitionBlocks
  Card total block number in user partition.
- uint32_t bootPartitionBlocks
  Boot partition size united as block size.
- uint32_t eraseGroupBlocks
  Erase group size united as block size.
- mmc_access_partition_t currentPartition
  Current access partition.
- mmc_voltage_window_t hostVoltageWindow
  Host voltage window.

45.2.3 struct mmc_boot_config_t

Data Fields

- bool enableBootAck
  Enable boot ACK.
- mmc_boot_partition_enable_t bootPartition
  Boot partition.
- bool retainBootBusWidth
  If retain boot bus width.
- mmc_data_bus_width_t bootDataBusWidth
  Boot data bus width.

45.3 Macro Definition Documentation

45.3.1 #define FSL_SDMMC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 1U)) /*2.1.1*/

45.4 Enumeration Type Documentation

45.4.1 enum _sdmmc_status

Enumerator

kStatus_SDMMC_NotSupportYet Haven't supported.
kStatus_SDMMC_TransferFailed Send command failed.
kStatus_SDMMC_SetCardBlockSizeFailed Set block size failed.

kStatus_SDMMC_HostNotSupport Host doesn't support.
kStatus_SDMMC_CardNotSupport Card doesn't support.
kStatus_SDMMC_AllSendCidFailed Send CID failed.
kStatus_SDMMC_SendRelativeAddressFailed Send relative address failed.
kStatus_SDMMC_SendCsdFailed Send CSD failed.
kStatus_SDMMC_SelectCardFailed Select card failed.
kStatus_SDMMC_SendScrFailed Send SCR failed.
kStatus_SDMMC_SetDataBusWidthFailed Set bus width failed.
kStatus_SDMMC_GoIdleFailed Go idle failed.
kStatus_SDMMC_HandShakeOperationConditionFailed Send Operation Condition failed.
kStatus_SDMMC_SendApplicationCommandFailed Send application command failed.
kStatus_SDMMC_SwitchFailed Switch command failed.
kStatus_SDMMC_StopTransmissionFailed Stop transmission failed.
kStatus_SDMMC_WaitWriteCompleteFailed Wait write complete failed.
kStatus_SDMMC_SetBlockCountFailed Set block count failed.
kStatus_SDMMC_SetRelativeAddressFailed Set relative address failed.
kStatus_SDMMC_SwitchHighSpeedFailed Switch high speed failed.
kStatus_SDMMC_SendExtendedCsdFailed Send EXT_CSD failed.
kStatus_SDMMC_ConfigureBootFailed Configure boot failed.
kStatus_SDMMC_ConfigureExtendedCsdFailed Configure EXT_CSD failed.
kStatus_SDMMC_EnableHighCapacityEraseFailed Enable high capacity erase failed.
kStatus_SDMMC_SendTestPatternFailed Send test pattern failed.
kStatus_SDMMC_ReceiveTestPatternFailed Receive test pattern failed.

45.4.2 enum _sd_card_flag

Enumerator

kSD_SupportHighCapacityFlag Support high capacity.
kSD_Support4BitWidthFlag Support 4-bit data width.
kSD_SupportSdhcFlag Card is SDHC.
kSD_SupportSdxcFlag Card is SDXC.

45.4.3 enum _mmc_card_flag

Enumerator

kMMC_SupportHighCapacityFlag Support high capacity.
kMMC_SupportHighSpeedFlag Support high speed.
kMMC_SupportHighSpeed52MHZFlag Support high speed 52MHZ.
kMMC_SupportHighSpeed26MHZFlag Support high speed 26MHZ.
kMMC_SupportAlternateBootFlag Support alternate boot.

45.5 Function Documentation

45.5.1 status_t SD_Init (sd_card_t * *card*)

This function initializes the card on a specific host controller.

Parameters

card	Card descriptor.

Return values

kStatus_SDMMC_GoIdleFailed	Go idle failed.
kStatus_SDMMC_NotSupportYet	Card not support.
kStatus_SDMMC_SendOperationConditionFailed	Send operation condition failed.
kStatus_SDMMC_AllSendCidFailed	Send CID failed.
kStatus_SDMMC_SendRelativeAddressFailed	Send relative address failed.
kStatus_SDMMC_SendCsdFailed	Send CSD failed.
kStatus_SDMMC_SelectCardFailed	Send SELECT_CARD command failed.
kStatus_SDMMC_SendScrFailed	Send SCR failed.
kStatus_SDMMC_SetBusWidthFailed	Set bus width failed.
kStatus_SDMMC_SwitchHighSpeedFailed	Switch high speed failed.
kStatus_SDMMC_SetCardBlockSizeFailed	Set card block size failed.
kStatus_Success	Operate successfully.

