Kinetis SDK v.2.0 API Reference Manual

NXP Semiconductors

Document Number: KSDK20K80FAPIRM
Rev. 0
Aug 2016

Contents

Contents

Contents

Contents

Section Number | Title | Page Number

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Section Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Section Number	Title	Page Number

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_FLEXIO_I2S_Idle = 2300
- kStatus_FLEXIO_I2S_TxBusy = 2301
- kStatus_FLEXIO_I2S_RxBusy = 2302
- kStatus_FLEXIO_I2S_Error = 2303
- kStatus_FLEXIO_I2S_QueueFull = 2304
- kStatus_QSPI_Idle = 4500
- kStatus_QSPI_Busy = 4501
- kStatus_QSPI_Error = 4502
- kStatus_SAI_TxBusy = 1900
- kStatus_SAI_RxBusy = 1901
- kStatus_SAI_TxError = 1902
- kStatus_SAI_RxError = 1903
- kStatus_SAI_QueueFull = 1904
- kStatus_SAI_TxIdle = 1905
- kStatus_SAI_RxIdle = 1906
- kStatus_SMARTCARD_Success = 4300
- kStatus_SMARTCARD_TxBusy = 4301
- kStatus_SMARTCARD_RxBusy = 4302
- kStatus_SMARTCARD_NoTransferInProgress = 4303
- kStatus_SMARTCARD_Timeout = 4304
- kStatus_SMARTCARD_Initialized = 4305
- kStatus_SMARTCARD_PhyInitialized = 4306
- kStatus_SMARTCARD_CardNotActivated = 4307
- kStatus_SMARTCARD_InvalidInput = 4308
- kStatus_SMARTCARD_OtherError = 4309
- kStatus_SMC_StopAbort = 3900
- kStatus_NOTIFIER_ErrorNotificationBefore = 9800
- kStatus_NOTIFIER_ErrorNotificationAfter = 9801

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

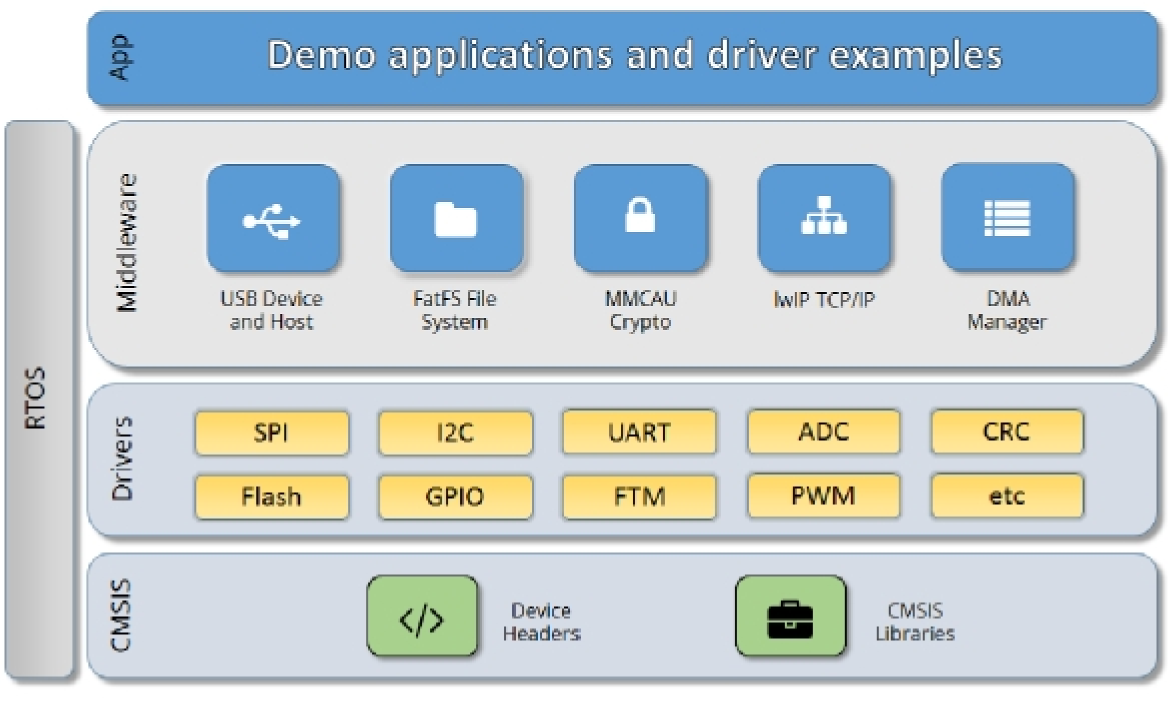

Figure 1: KSDK Block Diagram

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 KS-DK 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 KSD-K-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++;
    }
```

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,
kADC16_CalibrationFailedFlag = ADC_SC3_CALF_MASK }
Converter status flags.
- enum adc16_channel_mux_mode_t {
kADC16_ChannelMuxA = 0U,
kADC16_ChannelMuxB = 1U }
Channel multiplexer mode for each channel.
- 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,
kADC16_ResolutionDF9Bit = kADC16_Resolution8or9Bit,
kADC16_ResolutionDF13Bit = kADC16_Resolution12or13Bit,
kADC16_ResolutionDF11Bit = kADC16_Resolution10or11Bit,
kADC16_Resolution16Bit = 3U,
kADC16_ResolutionSE16Bit = kADC16_Resolution16Bit,
kADC16_ResolutionDF16Bit = kADC16_Resolution16Bit }
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_average_mode_t {
kADC16_HardwareAverageCount4 = 0U,
kADC16_HardwareAverageCount8 = 1U,
kADC16_HardwareAverageCount16 = 2U,
kADC16_HardwareAverageCount32 = 3U,
kADC16_HardwareAverageDisabled = 4U }
  Hardware average mode.
- 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.
- status_t ADC16_DoAutoCalibration (ADC_Type ∗base)
  Automates the hardware calibration.
- static void ADC16_SetOffsetValue (ADC_Type ∗base, int16_t value)
  Sets the offset value for the conversion result.

Advanced Features

- static void ADC16_EnableDMA (ADC_Type ∗base, bool enable)
  Enables generating the DMA trigger when the conversion is complete.
- static void ADC16_EnableHardwareTrigger (ADC_Type ∗base, bool enable)
  Enables the hardware trigger mode.
- void ADC16_SetChannelMuxMode (ADC_Type ∗base, adc16_channel_mux_mode_t mode)
  Sets the channel mux mode.
- void ADC16_SetHardwareCompareConfig (ADC_Type ∗base, const adc16_hardware_compare_config_t ∗config)
  Configures the hardware compare mode.
- void ADC16_SetHardwareAverage (ADC_Type ∗base, adc16_hardware_average_mode_t mode)
  Sets the hardware average 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::hardware-CompareMode

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 an interrupt request once the conversion is completed.
- bool enableDifferentialConversion
  Using Differential sample mode.

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

5.3.3.0.0.3.3 bool adc16_channel_config_t::enableDifferentialConversion

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.
kADC16_CalibrationFailedFlag Calibration is failed.

5.5.3 enum adc16_channel_mux_mode_t

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

Enumerator

kADC16_ChannelMuxA For channel with channel mux a.
kADC16_ChannelMuxB For channel with channel mux b.

5.5.4 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.

5.5.5 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.
kADC16_ResolutionDF9Bit Differential Sample 9-bit.
kADC16_ResolutionDF13Bit Differential Sample 13-bit.
kADC16_ResolutionDF11Bit Differential Sample 11-bit.
kADC16_Resolution16Bit Single End 16-bit or Differential Sample 16-bit.
kADC16_ResolutionSE16Bit Single End 16-bit.
kADC16_ResolutionDF16Bit Differential Sample 16-bit.

5.5.6 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.

5.5.7 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.

5.5.8 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.

5.5.9 enum adc16_hardware_average_mode_t

Enumerator

kADC16_HardwareAverageCount4 For hardware average with 4 samples.
kADC16_HardwareAverageCount8 For hardware average with 8 samples.
kADC16_HardwareAverageCount16 For hardware average with 16 samples.
kADC16_HardwareAverageCount32 For hardware average with 32 samples.
kADC16_HardwareAverageDisabled Disable the hardware average feature.

5.5.10 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 as follows.

```
*   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 status_t ADC16_DoAutoCalibration (ADC_Type * *base*)

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

Parameters

base	ADC16 peripheral base address.

Returns

Execution status.

Return values

kStatus_Success	Calibration is done successfully.
kStatus_Fail	Calibration has failed.

5.6.5 static void ADC16_SetOffsetValue (ADC_Type ∗ *base,* int16_t *value*) [inline], [static]

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

Parameters

base	ADC16 peripheral base address.
value	Setting offset value.

5.6.6 static void ADC16_EnableDMA (ADC_Type ∗ *base,* bool *enable*) [inline], [static]

Parameters

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

5.6.7 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.

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

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

Parameters

base	ADC16 peripheral base address.
mode	Setting channel mux mode. See "adc16_channel_mux_mode_t".

5.6.9 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.

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

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

Parameters

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

5.6.11 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".

5.6.12 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".

5.6.13 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 group 1 and greater indicates multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual for the number of SC1n registers (channel groups) specific to this device. Channel group 1 or greater are not used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

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

5.6.14 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.

5.6.15 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
CMP: Analog Comparator Driver

6.1 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.

6.2 Typical use case

6.2.1 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.
        }
    }
}
```

6.2.2 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)
    {
    }
}
```

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_EnableDMA (CMP_Type ∗base, bool enable)
    Enables/disables the DMA request for rising/falling events.
- static void CMP_EnableWindowMode (CMP_Type ∗base, bool enable)
    Enables/disables the window mode.
- 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.

6.3 Data Structure Documentation

6.3.1 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.
- bool enableTriggerMode
  Enable the trigger mode.

6.3.1.0.0.4 Field Documentation

6.3.1.0.0.4.1 bool cmp_config_t::enableCmp

6.3.1.0.0.4.2 cmp_hysteresis_mode_t cmp_config_t::hysteresisMode

6.3.1.0.0.4.3 bool cmp_config_t::enableHighSpeed

6.3.1.0.0.4.4 bool cmp_config_t::enableInvertOutput

6.3.1.0.0.4.5 bool cmp_config_t::useUnfilteredOutput

6.3.1.0.0.4.6 bool cmp_config_t::enablePinOut

6.3.1.0.0.4.7 bool cmp_config_t::enableTriggerMode

6.3.2 struct cmp_filter_config_t

Data Fields

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

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

6.3.2.0.0.5 Field Documentation

6.3.2.0.0.5.1 bool cmp_filter_config_t::enableSample

6.3.2.0.0.5.2 uint8_t cmp_filter_config_t::filterCount

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

6.3.2.0.0.5.3 uint8_t cmp_filter_config_t::filterPeriod

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

6.3.3 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.

6.3.3.0.0.6 Field Documentation

6.3.3.0.0.6.1 cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource

6.3.3.0.0.6.2 uint8_t cmp_dac_config_t::DACValue

Available range is 0-63.

6.4 Macro Definition Documentation

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

6.5 Enumeration Type Documentation

6.5.1 enum _cmp_interrupt_enable

Enumerator

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

6.5.2 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.

6.5.3 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.

6.5.4 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.

6.6 Function Documentation

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

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

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

Parameters

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

6.6.2 void CMP_Deinit (CMP_Type * *base*)

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

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

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

Parameters

base	CMP peripheral base address.

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

Parameters

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

6.6.4 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.

6.6.5 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 the same way in the application. When the user selects the same input from the analog mux to the positive and negative port, the comparator is disabled automatically.

Parameters

base	CMP peripheral base address.
positive-Channel	Positive side input channel number. Available range is 0-7.
negative-Channel	Negative side input channel number. Available range is 0-7.

6.6.6 void CMP_EnableDMA (CMP_Type ∗ *base,* bool *enable*)

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

Parameters

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

6.6.7 static void CMP_EnableWindowMode (CMP_Type ∗ *base,* bool *enable*) [inline], [static]

Parameters

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

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

Parameters

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

6.6.9 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.

6.6.10 void CMP_EnableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Parameters

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

6.6.11 void CMP_DisableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Parameters

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

6.6.12 uint32_t CMP_GetStatusFlags (CMP_Type * *base*)

Parameters

base	CMP peripheral base address.

Returns

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

6.6.13 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 7
CMT: Carrier Modulator Transmitter Driver

7.1 Overview

The carrier modulator transmitter (CMT) module provides the means to generate the protocol timing and carrier signals for a side variety of encoding schemes. The CMT incorporates hardware to off-load the critical and/or lengthy timing requirements associated with signal generation from the CPU. The KSDK provides a driver for the CMT module of the Kinetis devices.

7.2 Clock formulas

The CMT module has internal clock dividers. It was originally designed to be based on an 8 MHz bus clock that can be divided by 1, 2, 4, or 8 according to the specification. To be compatible with a higher bus frequency, the primary prescaler (PPS) was developed to receive a higher frequency and generate a clock enable signal called an intermediate frequency (IF). The IF must be approximately equal to 8 MHz and works as a clock enable to the secondary prescaler. For the PPS, the prescaler is selected according to the bus clock to generate an intermediate clock approximate to 8 MHz and is selected as (bus_clock-_hz/8000000). The secondary prescaler is the "cmtDivider". The clocks for the CMT module are listed below.

1. CMT clock frequency = bus_clock_Hz / (bus_clock_Hz / 8000000) / cmtDivider
2. CMT carrier and generator frequency = CMT clock frequency / (highCount1 + lowCount1)
   (In FSK mode, the second frequency = CMT clock frequency / (highCount2 + lowCount2))
3. CMT infrared output signal frequency
   a. In Time and Baseband mode
   CMT IRO signal mark time = (markCount + 1) / (CMT clock frequency / 8)
   CMT IRO signal space time = spaceCount / (CMT clock frequency / 8)
   b. In FSK mode
   CMT IRO signal mark time = (markCount + 1) / CMT carrier and generator frequency
   CMT IRO signal space time = spaceCount / CMT carrier and generator frequency

7.3 Typical use case

This is an example code to initialize data.

```
cmt_config_t config;
cmt_modulate_config_t modulateConfig;
uint32_t busClock;

// Gets the bus clock for the CMT module.
busClock = CLOCK_GetFreq(kCLOCK_BusClk);

CMT_GetDefaultConfig(&config);

// Interrupts is enabled to change the modulate mark and space count.
config.isInterruptEnabled = true;
```

```
CMT_Init(CMT, &config, busClock);

// Prepares the modulate configuration for a use case.
modulateConfig.highCount1 = ...;
modulateConfig.lowCount1 = ...;
modulateConfig.markCount = ...;
modulateConfig.spaceCount = ...;

// Sets the time mode.
CMT_SetMode(CMT, kCMT_TimeMode, &modulateConfig);
```

This is an example IRQ handler to change the mark and space count to complete data modulation.

```
// The data length has been transmitted.
uint32_t g_CmtDataBitLen;

void CMT_IRQHandler(void)
{
    if (CMT_GetStatusFlags(CMT))
    {
        if (g_CmtDataBitLen <= CMT_TEST_DATA_BITS)
        {
            // LSB.
            if (data & ((uint32_t)0x01 << g_CmtDataBitLen))
            {
                CMT_SetModulateMarkSpace(CMT, g_CmtModDataOneMarkCount,
      g_CmtModDataOneSpaceCount);
            }
            else
            {
                CMT_SetModulateMarkSpace(CMT, g_CmtModDataZeroMarkCount,
      g_CmtModDataZeroSpaceCount);
            }
        }
    }
}
```

Data Structures

- struct cmt_modulate_config_t
  CMT carrier generator and modulator configuration structure. More...
- struct cmt_config_t
  CMT basic configuration structure. More...

Enumerations

- enum cmt_mode_t {
  kCMT_DirectIROCtl = 0x00U,
  kCMT_TimeMode = 0x01U,
  kCMT_FSKMode = 0x05U,
  kCMT_BasebandMode = 0x09U }
  The modes of CMT.
- enum cmt_primary_clkdiv_t {

kCMT_PrimaryClkDiv1 = 0U,
kCMT_PrimaryClkDiv2 = 1U,
kCMT_PrimaryClkDiv3 = 2U,
kCMT_PrimaryClkDiv4 = 3U,
kCMT_PrimaryClkDiv5 = 4U,
kCMT_PrimaryClkDiv6 = 5U,
kCMT_PrimaryClkDiv7 = 6U,
kCMT_PrimaryClkDiv8 = 7U,
kCMT_PrimaryClkDiv9 = 8U,
kCMT_PrimaryClkDiv10 = 9U,
kCMT_PrimaryClkDiv11 = 10U,
kCMT_PrimaryClkDiv12 = 11U,
kCMT_PrimaryClkDiv13 = 12U,
kCMT_PrimaryClkDiv14 = 13U,
kCMT_PrimaryClkDiv15 = 14U,
kCMT_PrimaryClkDiv16 = 15U }
The CMT clock divide primary prescaler.
- enum cmt_second_clkdiv_t {
kCMT_SecondClkDiv1 = 0U,
kCMT_SecondClkDiv2 = 1U,
kCMT_SecondClkDiv4 = 2U,
kCMT_SecondClkDiv8 = 3U }
The CMT clock divide secondary prescaler.
- enum cmt_infrared_output_polarity_t {
kCMT_IROActiveLow = 0U,
kCMT_IROActiveHigh = 1U }
The CMT infrared output polarity.
- enum cmt_infrared_output_state_t {
kCMT_IROCtlLow = 0U,
kCMT_IROCtlHigh = 1U }
The CMT infrared output signal state control.
- enum _cmt_interrupt_enable { kCMT_EndOfCycleInterruptEnable = CMT_MSC_EOCIE_MASK }
CMT interrupt configuration structure, default settings all disabled.

Driver version

- #define FSL_CMT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
CMT driver version 2.0.1.

Initialization and deinitialization

- void CMT_GetDefaultConfig (cmt_config_t ∗config)
Gets the CMT default configuration structure.
- void CMT_Init (CMT_Type ∗base, const cmt_config_t ∗config, uint32_t busClock_Hz)
Initializes the CMT module.
- void CMT_Deinit (CMT_Type ∗base)

Disables the CMT module and gate control.

Basic Control Operations

- void CMT_SetMode (CMT_Type ∗base, cmt_mode_t mode, cmt_modulate_config_t ∗modulateConfig)
  Selects the mode for CMT.
- cmt_mode_t CMT_GetMode (CMT_Type ∗base)
  Gets the mode of the CMT module.
- uint32_t CMT_GetCMTFrequency (CMT_Type ∗base, uint32_t busClock_Hz)
  Gets the actual CMT clock frequency.
- static void CMT_SetCarrirGenerateCountOne (CMT_Type ∗base, uint32_t highCount, uint32_t lowCount)
  Sets the primary data set for the CMT carrier generator counter.
- static void CMT_SetCarrirGenerateCountTwo (CMT_Type ∗base, uint32_t highCount, uint32_t lowCount)
  Sets the secondary data set for the CMT carrier generator counter.
- void CMT_SetModulateMarkSpace (CMT_Type ∗base, uint32_t markCount, uint32_t spaceCount)
  Sets the modulation mark and space time period for the CMT modulator.
- static void CMT_EnableExtendedSpace (CMT_Type ∗base, bool enable)
  Enables or disables the extended space operation.
- void CMT_SetIroState (CMT_Type ∗base, cmt_infrared_output_state_t state)
  Sets the IRO (infrared output) signal state.
- static void CMT_EnableInterrupts (CMT_Type ∗base, uint32_t mask)
  Enables the CMT interrupt.
- static void CMT_DisableInterrupts (CMT_Type ∗base, uint32_t mask)
  Disables the CMT interrupt.
- static uint32_t CMT_GetStatusFlags (CMT_Type ∗base)
  Gets the end of the cycle status flag.

7.4 Data Structure Documentation

7.4.1 struct cmt_modulate_config_t

Data Fields

- uint8_t highCount1
  The high-time for carrier generator first register.
- uint8_t lowCount1
  The low-time for carrier generator first register.
- uint8_t highCount2
  The high-time for carrier generator second register for FSK mode.
- uint8_t lowCount2
  The low-time for carrier generator second register for FSK mode.
- uint16_t markCount
  The mark time for the modulator gate.
- uint16_t spaceCount
  The space time for the modulator gate.

7.4.1.0.0.7 Field Documentation

7.4.1.0.0.7.1 uint8_t cmt_modulate_config_t::highCount1

7.4.1.0.0.7.2 uint8_t cmt_modulate_config_t::lowCount1

7.4.1.0.0.7.3 uint8_t cmt_modulate_config_t::highCount2

7.4.1.0.0.7.4 uint8_t cmt_modulate_config_t::lowCount2

7.4.1.0.0.7.5 uint16_t cmt_modulate_config_t::markCount

7.4.1.0.0.7.6 uint16_t cmt_modulate_config_t::spaceCount

7.4.2 struct cmt_config_t

Data Fields

- bool isInterruptEnabled
  Timer interrupt 0-disable, 1-enable.
- bool isIroEnabled
  The IRO output 0-disabled, 1-enabled.
- cmt_infrared_output_polarity_t iroPolarity
  The IRO polarity.
- cmt_second_clkdiv_t divider
  The CMT clock divide prescaler.

7.4.2.0.0.8 Field Documentation

7.4.2.0.0.8.1 bool cmt_config_t::isInterruptEnabled

7.4.2.0.0.8.2 bool cmt_config_t::isIroEnabled

7.4.2.0.0.8.3 cmt_infrared_output_polarity_t cmt_config_t::iroPolarity

7.4.2.0.0.8.4 cmt_second_clkdiv_t cmt_config_t::divider

7.5 Macro Definition Documentation

7.5.1 #define FSL_CMT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

7.6 Enumeration Type Documentation

7.6.1 enum cmt_mode_t

Enumerator

kCMT_DirectIROCtl Carrier modulator is disabled and the IRO signal is directly in software control.

kCMT_TimeMode Carrier modulator is enabled in time mode.

kCMT_FSKMode Carrier modulator is enabled in FSK mode.
kCMT_BasebandMode Carrier modulator is enabled in baseband mode.

7.6.2 enum cmt_primary_clkdiv_t

The primary clock divider is used to divider the bus clock to get the intermediate frequency to approximately equal to 8 MHZ. When the bus clock is 8 MHZ, set primary prescaler to "kCMT_PrimaryClkDiv1".

Enumerator

kCMT_PrimaryClkDiv1 The intermediate frequency is the bus clock divided by 1.
kCMT_PrimaryClkDiv2 The intermediate frequency is the bus clock divided by 2.
kCMT_PrimaryClkDiv3 The intermediate frequency is the bus clock divided by 3.
kCMT_PrimaryClkDiv4 The intermediate frequency is the bus clock divided by 4.
kCMT_PrimaryClkDiv5 The intermediate frequency is the bus clock divided by 5.
kCMT_PrimaryClkDiv6 The intermediate frequency is the bus clock divided by 6.
kCMT_PrimaryClkDiv7 The intermediate frequency is the bus clock divided by 7.
kCMT_PrimaryClkDiv8 The intermediate frequency is the bus clock divided by 8.
kCMT_PrimaryClkDiv9 The intermediate frequency is the bus clock divided by 9.
kCMT_PrimaryClkDiv10 The intermediate frequency is the bus clock divided by 10.
kCMT_PrimaryClkDiv11 The intermediate frequency is the bus clock divided by 11.
kCMT_PrimaryClkDiv12 The intermediate frequency is the bus clock divided by 12.
kCMT_PrimaryClkDiv13 The intermediate frequency is the bus clock divided by 13.
kCMT_PrimaryClkDiv14 The intermediate frequency is the bus clock divided by 14.
kCMT_PrimaryClkDiv15 The intermediate frequency is the bus clock divided by 15.
kCMT_PrimaryClkDiv16 The intermediate frequency is the bus clock divided by 16.

7.6.3 enum cmt_second_clkdiv_t

The second prescaler can be used to divide the 8 MHZ CMT clock by 1, 2, 4, or 8 according to the specification.

Enumerator

kCMT_SecondClkDiv1 The CMT clock is the intermediate frequency frequency divided by 1.
kCMT_SecondClkDiv2 The CMT clock is the intermediate frequency frequency divided by 2.
kCMT_SecondClkDiv4 The CMT clock is the intermediate frequency frequency divided by 4.
kCMT_SecondClkDiv8 The CMT clock is the intermediate frequency frequency divided by 8.

7.6.4 enum cmt_infrared_output_polarity_t

Enumerator

kCMT_IROActiveLow The CMT infrared output signal polarity is active-low.
kCMT_IROActiveHigh The CMT infrared output signal polarity is active-high.

7.6.5 enum cmt_infrared_output_state_t

Enumerator

kCMT_IROCtlLow The CMT Infrared output signal state is controlled to low.
kCMT_IROCtlHigh The CMT Infrared output signal state is controlled to high.

7.6.6 enum _cmt_interrupt_enable

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

Enumerator

kCMT_EndOfCycleInterruptEnable CMT end of cycle interrupt.

7.7 Function Documentation

7.7.1 void CMT_GetDefaultConfig (cmt_config_t ∗ *config*)

This API gets the default configuration structure for the CMT_Init(). Use the initialized structure unchanged in CMT_Init() or modify fields of the structure before calling the CMT_Init().

Parameters

config	The CMT configuration structure pointer.

7.7.2 void CMT_Init (CMT_Type ∗ *base,* const cmt_config_t ∗ *config,* uint32_t *busClock_Hz*)

This function ungates the module clock and sets the CMT internal clock, interrupt, and infrared output signal for the CMT module.