45.5.2 void SD_Deinit (sd_card_t * *card*)

This function deinitializes the specific card.

Parameters

card	Card descriptor.

45.5.3 bool SD_CheckReadOnly (sd_card_t * *card*)

This function checks if the card is write-protected via CSD register.

Parameters

card	The specific card.

Return values

true	Card is read only.
false	Card isn't read only.

45.5.4 status_t SD_ReadBlocks (sd_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function reads blocks from specific card, with default block size defined by SDHC_CARD_DEFAULT_BLOCK_SIZE.

Parameters

card	Card descriptor.
buffer	The buffer to save the data read from card.
startBlock	The start block index.
blockCount	The number of blocks to read.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_CardNotSupport	Card not support.

kStatus_SDMMC_NotSupportYet	Not support now.
kStatus_SDMMC_WaitWriteCompleteFailed	Send status failed.
kStatus_SDMMC_TransferFailed	Transfer failed.
kStatus_SDMMC_StopTransmissionFailed	Stop transmission failed.
kStatus_Success	Operate successfully.

45.5.5 status_t SD_WriteBlocks (sd_card_t * *card*, const uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function writes blocks to specific card, with default block size 512 bytes.

Parameters

card	Card descriptor.
buffer	The buffer holding the data to be written to the card.
startBlock	The start block index.
blockCount	The number of blocks to write.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_NotSupportYet	Not support now.
kStatus_SDMMC_CardNotSupport	Card not support.
kStatus_SDMMC_WaitWriteCompleteFailed	Send status failed.
kStatus_SDMMC_TransferFailed	Transfer failed.

kStatus_SDMMC_Stop-TransmissionFailed	Stop transmission failed.
kStatus_Success	Operate successfully.

45.5.6 status_t SD_EraseBlocks (sd_card_t * *card,* uint32_t *startBlock,* uint32_t *blockCount*)

This function erases blocks of a specific card, with default block size 512 bytes.

Parameters

card	Card descriptor.
startBlock	The start block index.
blockCount	The number of blocks to erase.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_Wait-WriteCompleteFailed	Send status failed.
kStatus_SDMMC_-TransferFailed	Transfer failed.
kStatus_SDMMC_Wait-WriteCompleteFailed	Send status failed.
kStatus_Success	Operate successfully.

45.5.7 status_t MMC_Init (mmc_card_t * *card*)

Parameters

card	Card descriptor.

Return values

kStatus_SDMMC_GoIdleFailed	Go idle failed.
kStatus_SDMMC_SendOperationConditionFailed	Send operation condition failed.
kStatus_SDMMC_AllSendCidFailed	Send CID failed.
kStatus_SDMMC_SetRelativeAddressFailed	Set relative address failed.
kStatus_SDMMC_SendCsdFailed	Send CSD failed.
kStatus_SDMMC_CardNotSupport	Card not support.
kStatus_SDMMC_SelectCardFailed	Send SELECT_CARD command failed.
kStatus_SDMMC_SendExtendedCsdFailed	Send EXT_CSD failed.
kStatus_SDMMC_SetBusWidthFailed	Set bus width failed.
kStatus_SDMMC_SwitchHighSpeedFailed	Switch high speed failed.
kStatus_SDMMC_SetCardBlockSizeFailed	Set card block size failed.
kStatus_Success	Operate successfully.

45.5.8 void MMC_Deinit (mmc_card_t * *card*)

Parameters

card	Card descriptor.

45.5.9 bool MMC_CheckReadOnly (mmc_card_t * *card*)

Parameters

card	Card descriptor.

Return values

true	Card is read only.
false	Card isn't read only.

45.5.10 status_t MMC_ReadBlocks (mmc_card_t ∗ *card,* uint8_t ∗ *buffer,* uint32_t *startBlock,* uint32_t *blockCount*)

Parameters

card	Card descriptor.
buffer	The buffer to save data.
startBlock	The start block index.
blockCount	The number of blocks to read.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_CardNotSupport	Card not support.
kStatus_SDMMC_SetBlockCountFailed	Set block count failed.
kStatus_SDMMC_TransferFailed	Transfer failed.
kStatus_SDMMC_StopTransmissionFailed	Stop transmission failed.
kStatus_Success	Operate successfully.

45.5.11 status_t MMC_WriteBlocks (mmc_card_t ∗ *card,* const uint8_t ∗ *buffer,* uint32_t *startBlock,* uint32_t *blockCount*)

Parameters

card	Card descriptor.
buffer	The buffer to save data blocks.
startBlock	Start block number to write.
blockCount	Block count.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_NotSupportYet	Not support now.
kStatus_SDMMC_SetBlockCountFailed	Set block count failed.
kStatus_SDMMC_WaitWriteCompleteFailed	Send status failed.
kStatus_SDMMC_TransferFailed	Transfer failed.
kStatus_SDMMC_StopTransmissionFailed	Stop transmission failed.
kStatus_Success	Operate successfully.

45.5.12 status_t MMC_EraseGroups (mmc_card_t * *card*, uint32_t *startGroup*, uint32_t *endGroup*)

Erase group is the smallest erase unit in MMC card. The erase range is [startGroup, endGroup].

Parameters

card	Card descriptor.
startGroup	Start group number.
endGroup	End group number.

Return values

kStatus_InvalidArgument	Invalid argument.
kStatus_SDMMC_Wait-WriteCompleteFailed	Send status failed.
kStatus_SDMMC_-TransferFailed	Transfer failed.
kStatus_Success	Operate successfully.

45.5.13 status_t MMC_SelectPartition (mmc_card_t ∗ *card*, mmc_access_partition_t *partitionNumber*)

Parameters

card	Card descriptor.
partition-Number	The partition number.

Return values

kStatus_SDMMC_-ConfigureExtendedCsd-Failed	Configure EXT_CSD failed.
kStatus_Success	Operate successfully.

45.5.14 status_t MMC_SetBootConfig (mmc_card_t ∗ *card*, const mmc_boot_config_t ∗ *config*)

Parameters

card	Card descriptor.
config	Boot configuration structure.

Return values

kStatus_SDMMC_NotSupportYet	Not support now.
kStatus_SDMMC_ConfigureExtendedCsdFailed	Configure EXT_CSD failed.
kStatus_SDMMC_ConfigureBootFailed	Configure boot failed.
kStatus_Success	Operate successfully.

Chapter 46
SPI based Secured Digital Card (SDSPI)

46.1 Overview

The KSDK provides a driver to access the Secured Digital Card based on the SPI driver.

Function groups

This function group implements the SD card functional API in the SPI mode.

Typical use case

```
/* SPI_Init(). */

/* Register the SDSPI driver callback. */

/* Initializes card. */
if (kStatus_Success != SDSPI_Init(card))
{
    SDSPI_Deinit(card)
    return;
}

/* Read/Write card */
memset(g_testWriteBuffer, 0x17U, sizeof(g_testWriteBuffer));

while (true)
{
    memset(g_testReadBuffer, 0U, sizeof(g_testReadBuffer));

    SDSPI_WriteBlocks(card, g_testWriteBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    SDSPI_ReadBlocks(card, g_testReadBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    if (memcmp(g_testReadBuffer, g_testReadBuffer, sizeof(g_testWriteBuffer)))
    {
        break;
    }
}
```

Data Structures

- struct sdspi_command_t
  SDSPI command. More...
- struct sdspi_host_t
  SDSPI host state. More...
- struct sdspi_card_t
  SD Card Structure. More...