Parameters

base	CMT peripheral base address.
config	The CMT basic configuration structure.
busClock_Hz	The CMT module input clock - bus clock frequency.

7.7.3 void CMT_Deinit (CMT_Type ∗ *base*)

This function disables CMT modulator, interrupts, and gates the CMT clock control. CMT_Init must be called to use the CMT again.

Parameters

base	CMT peripheral base address.

7.7.4 void CMT_SetMode (CMT_Type ∗ *base*, cmt_mode_t *mode*, cmt_modulate_config_t ∗ *modulateConfig*)

Parameters

base	CMT peripheral base address.
mode	The CMT feature mode enumeration. See "cmt_mode_t".
modulate-Config	The carrier generation and modulator configuration.

7.7.5 cmt_mode_t CMT_GetMode (CMT_Type ∗ *base*)

Parameters

base	CMT peripheral base address.

Returns

The CMT mode. kCMT_DirectIROCtl Carrier modulator is disabled; the IRO signal is directly in software control. kCMT_TimeMode Carrier modulator is enabled in time mode. kCMT_FSKMode Carrier modulator is enabled in FSK mode. kCMT_BasebandMode Carrier modulator is enabled in baseband mode.

7.7.6 uint32_t CMT_GetCMTFrequency (CMT_Type * *base,* uint32_t *busClock_Hz*)

Parameters

base	CMT peripheral base address.
busClock_Hz	CMT module input clock - bus clock frequency.

Returns

The CMT clock frequency.

7.7.7 static void CMT_SetCarrirGenerateCountOne (CMT_Type * *base,* uint32_t *highCount,* uint32_t *lowCount*) [inline], [static]

This function sets the high-time and low-time of the primary data set for the CMT carrier generator counter to control the period and the duty cycle of the output carrier signal. If the CMT clock period is Tcmt, the period of the carrier generator signal equals (highCount + lowCount) * Tcmt. The duty cycle equals to highCount / (highCount + lowCount).

Parameters

base	CMT peripheral base address.
highCount	The number of CMT clocks for carrier generator signal high time, integer in the range of 1 ∼ 0xFF.
lowCount	The number of CMT clocks for carrier generator signal low time, integer in the range of 1 ∼ 0xFF.

7.7.8 static void CMT_SetCarrirGenerateCountTwo (CMT_Type * *base,* uint32_t *highCount,* uint32_t *lowCount*) [inline], [static]

This function is used for FSK mode setting the high-time and low-time of the secondary data set CMT carrier generator counter to control the period and the duty cycle of the output carrier signal. If the CMT clock period is Tcmt, the period of the carrier generator signal equals (highCount + lowCount) * Tcmt. The duty cycle equals highCount / (highCount + lowCount).

Parameters

base	CMT peripheral base address.
highCount	The number of CMT clocks for carrier generator signal high time, integer in the range of 1 ∼ 0xFF.
lowCount	The number of CMT clocks for carrier generator signal low time, integer in the range of 1 ∼ 0xFF.

7.7.9 void CMT_SetModulateMarkSpace (CMT_Type * *base,* uint32_t *markCount,* uint32_t *spaceCount*)

This function sets the mark time period of the CMT modulator counter to control the mark time of the output modulated signal from the carrier generator output signal. If the CMT clock frequency is Fcmt and the carrier out signal frequency is fcg:

- In Time and Baseband mode: The mark period of the generated signal equals (markCount + 1) / (Fcmt/8). The space period of the generated signal equals spaceCount / (Fcmt/8).
- In FSK mode: The mark period of the generated signal equals (markCount + 1)/fcg. The space period of the generated signal equals spaceCount / fcg.

Parameters

base	Base address for current CMT instance.
markCount	The number of clock period for CMT modulator signal mark period, in the range of $0 \sim$ 0xFFFF.
spaceCount	The number of clock period for CMT modulator signal space period, in the range of the $0 \sim$ 0xFFFF.

7.7.10 static void CMT_EnableExtendedSpace (CMT_Type * *base,* bool *enable*) [inline], [static]

This function is used to make the space period longer for time, baseband, and FSK modes.

Parameters

base	CMT peripheral base address.
enable	True enable the extended space, false disable the extended space.

7.7.11 void CMT_SetIroState (CMT_Type * *base,* cmt_infrared_output_state_t *state*)

Changes the states of the IRO signal when the kCMT_DirectIROMode mode is set and the IRO signal is enabled.

Parameters

base	CMT peripheral base address.
state	The control of the IRO signal. See "cmt_infrared_output_state_t"

7.7.12 static void CMT_EnableInterrupts (CMT_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the CMT interrupts according to the provided mask if enabled. The CMT only has the end of the cycle interrupt - an interrupt occurs at the end of the modulator cycle. This interrupt provides a means for the user to reload the new mark/space values into the CMT modulator data registers and verify the modulator mark and space. For example, to enable the end of cycle, do the following.

```
*     CMT_EnableInterrupts(CMT,
      kCMT_EndOfCycleInterruptEnable);
*
```

Parameters

base	CMT peripheral base address.
mask	The interrupts to enable. Logical OR of _cmt_interrupt_enable.

7.7.13 static void CMT_DisableInterrupts (CMT_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the CMT interrupts according to the provided maskIf enabled. The CMT only has the end of the cycle interrupt. For example, to disable the end of cycle, do the following.

```
*     CMT_DisableInterrupts(CMT,
      kCMT_EndOfCycleInterruptEnable);
*
```

Parameters

base	CMT peripheral base address.
mask	The interrupts to enable. Logical OR of _cmt_interrupt_enable.

7.7.14 static uint32_t CMT_GetStatusFlags (CMT_Type * *base*) [inline], [static]

The flag is set:

- When the modulator is not currently active and carrier and modulator are set to start the initial CMT transmission.
- At the end of each modulation cycle when the counter is reloaded and the carrier and modulator are enabled.

Parameters

base	CMT peripheral base address.

Returns

Current status of the end of cycle status flag
- non-zero: End-of-cycle has occurred.
- zero: End-of-cycle has not yet occurred since the flag last cleared.

Chapter 8
CRC: Cyclic Redundancy Check Driver

8.1 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 also provides a programmable polynomial, seed, and other parameters required to implement a 16-bit or 32-bit CRC standard.

8.2 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 the configuration structure. The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting a new checksum computation, the seed is set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed is set to the intermediate checksum value as obtained from previous calls to CRC_Get16bitResult() or CRC_Get32bitResult() function. After calling the CRC_Init(), one or multiple CRC_WriteData() calls follow to update the checksum with data and CRC_Get16bitResult() or CRC_Get32bitResult() follow to read the result. The crcResult member of the configuration structure determines whether the CRC_Get16bitResult() or CRC_Get32bitResult() return value is a final checksum or an intermediate checksum. The CRC_Init() function can be called as many times as required allowing for runtime changes of the CRC protocol.

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

8.3 CRC Write Data

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

8.4 CRC Get Checksum

The CRC_Get16bitResult() or CRC_Get32bitResult() function reads the CRC module data register. Depending on the prior CRC module usage, the return value is either an intermediate checksum or the final checksum. For example, for 16-bit CRCs the following call sequences can be used.

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

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

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

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

8.5 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()

The triplets are protected by the RTOS mutex to protect the CRC module against concurrent accesses from different tasks. This is an example.

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

8.6 Comments about API usage in interrupt handler

All APIs can be used from an interrupt handler although an interrupt latency of equal and lower priority interrupts increases. The user must protect against concurrent accesses from different interrupt handlers and/or tasks.

8.7 CRC Driver Examples

8.7.1 Simple examples

This is an example with the 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);
```

This is an example with the 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);
```

8.7.2 Advanced examples

Assuming there are three tasks/threads, each using the CRC module to compute checksums of a different protocol, with context switches.

First, prepare the three CRC module initialization functions for three different protocols CRC-16 (ARC), CRC-16/CCIT-FALSE, and CRC-32. The 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

These are the corresponding initialization 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 a 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);
```

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 the 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 the 32-bit checksum from the CRC module.
- uint16_t CRC_Get16bitResult (CRC_Type ∗base)
  Reads a 16-bit checksum from the CRC module.

Driver version

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

8.8 Data Structure Documentation

8.8.1 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()

8.8.1.0.0.9 Field Documentation

8.8.1.0.0.9.1 uint32_t crc_config_t::polynomial

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

8.8.1.0.0.9.2 bool crc_config_t::reflectIn

8.8.1.0.0.9.3 bool crc_config_t::reflectOut

8.8.1.0.0.9.4 bool crc_config_t::complementChecksum

8.8.1.0.0.9.5 crc_bits_t crc_config_t::crcBits

8.9 Macro Definition Documentation

8.9.1 #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

8.9.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as defeault.

8.10 Enumeration Type Documentation

8.10.1 enum crc_bits_t

Enumerator

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

8.10.2 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.

8.11 Function Documentation

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

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

Parameters

base	CRC peripheral address.
config	CRC module configuration structure.

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

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

Parameters

base	CRC peripheral address.

8.11.3 void CRC_GetDefaultConfig (crc_config_t * *config*)

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

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

Parameters

config	CRC protocol configuration structure.

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

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

Parameters

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

8.11.5 uint32_t CRC_Get32bitResult (CRC_Type * *base*)

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

Parameters

base	CRC peripheral address.

Returns

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

8.11.6 uint16_t CRC_Get16bitResult (CRC_Type * *base*)

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

Parameters

base	CRC peripheral address.

Returns

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

Chapter 9
DAC: Digital-to-Analog Converter Driver

9.1 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 enables 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.

9.2 Typical use case

9.2.1 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);
```

9.2.2 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. */
}
```

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_BufferWatermarkFlag = DAC_SR_DACBFWMF_MASK,
kDAC_BufferReadPointerTopPositionFlag = DAC_SR_DACBFRPTF_MASK,
kDAC_BufferReadPointerBottomPositionFlag = DAC_SR_DACBFRPBF_MASK }
  DAC buffer flags.
- enum _dac_buffer_interrupt_enable {
kDAC_BufferWatermarkInterruptEnable = DAC_C0_DACBWIEN_MASK,
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_watermark_t {
kDAC_BufferWatermark1Word = 0U,
kDAC_BufferWatermark2Word = 1U,
kDAC_BufferWatermark3Word = 2U,
kDAC_BufferWatermark4Word = 3U }
  DAC buffer watermark.
- enum dac_buffer_work_mode_t {
kDAC_BufferWorkAsNormalMode = 0U,
kDAC_BufferWorkAsSwingMode,
kDAC_BufferWorkAsOneTimeScanMode,
kDAC_BufferWorkAsFIFOMode }
  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 using 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.

9.3 Data Structure Documentation

9.3.1 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.

9.3.1.0.0.10 Field Documentation

9.3.1.0.0.10.1 dac_reference_voltage_source_t dac_config_t::referenceVoltageSource

9.3.1.0.0.10.2 bool dac_config_t::enableLowPowerMode

9.3.2 struct dac_buffer_config_t

Data Fields

- dac_buffer_trigger_mode_t triggerMode
  Select the buffer's trigger mode.

- dac_buffer_watermark_t watermark
  Select the buffer's watermark.
- dac_buffer_work_mode_t workMode
  Select the buffer's work mode.
- uint8_t upperLimit
  Set the upper limit for the buffer index.

9.3.2.0.0.11 Field Documentation

9.3.2.0.0.11.1 dac_buffer_trigger_mode_t dac_buffer_config_t::triggerMode

9.3.2.0.0.11.2 dac_buffer_watermark_t dac_buffer_config_t::watermark

9.3.2.0.0.11.3 dac_buffer_work_mode_t dac_buffer_config_t::workMode

9.3.2.0.0.11.4 uint8_t dac_buffer_config_t::upperLimit

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

9.4 Macro Definition Documentation

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

9.5 Enumeration Type Documentation

9.5.1 enum _dac_buffer_status_flags

Enumerator

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

9.5.2 enum _dac_buffer_interrupt_enable

Enumerator

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

9.5.3 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.

9.5.4 enum dac_buffer_trigger_mode_t

Enumerator

kDAC_BufferTriggerByHardwareMode The DAC hardware trigger is selected.
kDAC_BufferTriggerBySoftwareMode The DAC software trigger is selected.

9.5.5 enum dac_buffer_watermark_t

Enumerator

kDAC_BufferWatermark1Word 1 word away from the upper limit.
kDAC_BufferWatermark2Word 2 words away from the upper limit.
kDAC_BufferWatermark3Word 3 words away from the upper limit.
kDAC_BufferWatermark4Word 4 words away from the upper limit.

9.5.6 enum dac_buffer_work_mode_t

Enumerator

kDAC_BufferWorkAsNormalMode Normal mode.
kDAC_BufferWorkAsSwingMode Swing mode.
kDAC_BufferWorkAsOneTimeScanMode One-Time Scan mode.
kDAC_BufferWorkAsFIFOMode FIFO mode.

9.6 Function Documentation

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

This function initializes the DAC module including the following operations.

- 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".

9.6.2 void DAC_Deinit (DAC_Type * *base*)

This function de-initializes the DAC module including the following operations.

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

Parameters

base	DAC peripheral base address.

9.6.3 void DAC_GetDefaultConfig (dac_config_t * *config*)

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

9.6.6 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".

9.6.7 void DAC_GetDefaultBufferConfig (dac_buffer_config_t * *config*)

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

```
*   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".

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

Parameters

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

9.6.9 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.

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

This function triggers the function using 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.

9.6.11 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 a software trigger or a hardware trigger.

Parameters

base	DAC peripheral base address.

Returns

The current read pointer of the DAC buffer.

9.6.12 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 a software trigger or a hardware trigger. After the read pointer changes, the DAC output value also changes.

Parameters

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

9.6.13 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".

9.6.14 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".

9.6.15 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".

9.6.16 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 10
DMAMUX: Direct Memory Access Multiplexer Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMAMUX Operation

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

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.
- static void DMAMUX_EnablePeriodTrigger (DMAMUX_Type ∗base, uint32_t channel)
  Enables the DMAMUX period trigger.
- static void DMAMUX_DisablePeriodTrigger (DMAMUX_Type ∗base, uint32_t channel)
  Disables the DMAMUX period trigger.

10.3 Macro Definition Documentation

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

10.4 Function Documentation

10.4.1 void DMAMUX_Init (DMAMUX_Type * *base*)

This function ungates the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

10.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

10.4.3 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.

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

This function disables the DMAMUX channel.

Note

The user must disable the DMAMUX channel before configuring it.

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.

10.4.5 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 the DMA transfer.

10.4.6 static void DMAMUX_EnablePeriodTrigger (DMAMUX_Type * *base*, uint32_t *channel*) [inline], [static]

This function enables the DMAMUX period trigger feature.

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.

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

This function disables the DMAMUX period trigger.

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.

Chapter 11
DSPI: Serial Peripheral Interface Driver

11.1 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

11.2 DSPI Driver

11.2.1 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.

11.2.2 Typical use case

11.2.2.1 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);
```

11.2.2.2 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");
//}
```

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, 3))
  DSPI driver version 2.1.3.

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 a default value.
- 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 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.

11.2.3 Data Structure Documentation

11.2.3.1 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.

11.2.3.1.0.12 Field Documentation

11.2.3.1.0.12.1 bool dspi_command_data_config_t::isPcsContinuous

11.2.3.1.0.12.2 dspi_ctar_selection_t dspi_command_data_config_t::whichCtar

11.2.3.1.0.12.3 dspi_which_pcs_t dspi_command_data_config_t::whichPcs

11.2.3.1.0.12.4 bool dspi_command_data_config_t::isEndOfQueue

11.2.3.1.0.12.5 bool dspi_command_data_config_t::clearTransferCount

11.2.3.2 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.*

11.2.3.2.0.13 Field Documentation

11.2.3.2.0.13.1 uint32_t dspi_master_ctar_config_t::baudRate

11.2.3.2.0.13.2 uint32_t dspi_master_ctar_config_t::bitsPerFrame

11.2.3.2.0.13.3 dspi_clock_polarity_t dspi_master_ctar_config_t::cpol

11.2.3.2.0.13.4 dspi_clock_phase_t dspi_master_ctar_config_t::cpha

11.2.3.2.0.13.5 dspi_shift_direction_t dspi_master_ctar_config_t::direction

11.2.3.2.0.13.6 uint32_t dspi_master_ctar_config_t::pcsToSckDelayInNanoSec

It also sets the boundary value if out of range.

11.2.3.2.0.13.7 uint32_t dspi_master_ctar_config_t::lastSckToPcsDelayInNanoSec

It also sets the boundary value if out of range.

11.2.3.2.0.13.8 uint32_t dspi_master_ctar_config_t::betweenTransferDelayInNanoSec

It also sets the boundary value if out of range.

11.2.3.3 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.

11.2.3.3.0.14 Field Documentation

11.2.3.3.0.14.1 dspi_ctar_selection_t dspi_master_config_t::whichCtar

11.2.3.3.0.14.2 dspi_master_ctar_config_t dspi_master_config_t::ctarConfig

11.2.3.3.0.14.3 dspi_which_pcs_t dspi_master_config_t::whichPcs

11.2.3.3.0.14.4 dspi_pcs_polarity_config_t dspi_master_config_t::pcsActiveHighOrLow

11.2.3.3.0.14.5 bool dspi_master_config_t::enableContinuousSCK

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

11.2.3.3.0.14.6 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.

11.2.3.3.0.14.7 bool dspi_master_config_t::enableModifiedTimingFormat

11.2.3.3.0.14.8 dspi_master_sample_point_t dspi_master_config_t::samplePoint

It's valid only when CPHA=0.

11.2.3.4 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.

11.2.3.4.0.15 Field Documentation

11.2.3.4.0.15.1 uint32_t dspi_slave_ctar_config_t::bitsPerFrame

11.2.3.4.0.15.2 dspi_clock_polarity_t dspi_slave_ctar_config_t::cpol

11.2.3.4.0.15.3 dspi_clock_phase_t dspi_slave_ctar_config_t::cpha

Slave only supports MSB and does not support LSB.

11.2.3.5 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.

11.2.3.5.0.16 Field Documentation

11.2.3.5.0.16.1 dspi_ctar_selection_t dspi_slave_config_t::whichCtar

11.2.3.5.0.16.2 dspi_slave_ctar_config_t dspi_slave_config_t::ctarConfig

11.2.3.5.0.16.3 bool dspi_slave_config_t::enableContinuousSCK

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

11.2.3.5.0.16.4 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.

11.2.3.5.0.16.5 bool dspi_slave_config_t::enableModifiedTimingFormat

11.2.3.5.0.16.6 dspi_master_sample_point_t dspi_slave_config_t::samplePoint

It's valid only when CPHA=0.

11.2.3.6 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.

11.2.3.6.0.17 Field Documentation

11.2.3.6.0.17.1 uint8_t* dspi_transfer_t::txData

11.2.3.6.0.17.2 uint8_t* dspi_transfer_t::rxData

11.2.3.6.0.17.3 volatile size_t dspi_transfer_t::dataSize

11.2.3.6.0.17.4 uint32_t dspi_transfer_t::configFlags

11.2.3.7 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.

11.2.3.7.0.18 Field Documentation

11.2.3.7.0.18.1 uint32_t dspi_master_handle_t::bitsPerFrame

11.2.3.7.0.18.2 volatile uint32_t dspi_master_handle_t::command

11.2.3.7.0.18.3 volatile uint32_t dspi_master_handle_t::lastCommand

11.2.3.7.0.18.4 uint8_t dspi_master_handle_t::fifoSize

11.2.3.7.0.18.5 volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer

11.2.3.7.0.18.6 volatile bool dspi_master_handle_t::isThereExtraByte

11.2.3.7.0.18.7 uint8_t* volatile dspi_master_handle_t::txData

11.2.3.7.0.18.8 uint8_t* volatile dspi_master_handle_t::rxData

11.2.3.7.0.18.9 volatile size_t dspi_master_handle_t::remainingSendByteCount

11.2.3.7.0.18.10 volatile size_t dspi_master_handle_t::remainingReceiveByteCount

11.2.3.7.0.18.11 volatile uint8_t dspi_master_handle_t::state

11.2.3.7.0.18.12 dspi_master_transfer_callback_t dspi_master_handle_t::callback

11.2.3.7.0.18.13 void* dspi_master_handle_t::userData

11.2.3.8 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.

11.2.3.8.0.19 Field Documentation

11.2.3.8.0.19.1 uint32_t dspi_slave_handle_t::bitsPerFrame

11.2.3.8.0.19.2 volatile bool dspi_slave_handle_t::isThereExtraByte

11.2.3.8.0.19.3 uint8_t∗ volatile dspi_slave_handle_t::txData

11.2.3.8.0.19.4 uint8_t∗ volatile dspi_slave_handle_t::rxData

11.2.3.8.0.19.5 volatile size_t dspi_slave_handle_t::remainingSendByteCount

11.2.3.8.0.19.6 volatile size_t dspi_slave_handle_t::remainingReceiveByteCount

11.2.3.8.0.19.7 volatile uint8_t dspi_slave_handle_t::state

11.2.3.8.0.19.8 volatile uint32_t dspi_slave_handle_t::errorCount

11.2.3.8.0.19.9 dspi_slave_transfer_callback_t dspi_slave_handle_t::callback

11.2.3.8.0.19.10 void∗ dspi_slave_handle_t::userData

11.2.4 Macro Definition Documentation

11.2.4.1 #define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 3))

11.2.4.2 #define DSPI_DUMMY_DATA (0x00U)

Dummy data used for Tx if there is no txData.

11.2.4.3 #define DSPI_MASTER_CTAR_SHIFT (0U)

11.2.4.4 #define DSPI_MASTER_CTAR_MASK (0x0FU)

11.2.4.5 #define DSPI_MASTER_PCS_SHIFT (4U)

11.2.4.6 #define DSPI_MASTER_PCS_MASK (0xF0U)

11.2.4.7 #define DSPI_SLAVE_CTAR_SHIFT (0U)

11.2.4.8 #define DSPI_SLAVE_CTAR_MASK (0x07U)

11.2.5 Typedef Documentation

11.2.5.1 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.

11.2.5.2 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.

11.2.6 Enumeration Type Documentation

11.2.6.1 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.

11.2.6.2 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.

11.2.6.3 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.

11.2.6.4 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.

11.2.6.5 enum dspi_master_slave_mode_t

Enumerator

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

11.2.6.6 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.

11.2.6.7 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].

11.2.6.8 enum dspi_pcs_polarity_config_t

Enumerator

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

11.2.6.9 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).

11.2.6.10 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).

11.2.6.11 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.

11.2.6.12 enum dspi_shift_direction_t

Enumerator

kDSPI_MsbFirst Data transfers start with most significant bit.
kDSPI_LsbFirst Data transfers start with least significant bit. Shifting out of LSB is not supported for slave

11.2.6.13 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.

11.2.6.14 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.

11.2.6.15 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.

11.2.6.16 enum _dspi_transfer_config_flag_for_slave

Enumerator

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

11.2.6.17 enum _dspi_transfer_state

Enumerator

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

11.2.7 Function Documentation

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

This function initializes the DSPI master configuration. This is an example use case.

```
*   dspi_master_config_t  masterConfig;
*   masterConfig.whichCtar                                  = kDSPI_Ctar0;
*   masterConfig.ctarConfig.baudRate                        = 500000000U;
*   masterConfig.ctarConfig.bitsPerFrame                    = 8;
*   masterConfig.ctarConfig.cpol                            =
      kDSPI_ClockPolarityActiveHigh;
*   masterConfig.ctarConfig.cpha                            =
      kDSPI_ClockPhaseFirstEdge;
*   masterConfig.ctarConfig.direction                       =
```

```
     kDSPI_MsbFirst;
*   masterConfig.ctarConfig.pcsToSckDelayInNanoSec        = 1000000000U /
      masterConfig.ctarConfig.baudRate ;
*   masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec    = 1000000000U
      / masterConfig.ctarConfig.baudRate ;
*   masterConfig.ctarConfig.betweenTransferDelayInNanoSec =
      1000000000U / 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.

11.2.7.2 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

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

This function initializes the DSPI slave configuration. This is an example use case.

```
*   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.

11.2.7.4 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(). This is an example.

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

Parameters

slaveConfig	Pointer to the dspi_slave_config_t structure.

11.2.7.5 void DSPI_Deinit (SPI_Type * *base*)

Call this API to disable the DSPI clock.

Parameters

base	DSPI peripheral address.

11.2.7.6 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.

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

Parameters

base	DSPI peripheral address.

Returns

DSPI status (in SR register).