Enumerations

- enum _sdspi_status {
kStatus_SDSPI_SetFrequencyFailed = MAKE_STATUS(kStatusGroup_SDSPI, 0U),
kStatus_SDSPI_ExchangeFailed = MAKE_STATUS(kStatusGroup_SDSPI, 1U),
kStatus_SDSPI_WaitReadyFailed = MAKE_STATUS(kStatusGroup_SDSPI, 2U),
kStatus_SDSPI_ResponseError = MAKE_STATUS(kStatusGroup_SDSPI, 3U),
kStatus_SDSPI_WriteProtected = MAKE_STATUS(kStatusGroup_SDSPI, 4U),
kStatus_SDSPI_GoIdleFailed = MAKE_STATUS(kStatusGroup_SDSPI, 5U),
kStatus_SDSPI_SendCommandFailed = MAKE_STATUS(kStatusGroup_SDSPI, 6U),
kStatus_SDSPI_ReadFailed = MAKE_STATUS(kStatusGroup_SDSPI, 7U),
kStatus_SDSPI_WriteFailed = MAKE_STATUS(kStatusGroup_SDSPI, 8U),
kStatus_SDSPI_SendInterfaceConditionFailed,
kStatus_SDSPI_SendOperationConditionFailed,
kStatus_SDSPI_ReadOcrFailed = MAKE_STATUS(kStatusGroup_SDSPI, 11U),
kStatus_SDSPI_SetBlockSizeFailed = MAKE_STATUS(kStatusGroup_SDSPI, 12U),
kStatus_SDSPI_SendCsdFailed = MAKE_STATUS(kStatusGroup_SDSPI, 13U),
kStatus_SDSPI_SendCidFailed = MAKE_STATUS(kStatusGroup_SDSPI, 14U),
kStatus_SDSPI_StopTransmissionFailed = MAKE_STATUS(kStatusGroup_SDSPI, 15U),
kStatus_SDSPI_SendApplicationCommandFailed }
SDSPI API status.
- enum _sdspi_card_flag {
kSDSPI_SupportHighCapacityFlag = (1U << 0U),
kSDSPI_SupportSdhcFlag = (1U << 1U),
kSDSPI_SupportSdxcFlag = (1U << 2U),
kSDSPI_SupportSdscFlag = (1U << 3U) }
SDSPI card flag.
- enum sdspi_response_type_t {
kSDSPI_ResponseTypeR1 = 0U,
kSDSPI_ResponseTypeR1b = 1U,
kSDSPI_ResponseTypeR2 = 2U,
kSDSPI_ResponseTypeR3 = 3U,
kSDSPI_ResponseTypeR7 = 4U }
SDSPI response type.

SDSPI Function

- status_t SDSPI_Init (sdspi_card_t ∗card)
Initialize the card on a specific SPI instance.
- void SDSPI_Deinit (sdspi_card_t ∗card)
Deinitialize the card.
- bool SDSPI_CheckReadOnly (sdspi_card_t ∗card)
Check whether the card is write-protected.
- status_t SDSPI_ReadBlocks (sdspi_card_t ∗card, uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)
Read blocks from the specific card.
- status_t SDSPI_WriteBlocks (sdspi_card_t ∗card, uint8_t ∗buffer, uint32_t startBlock, uint32_t blockCount)

Write blocks of data to the specific card.

46.2 Data Structure Documentation

46.2.1 struct sdspi_command_t

Data Fields

- uint8_t index
  Command index.
- uint32_t argument
  Command argument.
- uint8_t responseType
  Response type.
- uint8_t response [5U]
  Response content.

46.2.2 struct sdspi_host_t

Data Fields

- uint32_t busBaudRate
  Bus baud rate.
- status_t(∗ setFrequency)(uint32_t frequency)
  Set frequency of SPI.
- status_t(∗ exchange)(uint8_t ∗in, uint8_t ∗out, uint32_t size)
  Exchange data over SPI.
- uint32_t(∗ getCurrentMilliseconds)(void)
  Get current time in milliseconds.

46.2.3 struct sdspi_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- sdspi_host_t ∗ host
  Host state information.
- uint32_t relativeAddress
  Relative address of the card.
- uint32_t flags
  Flags defined in _sdspi_card_flag.
- uint8_t rawCid [16U]
  Raw CID content.
- uint8_t rawCsd [16U]

Raw CSD content.
- uint8_t rawScr [8U]
  Raw SCR content.
- uint32_t ocr
  Raw OCR content.
- sd_cid_t cid
  CID.
- sd_csd_t csd
  CSD.
- sd_scr_t scr
  SCR.
- uint32_t blockCount
  Card total block number.
- uint32_t blockSize
  Card block size.