11.2.7.8 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. This is an example.

```
*  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).

11.2.7.9 void DSPI_EnableInterrupts (SPI_Type ∗ *base,* uint32_t *mask*)

This function configures the various interrupt masks of the DSPI. The parameters are a 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.

11.2.7.10 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.

11.2.7.11 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.

11.2.7.12 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.

11.2.7.13 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.

11.2.7.14 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.

11.2.7.15 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.

11.2.7.16 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.

11.2.7.17 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.

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

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

Parameters

base	DSPI peripheral address.

11.2.7.19 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.

11.2.7.20 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; Otherwise, enables (true) the TX FIFO
enableRxFifo	Disables (false) the RX FIFO; Otherwise, enables (true) the RX FIFO

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

Parameters

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

11.2.7.22 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.

11.2.7.23 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

11.2.7.24 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 the configuration 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

11.2.7.25 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 the following. 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. Note that returning the 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.

11.2.7.26 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.

11.2.7.27 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(). This is an example.

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

Parameters

command	Pointer to the dspi_command_data_config_t structure.

11.2.7.28 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.

11.2.7.29 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_WriteCommandDataMastermode 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 the command structure.

Returns

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

11.2.7.30 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_MasterWriteCommandDataBlocking(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.

11.2.7.31 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.

11.2.7.32 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.

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

Parameters

base	DSPI peripheral address.

Returns

The data from the read data buffer.

11.2.7.34 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.

11.2.7.35 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.

11.2.7.36 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.

11.2.7.37 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.

11.2.7.38 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.

11.2.7.39 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.

11.2.7.40 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.

11.2.7.41 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.

11.2.7.42 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.

11.2.7.43 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.

11.2.7.44 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.

11.3 DSPI DMA Driver

11.3.1 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.

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.

11.3.2 Data Structure Documentation

11.3.2.1 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

11.3.2.1.0.20 Field Documentation

11.3.2.1.0.20.1 uint32_t dspi_master_dma_handle_t::bitsPerFrame

11.3.2.1.0.20.2 volatile uint32_t dspi_master_dma_handle_t::command

11.3.2.1.0.20.3 volatile uint32_t dspi_master_dma_handle_t::lastCommand

11.3.2.1.0.20.4 uint8_t dspi_master_dma_handle_t::fifoSize

11.3.2.1.0.20.5 volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer

11.3.2.1.0.20.6 volatile bool dspi_master_dma_handle_t::isThereExtraByte

11.3.2.1.0.20.7 uint8_t∗ volatile dspi_master_dma_handle_t::txData

11.3.2.1.0.20.8 uint8_t∗ volatile dspi_master_dma_handle_t::rxData

11.3.2.1.0.20.9 volatile size_t dspi_master_dma_handle_t::remainingSendByteCount

11.3.2.1.0.20.10 volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount

11.3.2.1.0.20.11 uint32_t dspi_master_dma_handle_t::rxBuffIfNull

11.3.2.1.0.20.12 uint32_t dspi_master_dma_handle_t::txBuffIfNull

11.3.2.1.0.20.13 volatile uint8_t dspi_master_dma_handle_t::state

11.3.2.1.0.20.14 dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback

11.3.2.1.0.20.15 void∗ dspi_master_dma_handle_t::userData

11.3.2.2 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

11.3.2.2.0.21 Field Documentation

11.3.2.2.0.21.1 uint32_t dspi_slave_dma_handle_t::bitsPerFrame

11.3.2.2.0.21.2 volatile bool dspi_slave_dma_handle_t::isThereExtraByte

11.3.2.2.0.21.3 uint8_t* volatile dspi_slave_dma_handle_t::txData

11.3.2.2.0.21.4 uint8_t* volatile dspi_slave_dma_handle_t::rxData

11.3.2.2.0.21.5 volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount

11.3.2.2.0.21.6 volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount

11.3.2.2.0.21.7 uint32_t dspi_slave_dma_handle_t::rxBuffIfNull

11.3.2.2.0.21.8 uint32_t dspi_slave_dma_handle_t::txBuffIfNull

11.3.2.2.0.21.9 uint32_t dspi_slave_dma_handle_t::txLastData

11.3.2.2.0.21.10 volatile uint8_t dspi_slave_dma_handle_t::state

11.3.2.2.0.21.11 uint32_t dspi_slave_dma_handle_t::errorCount

11.3.2.2.0.21.12 dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback

11.3.2.2.0.21.13 void* dspi_slave_dma_handle_t::userData

11.3.3 Typedef Documentation

11.3.3.1 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.

11.3.3.2 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.

11.3.4 Function Documentation

11.3.4.1 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.

11.3.4.2 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.

11.3.4.3 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.

11.3.4.4 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.

11.3.4.5 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.
dmaRxRegTo-RxDataHandle	dmaRxRegToRxDataHandle pointer to dma_handle_t.
dmaTxDataTo-TxRegHandle	dmaTxDataToTxRegHandle pointer to dma_handle_t.

11.3.4.6 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 eight.

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.

11.3.4.7 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.

11.3.4.8 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.

11.4 DSPI eDMA Driver

11.4.1 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.

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.

11.4.2 Data Structure Documentation

11.4.2.1 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.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- volatile uint8_t state
  DSPI transfer state , _dspi_transfer_state.
- 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.
- 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.

11.4.2.1.0.22 Field Documentation

11.4.2.1.0.22.1 uint32_t dspi_master_edma_handle_t::bitsPerFrame

11.4.2.1.0.22.2 volatile uint32_t dspi_master_edma_handle_t::command

11.4.2.1.0.22.3 volatile uint32_t dspi_master_edma_handle_t::lastCommand

11.4.2.1.0.22.4 uint8_t dspi_master_edma_handle_t::fifoSize

11.4.2.1.0.22.5 volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer

11.4.2.1.0.22.6 uint8_t dspi_master_edma_handle_t::nbytes

11.4.2.1.0.22.7 volatile uint8_t dspi_master_edma_handle_t::state

11.4.2.1.0.22.8 uint8_t∗ volatile dspi_master_edma_handle_t::txData

11.4.2.1.0.22.9 uint8_t∗ volatile dspi_master_edma_handle_t::rxData

11.4.2.1.0.22.10 volatile size_t dspi_master_edma_handle_t::remainingSendByteCount

11.4.2.1.0.22.11 volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount

11.4.2.1.0.22.12 uint32_t dspi_master_edma_handle_t::rxBuffIfNull

11.4.2.1.0.22.13 uint32_t dspi_master_edma_handle_t::txBuffIfNull

11.4.2.1.0.22.14 dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback

11.4.2.1.0.22.15 void∗ dspi_master_edma_handle_t::userData

11.4.2.2 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.
- 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.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- volatile uint8_t state
  DSPI transfer state.
- 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

11.4.2.2.0.23 Field Documentation

11.4.2.2.0.23.1 uint32_t dspi_slave_edma_handle_t::bitsPerFrame

11.4.2.2.0.23.2 uint8_t* volatile dspi_slave_edma_handle_t::txData

11.4.2.2.0.23.3 uint8_t* volatile dspi_slave_edma_handle_t::rxData

11.4.2.2.0.23.4 volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount

11.4.2.2.0.23.5 volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount

11.4.2.2.0.23.6 uint32_t dspi_slave_edma_handle_t::rxBuffIfNull

11.4.2.2.0.23.7 uint32_t dspi_slave_edma_handle_t::txBuffIfNull

11.4.2.2.0.23.8 uint32_t dspi_slave_edma_handle_t::txLastData

11.4.2.2.0.23.9 uint8_t dspi_slave_edma_handle_t::nbytes

11.4.2.2.0.23.10 volatile uint8_t dspi_slave_edma_handle_t::state

11.4.2.2.0.23.11 dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback

11.4.2.2.0.23.12 void* dspi_slave_edma_handle_t::userData

11.4.3 Typedef Documentation

11.4.3.1 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.

11.4.3.2 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.

11.4.4 Function Documentation

11.4.4.1 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.

11.4.4.2 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.

11.4.4.3 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.

11.4.4.4 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.

11.4.4.5 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.

11.4.4.6 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 is 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 eight.

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.

11.4.4.7 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.

11.4.4.8 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.

11.5 DSPI FreeRTOS Driver

11.5.1 Overview

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.

11.5.2 Function Documentation

11.5.2.1 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.

11.5.2.2 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.

11.5.2.3 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.

11.6 DSPI µCOS/II Driver

11.6.1 Overview

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.

11.6.2 Function Documentation

11.6.2.1 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.

11.6.2.2 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.

11.6.2.3 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.

11.7 DSPI µCOS/III Driver

11.7.1 Overview

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.

11.7.2 Function Documentation

11.7.2.1 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.

11.7.2.2 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.

11.7.2.3 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 12
eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

12.1 Overview

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

12.2 Typical use case

12.2.1 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);
```

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_GroupPriorityErrorFlag = DMA_ES_GPE_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, 1, 1))
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_EnableAsyncRequest (DMA_Type *base, uint32_t channel, bool enable)
  Enables an async request 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 dest-Modulo)
  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_GetRemainingMajorLoopCount (DMA_Type ∗base, uint32_t channel)
  Gets the remaining major loop count 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.

12.3 Data Structure Documentation

12.3.1 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.

12.3.1.0.0.24 Field Documentation

12.3.1.0.0.24.1 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.

12.3.1.0.0.24.2 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.

12.3.1.0.0.24.3 bool edma_config_t::enableDebugMode

When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

12.3.2 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.

12.3.2.0.0.25 Field Documentation

12.3.2.0.0.25.1 uint32_t edma_transfer_config_t::srcAddr

12.3.2.0.0.25.2 uint32_t edma_transfer_config_t::destAddr

12.3.2.0.0.25.3 edma_transfer_size_t edma_transfer_config_t::srcTransferSize

12.3.2.0.0.25.4 edma_transfer_size_t edma_transfer_config_t::destTransferSize

12.3.2.0.0.25.5 int16_t edma_transfer_config_t::srcOffset

12.3.2.0.0.25.6 int16_t edma_transfer_config_t::destOffset

12.3.2.0.0.25.7 uint32_t edma_transfer_config_t::majorLoopCounts

12.3.3 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.

12.3.4 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.

12.3.4.0.0.26 Field Documentation

12.3.4.0.0.26.1 bool edma_minor_offset_config_t::enableSrcMinorOffset

12.3.4.0.0.26.2 bool edma_minor_offset_config_t::enableDestMinorOffset

12.3.4.0.0.26.3 uint32_t edma_minor_offset_config_t::minorOffset

12.3.5 struct edma_tcd_t

This structure is same as TCD register which is described in reference manual, and is used to configure the 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.

12.3.5.0.0.27 Field Documentation

12.3.5.0.0.27.1 __IO uint16_t edma_tcd_t::CITER

12.3.5.0.0.27.2 __IO uint16_t edma_tcd_t::BITER

12.3.6 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.

12.3.6.0.0.28 Field Documentation

12.3.6.0.0.28.1 edma_callback edma_handle_t::callback

12.3.6.0.0.28.2 void* edma_handle_t::userData

12.3.6.0.0.28.3 DMA_Type* edma_handle_t::base

12.3.6.0.0.28.4 edma_tcd_t* edma_handle_t::tcdPool

12.3.6.0.0.28.5 uint8_t edma_handle_t::channel

12.3.6.0.0.28.6 volatile int8_t edma_handle_t::header

Should point to the next TCD to be loaded into the eDMA engine.

12.3.6.0.0.28.7 volatile int8_t edma_handle_t::tail

Should point to the next TCD to be stored into the memory pool.

12.3.6.0.0.28.8 volatile int8_t edma_handle_t::tcdUsed

Should reflect the number of TCDs can be used/loaded in the memory.

12.3.6.0.0.28.9 volatile int8_t edma_handle_t::tcdSize

12.3.6.0.0.28.10 uint8_t edma_handle_t::flags

12.4 Macro Definition Documentation

12.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

Version 2.1.1.

12.5 Typedef Documentation

12.5.1 typedef void(∗ edma_callback)(struct _edma_handle ∗handle, void ∗userData, bool transferDone, uint32_t tcds)

12.6 Enumeration Type Documentation

12.6.1 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.

12.6.2 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.

12.6.3 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.

12.6.4 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.

12.6.5 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

12.6.6 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_GroupPriorityErrorFlag Group priority is not unique.
kEDMA_ValidFlag No error occurred, this bit is 0. Otherwise, it is 1.

12.6.7 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.

12.6.8 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.

12.6.9 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.

12.7 Function Documentation

12.7.1 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.

12.7.2 void EDMA_Deinit (DMA_Type ∗ *base*)

This function gates the eDMA clock.

Parameters

base	eDMA peripheral base address.

12.7.3 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.

12.7.4 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.

12.7.5 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 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	Pointer to eDMA transfer configuration structure.
nextTcd	Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

Note

If 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.

12.7.6 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.

12.7.7 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.

12.7.8 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 DONE flag is cleared before calling this interface, or the configuration is invalid.

12.7.9 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

12.7.10 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.

12.7.11 static void EDMA_EnableAsyncRequest (DMA_Type * *base,* uint32_t *channel,* bool *enable*) [inline], [static]

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
enable	The command to enable (true) or disable (false).

12.7.12 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).

12.7.13 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 interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

12.7.14 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.

12.7.15 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.

12.7.16 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 TCD 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 users 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.

12.7.17 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.

12.7.18 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

Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

tcd	Point to the TCD structure.
type	Channel link type, it can be one of: • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink
linkedChannel	The linked channel number.

12.7.19 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

12.7.20 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.

12.7.21 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).

12.7.22 void EDMA_TcdEnableInterrupts (edma_tcd_t * *tcd,* uint32_t *mask*)

Parameters

tcd	Point to the TCD structure.
mask	The mask of interrupt source to be set. Users need to use the defined edma_interrupt-_enable_t type.

12.7.23 void EDMA_TcdDisableInterrupts (edma_tcd_t * *tcd*, uint32_t *mask*)

Parameters

tcd	Point to the TCD structure.
mask	The mask of interrupt source to be set. Users need to use the defined edma_interrupt-_enable_t type.

12.7.24 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.

12.7.25 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.

12.7.26 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.

12.7.27 uint32_t EDMA_GetRemainingMajorLoopCount (DMA_Type * *base*, uint32_t *channel*)

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the the number of major loop count that has not finished.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Returns

Major loop count which has not been transferred yet for the current TCD.

Note

1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.
    1. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTES(initially configured)

12.7.28 static uint32_t EDMA_GetErrorStatusFlags (DMA_Type * *base*) [inline], [static]

Parameters

base	eDMA peripheral base address.

Returns

The mask of error status flags. Users need to use the _edma_error_status_flags type to decode the return variables.

12.7.29 uint32_t EDMA_GetChannelStatusFlags (DMA_Type ∗ *base*, uint32_t *channel*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Returns

The mask of channel status flags. Users need to use the _edma_channel_status_flags type to decode the return variables.

12.7.30 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 channel status to be cleared. Users need to use the defined _edma_-channel_status_flags type.

12.7.31 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.

12.7.32 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.

12.7.33 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.

12.7.34 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).

12.7.35 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_Queue-Full	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.

12.7.36 void EDMA_StartTransfer (edma_handle_t * *handle*)

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

handle	eDMA handle pointer.

12.7.37 void EDMA_StopTransfer (edma_handle_t ∗ *handle*)

This function disables the channel request to pause the transfer. Users can call EDMA_StartTransfer() again to resume the transfer.

Parameters

handle	eDMA handle pointer.

12.7.38 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.

12.7.39 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.

Note: For the case using TCD queue, when the major iteration count is exhausted, additional operations are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As sga and sga_index are calculated based on the DLAST_SGA bitfield lies in the TCD_CSR register, the sga_index in this case should be 2 (DLAST_SGA of TCD[1] stores the address of TCD[2]). Thus, the "tcdUsed" updated should be (tcdUsed - 2U) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already.).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and tcdUsed updated are identical for them. tcdUsed are both 0 in this case as no

TCD to be loaded.

See the "eDMA basic data flow" in the eDMA Functional description part of the Reference Manual for further details.

Parameters

handle	eDMA handle pointer.

Chapter 13
EWM: External Watchdog Monitor Driver

13.1 Overview

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

13.2 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);
```

Data Structures

- struct ewm_config_t
  Data structure for EWM configuration. More...

Enumerations

- enum ewm_lpo_clock_source_t {
  kEWM_LpoClockSource0 = 0U,
  kEWM_LpoClockSource1 = 1U,
  kEWM_LpoClockSource2 = 2U,
  kEWM_LpoClockSource3 = 3U }
  Describes EWM clock source.
- enum _ewm_interrupt_enable_t { kEWM_InterruptEnable = EWM_CTRL_INTEN_MASK }
  EWM interrupt configuration structure with 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 all status flags.
- void EWM_Refresh (EWM_Type *base)
  Services the EWM.

13.3 Data Structure Documentation

13.3.1 struct ewm_config_t

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.
- ewm_lpo_clock_source_t clockSource
  Clock source select.
- uint8_t prescaler
  Clock prescaler value.
- uint8_t compareLowValue
  Compare low-register value.
- uint8_t compareHighValue
  Compare high-register value.

13.4 Macro Definition Documentation

13.4.1 #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

13.5 Enumeration Type Documentation

13.5.1 enum ewm_lpo_clock_source_t

Enumerator

kEWM_LpoClockSource0	EWM clock sourced from lpo_clk[0].
kEWM_LpoClockSource1	EWM clock sourced from lpo_clk[1].
kEWM_LpoClockSource2	EWM clock sourced from lpo_clk[2].

kEWM_LpoClockSource3 EWM clock sourced from lpo_clk[3].

13.5.2 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM_InterruptEnable Enable the EWM to generate an interrupt.

13.5.3 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.

13.6 Function Documentation

13.6.1 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 the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

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

Parameters

base	EWM peripheral base address
config	The configuration of the EWM

13.6.2 void EWM_Deinit (EWM_Type ∗ *base*)

This function is used to shut down the EWM.

Parameters

base	EWM peripheral base address

13.6.3 void EWM_GetDefaultConfig (ewm_config_t * *config*)

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

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

Parameters

config	Pointer to the EWM configuration structure.

See Also

ewm_config_t

13.6.4 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

13.6.5 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

13.6.6 static uint32_t EWM_GetStatusFlags (EWM_Type * *base*) [inline], [static]

This function gets all status flags.

This is an example for getting the running flag.

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

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.

13.6.7 void EWM_Refresh (EWM_Type * *base*)

This function resets the EWM counter to zero.

Parameters

base	EWM peripheral base address

Chapter 14
C90TFS Flash Driver

14.1 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...
- union pflash_protection_status_low_t
  PFlash protection status - low 32bit. More...
- struct pflash_protection_status_t
  PFlash protection status - full. More...
- struct flash_prefetch_speculation_status_t
  Flash prefetch speculation status. More...
- struct flash_protection_config_t
  Active flash protection information for the current operation. More...
- struct flash_access_config_t
  Active flash Execute-Only access information for the current operation. 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,
kFLASH_PropertyEepromTotalSize = 0x15U,
kFLASH_PropertyFlashMemoryIndex = 0x20U }

  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_ResourceRangePflashSwapIfrStart = 0x10000U,
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.
- enum flash_memory_index_t {
kFLASH_MemoryIndexPrimaryFlash = 0x00U,
kFLASH_MemoryIndexSecondaryFlash = 0x01U }
Enumeration for the flash memory index.
- enum flash_prefetch_speculation_option_t

Enumeration for the two possible options of flash prefetch speculation.

Flash version

- enum _flash_driver_version_constants {
  kFLASH_DriverVersionName = 'F',
  kFLASH_DriverVersionMajor = 2,
  kFLASH_DriverVersionMinor = 2,
  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, 2, 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_SSD_IS_SECONDARY_FLASH_SUPPORTED (0)
  Indicates whether the secondary flash is supported in the Flash driver.
- #define FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_PROTECTION_REGISTER (0)
  Indicates whether the secondary flash has its own protection register in flash module.
- #define FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_ACCESS_REGISTER (0)
  Indicates whether the secondary flash has its own Execute-Only access register in flash module.
- #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,
kStatus_FLASH_ReadOnlyProperty = MAKE_STATUS(kStatusGroupFlashDriver, 18),
kStatus_FLASH_InvalidPropertyValue,
kStatus_FLASH_InvalidSpeculationOption }
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_EraseAllUnsecure (flash_config_t ∗config, uint32_t key)
  Erases the entire flash, including protected sectors.
- status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t ∗config, uint32_t key)
  Erases all program flash execute-only segments defined by the FXACC registers.

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_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_Set-FlexramAsRamError	Failed to set flexram as RAM.
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_Recover-FlexramAsEepromError	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_ReadResource (flash_config_t ∗config, uint32_t start, uint32_t ∗dst, uint32_-t lengthInBytes, flash_read_resource_option_t option)
  Reads the resource with data at locations passed in through parameters.
- status_t FLASH_ReadOnce (flash_config_t ∗config, uint32_t index, uint32_t ∗dst, uint32_t length-InBytes)
  Reads the Program Once Field 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.
eepromData-SizeCode	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
flexnvm-PartitionCode	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_Invalid-Argument	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 command execution.

- status_t FLASH_PflashSetProtection (flash_config_t *config, pflash_protection_status_t *protect-Status)

Sets the PFlash Protection to the intended protection status.
- status_t FLASH_PflashGetProtection (flash_config_t *config, pflash_protection_status_t *protectStatus)
  Gets the PFlash protection status.

14.2 Data Structure Documentation

14.2.1 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 * flashCommonBitOperation
  Execute-in-RAM function: flash_common_bit_operation.

14.2.1.0.0.29 Field Documentation

14.2.1.0.0.29.1 uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

14.2.1.0.0.29.2 uint32_t* flash_execute_in_ram_function_config_t::flashRunCommand

14.2.1.0.0.29.3 uint32_t* flash_execute_in_ram_function_config_t::flashCommonBitOperation

14.2.2 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.

14.2.2.0.0.30 Field Documentation

14.2.2.0.0.30.1 flash_swap_state_t flash_swap_state_config_t::flashSwapState

14.2.2.0.0.30.2 flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus

14.2.2.0.0.30.3 flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus

14.2.3 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.

14.2.3.0.0.31 Field Documentation

14.2.3.0.0.31.1 uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress

14.2.3.0.0.31.2 uint16_t flash_swap_ifr_field_config_t::swapEnableWord

14.2.3.0.0.31.3 uint8_t flash_swap_ifr_field_config_t::reserved0[4]

14.2.4 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.

14.2.4.0.0.32 Field Documentation

14.2.4.0.0.32.1 uint32_t flash_swap_ifr_field_data_t::flashSwapIfrData[2]

14.2.4.0.0.32.2 flash_swap_ifr_field_config_t flash_swap_ifr_field_data_t::flashSwapIfrField

14.2.5 union pflash_protection_status_low_t

Data Fields

- uint32_t protl32b
  PROT[31:0] .

- uint8_t protsl
  PROTS[7:0] .
- uint8_t protsh
  PROTS[15:8] .

14.2.5.0.0.33 Field Documentation

14.2.5.0.0.33.1 uint32_t pflash_protection_status_low_t::protl32b

14.2.5.0.0.33.2 uint8_t pflash_protection_status_low_t::protsl

14.2.5.0.0.33.3 uint8_t pflash_protection_status_low_t::protsh

14.2.6 struct pflash_protection_status_t

Data Fields

- pflash_protection_status_low_t valueLow32b
  PROT[31:0] or PROTS[15:0].

14.2.6.0.0.34 Field Documentation

14.2.6.0.0.34.1 pflash_protection_status_low_t pflash_protection_status_t::valueLow32b

14.2.7 struct flash_prefetch_speculation_status_t

Data Fields

- flash_prefetch_speculation_option_t instructionOption
  Instruction speculation.
- flash_prefetch_speculation_option_t dataOption
  Data speculation.

14.2.7.0.0.35 Field Documentation

14.2.7.0.0.35.1 flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t-::instructionOption

14.2.7.0.0.35.2 flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::data-Option

14.2.8 struct flash_protection_config_t

Data Fields

- uint32_t regionBase
  Base address of flash protection region.
- uint32_t regionSize

size of flash protection region.
- uint32_t regionCount
  flash protection region count.

14.2.8.0.0.36 Field Documentation

14.2.8.0.0.36.1 uint32_t flash_protection_config_t::regionBase

14.2.8.0.0.36.2 uint32_t flash_protection_config_t::regionSize

14.2.8.0.0.36.3 uint32_t flash_protection_config_t::regionCount

14.2.9 struct flash_access_config_t

Data Fields

- uint32_t SegmentBase
  Base address of flash Execute-Only segment.
- uint32_t SegmentSize
  size of flash Execute-Only segment.
- uint32_t SegmentCount
  flash Execute-Only segment count.

14.2.9.0.0.37 Field Documentation

14.2.9.0.0.37.1 uint32_t flash_access_config_t::SegmentBase

14.2.9.0.0.37.2 uint32_t flash_access_config_t::SegmentSize

14.2.9.0.0.37.3 uint32_t flash_access_config_t::SegmentCount

14.2.10 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.

14.2.10.0.0.38 Field Documentation

14.2.10.0.0.38.1 uint32_t flash_operation_config_t::convertedAddress

14.2.10.0.0.38.2 uint32_t flash_operation_config_t::activeSectorSize

14.2.10.0.0.38.3 uint32_t flash_operation_config_t::activeBlockSize

14.2.10.0.0.38.4 uint32_t flash_operation_config_t::blockWriteUnitSize

14.2.10.0.0.38.5 uint32_t flash_operation_config_t::sectorCmdAddressAligment

14.2.10.0.0.38.6 uint32_t flash_operation_config_t::partCmdAddressAligment

14.2.10.0.0.38.7 uint32_t flash_operation_config_t::resourceCmdAddressAligment

14.2.10.0.0.38.8 uint32_t flash_operation_config_t::checkCmdAddressAligment

14.2.11 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.
- uint32_t FlashMemoryIndex
  0 - primary flash; 1 - secondary flash

14.2.11.0.0.39 Field Documentation

14.2.11.0.0.39.1 uint32_t flash_config_t::PFlashTotalSize

14.2.11.0.0.39.2 uint32_t flash_config_t::PFlashBlockCount

14.2.11.0.0.39.3 uint32_t flash_config_t::PFlashSectorSize

14.2.11.0.0.39.4 flash_callback_t flash_config_t::PFlashCallback

14.2.11.0.0.39.5 uint32_t flash_config_t::PFlashAccessSegmentSize

14.2.11.0.0.39.6 uint32_t flash_config_t::PFlashAccessSegmentCount

14.2.11.0.0.39.7 uint32_t* flash_config_t::flashExecuteInRamFunctionInfo

14.3 Macro Definition Documentation

14.3.1 #define MAKE_VERSION(*major*, *minor*, *bugfix*) (((major) << 16) | ((minor) << 8) | (bugfix))

14.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))

Version 2.2.0.

14.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

14.3.4 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

14.3.5 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

14.3.6 #define kStatusGroupGeneric 0

14.3.7 #define MAKE_STATUS(*group*, *code*) ((((group)*100) + (code)))

14.3.8 #define FOUR_CHAR_CODE(*a*, *b*, *c*, *d*) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))

14.4 Enumeration Type Documentation

14.4.1 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.

14.4.2 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.
kStatus_FLASH_ReadOnlyProperty The flash property is read-only.
kStatus_FLASH_InvalidPropertyValue The flash property value is out of range.
kStatus_FLASH_InvalidSpeculationOption The option of flash prefetch speculation is invalid.

14.4.3 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.

14.4.4 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.

14.4.5 enum flash_security_state_t

Enumerator

kFLASH_SecurityStateNotSecure Flash is not secure.
kFLASH_SecurityStateBackdoorEnabled Flash backdoor is enabled.
kFLASH_SecurityStateBackdoorDisabled Flash backdoor is disabled.

14.4.6 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.

14.4.7 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.

14.4.8 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 size property.
kFLASH_PropertyDflashBlockCount Dflash block count property.
kFLASH_PropertyDflashBlockBaseAddr Dflash block base address property.
kFLASH_PropertyEepromTotalSize EEPROM total size property.
kFLASH_PropertyFlashMemoryIndex Flash memory index property.

14.4.9 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.

14.4.10 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.

14.4.11 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_ResourceRangePflashSwapIfrStart Pflash swap IFR start address.

kFLASH_ResourceRangePflashSwapIfrEnd Pflash swap IFR end address.

kFLASH_ResourceRangeDflashIfrStart Dflash IFR start address.

kFLASH_ResourceRangeDflashIfrEnd Dflash IFR end address.

14.4.12 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.

14.4.13 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.

14.4.14 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.

14.4.15 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.

14.4.16 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.

14.4.17 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.

14.4.18 enum flash_memory_index_t

Enumerator

kFLASH_MemoryIndexPrimaryFlash Current flash memory is primary flash.

kFLASH_MemoryIndexSecondaryFlash Current flash memory is secondary flash.

14.5 Function Documentation

14.5.1 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_InvalidArgument	An invalid argument is provided.
kStatus_FLASH_ExecuteInRamFunctionNotReady	Execute-in-RAM function is not available.
kStatus_FLASH_PartitionStatusUpdateFailure	Failed to update the partition status.

14.5.2 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.

14.5.3 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.

14.5.4 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.

14.5.5 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_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-AlignmentError	The parameter is not aligned with the specified baseline.
kStatus_FLASH_Address-Error	The address is out of range.
kStatus_FLASH_Erase-KeyError	The 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 the command execution.

14.5.6 status_t FLASH_EraseAllUnsecure (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.

14.5.7 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_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 the command execution.

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

14.5.9 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.

14.5.10 status_t FLASH_ReadResource (flash_config_t * *config*, uint32_t *start*, uint32_t * *dst*, uint32_t *lengthInBytes*, flash_read_resource_option_t *option*)

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_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with the specified baseline.
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.

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

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

Parameters

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

Return values

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

14.5.12 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 backdoor key enabling state.

Parameters

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

Return values

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

14.5.13 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.

14.5.14 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.

14.5.15 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.

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

14.5.17 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_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.

14.5.18 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.

14.5.19 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.

14.5.20 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_InvalidArgument	An invalid argument is provided.

kStatus_FLASH_-UnknownProperty	An unknown property tag.

14.5.21 status_t FLASH_PflashSetProtection (flash_config_t ∗ *config*, pflash_protection_status_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(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_Invalid-Argument	An invalid argument is provided.
kStatus_FLASH_-CommandFailure	Run-time error during command execution.

14.5.22 status_t FLASH_PflashGetProtection (flash_config_t ∗ *config*, pflash_protection_status_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(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

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

Chapter 15
FlexBus: External Bus Interface Driver

15.1 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 number of chip selects available depends on the device and its pin configuration.

15.2 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.

15.3 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);
```

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, 1))
Version 2.0.1.

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.

15.4 Data Structure Documentation

15.4.1 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.

15.5 Macro Definition Documentation

15.5.1 #define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

15.6 Enumeration Type Documentation

15.6.1 enum flexbus_port_size_t

Enumerator

kFLEXBUS_4Bytes 32-bit port size
kFLEXBUS_1Byte 8-bit port size
kFLEXBUS_2Bytes 16-bit port size

15.6.2 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.

15.6.3 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.

15.6.4 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.

15.6.5 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

15.6.6 enum flexbus_multiplex_group1_t

Enumerator

kFLEXBUS_MultiplexGroup1_FB_ALE FB_ALE.
kFLEXBUS_MultiplexGroup1_FB_CS1 FB_CS1.
kFLEXBUS_MultiplexGroup1_FB_TS FB_TS.

15.6.7 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.

15.6.8 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.

15.6.9 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.

15.6.10 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.

15.7 Function Documentation

15.7.1 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 that in this function, certain parameters, depending on external memories, must be set before using the 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

15.7.2 void FLEXBUS_Deinit (FB_Type * *base*)

This function disables the clock gate of the FlexBus module clock.

Parameters

base	FlexBus peripheral address.

15.7.3 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 16
FlexIO: FlexIO Driver

16.1 Overview

The KSDK provides a generic driver and multiple protocol-specific FlexIO drivers for the FlexIO module of Kinetis devices.

Modules

- FlexIO Camera Driver
- FlexIO Driver
- FlexIO I2C Master Driver
- FlexIO I2S Driver
- FlexIO SPI Driver
- FlexIO UART Driver

16.2 FlexIO Driver

16.2.1 Overview

Data Structures

- struct flexio_config_t
  Define FlexIO user configuration structure. More...
- struct flexio_timer_config_t
  Define FlexIO timer configuration structure. More...
- struct flexio_shifter_config_t
  Define FlexIO shifter configuration structure. More...

Macros

- #define FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x) ((uint32_t)(x) << 1U)
  Calculate FlexIO timer trigger.

Typedefs

- typedef void(∗ flexio_isr_t)(void ∗base, void ∗handle)
  typedef for FlexIO simulated driver interrupt handler.

Enumerations

- enum flexio_timer_trigger_polarity_t {
  kFLEXIO_TimerTriggerPolarityActiveHigh = 0x0U,
  kFLEXIO_TimerTriggerPolarityActiveLow = 0x1U }
  Define time of timer trigger polarity.
- enum flexio_timer_trigger_source_t {
  kFLEXIO_TimerTriggerSourceExternal = 0x0U,
  kFLEXIO_TimerTriggerSourceInternal = 0x1U }
  Define type of timer trigger source.
- enum flexio_pin_config_t {
  kFLEXIO_PinConfigOutputDisabled = 0x0U,
  kFLEXIO_PinConfigOpenDrainOrBidirection = 0x1U,
  kFLEXIO_PinConfigBidirectionOutputData = 0x2U,
  kFLEXIO_PinConfigOutput = 0x3U }
  Define type of timer/shifter pin configuration.
- enum flexio_pin_polarity_t {
  kFLEXIO_PinActiveHigh = 0x0U,
  kFLEXIO_PinActiveLow = 0x1U }
  Definition of pin polarity.

- enum flexio_timer_mode_t {
  kFLEXIO_TimerModeDisabled = 0x0U,
  kFLEXIO_TimerModeDual8BitBaudBit = 0x1U,
  kFLEXIO_TimerModeDual8BitPWM = 0x2U,
  kFLEXIO_TimerModeSingle16Bit = 0x3U }
  Define type of timer work mode.
- enum flexio_timer_output_t {
  kFLEXIO_TimerOutputOneNotAffectedByReset = 0x0U,
  kFLEXIO_TimerOutputZeroNotAffectedByReset = 0x1U,
  kFLEXIO_TimerOutputOneAffectedByReset = 0x2U,
  kFLEXIO_TimerOutputZeroAffectedByReset = 0x3U }
  Define type of timer initial output or timer reset condition.
- enum flexio_timer_decrement_source_t {
  kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput = 0x0U,
  kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput = 0x1U,
  kFLEXIO_TimerDecSrcOnPinInputShiftPinInput = 0x2U,
  kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput = 0x3U }
  Define type of timer decrement.
- enum flexio_timer_reset_condition_t {
  kFLEXIO_TimerResetNever = 0x0U,
  kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput = 0x2U,
  kFLEXIO_TimerResetOnTimerTriggerEqualToTimerOutput = 0x3U,
  kFLEXIO_TimerResetOnTimerPinRisingEdge = 0x4U,
  kFLEXIO_TimerResetOnTimerTriggerRisingEdge = 0x6U,
  kFLEXIO_TimerResetOnTimerTriggerBothEdge = 0x7U }
  Define type of timer reset condition.
- enum flexio_timer_disable_condition_t {
  kFLEXIO_TimerDisableNever = 0x0U,
  kFLEXIO_TimerDisableOnPreTimerDisable = 0x1U,
  kFLEXIO_TimerDisableOnTimerCompare = 0x2U,
  kFLEXIO_TimerDisableOnTimerCompareTriggerLow = 0x3U,
  kFLEXIO_TimerDisableOnPinBothEdge = 0x4U,
  kFLEXIO_TimerDisableOnPinBothEdgeTriggerHigh = 0x5U,
  kFLEXIO_TimerDisableOnTriggerFallingEdge = 0x6U }
  Define type of timer disable condition.
- enum flexio_timer_enable_condition_t {
  kFLEXIO_TimerEnabledAlways = 0x0U,
  kFLEXIO_TimerEnableOnPrevTimerEnable = 0x1U,
  kFLEXIO_TimerEnableOnTriggerHigh = 0x2U,
  kFLEXIO_TimerEnableOnTriggerHighPinHigh = 0x3U,
  kFLEXIO_TimerEnableOnPinRisingEdge = 0x4U,
  kFLEXIO_TimerEnableOnPinRisingEdgeTriggerHigh = 0x5U,
  kFLEXIO_TimerEnableOnTriggerRisingEdge = 0x6U,
  kFLEXIO_TimerEnableOnTriggerBothEdge = 0x7U }
  Define type of timer enable condition.
- enum flexio_timer_stop_bit_condition_t {

kFLEXIO_TimerStopBitDisabled = 0x0U,
kFLEXIO_TimerStopBitEnableOnTimerCompare = 0x1U,
kFLEXIO_TimerStopBitEnableOnTimerDisable = 0x2U,
kFLEXIO_TimerStopBitEnableOnTimerCompareDisable = 0x3U }
Define type of timer stop bit generate condition.
- enum flexio_timer_start_bit_condition_t {
kFLEXIO_TimerStartBitDisabled = 0x0U,
kFLEXIO_TimerStartBitEnabled = 0x1U }
Define type of timer start bit generate condition.
- enum flexio_shifter_timer_polarity_t
Define type of timer polarity for shifter control.
- enum flexio_shifter_mode_t {
kFLEXIO_ShifterDisabled = 0x0U,
kFLEXIO_ShifterModeReceive = 0x1U,
kFLEXIO_ShifterModeTransmit = 0x2U,
kFLEXIO_ShifterModeMatchStore = 0x4U,
kFLEXIO_ShifterModeMatchContinuous = 0x5U,
kFLEXIO_ShifterModeState = 0x6U,
kFLEXIO_ShifterModeLogic = 0x7U }
Define type of shifter working mode.
- enum flexio_shifter_input_source_t {
kFLEXIO_ShifterInputFromPin = 0x0U,
kFLEXIO_ShifterInputFromNextShifterOutput = 0x1U }
Define type of shifter input source.
- enum flexio_shifter_stop_bit_t {
kFLEXIO_ShifterStopBitDisable = 0x0U,
kFLEXIO_ShifterStopBitLow = 0x2U,
kFLEXIO_ShifterStopBitHigh = 0x3U }
Define of STOP bit configuration.
- enum flexio_shifter_start_bit_t {
kFLEXIO_ShifterStartBitDisabledLoadDataOnEnable = 0x0U,
kFLEXIO_ShifterStartBitDisabledLoadDataOnShift = 0x1U,
kFLEXIO_ShifterStartBitLow = 0x2U,
kFLEXIO_ShifterStartBitHigh = 0x3U }
Define type of START bit configuration.
- enum flexio_shifter_buffer_type_t {
kFLEXIO_ShifterBuffer = 0x0U,
kFLEXIO_ShifterBufferBitSwapped = 0x1U,
kFLEXIO_ShifterBufferByteSwapped = 0x2U,
kFLEXIO_ShifterBufferBitByteSwapped = 0x3U,
kFLEXIO_ShifterBufferNibbleByteSwapped = 0x4U,
kFLEXIO_ShifterBufferHalfWordSwapped = 0x5U,
kFLEXIO_ShifterBufferNibbleSwapped = 0x6U }
Define FlexIO shifter buffer type.

Driver version

- #define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  FlexIO driver version 2.0.1.

FlexIO Initialization and De-initialization

- void FLEXIO_GetDefaultConfig (flexio_config_t ∗userConfig)
  Gets the default configuration to configure the FlexIO module.
- void FLEXIO_Init (FLEXIO_Type ∗base, const flexio_config_t ∗userConfig)
  Configures the FlexIO with a FlexIO configuration.
- void FLEXIO_Deinit (FLEXIO_Type ∗base)
  Gates the FlexIO clock.

FlexIO Basic Operation

- void FLEXIO_Reset (FLEXIO_Type ∗base)
  Resets the FlexIO module.
- static void FLEXIO_Enable (FLEXIO_Type ∗base, bool enable)
  Enables the FlexIO module operation.
- static uint32_t FLEXIO_ReadPinInput (FLEXIO_Type ∗base)
  Reads the input data on each of the FlexIO pins.
- static uint8_t FLEXIO_GetShifterState (FLEXIO_Type ∗base)
  Gets the current state pointer for state mode use.
- void FLEXIO_SetShifterConfig (FLEXIO_Type ∗base, uint8_t index, const flexio_shifter_config_t ∗shifterConfig)
  Configures the shifter with the shifter configuration.
- void FLEXIO_SetTimerConfig (FLEXIO_Type ∗base, uint8_t index, const flexio_timer_config_t ∗timerConfig)
  Configures the timer with the timer configuration.

FlexIO Interrupt Operation

- static void FLEXIO_EnableShifterStatusInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Enables the shifter status interrupt.
- static void FLEXIO_DisableShifterStatusInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Disables the shifter status interrupt.
- static void FLEXIO_EnableShifterErrorInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Enables the shifter error interrupt.
- static void FLEXIO_DisableShifterErrorInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Disables the shifter error interrupt.
- static void FLEXIO_EnableTimerStatusInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Enables the timer status interrupt.
- static void FLEXIO_DisableTimerStatusInterrupts (FLEXIO_Type ∗base, uint32_t mask)
  Disables the timer status interrupt.

FlexIO Status Operation

- static uint32_t FLEXIO_GetShifterStatusFlags (FLEXIO_Type ∗base)
  Gets the shifter status flags.
- static void FLEXIO_ClearShifterStatusFlags (FLEXIO_Type ∗base, uint32_t mask)
  Clears the shifter status flags.
- static uint32_t FLEXIO_GetShifterErrorFlags (FLEXIO_Type ∗base)
  Gets the shifter error flags.
- static void FLEXIO_ClearShifterErrorFlags (FLEXIO_Type ∗base, uint32_t mask)
  Clears the shifter error flags.
- static uint32_t FLEXIO_GetTimerStatusFlags (FLEXIO_Type ∗base)
  Gets the timer status flags.
- static void FLEXIO_ClearTimerStatusFlags (FLEXIO_Type ∗base, uint32_t mask)
  Clears the timer status flags.

FlexIO DMA Operation

- static void FLEXIO_EnableShifterStatusDMA (FLEXIO_Type ∗base, uint32_t mask, bool enable)
  Enables/disables the shifter status DMA.
- uint32_t FLEXIO_GetShifterBufferAddress (FLEXIO_Type ∗base, flexio_shifter_buffer_type_t type, uint8_t index)
  Gets the shifter buffer address for the DMA transfer usage.
- status_t FLEXIO_RegisterHandleIRQ (void ∗base, void ∗handle, flexio_isr_t isr)
  Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.
- status_t FLEXIO_UnregisterHandleIRQ (void ∗base)
  Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

16.2.2 Data Structure Documentation

16.2.2.1 struct flexio_config_t

Data Fields

- bool enableFlexio
  Enable/disable FlexIO module.
- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  `Enable/disable fast access to FlexIO registers, fast access requires` *the FlexIO clock to be at least twice the frequency of the bus clock.*

16.2.2.1.0.40 Field Documentation

16.2.2.1.0.40.1 bool flexio_config_t::enableFastAccess

16.2.2.2 struct flexio_timer_config_t

Data Fields

- uint32_t triggerSelect
  The internal trigger selection number using MACROs.
- flexio_timer_trigger_polarity_t triggerPolarity
  Trigger Polarity.
- flexio_timer_trigger_source_t triggerSource
  Trigger Source, internal (see 'trgsel') or external.
- flexio_pin_config_t pinConfig
  Timer Pin Configuration.
- uint32_t pinSelect
  Timer Pin number Select.
- flexio_pin_polarity_t pinPolarity
  Timer Pin Polarity.
- flexio_timer_mode_t timerMode
  Timer work Mode.
- flexio_timer_output_t timerOutput
  `Configures the initial state of the Timer Output and`
  whether it is affected by the Timer reset.
- flexio_timer_decrement_source_t timerDecrement
  `Configures the source of the Timer decrement and the`
  source of the Shift clock.
- flexio_timer_reset_condition_t timerReset
  `Configures the condition that causes the timer counter`
  (and optionally the timer output) to be reset.
- flexio_timer_disable_condition_t timerDisable
  `Configures the condition that causes the Timer to be`
  disabled and stop decrementing.
- flexio_timer_enable_condition_t timerEnable
  `Configures the condition that causes the Timer to be`
  enabled and start decrementing.
- flexio_timer_stop_bit_condition_t timerStop
  Timer STOP Bit generation.
- flexio_timer_start_bit_condition_t timerStart
  Timer STRAT Bit generation.
- uint32_t timerCompare
  Value for Timer Compare N Register.

16.2.2.2.0.41 Field Documentation

16.2.2.2.0.41.1 uint32_t flexio_timer_config_t::triggerSelect

16.2.2.2.0.41.2 flexio_timer_trigger_polarity_t flexio_timer_config_t::triggerPolarity

16.2.2.2.0.41.3 flexio_timer_trigger_source_t flexio_timer_config_t::triggerSource

16.2.2.2.0.41.4 flexio_pin_config_t flexio_timer_config_t::pinConfig

16.2.2.2.0.41.5 uint32_t flexio_timer_config_t::pinSelect

16.2.2.2.0.41.6 flexio_pin_polarity_t flexio_timer_config_t::pinPolarity

16.2.2.2.0.41.7 flexio_timer_mode_t flexio_timer_config_t::timerMode

16.2.2.2.0.41.8 flexio_timer_output_t flexio_timer_config_t::timerOutput

16.2.2.2.0.41.9 flexio_timer_decrement_source_t flexio_timer_config_t::timerDecrement

16.2.2.2.0.41.10 flexio_timer_reset_condition_t flexio_timer_config_t::timerReset

16.2.2.2.0.41.11 flexio_timer_disable_condition_t flexio_timer_config_t::timerDisable

16.2.2.2.0.41.12 flexio_timer_enable_condition_t flexio_timer_config_t::timerEnable

16.2.2.2.0.41.13 flexio_timer_stop_bit_condition_t flexio_timer_config_t::timerStop

16.2.2.2.0.41.14 flexio_timer_start_bit_condition_t flexio_timer_config_t::timerStart

16.2.2.2.0.41.15 uint32_t flexio_timer_config_t::timerCompare

16.2.2.3 struct flexio_shifter_config_t

Data Fields

- uint32_t timerSelect
  `Selects which Timer is used for controlling the`
  logic/shift register and generating the Shift clock.
- flexio_shifter_timer_polarity_t timerPolarity
  Timer Polarity.
- flexio_pin_config_t pinConfig
  Shifter Pin Configuration.
- uint32_t pinSelect
  Shifter Pin number Select.
- flexio_pin_polarity_t pinPolarity
  Shifter Pin Polarity.
- flexio_shifter_mode_t shifterMode
  Configures the mode of the Shifter.
- uint32_t parallelWidth
  Configures the parallel width when using parallel mode.

- flexio_shifter_input_source_t inputSource
  Selects the input source for the shifter.
- flexio_shifter_stop_bit_t shifterStop
  Shifter STOP bit.
- flexio_shifter_start_bit_t shifterStart
  Shifter START bit.

16.2.2.3.0.42 Field Documentation

16.2.2.3.0.42.1 uint32_t flexio_shifter_config_t::timerSelect

16.2.2.3.0.42.2 flexio_shifter_timer_polarity_t flexio_shifter_config_t::timerPolarity

16.2.2.3.0.42.3 flexio_pin_config_t flexio_shifter_config_t::pinConfig

16.2.2.3.0.42.4 uint32_t flexio_shifter_config_t::pinSelect

16.2.2.3.0.42.5 flexio_pin_polarity_t flexio_shifter_config_t::pinPolarity

16.2.2.3.0.42.6 flexio_shifter_mode_t flexio_shifter_config_t::shifterMode

16.2.2.3.0.42.7 uint32_t flexio_shifter_config_t::parallelWidth

16.2.2.3.0.42.8 flexio_shifter_input_source_t flexio_shifter_config_t::inputSource

16.2.2.3.0.42.9 flexio_shifter_stop_bit_t flexio_shifter_config_t::shifterStop

16.2.2.3.0.42.10 flexio_shifter_start_bit_t flexio_shifter_config_t::shifterStart

16.2.3 Macro Definition Documentation

16.2.3.1 #define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

16.2.3.2 #define FLEXIO_TIMER_TRIGGER_SEL_PININPUT(*x*) ((uint32_t)(x) << 1U)

16.2.4 Typedef Documentation

16.2.4.1 typedef void(∗ flexio_isr_t)(void ∗base, void ∗handle)

16.2.5 Enumeration Type Documentation

16.2.5.1 enum flexio_timer_trigger_polarity_t

Enumerator

kFLEXIO_TimerTriggerPolarityActiveHigh Active high.
kFLEXIO_TimerTriggerPolarityActiveLow Active low.

16.2.5.2 enum flexio_timer_trigger_source_t

Enumerator

kFLEXIO_TimerTriggerSourceExternal External trigger selected.
kFLEXIO_TimerTriggerSourceInternal Internal trigger selected.

16.2.5.3 enum flexio_pin_config_t

Enumerator

kFLEXIO_PinConfigOutputDisabled Pin output disabled.
kFLEXIO_PinConfigOpenDrainOrBidirection Pin open drain or bidirectional output enable.
kFLEXIO_PinConfigBidirectionOutputData Pin bidirectional output data.
kFLEXIO_PinConfigOutput Pin output.

16.2.5.4 enum flexio_pin_polarity_t

Enumerator

kFLEXIO_PinActiveHigh Active high.
kFLEXIO_PinActiveLow Active low.

16.2.5.5 enum flexio_timer_mode_t

Enumerator

kFLEXIO_TimerModeDisabled Timer Disabled.
kFLEXIO_TimerModeDual8BitBaudBit Dual 8-bit counters baud/bit mode.
kFLEXIO_TimerModeDual8BitPWM Dual 8-bit counters PWM mode.
kFLEXIO_TimerModeSingle16Bit Single 16-bit counter mode.

16.2.5.6 enum flexio_timer_output_t

Enumerator

kFLEXIO_TimerOutputOneNotAffectedByReset Logic one when enabled and is not affected by timer reset.
kFLEXIO_TimerOutputZeroNotAffectedByReset Logic zero when enabled and is not affected by timer reset.
kFLEXIO_TimerOutputOneAffectedByReset Logic one when enabled and on timer reset.
kFLEXIO_TimerOutputZeroAffectedByReset Logic zero when enabled and on timer reset.

16.2.5.7 enum flexio_timer_decrement_source_t

Enumerator

kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput Decrement counter on FlexIO clock, Shift clock equals Timer output.
kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput Decrement counter on Trigger input (both edges), Shift clock equals Timer output.
kFLEXIO_TimerDecSrcOnPinInputShiftPinInput Decrement counter on Pin input (both edges), Shift clock equals Pin input.
kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

16.2.5.8 enum flexio_timer_reset_condition_t

Enumerator

kFLEXIO_TimerResetNever Timer never reset.
kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput Timer reset on Timer Pin equal to Timer Output.
kFLEXIO_TimerResetOnTimerTriggerEqualToTimerOutput Timer reset on Timer Trigger equal to Timer Output.
kFLEXIO_TimerResetOnTimerPinRisingEdge Timer reset on Timer Pin rising edge.
kFLEXIO_TimerResetOnTimerTriggerRisingEdge Timer reset on Trigger rising edge.
kFLEXIO_TimerResetOnTimerTriggerBothEdge Timer reset on Trigger rising or falling edge.