46.2.3.0.0.90 Field Documentation

46.2.3.0.0.90.1 uint32_t sdspi_card_t::flags

46.3 Enumeration Type Documentation

46.3.1 enum _sdspi_status

Enumerator

kStatus_SDSPI_SetFrequencyFailed Set frequency failed.
kStatus_SDSPI_ExchangeFailed Exchange data on SPI bus failed.
kStatus_SDSPI_WaitReadyFailed Wait card ready failed.
kStatus_SDSPI_ResponseError Response is error.
kStatus_SDSPI_WriteProtected Write protected.
kStatus_SDSPI_GoIdleFailed Go idle failed.
kStatus_SDSPI_SendCommandFailed Send command failed.
kStatus_SDSPI_ReadFailed Read data failed.
kStatus_SDSPI_WriteFailed Write data failed.
kStatus_SDSPI_SendInterfaceConditionFailed Send interface condition failed.
kStatus_SDSPI_SendOperationConditionFailed Send operation condition failed.
kStatus_SDSPI_ReadOcrFailed Read OCR failed.
kStatus_SDSPI_SetBlockSizeFailed Set block size failed.
kStatus_SDSPI_SendCsdFailed Send CSD failed.
kStatus_SDSPI_SendCidFailed Send CID failed.
kStatus_SDSPI_StopTransmissionFailed Stop transmission failed.
kStatus_SDSPI_SendApplicationCommandFailed Send application command failed.

46.3.2 enum _sdspi_card_flag

Enumerator

kSDSPI_SupportHighCapacityFlag Card is high capacity.
kSDSPI_SupportSdhcFlag Card is SDHC.
kSDSPI_SupportSdxcFlag Card is SDXC.
kSDSPI_SupportSdscFlag Card is SDSC.

46.3.3 enum sdspi_response_type_t

Enumerator

kSDSPI_ResponseTypeR1 Response 1.
kSDSPI_ResponseTypeR1b Response 1 with busy.
kSDSPI_ResponseTypeR2 Response 2.
kSDSPI_ResponseTypeR3 Response 3.
kSDSPI_ResponseTypeR7 Response 7.

46.4 Function Documentation

46.4.1 status_t SDSPI_Init (sdspi_card_t * *card*)

This function initializes the card on a specific SPI instance.

Parameters

card	Card descriptor

Return values

kStatus_SDSPI_SetFrequencyFailed	Set frequency failed.
kStatus_SDSPI_GoIdleFailed	Go idle failed.
kStatus_SDSPI_SendInterfaceConditionFailed	Send interface condition failed.

kStatus_SDSPI_SendOperationConditionFailed	Send operation condition failed.
kStatus_Timeout	Send command timeout.
kStatus_SDSPI_NotSupportYet	Not support yet.
kStatus_SDSPI_ReadOcrFailed	Read OCR failed.
kStatus_SDSPI_SetBlockSizeFailed	Set block size failed.
kStatus_SDSPI_SendCsdFailed	Send CSD failed.
kStatus_SDSPI_SendCidFailed	Send CID failed.
kStatus_Success	Operate successfully.

46.4.2 void SDSPI_Deinit (sdspi_card_t * *card*)

This function deinitializes the specific card.

Parameters

card	Card descriptor

46.4.3 bool SDSPI_CheckReadOnly (sdspi_card_t * *card*)

This function checks if the card is write-protected via CSD register.

Parameters

card	Card descriptor.

Return values

true	Card is read only.
false	Card isn't read only.

46.4.4 status_t SDSPI_ReadBlocks (sdspi_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function reads blocks from specific card.

Parameters

card	Card descriptor.
buffer	the buffer to hold the data read from card
startBlock	the start block index
blockCount	the number of blocks to read

Return values

kStatus_SDSPI_SendCommandFailed	Send command failed.
kStatus_SDSPI_ReadFailed	Read data failed.
kStatus_SDSPI_StopTransmissionFailed	Stop transmission failed.
kStatus_Success	Operate successfully.

46.4.5 status_t SDSPI_WriteBlocks (sdspi_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function writes blocks to specific card

Parameters

card	Card descriptor.
buffer	the buffer holding the data to be written to the card

startBlock	the start block index
blockCount	the number of blocks to write

Return values

kStatus_SDSPI_Write-Protected	Card is write protected.
kStatus_SDSPI_Send-CommandFailed	Send command failed.
kStatus_SDSPI_-ResponseError	Response is error.
kStatus_SDSPI_Write-Failed	Write data failed.
kStatus_SDSPI_-ExchangeFailed	Exchange data over SPI failed.
kStatus_SDSPI_Wait-ReadyFailed	Wait card to be ready status failed.
kStatus_Success	Operate successfully.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: KSDK20KV58APIRM
Rev. 0
Jun 2016