16.2.5.9 enum flexio_timer_disable_condition_t

Enumerator

kFLEXIO_TimerDisableNever Timer never disabled.
kFLEXIO_TimerDisableOnPreTimerDisable Timer disabled on Timer N-1 disable.
kFLEXIO_TimerDisableOnTimerCompare Timer disabled on Timer compare.
kFLEXIO_TimerDisableOnTimerCompareTriggerLow Timer disabled on Timer compare and Trigger Low.
kFLEXIO_TimerDisableOnPinBothEdge Timer disabled on Pin rising or falling edge.
kFLEXIO_TimerDisableOnPinBothEdgeTriggerHigh Timer disabled on Pin rising or falling edge provided Trigger is high.
kFLEXIO_TimerDisableOnTriggerFallingEdge Timer disabled on Trigger falling edge.

16.2.5.10 enum flexio_timer_enable_condition_t

Enumerator

kFLEXIO_TimerEnabledAlways Timer always enabled.

kFLEXIO_TimerEnableOnPrevTimerEnable Timer enabled on Timer N-1 enable.
kFLEXIO_TimerEnableOnTriggerHigh Timer enabled on Trigger high.
kFLEXIO_TimerEnableOnTriggerHighPinHigh Timer enabled on Trigger high and Pin high.
kFLEXIO_TimerEnableOnPinRisingEdge Timer enabled on Pin rising edge.
kFLEXIO_TimerEnableOnPinRisingEdgeTriggerHigh Timer enabled on Pin rising edge and Trigger high.
kFLEXIO_TimerEnableOnTriggerRisingEdge Timer enabled on Trigger rising edge.
kFLEXIO_TimerEnableOnTriggerBothEdge Timer enabled on Trigger rising or falling edge.

16.2.5.11 enum flexio_timer_stop_bit_condition_t

Enumerator

kFLEXIO_TimerStopBitDisabled Stop bit disabled.
kFLEXIO_TimerStopBitEnableOnTimerCompare Stop bit is enabled on timer compare.
kFLEXIO_TimerStopBitEnableOnTimerDisable Stop bit is enabled on timer disable.
kFLEXIO_TimerStopBitEnableOnTimerCompareDisable Stop bit is enabled on timer compare and timer disable.

16.2.5.12 enum flexio_timer_start_bit_condition_t

Enumerator

kFLEXIO_TimerStartBitDisabled Start bit disabled.
kFLEXIO_TimerStartBitEnabled Start bit enabled.

16.2.5.13 enum flexio_shifter_timer_polarity_t

16.2.5.14 enum flexio_shifter_mode_t

Enumerator

kFLEXIO_ShifterDisabled Shifter is disabled.
kFLEXIO_ShifterModeReceive Receive mode.
kFLEXIO_ShifterModeTransmit Transmit mode.
kFLEXIO_ShifterModeMatchStore Match store mode.
kFLEXIO_ShifterModeMatchContinuous Match continuous mode.
kFLEXIO_ShifterModeState SHIFTBUF contents are used for storing programmable state attributes.
kFLEXIO_ShifterModeLogic SHIFTBUF contents are used for implementing programmable logic look up table.

16.2.5.15 enum flexio_shifter_input_source_t

Enumerator

kFLEXIO_ShifterInputFromPin Shifter input from pin.
kFLEXIO_ShifterInputFromNextShifterOutput Shifter input from Shifter N+1.

16.2.5.16 enum flexio_shifter_stop_bit_t

Enumerator

kFLEXIO_ShifterStopBitDisable Disable shifter stop bit.
kFLEXIO_ShifterStopBitLow Set shifter stop bit to logic low level.
kFLEXIO_ShifterStopBitHigh Set shifter stop bit to logic high level.

16.2.5.17 enum flexio_shifter_start_bit_t

Enumerator

kFLEXIO_ShifterStartBitDisabledLoadDataOnEnable Disable shifter start bit, transmitter loads data on enable.
kFLEXIO_ShifterStartBitDisabledLoadDataOnShift Disable shifter start bit, transmitter loads data on first shift.
kFLEXIO_ShifterStartBitLow Set shifter start bit to logic low level.
kFLEXIO_ShifterStartBitHigh Set shifter start bit to logic high level.

16.2.5.18 enum flexio_shifter_buffer_type_t

Enumerator

kFLEXIO_ShifterBuffer Shifter Buffer N Register.
kFLEXIO_ShifterBufferBitSwapped Shifter Buffer N Bit Byte Swapped Register.
kFLEXIO_ShifterBufferByteSwapped Shifter Buffer N Byte Swapped Register.
kFLEXIO_ShifterBufferBitByteSwapped Shifter Buffer N Bit Swapped Register.
kFLEXIO_ShifterBufferNibbleByteSwapped Shifter Buffer N Nibble Byte Swapped Register.
kFLEXIO_ShifterBufferHalfWordSwapped Shifter Buffer N Half Word Swapped Register.
kFLEXIO_ShifterBufferNibbleSwapped Shifter Buffer N Nibble Swapped Register.

16.2.6 Function Documentation

16.2.6.1 void FLEXIO_GetDefaultConfig (flexio_config_t * *userConfig*)

The configuration can used directly to call the FLEXIO_Configure().

Example:

```
flexio_config_t config;
FLEXIO_GetDefaultConfig(&config);
```

Parameters

userConfig	pointer to flexio_config_t structure

16.2.6.2 void FLEXIO_Init (FLEXIO_Type * *base*, const flexio_config_t * *userConfig*)

The configuration structure can be filled by the user or be set with default values by FLEXIO_GetDefault-Config().

Example

```
flexio_config_t config = {
.enableFlexio = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false
};
FLEXIO_Configure(base, &config);
```

Parameters

base	FlexIO peripheral base address
userConfig	pointer to flexio_config_t structure

16.2.6.3 void FLEXIO_Deinit (FLEXIO_Type * *base*)

Call this API to stop the FlexIO clock.

Note

After calling this API, call the FLEXO_Init to use the FlexIO module.

Parameters

base	FlexIO peripheral base address

16.2.6.4 void FLEXIO_Reset (FLEXIO_Type * *base*)

Parameters

base	FlexIO peripheral base address

16.2.6.5 static void FLEXIO_Enable (FLEXIO_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	FlexIO peripheral base address
enable	true to enable, false to disable.

16.2.6.6 static uint32_t FLEXIO_ReadPinInput (FLEXIO_Type ∗ *base*) [inline], [static]

Parameters

base	FlexIO peripheral base address

Returns

FlexIO pin input data

16.2.6.7 static uint8_t FLEXIO_GetShifterState (FLEXIO_Type ∗ *base*) [inline], [static]

Parameters

base	FlexIO peripheral base address

Returns

current State pointer

16.2.6.8 void FLEXIO_SetShifterConfig (FLEXIO_Type ∗ *base*, uint8_t *index*, const flexio_shifter_config_t ∗ *shifterConfig*)

The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {
.timerSelect = 0,
.timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
.pinPolarity = kFLEXIO_PinActiveLow,
.shifterMode = kFLEXIO_ShifterModeTransmit,
.inputSource = kFLEXIO_ShifterInputFromPin,
.shifterStop = kFLEXIO_ShifterStopBitHigh,
.shifterStart = kFLEXIO_ShifterStartBitLow
};
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

base	FlexIO peripheral base address
index	Shifter index
shifterConfig	Pointer to flexio_shifter_config_t structure

16.2.6.9 void FLEXIO_SetTimerConfig (FLEXIO_Type ∗ *base,* uint8_t *index,* const flexio_timer_config_t ∗ *timerConfig*)

The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

Example

```
flexio_timer_config_t config = {
.triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(0),
.triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,
.triggerSource = kFLEXIO_TimerTriggerSourceInternal,
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
.pinSelect = 0,
.pinPolarity = kFLEXIO_PinActiveHigh,
.timerMode = kFLEXIO_TimerModeDual8BitBaudBit,
.timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,
.timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput
      ,
.timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,
.timerDisable = kFLEXIO_TimerDisableOnTimerCompare,
.timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,
.timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,
.timerStart = kFLEXIO_TimerStartBitEnabled
};
FLEXIO_SetTimerConfig(base, &config);
```

Parameters

base	FlexIO peripheral base address
index	Timer index
timerConfig	Pointer to the flexio_timer_config_t structure

16.2.6.10 static void FLEXIO_EnableShifterStatusInterrupts (FLEXIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

The interrupt generates when the corresponding SSF is set.

Parameters

base	FlexIO peripheral base address
mask	The shifter status mask which can be calculated by (1 << shifter index)

Note

For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 << shifter index0) \| (1 << shifter index1))

16.2.6.11 static void FLEXIO_DisableShifterStatusInterrupts (FLEXIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SSF is set.

Parameters

base	FlexIO peripheral base address
mask	The shifter status mask which can be calculated by (1 << shifter index)

Note

For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using ((1 << shifter index0) \| (1 << shifter index1))

16.2.6.12 static void FLEXIO_EnableShifterErrorInterrupts (FLEXIO_Type ∗ *base*, uint32_t *mask*) [inline], [static]

The interrupt generates when the corresponding SEF is set.

Parameters

base	FlexIO peripheral base address
mask	The shifter error mask which can be calculated by (1 << shifter index)

Note

For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 << shifter index0) | (1 << shifter index1))

16.2.6.13 static void FLEXIO_DisableShifterErrorInterrupts (FLEXIO_Type * *base,* uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SEF is set.

Parameters

base	FlexIO peripheral base address
mask	The shifter error mask which can be calculated by (1 << shifter index)

Note

For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using ((1 << shifter index0) | (1 << shifter index1))

16.2.6.14 static void FLEXIO_EnableTimerStatusInterrupts (FLEXIO_Type * *base,* uint32_t *mask*) [inline], [static]

The interrupt generates when the corresponding SSF is set.

Parameters

base	FlexIO peripheral base address
mask	The timer status mask which can be calculated by (1 << timer index)

Note

For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using ((1 << timer index0) | (1 << timer index1))

16.2.6.15 static void FLEXIO_DisableTimerStatusInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SSF is set.

Parameters

base	FlexIO peripheral base address
mask	The timer status mask which can be calculated by (1 << timer index)

Note

For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using ((1 << timer index0) | (1 << timer index1))

16.2.6.16 static uint32_t FLEXIO_GetShifterStatusFlags (FLEXIO_Type * *base*) [inline], [static]

Parameters

base	FlexIO peripheral base address

Returns

Shifter status flags

16.2.6.17 static void FLEXIO_ClearShifterStatusFlags (FLEXIO_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	FlexIO peripheral base address
mask	The shifter status mask which can be calculated by (1 << shifter index)

Note

For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using ((1 << shifter index0) | (1 << shifter index1))

16.2.6.18 static uint32_t FLEXIO_GetShifterErrorFlags (FLEXIO_Type * *base*) [inline], [static]

Parameters

base	FlexIO peripheral base address

Returns

Shifter error flags

16.2.6.19 static void FLEXIO_ClearShifterErrorFlags (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	FlexIO peripheral base address
mask	The shifter error mask which can be calculated by (1 << shifter index)

Note

For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using ((1 << shifter index0) \| (1 << shifter index1))

16.2.6.20 static uint32_t FLEXIO_GetTimerStatusFlags (FLEXIO_Type * *base*) [inline], [static]

Parameters

base	FlexIO peripheral base address

Returns

Timer status flags

16.2.6.21 static void FLEXIO_ClearTimerStatusFlags (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	FlexIO peripheral base address
mask	The timer status mask which can be calculated by (1 << timer index)

Note

For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using ((1 << timer index0) \| (1 << timer index1))

16.2.6.22 static void FLEXIO_EnableShifterStatusDMA (FLEXIO_Type * *base,* uint32_t *mask,* bool *enable*) [inline], [static]

The DMA request generates when the corresponding SSF is set.

Note

For multiple shifter status DMA enables, for example, calculate the mask by using ((1 << shifter index0) | (1 << shifter index1))

Parameters

base	FlexIO peripheral base address
mask	The shifter status mask which can be calculated by (1 << shifter index)
enable	True to enable, false to disable.

16.2.6.23 uint32_t FLEXIO_GetShifterBufferAddress (FLEXIO_Type * *base,* flexio_shifter_buffer_type_t *type,* uint8_t *index*)

Parameters

base	FlexIO peripheral base address
type	Shifter type of flexio_shifter_buffer_type_t
index	Shifter index

Returns

Corresponding shifter buffer index

16.2.6.24 status_t FLEXIO_RegisterHandleIRQ (void * *base,* void * *handle,* flexio_isr_t *isr*)

Parameters

base	Pointer to the FlexIO simulated peripheral type.
handle	Pointer to the handler for FlexIO simulated peripheral.
isr	FlexIO simulated peripheral interrupt handler.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/ISR table out of range.

16.2.6.25 status_t FLEXIO_UnregisterHandleIRQ (void * *base*)

Parameters

base	Pointer to the FlexIO simulated peripheral type.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/ISR table out of range.

16.3 FlexIO Camera Driver

16.3.1 Overview

The KSDK provides driver for the camera function using Flexible I/O.

FlexIO Camera driver includes functional APIs and eDMA transactional APIs. Functional APIs target low level APIs. Users can use functional APIs for FlexIO Camera initialization/configuration/operation purpose. Using the functional API requires knowledge of the FlexIO Camera peripheral and how to organize functional APIs to meet the requirements of the application. All functional API use the FLEXIO_CAMERA_Type ∗ as the first parameter. FlexIO Camera functional operation groups provide the functional APIs set.

eDMA transactional APIs target high-level APIs. Users can use the transactional API to enable the peripheral quickly and can also use in the application if the code size and performance of transactional APIs satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the flexio_camera_edma_handle_t as the second parameter. Users need to initialize the handle by calling the FLEXIO_CAMERA_TransferCreateHandleEDMA() API.

eDMA transactional APIs support asynchronous receive. This means that the functions FLEXIO_CAMERA_TransferReceiveEDMA() set up an interrupt for data receive. When the receive is complete, the upper layer is notified through a callback function with the status kStatus_FLEXIO_CAMERA_RxIdle.

16.3.2 Typical use case

16.3.2.1 FlexIO Camera Receive using eDMA method

```
volatile uint32_t isEDMAGetOnePictureFinish = false;
edma_handle_t g_edmaHandle;
flexio_camera_edma_handle_t g_cameraEdmaHandle;
edma_config_t edmaConfig;
FLEXIO_CAMERA_Type g_FlexioCameraDevice = {.flexioBase = FLEXIO0,
                                           .datPinStartIdx = 24U, /* fxio_pin 24 -31 are used. */
                                           .pclkPinIdx = 1U,      /* fxio_pin 1 is used as pclk pin. */
                                           .hrefPinIdx = 18U,     /* flexio_pin 18 is used as href pin. */
                                           .shifterStartIdx = 0U, /* Shifter 0 = 7 are used. */
                                           .shifterCount = 8U,
                                           .timerIdx = 0U};
flexio_camera_config_t cameraConfig;

/* Configure DMAMUX */
DMAMUX_Init(DMAMUX0);
/* Configure DMA */
EDMA_GetDefaultConfig(&edmaConfig);
EDMA_Init(DMA0, &edmaConfig);

DMAMUX_SetSource(DMAMUX0, DMA_CHN_FLEXIO_TO_FRAMEBUFF, (g_FlexioCameraDevice.
      shifterStartIdx + 1U));
DMAMUX_EnableChannel(DMAMUX0, DMA_CHN_FLEXIO_TO_FRAMEBUFF);
EDMA_CreateHandle(&g_edmaHandle, DMA0, DMA_CHN_FLEXIO_TO_FRAMEBUFF);

FLEXIO_CAMERA_GetDefaultConfig(&cameraConfig);
FLEXIO_CAMERA_Init(&g_FlexioCameraDevice, &cameraConfig);
/* Clear all the flag. */
```

```
FLEXIO_CAMERA_ClearStatusFlags(&g_FlexioCameraDevice,
                               kFLEXIO_CAMERA_RxDataRegFullFlag |
      kFLEXIO_CAMERA_RxErrorFlag);
FLEXIO_ClearTimerStatusFlags(FLEXIO0, 0xFF);
FLEXIO_CAMERA_TransferCreateHandleEDMA(&g_FlexioCameraDevice, &
      g_cameraEdmaHandle, FLEXIO_CAMERA_UserCallback, NULL,
                               &g_edmaHandle);
cameraTransfer.dataAddress = (uint32_t)u16CameraFrameBuffer;
cameraTransfer.dataNum = sizeof(u16CameraFrameBuffer);
FLEXIO_CAMERA_TransferReceiveEDMA(&g_FlexioCameraDevice, &
      g_cameraEdmaHandle, &cameraTransfer);
while (!(isEDMAGetOnePictureFinish))
{
    ;
}

/* A callback function is also needed */
void FLEXIO_CAMERA_UserCallback(FLEXIO_CAMERA_Type *base,
                            flexio_camera_edma_handle_t *handle,
                            status_t status,
                            void *userData)
{
    userData = userData;
    /* eDMA Transfer finished */
    if (kStatus_FLEXIO_CAMERA_RxIdle == status)
    {
        isEDMAGetOnePictureFinish = true;
    }
}
```

Modules

- FlexIO eDMA Camera Driver

Data Structures

- struct FLEXIO_CAMERA_Type
  Define structure of configuring the FlexIO Camera device. More...
- struct flexio_camera_config_t
  Define FlexIO Camera user configuration structure. More...
- struct flexio_camera_transfer_t
  Define FlexIO Camera transfer structure. More...

Macros

- #define FLEXIO_CAMERA_PARALLEL_DATA_WIDTH (8U)
  Define the Camera CPI interface is constantly 8-bit width.

Enumerations

- enum _flexio_camera_status {
  kStatus_FLEXIO_CAMERA_RxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_CAMERA,

0),
kStatus_FLEXIO_CAMERA_RxIdle = MAKE_STATUS(kStatusGroup_FLEXIO_CAMERA, 1)
}
Error codes for the Camera driver.
- enum _flexio_camera_status_flags {
kFLEXIO_CAMERA_RxDataRegFullFlag = 0x1U,
kFLEXIO_CAMERA_RxErrorFlag = 0x2U }
Define FlexIO Camera status mask.

Driver version

- #define FSL_FLEXIO_CAMERA_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
FlexIO Camera driver version 2.1.0.

Initialization and configuration

- void FLEXIO_CAMERA_Init (FLEXIO_CAMERA_Type ∗base, const flexio_camera_config_t ∗config)
Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO Camera.
- void FLEXIO_CAMERA_Deinit (FLEXIO_CAMERA_Type ∗base)
Disables the FlexIO Camera and gates the FlexIO clock.
- void FLEXIO_CAMERA_GetDefaultConfig (flexio_camera_config_t ∗config)
Gets the default configuration to configure the FlexIO Camera.
- static void FLEXIO_CAMERA_Enable (FLEXIO_CAMERA_Type ∗base, bool enable)
Enables/disables the FlexIO Camera module operation.

Status

- uint32_t FLEXIO_CAMERA_GetStatusFlags (FLEXIO_CAMERA_Type ∗base)
Gets the FlexIO Camera status flags.
- void FLEXIO_CAMERA_ClearStatusFlags (FLEXIO_CAMERA_Type ∗base, uint32_t mask)
Clears the receive buffer full flag manually.

Interrupts

- void FLEXIO_CAMERA_EnableInterrupt (FLEXIO_CAMERA_Type ∗base)
Switches on the interrupt for receive buffer full event.
- void FLEXIO_CAMERA_DisableInterrupt (FLEXIO_CAMERA_Type ∗base)
Switches off the interrupt for receive buffer full event.

DMA support

- static void FLEXIO_CAMERA_EnableRxDMA (FLEXIO_CAMERA_Type ∗base, bool enable)

Enables/disables the FlexIO Camera receive DMA.
- static uint32_t FLEXIO_CAMERA_GetRxBufferAddress (FLEXIO_CAMERA_Type ∗base)
  Gets the data from the receive buffer.

16.3.3 Data Structure Documentation

16.3.3.1 struct FLEXIO_CAMERA_Type

Data Fields

- FLEXIO_Type ∗ flexioBase
  FlexIO module base address.
- uint32_t datPinStartIdx
  First data pin (D0) index for flexio_camera.
- uint32_t pclkPinIdx
  Pixel clock pin (PCLK) index for flexio_camera.
- uint32_t hrefPinIdx
  Horizontal sync pin (HREF) index for flexio_camera.
- uint32_t shifterStartIdx
  First shifter index used for flexio_camera data FIFO.
- uint32_t shifterCount
  The count of shifters that are used as flexio_camera data FIFO.
- uint32_t timerIdx
  Timer index used for flexio_camera in FlexIO.

16.3.3.1.0.43 Field Documentation

16.3.3.1.0.43.1 FLEXIO_Type∗ FLEXIO_CAMERA_Type::flexioBase

16.3.3.1.0.43.2 uint32_t FLEXIO_CAMERA_Type::datPinStartIdx

Then the successive following FLEXIO_CAMERA_DATA_WIDTH-1 pins are used as D1-D7.

16.3.3.1.0.43.3 uint32_t FLEXIO_CAMERA_Type::pclkPinIdx

16.3.3.1.0.43.4 uint32_t FLEXIO_CAMERA_Type::hrefPinIdx

16.3.3.1.0.43.5 uint32_t FLEXIO_CAMERA_Type::shifterStartIdx

16.3.3.1.0.43.6 uint32_t FLEXIO_CAMERA_Type::shifterCount

16.3.3.1.0.43.7 uint32_t FLEXIO_CAMERA_Type::timerIdx

16.3.3.2 struct flexio_camera_config_t

Data Fields

- bool enablecamera
  Enable/disable FlexIO Camera TX & RX.

- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  `Enable/disable fast access to FlexIO registers,`
  fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

16.3.3.2.0.44 Field Documentation

16.3.3.2.0.44.1 bool flexio_camera_config_t::enablecamera

16.3.3.2.0.44.2 bool flexio_camera_config_t::enableFastAccess

16.3.3.3 struct flexio_camera_transfer_t

Data Fields

- uint32_t dataAddress
  Transfer buffer.
- uint32_t dataNum
  Transfer num.

16.3.4 Macro Definition Documentation

16.3.4.1 #define FSL_FLEXIO_CAMERA_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

16.3.4.2 #define FLEXIO_CAMERA_PARALLEL_DATA_WIDTH (8U)

16.3.5 Enumeration Type Documentation

16.3.5.1 enum _flexio_camera_status

Enumerator

kStatus_FLEXIO_CAMERA_RxBusy Receiver is busy.
kStatus_FLEXIO_CAMERA_RxIdle Camera receiver is idle.

16.3.5.2 enum _flexio_camera_status_flags

Enumerator

kFLEXIO_CAMERA_RxDataRegFullFlag Receive buffer full flag.
kFLEXIO_CAMERA_RxErrorFlag Receive buffer error flag.

16.3.6 Function Documentation

16.3.6.1 void FLEXIO_CAMERA_Init (FLEXIO_CAMERA_Type * *base*, const flexio_camera_config_t * *config*)

Parameters

base	Pointer to FLEXIO_CAMERA_Type structure
config	Pointer to flexio_camera_config_t structure

16.3.6.2 void FLEXIO_CAMERA_Deinit (FLEXIO_CAMERA_Type * *base*)

Note

After calling this API, call FLEXO_CAMERA_Init to use the FlexIO Camera module.

Parameters

base	Pointer to FLEXIO_CAMERA_Type structure

16.3.6.3 void FLEXIO_CAMERA_GetDefaultConfig (flexio_camera_config_t * *config*)

The configuration can be used directly for calling the FLEXIO_CAMERA_Init(). Example:

```
flexio_camera_config_t config;
FLEXIO_CAMERA_GetDefaultConfig(&userConfig);
```

Parameters

config	Pointer to the flexio_camera_config_t structure

16.3.6.4 static void FLEXIO_CAMERA_Enable (FLEXIO_CAMERA_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to the FLEXIO_CAMERA_Type
enable	True to enable, false to disable.

16.3.6.5 uint32_t FLEXIO_CAMERA_GetStatusFlags (FLEXIO_CAMERA_Type * *base*)

Parameters

base	Pointer to FLEXIO_CAMERA_Type structure

Returns

FlexIO shifter status flags
- FLEXIO_SHIFTSTAT_SSF_MASK
- 0

16.3.6.6 void FLEXIO_CAMERA_ClearStatusFlags (FLEXIO_CAMERA_Type ∗ *base*, uint32_t *mask*)

Parameters

base	Pointer to the device.
mask	status flag The parameter can be any combination of the following values: • kFLEXIO_CAMERA_RxDataRegFullFlag • kFLEXIO_CAMERA_RxErrorFlag

16.3.6.7 void FLEXIO_CAMERA_EnableInterrupt (FLEXIO_CAMERA_Type ∗ *base*)

Parameters

base	Pointer to the device.

16.3.6.8 void FLEXIO_CAMERA_DisableInterrupt (FLEXIO_CAMERA_Type ∗ *base*)

Parameters

base	Pointer to the device.

16.3.6.9 static void FLEXIO_CAMERA_EnableRxDMA (FLEXIO_CAMERA_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to FLEXIO_CAMERA_Type structure
enable	True to enable, false to disable.

The FlexIO Camera mode can't work without the DMA or eDMA support, Usually, it needs at least two DMA or eDMA channels, one for transferring data from Camera, such as 0V7670 to FlexIO buffer, another is for transferring data from FlexIO buffer to LCD.

16.3.6.10 static uint32_t FLEXIO_CAMERA_GetRxBufferAddress (FLEXIO_CAMERA_Type * *base*) [inline], [static]

Parameters

base	Pointer to the device.

Returns

data Pointer to the buffer that keeps the data with count of base->shifterCount .

16.3.7 FlexIO eDMA Camera Driver

16.3.7.1 Overview

Data Structures

- struct flexio_camera_edma_handle_t
  Camera eDMA handle. More...

Typedefs

- typedef void(∗ flexio_camera_edma_transfer_callback_t)(FLEXIO_CAMERA_Type ∗base, flexio-_camera_edma_handle_t ∗handle, status_t status, void ∗userData)
  Camera transfer callback function.

eDMA transactional

- status_t FLEXIO_CAMERA_TransferCreateHandleEDMA (FLEXIO_CAMERA_Type ∗base, flexio_camera_edma_handle_t ∗handle, flexio_camera_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗rxEdmaHandle)
  Initializes the Camera handle, which is used in transactional functions.
- status_t FLEXIO_CAMERA_TransferReceiveEDMA (FLEXIO_CAMERA_Type ∗base, flexio_-camera_edma_handle_t ∗handle, flexio_camera_transfer_t ∗xfer)
  Receives data using eDMA.
- void FLEXIO_CAMERA_TransferAbortReceiveEDMA (FLEXIO_CAMERA_Type ∗base, flexio-_camera_edma_handle_t ∗handle)
  Aborts the receive data which used the eDMA.
- status_t FLEXIO_CAMERA_TransferGetReceiveCountEDMA (FLEXIO_CAMERA_Type ∗base, flexio_camera_edma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be received.

16.3.7.2 Data Structure Documentation

16.3.7.2.1 struct _flexio_camera_edma_handle

Forward declaration of the handle typedef.

Data Fields

- flexio_camera_edma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  Camera callback function parameter.
- size_t rxSize
  Total bytes to be received.
- edma_handle_t ∗ rxEdmaHandle

The eDMA RX channel used.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- volatile uint8_t rxState
  RX transfer state.

16.3.7.2.1.1 Field Documentation

16.3.7.2.1.1.1 flexio_camera_edma_transfer_callback_t flexio_camera_edma_handle_t::callback

16.3.7.2.1.1.2 void* flexio_camera_edma_handle_t::userData

16.3.7.2.1.1.3 size_t flexio_camera_edma_handle_t::rxSize

16.3.7.2.1.1.4 edma_handle_t* flexio_camera_edma_handle_t::rxEdmaHandle

16.3.7.2.1.1.5 uint8_t flexio_camera_edma_handle_t::nbytes

16.3.7.3 Typedef Documentation

16.3.7.3.1 typedef void(* flexio_camera_edma_transfer_callback_t)(FLEXIO_CAMERA_Type *base, flexio_camera_edma_handle_t *handle, status_t status, void *userData)

16.3.7.4 Function Documentation

16.3.7.4.1 status_t FLEXIO_CAMERA_TransferCreateHandleEDMA (FLEXIO_CAMERA_Type * *base*, flexio_camera_edma_handle_t * *handle*, flexio_camera_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *rxEdmaHandle*)

Parameters

base	Pointer to the FLEXIO_CAMERA_Type.
handle	Pointer to flexio_camera_edma_handle_t structure.
callback	The callback function.
userData	The parameter of the callback function.
rxEdmaHandle	User requested DMA handle for RX DMA transfer.

Return values

kStatus_Success	Successfully create the handle.

kStatus_OutOfRange	The FlexIO Camera eDMA type/handle table out of range.

16.3.7.4.2 status_t FLEXIO_CAMERA_TransferReceiveEDMA (FLEXIO_CAMERA_Type * *base*, flexio_camera_edma_handle_t * *handle*, flexio_camera_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	Pointer to the FLEXIO_CAMERA_Type.
handle	Pointer to the flexio_camera_edma_handle_t structure.
xfer	Camera eDMA transfer structure, see flexio_camera_transfer_t.

Return values

kStatus_Success	if succeeded, others failed.
kStatus_CAMERA_Rx-Busy	Previous transfer on going.

16.3.7.4.3 void FLEXIO_CAMERA_TransferAbortReceiveEDMA (FLEXIO_CAMERA_Type * *base*, flexio_camera_edma_handle_t * *handle*)

This function aborts the receive data which used the eDMA.

Parameters

base	Pointer to the FLEXIO_CAMERA_Type.
handle	Pointer to the flexio_camera_edma_handle_t structure.

16.3.7.4.4 status_t FLEXIO_CAMERA_TransferGetReceiveCountEDMA (FLEXIO_CAMERA_Type * *base*, flexio_camera_edma_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes still not received.

Parameters

base	Pointer to the FLEXIO_CAMERA_Type.
handle	Pointer to the flexio_camera_edma_handle_t structure.
count	Number of bytes sent so far by the non-blocking transaction.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_InvalidArgument	The count parameter is invalid.

16.4 FlexIO I2C Master Driver

16.4.1 Overview

The KSDK provides a peripheral driver for I2C master function using Flexible I/O module of Kinetis devices.

The FlexIO I2C master driver includes functional APIs and transactional APIs.

Functional APIs target low level APIs. Functional APIs can be used for the FlexIO I2C master initialization/configuration/operation for the optimization/customization purpose. Using the functional APIs requires the knowledge of the FlexIO I2C master peripheral and how to organize functional APIs to meet the application requirements. The FlexIO I2C master functional operation groups provide the functional APIs set.

Transactional APIs target 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 an asynchronous transfer. This means that the functions FLEXIO_I2C_-MasterTransferNonBlocking() set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_Success status.

16.4.2 Typical use case

16.4.2.1 FlexIO I2C master transfer using an interrupt method

```
flexio_i2c_master_handle_t g_m_handle;
flexio_i2c_master_config_t masterConfig;
flexio_i2c_master_transfer_t masterXfer;
volatile bool completionFlag = false;
const uint8_t sendData[] = [......];
FLEXIO_I2C_Type i2cDev;

void FLEXIO_I2C_MasterCallback(FLEXIO_I2C_Type *base, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_Success == status)
    {
        completionFlag = true;
    }
}

void main(void)
{
    //...

    FLEXIO_I2C_MasterGetDefaultConfig(&masterConfig);

    FLEXIO_I2C_MasterInit(&i2cDev, &user_config);
    FLEXIO_I2C_MasterTransferCreateHandle(&i2cDev, &g_m_handle,
      FLEXIO_I2C_MasterCallback, NULL);

    // Prepares to send.
```

```
    masterXfer.slaveAddress = g_accel_address[0];
    masterXfer.direction = kI2C_Read;
    masterXfer.subaddress = &who_am_i_reg;
    masterXfer.subaddressSize = 1;
    masterXfer.data = &who_am_i_value;
    masterXfer.dataSize = 1;
    masterXfer.flags = kI2C_TransferDefaultFlag;

    // Sends out.
    FLEXIO_I2C_MasterTransferNonBlocking(&i2cDev, &g_m_handle, &
      masterXfer);

    // Wait for sending is complete.
    while (!completionFlag)
    {
    }

    // ...
}
```

Data Structures

- struct FLEXIO_I2C_Type
  Define FlexIO I2C master access structure typedef. More...
- struct flexio_i2c_master_config_t
  Define FlexIO I2C master user configuration structure. More...
- struct flexio_i2c_master_transfer_t
  Define FlexIO I2C master transfer structure. More...
- struct flexio_i2c_master_handle_t
  Define FlexIO I2C master handle structure. More...

Typedefs

- typedef void(∗ flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type ∗base, flexio_i2c_-master_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO I2C master transfer callback typedef.

Enumerations

- enum _flexio_i2c_status {
  kStatus_FLEXIO_I2C_Busy = MAKE_STATUS(kStatusGroup_FLEXIO_I2C, 0),
  kStatus_FLEXIO_I2C_Idle = MAKE_STATUS(kStatusGroup_FLEXIO_I2C, 1),
  kStatus_FLEXIO_I2C_Nak = MAKE_STATUS(kStatusGroup_FLEXIO_I2C, 2) }
  FlexIO I2C transfer status.
- enum _flexio_i2c_master_interrupt {
  kFLEXIO_I2C_TxEmptyInterruptEnable = 0x1U,
  kFLEXIO_I2C_RxFullInterruptEnable = 0x2U }
  Define FlexIO I2C master interrupt mask.
- enum _flexio_i2c_master_status_flags {

kFLEXIO_I2C_TxEmptyFlag = 0x1U,
kFLEXIO_I2C_RxFullFlag = 0x2U,
kFLEXIO_I2C_ReceiveNakFlag = 0x4U }
Define FlexIO I2C master status mask.
- enum flexio_i2c_direction_t {
kFLEXIO_I2C_Write = 0x0U,
kFLEXIO_I2C_Read = 0x1U }
Direction of master transfer.

Driver version

- #define FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
FlexIO I2C master driver version 2.1.2.

Initialization and deinitialization

- void FLEXIO_I2C_MasterInit (FLEXIO_I2C_Type ∗base, flexio_i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.
- void FLEXIO_I2C_MasterDeinit (FLEXIO_I2C_Type ∗base)
De-initializes the FlexIO I2C master peripheral.
- void FLEXIO_I2C_MasterGetDefaultConfig (flexio_i2c_master_config_t ∗masterConfig)
Gets the default configuration to configure the FlexIO module.
- static void FLEXIO_I2C_MasterEnable (FLEXIO_I2C_Type ∗base, bool enable)
Enables/disables the FlexIO module operation.

Status

- uint32_t FLEXIO_I2C_MasterGetStatusFlags (FLEXIO_I2C_Type ∗base)
Gets the FlexIO I2C master status flags.
- void FLEXIO_I2C_MasterClearStatusFlags (FLEXIO_I2C_Type ∗base, uint32_t mask)
Clears the FlexIO I2C master status flags.

Interrupts

- void FLEXIO_I2C_MasterEnableInterrupts (FLEXIO_I2C_Type ∗base, uint32_t mask)
Enables the FlexIO i2c master interrupt requests.
- void FLEXIO_I2C_MasterDisableInterrupts (FLEXIO_I2C_Type ∗base, uint32_t mask)
Disables the FlexIO I2C master interrupt requests.

Bus Operations

- void FLEXIO_I2C_MasterSetBaudRate (FLEXIO_I2C_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the FlexIO I2C master transfer baudrate.
- void FLEXIO_I2C_MasterStart (FLEXIO_I2C_Type ∗base, uint8_t address, flexio_i2c_direction_t direction)
  Sends START + 7-bit address to the bus.
- void FLEXIO_I2C_MasterStop (FLEXIO_I2C_Type ∗base)
  Sends the stop signal on the bus.
- void FLEXIO_I2C_MasterRepeatedStart (FLEXIO_I2C_Type ∗base)
  Sends the repeated start signal on the bus.
- void FLEXIO_I2C_MasterAbortStop (FLEXIO_I2C_Type ∗base)
  Sends the stop signal when transfer is still on-going.
- void FLEXIO_I2C_MasterEnableAck (FLEXIO_I2C_Type ∗base, bool enable)
  Configures the sent ACK/NAK for the following byte.
- status_t FLEXIO_I2C_MasterSetTransferCount (FLEXIO_I2C_Type ∗base, uint8_t count)
  Sets the number of bytes to be transferred from a start signal to a stop signal.
- static void FLEXIO_I2C_MasterWriteByte (FLEXIO_I2C_Type ∗base, uint32_t data)
  Writes one byte of data to the I2C bus.
- static uint8_t FLEXIO_I2C_MasterReadByte (FLEXIO_I2C_Type ∗base)
  Reads one byte of data from the I2C bus.
- status_t FLEXIO_I2C_MasterWriteBlocking (FLEXIO_I2C_Type ∗base, const uint8_t ∗txBuff, uint8_t txSize)
  Sends a buffer of data in bytes.
- void FLEXIO_I2C_MasterReadBlocking (FLEXIO_I2C_Type ∗base, uint8_t ∗rxBuff, uint8_t rxSize)
  Receives a buffer of bytes.
- status_t FLEXIO_I2C_MasterTransferBlocking (FLEXIO_I2C_Type ∗base, flexio_i2c_master_transfer_t ∗xfer)
  Performs a master polling transfer on the I2C bus.

Transactional

- status_t FLEXIO_I2C_MasterTransferCreateHandle (FLEXIO_I2C_Type ∗base, flexio_i2c_master_handle_t ∗handle, flexio_i2c_master_transfer_callback_t callback, void ∗userData)
  Initializes the I2C handle which is used in transactional functions.
- status_t FLEXIO_I2C_MasterTransferNonBlocking (FLEXIO_I2C_Type ∗base, flexio_i2c_master_handle_t ∗handle, flexio_i2c_master_transfer_t ∗xfer)
  Performs a master interrupt non-blocking transfer on the I2C bus.
- status_t FLEXIO_I2C_MasterTransferGetCount (FLEXIO_I2C_Type ∗base, flexio_i2c_master_handle_t ∗handle, size_t ∗count)
  Gets the master transfer status during a interrupt non-blocking transfer.
- void FLEXIO_I2C_MasterTransferAbort (FLEXIO_I2C_Type ∗base, flexio_i2c_master_handle_t ∗handle)
  Aborts an interrupt non-blocking transfer early.
- void FLEXIO_I2C_MasterTransferHandleIRQ (void ∗i2cType, void ∗i2cHandle)
  Master interrupt handler.

16.4.3 Data Structure Documentation

16.4.3.1 struct FLEXIO_I2C_Type

Data Fields

- FLEXIO_Type * flexioBase
  FlexIO base pointer.
- uint8_t SDAPinIndex
  Pin select for I2C SDA.
- uint8_t SCLPinIndex
  Pin select for I2C SCL.
- uint8_t shifterIndex [2]
  Shifter index used in FlexIO I2C.
- uint8_t timerIndex [2]
  Timer index used in FlexIO I2C.

16.4.3.1.0.1 Field Documentation

16.4.3.1.0.1.1 FLEXIO_Type* FLEXIO_I2C_Type::flexioBase

16.4.3.1.0.1.2 uint8_t FLEXIO_I2C_Type::SDAPinIndex

16.4.3.1.0.1.3 uint8_t FLEXIO_I2C_Type::SCLPinIndex

16.4.3.1.0.1.4 uint8_t FLEXIO_I2C_Type::shifterIndex[2]

16.4.3.1.0.1.5 uint8_t FLEXIO_I2C_Type::timerIndex[2]

16.4.3.2 struct flexio_i2c_master_config_t

Data Fields

- bool enableMaster
  Enables the FlexIO I2C peripheral at initialization time.
- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  `Enable/disable fast access to FlexIO registers, fast access requires`
  the FlexIO clock to be at least twice the frequency of the bus clock.
- uint32_t baudRate_Bps
  Baud rate in Bps.

16.4.3.2.0.2 Field Documentation

16.4.3.2.0.2.1 bool flexio_i2c_master_config_t::enableMaster

16.4.3.2.0.2.2 bool flexio_i2c_master_config_t::enableInDoze

16.4.3.2.0.2.3 bool flexio_i2c_master_config_t::enableInDebug

16.4.3.2.0.2.4 bool flexio_i2c_master_config_t::enableFastAccess

16.4.3.2.0.2.5 uint32_t flexio_i2c_master_config_t::baudRate_Bps

16.4.3.3 struct flexio_i2c_master_transfer_t

Data Fields

- uint32_t flags
  Transfer flag which controls the transfer, reserved for FlexIO I2C.
- uint8_t slaveAddress
  7-bit slave address.
- flexio_i2c_direction_t direction
  Transfer direction, read or write.
- uint32_t subaddress
  Sub address.
- uint8_t subaddressSize
  Size of command buffer.
- uint8_t volatile ∗ data
  Transfer buffer.
- volatile size_t dataSize
  Transfer size.

16.4.3.3.0.3 Field Documentation

16.4.3.3.0.3.1 uint32_t flexio_i2c_master_transfer_t::flags

16.4.3.3.0.3.2 uint8_t flexio_i2c_master_transfer_t::slaveAddress

16.4.3.3.0.3.3 flexio_i2c_direction_t flexio_i2c_master_transfer_t::direction

16.4.3.3.0.3.4 uint32_t flexio_i2c_master_transfer_t::subaddress

Transferred MSB first.

16.4.3.3.0.3.5 uint8_t flexio_i2c_master_transfer_t::subaddressSize

16.4.3.3.0.3.6 uint8_t volatile∗ flexio_i2c_master_transfer_t::data

16.4.3.3.0.3.7 volatile size_t flexio_i2c_master_transfer_t::dataSize

16.4.3.4 struct _flexio_i2c_master_handle

FlexIO I2C master handle typedef.

Data Fields

- flexio_i2c_master_transfer_t transfer
  FlexIO I2C master transfer copy.
- size_t transferSize
  Total bytes to be transferred.
- uint8_t state
  Transfer state maintained during transfer.
- flexio_i2c_master_transfer_callback_t completionCallback
  Callback function called at transfer event.
- void ∗ userData
  Callback parameter passed to callback function.

16.4.3.4.0.4 Field Documentation

16.4.3.4.0.4.1 flexio_i2c_master_transfer_t flexio_i2c_master_handle_t::transfer

16.4.3.4.0.4.2 size_t flexio_i2c_master_handle_t::transferSize

16.4.3.4.0.4.3 uint8_t flexio_i2c_master_handle_t::state

16.4.3.4.0.4.4 flexio_i2c_master_transfer_callback_t flexio_i2c_master_handle_t::completion-Callback

Callback function called at transfer event.

16.4.3.4.0.4.5 void* flexio_i2c_master_handle_t::userData

16.4.4 Macro Definition Documentation

16.4.4.1 #define FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

16.4.5 Typedef Documentation

16.4.5.1 typedef void(* flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type *base, flexio_i2c_master_handle_t *handle, status_t status, void *userData)

16.4.6 Enumeration Type Documentation

16.4.6.1 enum _flexio_i2c_status

Enumerator

kStatus_FLEXIO_I2C_Busy I2C is busy doing transfer.
kStatus_FLEXIO_I2C_Idle I2C is busy doing transfer.
kStatus_FLEXIO_I2C_Nak NAK received during transfer.

16.4.6.2 enum _flexio_i2c_master_interrupt

Enumerator

kFLEXIO_I2C_TxEmptyInterruptEnable Tx buffer empty interrupt enable.
kFLEXIO_I2C_RxFullInterruptEnable Rx buffer full interrupt enable.

16.4.6.3 enum _flexio_i2c_master_status_flags

Enumerator

kFLEXIO_I2C_TxEmptyFlag Tx shifter empty flag.
kFLEXIO_I2C_RxFullFlag Rx shifter full/Transfer complete flag.
kFLEXIO_I2C_ReceiveNakFlag Receive NAK flag.

16.4.6.4 enum flexio_i2c_direction_t

Enumerator

kFLEXIO_I2C_Write Master send to slave.
kFLEXIO_I2C_Read Master receive from slave.

16.4.7 Function Documentation

16.4.7.1 void FLEXIO_I2C_MasterInit (FLEXIO_I2C_Type * base, flexio_i2c_master_config_t * masterConfig, uint32_t srcClock_Hz)

Example

```
FLEXIO_I2C_Type base = {
.flexioBase = FLEXIO,
.SDAPinIndex = 0,
.SCLPinIndex = 1,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_i2c_master_config_t config = {
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 100000
};
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
masterConfig	Pointer to flexio_i2c_master_config_t structure.
srcClock_Hz	FlexIO source clock in Hz.

16.4.7.2 void FLEXIO_I2C_MasterDeinit (FLEXIO_I2C_Type * base)

Calling this API gates the FlexIO clock and the FlexIO I2C master module can't work unless the FLEXIO_I2C_MasterInit is called.

Parameters

base	pointer to FLEXIO_I2C_Type structure.

16.4.7.3 void FLEXIO_I2C_MasterGetDefaultConfig (flexio_i2c_master_config_t * masterConfig)

The configuration can be used directly for calling the FLEXIO_I2C_MasterInit().

Example:

```
flexio_i2c_master_config_t config;
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

Parameters

masterConfig	Pointer to flexio_i2c_master_config_t structure.

16.4.7.4 static void FLEXIO_I2C_MasterEnable (FLEXIO_I2C_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
enable	Pass true to enable module, false to disable module.

16.4.7.5 uint32_t FLEXIO_I2C_MasterGetStatusFlags (FLEXIO_I2C_Type ∗ *base*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure

Returns

Status flag, use status flag to AND _flexio_i2c_master_status_flags can get the related status.

16.4.7.6 void FLEXIO_I2C_MasterClearStatusFlags (FLEXIO_I2C_Type ∗ *base*, uint32_t *mask*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
mask	Status flag. The parameter can be any combination of the following values: • kFLEXIO_I2C_RxFullFlag • kFLEXIO_I2C_ReceiveNakFlag

16.4.7.7 void FLEXIO_I2C_MasterEnableInterrupts (FLEXIO_I2C_Type ∗ *base*, uint32_t *mask*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
mask	Interrupt source. Currently only one interrupt request source: • kFLEXIO_I2C_TransferCompleteInterruptEnable

16.4.7.8 void FLEXIO_I2C_MasterDisableInterrupts (FLEXIO_I2C_Type ∗ *base*, uint32_t *mask*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
mask	Interrupt source.

16.4.7.9 void FLEXIO_I2C_MasterSetBaudRate (FLEXIO_I2C_Type ∗ *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure
baudRate_Bps	the baud rate value in HZ
srcClock_Hz	source clock in HZ

16.4.7.10 void FLEXIO_I2C_MasterStart (FLEXIO_I2C_Type ∗ *base*, uint8_t *address*, flexio_i2c_direction_t *direction*)

Note

This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO_I2C_-RxFullFlag status is asserted before calling this API.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
address	7-bit address.
direction	transfer direction. This parameter is one of the values in flexio_i2c_direction_t: • kFLEXIO_I2C_Write: Transmit • kFLEXIO_I2C_Read: Receive

16.4.7.11 void FLEXIO_I2C_MasterStop (FLEXIO_I2C_Type * *base*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.

16.4.7.12 void FLEXIO_I2C_MasterRepeatedStart (FLEXIO_I2C_Type * *base*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.

16.4.7.13 void FLEXIO_I2C_MasterAbortStop (FLEXIO_I2C_Type * *base*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.

16.4.7.14 void FLEXIO_I2C_MasterEnableAck (FLEXIO_I2C_Type * *base,* bool *enable*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
enable	True to configure send ACK, false configure to send NAK.

16.4.7.15 status_t FLEXIO_I2C_MasterSetTransferCount (FLEXIO_I2C_Type * *base,* uint8_t *count*)

Note

Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
count	Number of bytes need to be transferred from a start signal to a re-start/stop signal

Return values

kStatus_Success	Successfully configured the count.
kStatus_InvalidArgument	Input argument is invalid.

16.4.7.16 static void FLEXIO_I2C_MasterWriteByte (FLEXIO_I2C_Type ∗ *base*, uint32_t *data*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
data	a byte of data.

16.4.7.17 static uint8_t FLEXIO_I2C_MasterReadByte (FLEXIO_I2C_Type ∗ *base*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.

Returns

data byte read.

16.4.7.18 status_t FLEXIO_I2C_MasterWriteBlocking (FLEXIO_I2C_Type ∗ *base*, const uint8_t ∗ *txBuff*, uint8_t *txSize*)

Note

This function blocks via polling until all bytes have been sent.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
txBuff	The data bytes to send.
txSize	The number of data bytes to send.

Return values

kStatus_Success	Successfully write data.
kStatus_FLEXIO_I2C_-Nak	Receive NAK during writing data.

16.4.7.19 void FLEXIO_I2C_MasterReadBlocking (FLEXIO_I2C_Type ∗ *base*, uint8_t ∗ *rxBuff*, uint8_t *rxSize*)

Note

This function blocks via polling until all bytes have been received.

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
rxBuff	The buffer to store the received bytes.
rxSize	The number of data bytes to be received.

16.4.7.20 status_t FLEXIO_I2C_MasterTransferBlocking (FLEXIO_I2C_Type * *base*, flexio_i2c_master_transfer_t * *xfer*)

Note

The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

base	pointer to FLEXIO_I2C_Type structure.
xfer	pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

16.4.7.21 status_t FLEXIO_I2C_MasterTransferCreateHandle (FLEXIO_I2C_Type * *base*, flexio_i2c_master_handle_t * *handle*, flexio_i2c_master_transfer_callback_t *callback*, void * *userData*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
handle	Pointer to flexio_i2c_master_handle_t structure to store the transfer state.
callback	Pointer to user callback function.
userData	User param passed to the callback function.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/isr table out of range.

16.4.7.22 status_t FLEXIO_I2C_MasterTransferNonBlocking (FLEXIO_I2C_Type * *base*, flexio_i2c_master_handle_t * *handle*, flexio_i2c_master_transfer_t * *xfer*)

Note

The API returns immediately after the transfer initiates. Call FLEXIO_I2C_MasterGetTransfer-Count to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_FLEXIO_I2C_Busy, the transfer is finished.

Parameters

base	Pointer to FLEXIO_I2C_Type structure
handle	Pointer to flexio_i2c_master_handle_t structure which stores the transfer state
xfer	pointer to flexio_i2c_master_transfer_t structure

Return values

kStatus_Success	Successfully start a transfer.
kStatus_FLEXIO_I2C_-Busy	FlexIO I2C is not idle, is running another transfer.

16.4.7.23 status_t FLEXIO_I2C_MasterTransferGetCount (FLEXIO_I2C_Type * *base*, flexio_i2c_master_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to FLEXIO_I2C_Type structure.
handle	Pointer to flexio_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.

16.4.7.24 void FLEXIO_I2C_MasterTransferAbort (FLEXIO_I2C_Type * *base*, flexio_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	Pointer to FLEXIO_I2C_Type structure
handle	Pointer to flexio_i2c_master_handle_t structure which stores the transfer state

16.4.7.25 void FLEXIO_I2C_MasterTransferHandleIRQ (void * *i2cType*, void * *i2cHandle*)

Parameters

i2cType	Pointer to FLEXIO_I2C_Type structure
i2cHandle	Pointer to flexio_i2c_master_transfer_t structure

16.5 FlexIO I2S Driver

16.5.1 Overview

The KSDK provides a peripheral driver for I2S function using Flexible I/O module of Kinetis devices.

The FlexIO I2S driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for FlexIO I2S initialization/configuration/operation for optimization/customization purpose. Using the functional API requires knowledge of the FlexIO I2S peripheral and how to organize functional APIs to meet the application requirements. All functional APIs use the peripheral base address as the first parameter. FlexIO I2S functional operation groups provide the functional APIs set.

Transactional APIs target high-level APIs. The transactional APIs can be used to enable the peripheral and can also be used 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 A-PI implementation and write custom code. All transactional APIs use the the sai_handle_t as the first parameter. Initialize the handle by calling the FlexIO_I2S_TransferTxCreateHandle() or FlexIO_I2S_-TransferRxCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions FLEXIO_I2S_Transfer-SendNonBlocking() and FLEXIO_I2S_TransferReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus-_FLEXIO_I2S_TxIdle and kStatus_FLEXIO_I2S_RxIdle status.

16.5.2 Typical use case

16.5.2.1 FlexIO I2S send/receive using an interrupt method

```
sai_handle_t g_saiTxHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
volatile bool rxFinished;
const uint8_t sendData[] = [......];

void FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_I2S_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...

    FLEXIO_I2S_TxGetDefaultConfig(&user_config);

    FLEXIO_I2S_TxInit(FLEXIO I2S0, &user_config);
    FLEXIO_I2S_TransferTxCreateHandle(FLEXIO I2S0, &g_saiHandle,
```

```
      FLEXIO_I2S_UserCallback, NULL);

    //Configures the SAI format.
    FLEXIO_I2S_TransferTxSetTransferFormat(FLEXIO I2S0, &g_saiHandle, mclkSource, mclk);

    // Prepares to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Sends out.
    FLEXIO_I2S_TransferSendNonBlocking(FLEXIO I2S0, &g_saiHandle, &
      sendXfer);

    // Waiting to send is finished.
    while (!txFinished)
    {
    }

    // ...
}
```

16.5.2.2 FLEXIO_I2S send/receive using a DMA method

```
sai_handle_t g_saiHandle;
dma_handle_t g_saiTxDmaHandle;
dma_handle_t g_saiRxDmaHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
uint8_t sendData[] = ...;

void FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_I2S_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...

    FLEXIO_I2S_TxGetDefaultConfig(&user_config);
    FLEXIO_I2S_TxInit(FLEXIO I2S0, &user_config);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, FLEXIO_I2S_TX_DMA_CHANNEL, FLEXIO_I2S_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, FLEXIO_I2S_TX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_TransferTxCreateHandle(&g_saiTxDmaHandle, DMA0, FLEXIO_I2S_TX_DMA_CHANNEL);

    FLEXIO_I2S_TransferTxCreateHandleDMA(FLEXIO I2S0, &g_saiTxDmaHandle
      , FLEXIO_I2S_UserCallback, NULL);

    // Prepares to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
```

```
    txFinished = false;

    // Sends out.
    FLEXIO_I2S_TransferSendDMA(&g_saiHandle, &sendXfer);

    // Waiting to send is finished.
    while (!txFinished)
    {
    }

    // ...
}
```

Modules

- FlexIO DMA I2S Driver
- FlexIO eDMA I2S Driver

Data Structures

- struct FLEXIO_I2S_Type
  Define FlexIO I2S access structure typedef. More...
- struct flexio_i2s_config_t
  FlexIO I2S configure structure. More...
- struct flexio_i2s_format_t
  FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx. More...
- struct flexio_i2s_transfer_t
  Define FlexIO I2S transfer structure. More...
- struct flexio_i2s_handle_t
  Define FlexIO I2S handle structure. More...

Macros

- #define FLEXIO_I2S_XFER_QUEUE_SIZE (4)
  FlexIO I2S transfer queue size, user can refine it according to use case.

Typedefs

- typedef void(∗ flexio_i2s_callback_t)(FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO I2S xfer callback prototype.

Enumerations

- enum _flexio_i2s_status {
kStatus_FLEXIO_I2S_Idle = MAKE_STATUS(kStatusGroup_FLEXIO_I2S, 0),
kStatus_FLEXIO_I2S_TxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_I2S, 1),
kStatus_FLEXIO_I2S_RxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_I2S, 2),
kStatus_FLEXIO_I2S_Error = MAKE_STATUS(kStatusGroup_FLEXIO_I2S, 3),
kStatus_FLEXIO_I2S_QueueFull = MAKE_STATUS(kStatusGroup_FLEXIO_I2S, 4) }
FlexIO I2S transfer status.
- enum flexio_i2s_master_slave_t {
kFLEXIO_I2S_Master = 0x0U,
kFLEXIO_I2S_Slave = 0x1U }
Master or slave mode.
- enum _flexio_i2s_interrupt_enable {
kFLEXIO_I2S_TxDataRegEmptyInterruptEnable = 0x1U,
kFLEXIO_I2S_RxDataRegFullInterruptEnable = 0x2U }
Define FlexIO FlexIO I2S interrupt mask.
- enum _flexio_i2s_status_flags {
kFLEXIO_I2S_TxDataRegEmptyFlag = 0x1U,
kFLEXIO_I2S_RxDataRegFullFlag = 0x2U }
Define FlexIO FlexIO I2S status mask.
- enum flexio_i2s_sample_rate_t {
kFLEXIO_I2S_SampleRate8KHz = 8000U,
kFLEXIO_I2S_SampleRate11025Hz = 11025U,
kFLEXIO_I2S_SampleRate12KHz = 12000U,
kFLEXIO_I2S_SampleRate16KHz = 16000U,
kFLEXIO_I2S_SampleRate22050Hz = 22050U,
kFLEXIO_I2S_SampleRate24KHz = 24000U,
kFLEXIO_I2S_SampleRate32KHz = 32000U,
kFLEXIO_I2S_SampleRate44100Hz = 44100U,
kFLEXIO_I2S_SampleRate48KHz = 48000U,
kFLEXIO_I2S_SampleRate96KHz = 96000U }
Audio sample rate.
- enum flexio_i2s_word_width_t {
kFLEXIO_I2S_WordWidth8bits = 8U,
kFLEXIO_I2S_WordWidth16bits = 16U,
kFLEXIO_I2S_WordWidth24bits = 24U,
kFLEXIO_I2S_WordWidth32bits = 32U }
Audio word width.

Driver version

- #define FSL_FLEXIO_I2S_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))
FlexIO I2S driver version 2.1.0.

Initialization and deinitialization

- void FLEXIO_I2S_Init (FLEXIO_I2S_Type ∗base, const flexio_i2s_config_t ∗config)
  Initializes the FlexIO I2S.
- void FLEXIO_I2S_GetDefaultConfig (flexio_i2s_config_t ∗config)
  Sets the FlexIO I2S configuration structure to default values.
- void FLEXIO_I2S_Deinit (FLEXIO_I2S_Type ∗base)
  De-initializes the FlexIO I2S.
- static void FLEXIO_I2S_Enable (FLEXIO_I2S_Type ∗base, bool enable)
  Enables/disables the FlexIO I2S module operation.

Status

- uint32_t FLEXIO_I2S_GetStatusFlags (FLEXIO_I2S_Type ∗base)
  Gets the FlexIO I2S status flags.

Interrupts

- void FLEXIO_I2S_EnableInterrupts (FLEXIO_I2S_Type ∗base, uint32_t mask)
  Enables the FlexIO I2S interrupt.
- void FLEXIO_I2S_DisableInterrupts (FLEXIO_I2S_Type ∗base, uint32_t mask)
  Disables the FlexIO I2S interrupt.

DMA Control

- static void FLEXIO_I2S_TxEnableDMA (FLEXIO_I2S_Type ∗base, bool enable)
  Enables/disables the FlexIO I2S Tx DMA requests.
- static void FLEXIO_I2S_RxEnableDMA (FLEXIO_I2S_Type ∗base, bool enable)
  Enables/disables the FlexIO I2S Rx DMA requests.
- static uint32_t FLEXIO_I2S_TxGetDataRegisterAddress (FLEXIO_I2S_Type ∗base)
  Gets the FlexIO I2S send data register address.
- static uint32_t FLEXIO_I2S_RxGetDataRegisterAddress (FLEXIO_I2S_Type ∗base)
  Gets the FlexIO I2S receive data register address.

Bus Operations

- void FLEXIO_I2S_MasterSetFormat (FLEXIO_I2S_Type ∗base, flexio_i2s_format_t ∗format, uint32_t srcClock_Hz)
  Configures the FlexIO I2S audio format in master mode.
- void FLEXIO_I2S_SlaveSetFormat (FLEXIO_I2S_Type ∗base, flexio_i2s_format_t ∗format)
  Configures the FlexIO I2S audio format in slave mode.
- void FLEXIO_I2S_WriteBlocking (FLEXIO_I2S_Type ∗base, uint8_t bitWidth, uint8_t ∗txData, size_t size)
  Sends data using a blocking method.
- static void FLEXIO_I2S_WriteData (FLEXIO_I2S_Type ∗base, uint8_t bitWidth, uint32_t data)

Writes data into a data register.
- void FLEXIO_I2S_ReadBlocking (FLEXIO_I2S_Type ∗base, uint8_t bitWidth, uint8_t ∗rxData, size_t size)
  Receives a piece of data using a blocking method.
- static uint32_t FLEXIO_I2S_ReadData (FLEXIO_I2S_Type ∗base)
  Reads a data from the data register.

Transactional

- void FLEXIO_I2S_TransferTxCreateHandle (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, flexio_i2s_callback_t callback, void ∗userData)
  Initializes the FlexIO I2S handle.
- void FLEXIO_I2S_TransferSetFormat (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, flexio_i2s_format_t ∗format, uint32_t srcClock_Hz)
  Configures the FlexIO I2S audio format.
- void FLEXIO_I2S_TransferRxCreateHandle (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, flexio_i2s_callback_t callback, void ∗userData)
  Initializes the FlexIO I2S receive handle.
- status_t FLEXIO_I2S_TransferSendNonBlocking (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs an interrupt non-blocking send transfer on FlexIO I2S.
- status_t FLEXIO_I2S_TransferReceiveNonBlocking (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs an interrupt non-blocking receive transfer on FlexIO I2S.
- void FLEXIO_I2S_TransferAbortSend (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle)
  Aborts the current send.
- void FLEXIO_I2S_TransferAbortReceive (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle)
  Aborts the current receive.
- status_t FLEXIO_I2S_TransferGetSendCount (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be sent.
- status_t FLEXIO_I2S_TransferGetReceiveCount (FLEXIO_I2S_Type ∗base, flexio_i2s_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be received.
- void FLEXIO_I2S_TransferTxHandleIRQ (void ∗i2sBase, void ∗i2sHandle)
  Tx interrupt handler.
- void FLEXIO_I2S_TransferRxHandleIRQ (void ∗i2sBase, void ∗i2sHandle)
  Rx interrupt handler.

16.5.3 Data Structure Documentation

16.5.3.1 struct FLEXIO_I2S_Type

Data Fields

- FLEXIO_Type ∗ flexioBase
  FlexIO base pointer.

- uint8_t txPinIndex
  Tx data pin index in FlexIO pins.
- uint8_t rxPinIndex
  Rx data pin index.
- uint8_t bclkPinIndex
  Bit clock pin index.
- uint8_t fsPinIndex
  Frame sync pin index.
- uint8_t txShifterIndex
  Tx data shifter index.
- uint8_t rxShifterIndex
  Rx data shifter index.
- uint8_t bclkTimerIndex
  Bit clock timer index.
- uint8_t fsTimerIndex
  Frame sync timer index.

16.5.3.2 struct flexio_i2s_config_t

Data Fields

- bool enableI2S
  Enable FlexIO I2S.
- flexio_i2s_master_slave_t masterSlave
  Master or slave.

16.5.3.3 struct flexio_i2s_format_t

Data Fields

- uint8_t bitWidth
  Bit width of audio data, always 8/16/24/32 bits.
- uint32_t sampleRate_Hz
  Sample rate of the audio data.

16.5.3.4 struct flexio_i2s_transfer_t

Data Fields

- uint8_t ∗ data
  Data buffer start pointer.
- size_t dataSize
  Bytes to be transferred.

16.5.3.4.0.5 Field Documentation

16.5.3.4.0.5.1 size_t flexio_i2s_transfer_t::dataSize

16.5.3.5 struct _flexio_i2s_handle

Data Fields

- uint32_t state
  Internal state.
- flexio_i2s_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback function.
- uint8_t bitWidth
  Bit width for transfer, 8/16/24/32bits.
- flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.

16.5.4 Macro Definition Documentation

16.5.4.1 #define FSL_FLEXIO_I2S_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

16.5.4.2 #define FLEXIO_I2S_XFER_QUEUE_SIZE (4)

16.5.5 Enumeration Type Documentation

16.5.5.1 enum _flexio_i2s_status

Enumerator

kStatus_FLEXIO_I2S_Idle FlexIO I2S is in idle state.
kStatus_FLEXIO_I2S_TxBusy FlexIO I2S Tx is busy.
kStatus_FLEXIO_I2S_RxBusy FlexIO I2S Tx is busy.
kStatus_FLEXIO_I2S_Error FlexIO I2S error occurred.
kStatus_FLEXIO_I2S_QueueFull FlexIO I2S transfer queue is full.

16.5.5.2 enum flexio_i2s_master_slave_t

Enumerator

kFLEXIO_I2S_Master Master mode.
kFLEXIO_I2S_Slave Slave mode.

16.5.5.3 enum _flexio_i2s_interrupt_enable

Enumerator

kFLEXIO_I2S_TxDataRegEmptyInterruptEnable Transmit buffer empty interrupt enable.
kFLEXIO_I2S_RxDataRegFullInterruptEnable Receive buffer full interrupt enable.

16.5.5.4 enum _flexio_i2s_status_flags

Enumerator

kFLEXIO_I2S_TxDataRegEmptyFlag Transmit buffer empty flag.
kFLEXIO_I2S_RxDataRegFullFlag Receive buffer full flag.

16.5.5.5 enum flexio_i2s_sample_rate_t

Enumerator

kFLEXIO_I2S_SampleRate8KHz Sample rate 8000Hz.
kFLEXIO_I2S_SampleRate11025Hz Sample rate 11025Hz.
kFLEXIO_I2S_SampleRate12KHz Sample rate 12000Hz.
kFLEXIO_I2S_SampleRate16KHz Sample rate 16000Hz.
kFLEXIO_I2S_SampleRate22050Hz Sample rate 22050Hz.
kFLEXIO_I2S_SampleRate24KHz Sample rate 24000Hz.
kFLEXIO_I2S_SampleRate32KHz Sample rate 32000Hz.
kFLEXIO_I2S_SampleRate44100Hz Sample rate 44100Hz.
kFLEXIO_I2S_SampleRate48KHz Sample rate 48000Hz.
kFLEXIO_I2S_SampleRate96KHz Sample rate 96000Hz.

16.5.5.6 enum flexio_i2s_word_width_t

Enumerator

kFLEXIO_I2S_WordWidth8bits Audio data width 8 bits.
kFLEXIO_I2S_WordWidth16bits Audio data width 16 bits.
kFLEXIO_I2S_WordWidth24bits Audio data width 24 bits.
kFLEXIO_I2S_WordWidth32bits Audio data width 32 bits.

16.5.6 Function Documentation

16.5.6.1 void FLEXIO_I2S_Init (FLEXIO_I2S_Type * *base*, const flexio_i2s_config_t * *config*)

This API configures FlexIO pins and shifter to I2S and configures the FlexIO I2S with a configuration structure. The configuration structure can be filled by the user, or be set with default values by FLEXIO_I2S_GetDefaultConfig().

Note

This API should be called at the beginning of the application to use the FlexIO I2S driver. Otherwise, any access to the FlexIO I2S module can cause hard fault because the clock is not enabled.

Parameters

base	FlexIO I2S base pointer
config	FlexIO I2S configure structure.

16.5.6.2 void FLEXIO_I2S_GetDefaultConfig (flexio_i2s_config_t * *config*)

The purpose of this API is to get the configuration structure initialized for use in FLEXIO_I2S_Init(). Users may use the initialized structure unchanged in FLEXIO_I2S_Init() or modify some fields of the structure before calling FLEXIO_I2S_Init().

Parameters

config	pointer to master configuration structure

16.5.6.3 void FLEXIO_I2S_Deinit (FLEXIO_I2S_Type * *base*)

Calling this API gates the FlexIO i2s clock. After calling this API, call the FLEXO_I2S_Init to use the FlexIO I2S module.

Parameters

base	FlexIO I2S base pointer

16.5.6.4 static void FLEXIO_I2S_Enable (FLEXIO_I2S_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to FLEXIO_I2S_Type
enable	True to enable, false to disable.

16.5.6.5 uint32_t FLEXIO_I2S_GetStatusFlags (FLEXIO_I2S_Type ∗ *base*)

Parameters

base	Pointer to FLEXIO_I2S_Type structure

Returns

Status flag, which are ORed by the enumerators in the _flexio_i2s_status_flags.

16.5.6.6 void FLEXIO_I2S_EnableInterrupts (FLEXIO_I2S_Type ∗ *base*, uint32_t *mask*)

This function enables the FlexIO UART interrupt.

Parameters

base	Pointer to FLEXIO_I2S_Type structure
mask	interrupt source

16.5.6.7 void FLEXIO_I2S_DisableInterrupts (FLEXIO_I2S_Type ∗ *base*, uint32_t *mask*)

This function enables the FlexIO UART interrupt.

Parameters

base	pointer to FLEXIO_I2S_Type structure
mask	interrupt source

16.5.6.8 static void FLEXIO_I2S_TxEnableDMA (FLEXIO_I2S_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	FlexIO I2S base pointer
enable	True means enable DMA, false means disable DMA.

16.5.6.9 static void FLEXIO_I2S_RxEnableDMA (FLEXIO_I2S_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	FlexIO I2S base pointer
enable	True means enable DMA, false means disable DMA.

16.5.6.10 static uint32_t FLEXIO_I2S_TxGetDataRegisterAddress (FLEXIO_I2S_Type * *base*) [inline], [static]

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

base	Pointer to FLEXIO_I2S_Type structure

Returns

FlexIO i2s send data register address.

16.5.6.11 static uint32_t FLEXIO_I2S_RxGetDataRegisterAddress (FLEXIO_I2S_Type * *base*) [inline], [static]

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

base	Pointer to FLEXIO_I2S_Type structure

Returns

FlexIO i2s receive data register address.

16.5.6.12 void FLEXIO_I2S_MasterSetFormat (FLEXIO_I2S_Type * *base*, flexio_i2s_format_t * *format*, uint32_t *srcClock_Hz*)

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

base	Pointer to FLEXIO_I2S_Type structure
format	Pointer to FlexIO I2S audio data format structure.
srcClock_Hz	I2S master clock source frequency in Hz.

16.5.6.13 void FLEXIO_I2S_SlaveSetFormat (FLEXIO_I2S_Type * *base*, flexio_i2s_format_t * *format*)

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

base	Pointer to FLEXIO_I2S_Type structure
format	Pointer to FlexIO I2S audio data format structure.

16.5.6.14 void FLEXIO_I2S_WriteBlocking (FLEXIO_I2S_Type * *base*, uint8_t *bitWidth*, uint8_t * *txData*, size_t *size*)

Note

This function blocks via polling until data is ready to be sent.

Parameters

base	FlexIO I2S base pointer.
bitWidth	How many bits in a audio word, usually 8/16/24/32 bits.
txData	Pointer to the data to be written.
size	Bytes to be written.

16.5.6.15 static void FLEXIO_I2S_WriteData (FLEXIO_I2S_Type * *base*, uint8_t *bitWidth*, uint32_t *data*) [inline], [static]

Parameters

base	FlexIO I2S base pointer.
bitWidth	How many bits in a audio word, usually 8/16/24/32 bits.
data	Data to be written.

16.5.6.16 void FLEXIO_I2S_ReadBlocking (FLEXIO_I2S_Type * *base*, uint8_t *bitWidth*, uint8_t * *rxData*, size_t *size*)

Note

This function blocks via polling until data is ready to be sent.

Parameters

base	FlexIO I2S base pointer
bitWidth	How many bits in a audio word, usually 8/16/24/32 bits.
rxData	Pointer to the data to be read.
size	Bytes to be read.

16.5.6.17 static uint32_t FLEXIO_I2S_ReadData (FLEXIO_I2S_Type * *base*) [inline], [static]

Parameters

base	FlexIO I2S base pointer

Returns

Data read from data register.

16.5.6.18 void FLEXIO_I2S_TransferTxCreateHandle (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*, flexio_i2s_callback_t *callback*, void * *userData*)

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

base	Pointer to FLEXIO_I2S_Type structure
handle	Pointer to flexio_i2s_handle_t structure to store the transfer state.
callback	FlexIO I2S callback function, which is called while finished a block.
userData	User parameter for the FlexIO I2S callback.

16.5.6.19 void FLEXIO_I2S_TransferSetFormat (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*, flexio_i2s_format_t * *format*, uint32_t *srcClock_Hz*)

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	FlexIO I2S handle pointer.
format	Pointer to audio data format structure.
srcClock_Hz	FlexIO I2S bit clock source frequency in Hz. This parameter should be 0 while in slave mode.

16.5.6.20 void FLEXIO_I2S_TransferRxCreateHandle (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*, flexio_i2s_callback_t *callback*, void * *userData*)

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure to store the transfer state.
callback	FlexIO I2S callback function, which is called while finished a block.
userData	User parameter for the FlexIO I2S callback.

16.5.6.21 status_t FLEXIO_I2S_TransferSendNonBlocking (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*, flexio_i2s_transfer_t * *xfer*)

Note

The API returns immediately after transfer initiates. Call FLEXIO_I2S_GetRemainingBytes to poll the transfer status and check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state
xfer	Pointer to flexio_i2s_transfer_t structure

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_FLEXIO_I2S_Tx-Busy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	The input parameter is invalid.

16.5.6.22 status_t FLEXIO_I2S_TransferReceiveNonBlocking (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*, flexio_i2s_transfer_t * *xfer*)

Note

The API returns immediately after transfer initiates. Call FLEXIO_I2S_GetRemainingBytes to poll the transfer status to check whether the transfer is finished. If the return status is 0, the transfer is finished.

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state
xfer	Pointer to flexio_i2s_transfer_t structure

Return values

kStatus_Success	Successfully start the data receive.
kStatus_FLEXIO_I2S_-RxBusy	Previous receive still not finished.
kStatus_InvalidArgument	The input parameter is invalid.

16.5.6.23 void FLEXIO_I2S_TransferAbortSend (FLEXIO_I2S_Type * *base*, flexio_i2s_handle_t * *handle*)

Note

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

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state

16.5.6.24 void FLEXIO_I2S_TransferAbortReceive (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_handle_t ∗ *handle*)

Note

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

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state

16.5.6.25 status_t FLEXIO_I2S_TransferGetSendCount (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state
count	Bytes sent.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

16.5.6.26 status_t FLEXIO_I2S_TransferGetReceiveCount (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	Pointer to FLEXIO_I2S_Type structure.
handle	Pointer to flexio_i2s_handle_t structure which stores the transfer state

Returns

count Bytes received.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

16.5.6.27 void FLEXIO_I2S_TransferTxHandleIRQ (void * *i2sBase,* void * *i2sHandle*)

Parameters

i2sBase	Pointer to FLEXIO_I2S_Type structure.
i2sHandle	Pointer to flexio_i2s_handle_t structure

16.5.6.28 void FLEXIO_I2S_TransferRxHandleIRQ (void * *i2sBase,* void * *i2sHandle*)

Parameters

i2sBase	Pointer to FLEXIO_I2S_Type structure.
i2sHandle	Pointer to flexio_i2s_handle_t structure.

16.5.7 FlexIO eDMA I2S Driver

16.5.7.1 Overview

Data Structures

- struct flexio_i2s_edma_handle_t
  FlexIO I2S DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ flexio_i2s_edma_callback_t)(FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO I2S eDMA transfer callback function for finish and error.

eDMA Transactional

- void FLEXIO_I2S_TransferTxCreateHandleEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, flexio_i2s_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the FlexIO I2S eDMA handle.
- void FLEXIO_I2S_TransferRxCreateHandleEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, flexio_i2s_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the FlexIO I2S Rx eDMA handle.
- void FLEXIO_I2S_TransferSetFormatEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, flexio_i2s_format_t ∗format, uint32_t srcClock_Hz)
  Configures the FlexIO I2S Tx audio format.
- status_t FLEXIO_I2S_TransferSendEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs a non-blocking FlexIO I2S transfer using DMA.
- status_t FLEXIO_I2S_TransferReceiveEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs a non-blocking FlexIO I2S receive using eDMA.
- void FLEXIO_I2S_TransferAbortSendEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle)
  Aborts a FlexIO I2S transfer using eDMA.
- void FLEXIO_I2S_TransferAbortReceiveEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle)
  Aborts a FlexIO I2S receive using eDMA.
- status_t FLEXIO_I2S_TransferGetSendCountEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be sent.
- status_t FLEXIO_I2S_TransferGetReceiveCountEDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_edma_handle_t ∗handle, size_t ∗count)
  Get the remaining bytes to be received.

16.5.7.2 Data Structure Documentation

16.5.7.2.1 struct _flexio_i2s_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  DMA handler for FlexIO I2S send.
- uint8_t bytesPerFrame
  Bytes in a frame.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- uint32_t state
  Internal state for FlexIO I2S eDMA transfer.
- flexio_i2s_edma_callback_t callback
  Callback for users while transfer finish or error occurred.
- void * userData
  User callback parameter.
- edma_tcd_t tcd [FLEXIO_I2S_XFER_QUEUE_SIZE+1U]
  TCD pool for eDMA transfer.
- flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.

16.5.7.2.1.1 Field Documentation

16.5.7.2.1.1.1 uint8_t flexio_i2s_edma_handle_t::nbytes

16.5.7.2.1.1.2 edma_tcd_t flexio_i2s_edma_handle_t::tcd[FLEXIO_I2S_XFER_QUEUE_SIZE+1U]

16.5.7.2.1.1.3 flexio_i2s_transfer_t flexio_i2s_edma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]

16.5.7.2.1.1.4 volatile uint8_t flexio_i2s_edma_handle_t::queueUser

16.5.7.3 Function Documentation

16.5.7.3.1 void FLEXIO_I2S_TransferTxCreateHandleEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*, flexio_i2s_edma_callback_t *callback*, void * *userData*, edma_handle_t * *dmaHandle*)

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S eDMA handle pointer.
callback	FlexIO I2S eDMA callback function called while finished a block.
userData	User parameter for callback.
dmaHandle	eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

16.5.7.3.2 void FLEXIO_I2S_TransferRxCreateHandleEDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_edma_handle_t ∗ *handle*, flexio_i2s_edma_callback_t *callback*, void ∗ *userData*, edma_handle_t ∗ *dmaHandle*)

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S eDMA handle pointer.
callback	FlexIO I2S eDMA callback function called while finished a block.
userData	User parameter for callback.
dmaHandle	eDMA handle for FlexIO I2S. This handle is a static value allocated by users.

16.5.7.3.3 void FLEXIO_I2S_TransferSetFormatEDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_edma_handle_t ∗ *handle*, flexio_i2s_format_t ∗ *format*, uint32_t *srcClock_Hz*)

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to format.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S eDMA handle pointer
format	Pointer to FlexIO I2S audio data format structure.
srcClock_Hz	FlexIO I2S clock source frequency in Hz, it should be 0 while in slave mode.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input arguments is invalid.

16.5.7.3.4 status_t FLEXIO_I2S_TransferSendEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*, flexio_i2s_transfer_t * *xfer*)

Note

This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetTransferStatus to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Start a FlexIO I2S eDMA send successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_TxBusy	FlexIO I2S is busy sending data.

16.5.7.3.5 status_t FLEXIO_I2S_TransferReceiveEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*, flexio_i2s_transfer_t * *xfer*)

Note

This interface returned immediately after transfer initiates. Users should call FLEXIO_I2S_GetReceiveRemainingBytes to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Start a FlexIO I2S eDMA receive successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_RxBusy	FlexIO I2S is busy receiving data.

16.5.7.3.6 void FLEXIO_I2S_TransferAbortSendEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.

16.5.7.3.7 void FLEXIO_I2S_TransferAbortReceiveEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.

16.5.7.3.8 status_t FLEXIO_I2S_TransferGetSendCountEDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
count	Bytes sent.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

16.5.7.3.9 status_t FLEXIO_I2S_TransferGetReceiveCountEDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_edma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
count	Bytes received.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

16.5.8 FlexIO DMA I2S Driver

16.5.8.1 Overview

Data Structures

- struct flexio_i2s_dma_handle_t
  FlexIO I2S DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ flexio_i2s_dma_callback_t)(FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO I2S DMA transfer callback function for finish and error.

DMA Transactional

- void FLEXIO_I2S_TransferTxCreateHandleDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, flexio_i2s_dma_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Initializes the FlexIO I2S DMA handle.
- void FLEXIO_I2S_TransferRxCreateHandleDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, flexio_i2s_dma_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Initializes the FlexIO I2S Rx DMA handle.
- void FLEXIO_I2S_TransferSetFormatDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, flexio_i2s_format_t ∗format, uint32_t srcClock_Hz)
  Configures the FlexIO I2S Tx audio format.
- status_t FLEXIO_I2S_TransferSendDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs a non-blocking FlexIO I2S transfer using DMA.
- status_t FLEXIO_I2S_TransferReceiveDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, flexio_i2s_transfer_t ∗xfer)
  Performs a non-blocking FlexIO I2S receive using DMA.
- void FLEXIO_I2S_TransferAbortSendDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle)
  Aborts a FlexIO I2S transfer using DMA.
- void FLEXIO_I2S_TransferAbortReceiveDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle)
  Aborts a FlexIO I2S receive using DMA.
- status_t FLEXIO_I2S_TransferGetSendCountDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be sent.
- status_t FLEXIO_I2S_TransferGetReceiveCountDMA (FLEXIO_I2S_Type ∗base, flexio_i2s_dma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be received.

16.5.8.2 Data Structure Documentation

16.5.8.2.1 struct _flexio_i2s_dma_handle

Data Fields

- dma_handle_t ∗ dmaHandle
  DMA handler for FlexIO I2S send.
- uint8_t bytesPerFrame
  Bytes in a frame.
- uint32_t state
  Internal state for FlexIO I2S DMA transfer.
- flexio_i2s_dma_callback_t callback
  Callback for users while transfer finish or error occurred.
- void ∗ userData
  User callback parameter.
- flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [FLEXIO_I2S_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.

16.5.8.2.1.1 Field Documentation

16.5.8.2.1.1.1 flexio_i2s_transfer_t flexio_i2s_dma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]

16.5.8.2.1.1.2 volatile uint8_t flexio_i2s_dma_handle_t::queueUser

16.5.8.3 Function Documentation

16.5.8.3.1 void FLEXIO_I2S_TransferTxCreateHandleDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_dma_handle_t ∗ *handle*, flexio_i2s_dma_callback_t *callback*, void ∗ *userData*, dma_handle_t ∗ *dmaHandle*)

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
callback	FlexIO I2S DMA callback function called while finished a block.
userData	User parameter for callback.
dmaHandle	DMA handle for FlexIO I2S. This handle is a static value allocated by users.

16.5.8.3.2 void FLEXIO_I2S_TransferRxCreateHandleDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_dma_handle_t * *handle*, flexio_i2s_dma_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
callback	FlexIO I2S DMA callback function called while finished a block.
userData	User parameter for callback.
dmaHandle	DMA handle for FlexIO I2S. This handle is a static value allocated by users.

16.5.8.3.3 void FLEXIO_I2S_TransferSetFormatDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_dma_handle_t * *handle*, flexio_i2s_format_t * *format*, uint32_t *srcClock_Hz*)

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the DMA parameter according to the format.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer
format	Pointer to FlexIO I2S audio data format structure.
srcClock_Hz	FlexIO I2S clock source frequency in Hz. It should be 0 while in slave mode.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input arguments is invalid.

16.5.8.3.4 status_t FLEXIO_I2S_TransferSendDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_dma_handle_t ∗ *handle*, flexio_i2s_transfer_t ∗ *xfer*)

Note

This interface returns immediately after transfer initiates. Call FLEXIO_I2S_GetTransferStatus to poll the transfer status and check whether FLEXIO I2S transfer finished.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Start a FlexIO I2S DMA send successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_TxBusy	FlexIO I2S is busy sending data.

16.5.8.3.5 status_t FLEXIO_I2S_TransferReceiveDMA (FLEXIO_I2S_Type ∗ *base*, flexio_i2s_dma_handle_t ∗ *handle*, flexio_i2s_transfer_t ∗ *xfer*)

Note

This interface returns immediately after transfer initiates. Call FLEXIO_I2S_GetReceive-RemainingBytes to poll the transfer status to check whether the FlexIO I2S transfer is finished.

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Start a FlexIO I2S DMA receive successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_RxBusy	FlexIO I2S is busy receiving data.

16.5.8.3.6 void FLEXIO_I2S_TransferAbortSendDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_dma_handle_t * *handle*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.

16.5.8.3.7 void FLEXIO_I2S_TransferAbortReceiveDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_dma_handle_t * *handle*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.

16.5.8.3.8 status_t FLEXIO_I2S_TransferGetSendCountDMA (FLEXIO_I2S_Type * *base*, flexio_i2s_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
count	Bytes sent.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

16.5.8.3.9 status_t FLEXIO_I2S_TransferGetReceiveCountDMA (FLEXIO_I2S_Type ∗ *base,* flexio_i2s_dma_handle_t ∗ *handle,* size_t ∗ *count*)

Parameters

base	FlexIO I2S peripheral base address.
handle	FlexIO I2S DMA handle pointer.
count	Bytes received.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

16.6 FlexIO SPI Driver

16.6.1 Overview

The KSDK provides a peripheral driver for an SPI function using the Flexible I/O module of Kinetis devices.

FlexIO SPI driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for FlexIO SPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the FlexIO SPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the FLEXIO_SPI_Type *base as the first parameter. FlexIO SPI functional operation groups provide the functional API set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and also in the application if the code size and performance of transactional APIs can satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the flexio_spi_master_handle_t/flexio_spi_slave_handle_t as the second parameter. Initialize the handle by calling the FLEXIO_SPI_MasterTransferCreateHandle() or FLEXIO_SPI_SlaveTransferCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions FLEXIO_SPI_MasterTransferNonBlocking()/FLEXIO_SPI_SlaveTransferNonBlocking() set up an interrupt for data transfer. When the transfer is complete, the upper layer is notified through a callback function with the kStatus_FLEXIO_SPI_Idle status. Note that the FlexIO SPI slave driver only supports discontinuous PCS access, which is a limitation. The FlexIO SPI slave driver can support continuous PCS, but the slave can't adapt discontinuous and continuous PCS automatically. Users can change the timer disable mode in FLEXIO_SPI_SlaveInit manually, from kFLEXIO_TimerDisableOnTimerCompare to kFLEXIO_TimerDisableNever to enable a discontinuous PCS access. Only CPHA = 0 is supported.

16.6.2 Typical use case

16.6.2.1 FlexIO SPI send/receive using an interrupt method

```
flexio_spi_master_handle_t g_spiHandle;
FLEXIO_SPI_Type spiDev;
volatile bool txFinished;
static uint8_t srcBuff[BUFFER_SIZE];
static uint8_t destBuff[BUFFER_SIZE];

void FLEXIO_SPI_MasterUserCallback(FLEXIO_SPI_Type *base, flexio_spi_master_handle_t *handle
      , status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_SPI_Idle == status)
    {
        txFinished = true;
    }
}

void main(void)
```

```
{
    //...
    flexio_spi_transfer_t xfer = {0};
    flexio_spi_master_config_t userConfig;

    FLEXIO_SPI_MasterGetDefaultConfig(&userConfig);
    userConfig.baudRate_Bps = 500000U;

    spiDev.flexioBase = BOARD_FLEXIO_BASE;
    spiDev.SDOPinIndex = FLEXIO_SPI_MOSI_PIN;
    spiDev.SDIPinIndex = FLEXIO_SPI_MISO_PIN;
    spiDev.SCKPinIndex = FLEXIO_SPI_SCK_PIN;
    spiDev.CSnPinIndex = FLEXIO_SPI_CSn_PIN;
    spiDev.shifterIndex[0] = 0U;
    spiDev.shifterIndex[1] = 1U;
    spiDev.timerIndex[0] = 0U;
    spiDev.timerIndex[1] = 1U;

    FLEXIO_SPI_MasterInit(&spiDev, &userConfig, FLEXIO_CLOCK_FREQUENCY);

    xfer.txData = srcBuff;
    xfer.rxData = destBuff;
    xfer.dataSize = BUFFER_SIZE;
    xfer.flags = kFLEXIO_SPI_8bitMsb;
    FLEXIO_SPI_MasterTransferCreateHandle(&spiDev, &g_spiHandle,
      FLEXIO_SPI_MasterUserCallback, NULL);
    FLEXIO_SPI_MasterTransferNonBlocking(&spiDev, &g_spiHandle, &xfer);

    // Send finished.
    while (!txFinished)
    {
    }

        // ...
}
```

16.6.2.2 FlexIO_SPI Send/Receive using a DMA method

```
dma_handle_t g_spiTxDmaHandle;
dma_handle_t g_spiRxDmaHandle;
flexio_spi_master_handle_t g_spiHandle;
FLEXIO_SPI_Type spiDev;
volatile bool txFinished;
static uint8_t srcBuff[BUFFER_SIZE];
static uint8_t destBuff[BUFFER_SIZE];
void FLEXIO_SPI_MasterUserCallback(FLEXIO_SPI_Type *base, flexio_spi_master_dma_handle_t *
      handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_SPI_Idle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    flexio_spi_transfer_t xfer = {0};
    flexio_spi_master_config_t userConfig;

    FLEXIO_SPI_MasterGetDefaultConfig(&userConfig);
    userConfig.baudRate_Bps = 500000U;

    spiDev.flexioBase = BOARD_FLEXIO_BASE;
```

```
    spiDev.SDOPinIndex = FLEXIO_SPI_MOSI_PIN;
    spiDev.SDIPinIndex = FLEXIO_SPI_MISO_PIN;
    spiDev.SCKPinIndex = FLEXIO_SPI_SCK_PIN;
    spiDev.CSnPinIndex = FLEXIO_SPI_CSn_PIN;
    spiDev.shifterIndex[0] = 0U;
    spiDev.shifterIndex[1] = 1U;
    spiDev.timerIndex[0] = 0U;
    spiDev.timerIndex[1] = 1U;

    /*Initializes the DMA for the example.*/
    DMAMGR_Init();

    dma_request_source_tx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + spiDev.
      shifterIndex[0]);
    dma_request_source_rx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + spiDev.
      shifterIndex[1]);

    /* Requests  DMA channels for transmit and receive. */
    DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_tx, 0, &txHandle);
    DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_rx, 1, &rxHandle);

    FLEXIO_SPI_MasterInit(&spiDev, &userConfig, FLEXIO_CLOCK_FREQUENCY);

    /* Initializes the buffer. */
    for (i = 0; i < BUFFER_SIZE; i++)
    {
        srcBuff[i] = i;
    }

    /* Sends to the slave. */
    xfer.txData = srcBuff;
    xfer.rxData = destBuff;
    xfer.dataSize = BUFFER_SIZE;
    xfer.flags = kFLEXIO_SPI_8bitMsb;
    FLEXIO_SPI_MasterTransferCreateHandleDMA(&spiDev, &g_spiHandle,
       FLEXIO_SPI_MasterUserCallback, NULL, &g_spiTxDmaHandle, &g_spiRxDmaHandle);
    FLEXIO_SPI_MasterTransferDMA(&spiDev, &g_spiHandle, &xfer);

    // Send finished.
    while (!txFinished)
    {
    }

    // ...
}
```

Modules

- FlexIO DMA SPI Driver
- FlexIO eDMA SPI Driver

Data Structures

- struct FLEXIO_SPI_Type
  Define FlexIO SPI access structure typedef. More...
- struct flexio_spi_master_config_t
  Define FlexIO SPI master configuration structure. More...
- struct flexio_spi_slave_config_t
  Define FlexIO SPI slave configuration structure. More...

- struct flexio_spi_transfer_t
  Define FlexIO SPI transfer structure. More...
- struct flexio_spi_master_handle_t
  Define FlexIO SPI handle structure. More...

Macros

- #define FLEXIO_SPI_DUMMYDATA (0xFFFFU)
  FlexIO SPI dummy transfer data, the data is sent while txData is NULL.

Typedefs

- typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t
  Slave handle is the same with master handle.
- typedef void(∗ flexio_spi_master_transfer_callback_t)(FLEXIO_SPI_Type ∗base, flexio_spi_-master_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO SPI master callback for finished transmit.
- typedef void(∗ flexio_spi_slave_transfer_callback_t)(FLEXIO_SPI_Type ∗base, flexio_spi_slave-_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO SPI slave callback for finished transmit.

Enumerations

- enum _flexio_spi_status {
  kStatus_FLEXIO_SPI_Busy = MAKE_STATUS(kStatusGroup_FLEXIO_SPI, 1),
  kStatus_FLEXIO_SPI_Idle = MAKE_STATUS(kStatusGroup_FLEXIO_SPI, 2),
  kStatus_FLEXIO_SPI_Error = MAKE_STATUS(kStatusGroup_FLEXIO_SPI, 3) }
  Error codes for the FlexIO SPI driver.
- enum flexio_spi_clock_phase_t {
  kFLEXIO_SPI_ClockPhaseFirstEdge = 0x0U,
  kFLEXIO_SPI_ClockPhaseSecondEdge = 0x1U }
  FlexIO SPI clock phase configuration.
- enum flexio_spi_shift_direction_t {
  kFLEXIO_SPI_MsbFirst = 0,
  kFLEXIO_SPI_LsbFirst = 1 }
  FlexIO SPI data shifter direction options.
- enum flexio_spi_data_bitcount_mode_t {
  kFLEXIO_SPI_8BitMode = 0x08U,
  kFLEXIO_SPI_16BitMode = 0x10U }
  FlexIO SPI data length mode options.
- enum _flexio_spi_interrupt_enable {
  kFLEXIO_SPI_TxEmptyInterruptEnable = 0x1U,
  kFLEXIO_SPI_RxFullInterruptEnable = 0x2U }
  Define FlexIO SPI interrupt mask.

- enum _flexio_spi_status_flags {
  kFLEXIO_SPI_TxBufferEmptyFlag = 0x1U,
  kFLEXIO_SPI_RxBufferFullFlag = 0x2U }
  Define FlexIO SPI status mask.
- enum _flexio_spi_dma_enable {
  kFLEXIO_SPI_TxDmaEnable = 0x1U,
  kFLEXIO_SPI_RxDmaEnable = 0x2U,
  kFLEXIO_SPI_DmaAllEnable = 0x3U }
  Define FlexIO SPI DMA mask.
- enum _flexio_spi_transfer_flags {
  kFLEXIO_SPI_8bitMsb = 0x1U,
  kFLEXIO_SPI_8bitLsb = 0x2U,
  kFLEXIO_SPI_16bitMsb = 0x9U,
  kFLEXIO_SPI_16bitLsb = 0xaU }
  Define FlexIO SPI transfer flags.

Driver version

- #define FSL_FLEXIO_SPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  FlexIO SPI driver version 2.1.0.

FlexIO SPI Configuration

- void FLEXIO_SPI_MasterInit (FLEXIO_SPI_Type ∗base, flexio_spi_master_config_t ∗masterConfig, uint32_t srcClock_Hz)
  Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI master hardware, and configures the FlexIO SPI with FlexIO SPI master configuration.
- void FLEXIO_SPI_MasterDeinit (FLEXIO_SPI_Type ∗base)
  Gates the FlexIO clock.
- void FLEXIO_SPI_MasterGetDefaultConfig (flexio_spi_master_config_t ∗masterConfig)
  Gets the default configuration to configure the FlexIO SPI master.
- void FLEXIO_SPI_SlaveInit (FLEXIO_SPI_Type ∗base, flexio_spi_slave_config_t ∗slaveConfig)
  Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI slave hardware configuration, and configures the FlexIO SPI with FlexIO SPI slave configuration.
- void FLEXIO_SPI_SlaveDeinit (FLEXIO_SPI_Type ∗base)
  Gates the FlexIO clock.
- void FLEXIO_SPI_SlaveGetDefaultConfig (flexio_spi_slave_config_t ∗slaveConfig)
  Gets the default configuration to configure the FlexIO SPI slave.

Status

- uint32_t FLEXIO_SPI_GetStatusFlags (FLEXIO_SPI_Type ∗base)
  Gets FlexIO SPI status flags.
- void FLEXIO_SPI_ClearStatusFlags (FLEXIO_SPI_Type ∗base, uint32_t mask)
  Clears FlexIO SPI status flags.

Interrupts

- void FLEXIO_SPI_EnableInterrupts (FLEXIO_SPI_Type ∗base, uint32_t mask)
  Enables the FlexIO SPI interrupt.
- void FLEXIO_SPI_DisableInterrupts (FLEXIO_SPI_Type ∗base, uint32_t mask)
  Disables the FlexIO SPI interrupt.

DMA Control

- void FLEXIO_SPI_EnableDMA (FLEXIO_SPI_Type ∗base, uint32_t mask, bool enable)
  Enables/disables the FlexIO SPI transmit DMA.
- static uint32_t FLEXIO_SPI_GetTxDataRegisterAddress (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction)
  Gets the FlexIO SPI transmit data register address for MSB first transfer.
- static uint32_t FLEXIO_SPI_GetRxDataRegisterAddress (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction)
  Gets the FlexIO SPI receive data register address for the MSB first transfer.

Bus Operations

- static void FLEXIO_SPI_Enable (FLEXIO_SPI_Type ∗base, bool enable)
  Enables/disables the FlexIO SPI module operation.
- void FLEXIO_SPI_MasterSetBaudRate (FLEXIO_SPI_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClockHz)
  Sets baud rate for the FlexIO SPI transfer, which is only used for the master.
- static void FLEXIO_SPI_WriteData (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction, uint16_t data)
  Writes one byte of data, which is sent using the MSB method.
- static uint16_t FLEXIO_SPI_ReadData (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction)
  Reads 8 bit/16 bit data.
- void FLEXIO_SPI_WriteBlocking (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction, const uint8_t ∗buffer, size_t size)
  Sends a buffer of data bytes.
- void FLEXIO_SPI_ReadBlocking (FLEXIO_SPI_Type ∗base, flexio_spi_shift_direction_t direction, uint8_t ∗buffer, size_t size)
  Receives a buffer of bytes.
- void FLEXIO_SPI_MasterTransferBlocking (FLEXIO_SPI_Type ∗base, flexio_spi_transfer_t ∗xfer)
  Receives a buffer of bytes.

Transactional

- status_t FLEXIO_SPI_MasterTransferCreateHandle (FLEXIO_SPI_Type ∗base, flexio_spi_master_handle_t ∗handle, flexio_spi_master_transfer_callback_t callback, void ∗userData)
  Initializes the FlexIO SPI Master handle, which is used in transactional functions.

- status_t FLEXIO_SPI_MasterTransferNonBlocking (FLEXIO_SPI_Type ∗base, flexio_spi_master_handle_t ∗handle, flexio_spi_transfer_t ∗xfer)
  Master transfer data using IRQ.
- void FLEXIO_SPI_MasterTransferAbort (FLEXIO_SPI_Type ∗base, flexio_spi_master_handle_t ∗handle)
  Aborts the master data transfer, which used IRQ.
- status_t FLEXIO_SPI_MasterTransferGetCount (FLEXIO_SPI_Type ∗base, flexio_spi_master_handle_t ∗handle, size_t ∗count)
  Gets the data transfer status which used IRQ.
- void FLEXIO_SPI_MasterTransferHandleIRQ (void ∗spiType, void ∗spiHandle)
  FlexIO SPI master IRQ handler function.
- status_t FLEXIO_SPI_SlaveTransferCreateHandle (FLEXIO_SPI_Type ∗base, flexio_spi_slave_handle_t ∗handle, flexio_spi_slave_transfer_callback_t callback, void ∗userData)
  Initializes the FlexIO SPI Slave handle, which is used in transactional functions.
- status_t FLEXIO_SPI_SlaveTransferNonBlocking (FLEXIO_SPI_Type ∗base, flexio_spi_slave_handle_t ∗handle, flexio_spi_transfer_t ∗xfer)
  Slave transfer data using IRQ.
- static void FLEXIO_SPI_SlaveTransferAbort (FLEXIO_SPI_Type ∗base, flexio_spi_slave_handle_t ∗handle)
  Aborts the slave data transfer which used IRQ, share same API with master.
- static status_t FLEXIO_SPI_SlaveTransferGetCount (FLEXIO_SPI_Type ∗base, flexio_spi_slave_handle_t ∗handle, size_t ∗count)
  Gets the data transfer status which used IRQ, share same API with master.
- void FLEXIO_SPI_SlaveTransferHandleIRQ (void ∗spiType, void ∗spiHandle)
  FlexIO SPI slave IRQ handler function.

16.6.3 Data Structure Documentation

16.6.3.1 struct FLEXIO_SPI_Type

Data Fields

- FLEXIO_Type ∗ flexioBase
  FlexIO base pointer.
- uint8_t SDOPinIndex
  Pin select for data output.
- uint8_t SDIPinIndex
  Pin select for data input.
- uint8_t SCKPinIndex
  Pin select for clock.
- uint8_t CSnPinIndex
  Pin select for enable.
- uint8_t shifterIndex [2]
  Shifter index used in FlexIO SPI.
- uint8_t timerIndex [2]
  Timer index used in FlexIO SPI.

16.6.3.1.0.1 Field Documentation

16.6.3.1.0.1.1 FLEXIO_Type* FLEXIO_SPI_Type::flexioBase

16.6.3.1.0.1.2 uint8_t FLEXIO_SPI_Type::SDOPinIndex

16.6.3.1.0.1.3 uint8_t FLEXIO_SPI_Type::SDIPinIndex

16.6.3.1.0.1.4 uint8_t FLEXIO_SPI_Type::SCKPinIndex

16.6.3.1.0.1.5 uint8_t FLEXIO_SPI_Type::CSnPinIndex

16.6.3.1.0.1.6 uint8_t FLEXIO_SPI_Type::shifterIndex[2]

16.6.3.1.0.1.7 uint8_t FLEXIO_SPI_Type::timerIndex[2]

16.6.3.2 struct flexio_spi_master_config_t

Data Fields

- bool enableMaster
  Enable/disable FlexIO SPI master after configuration.
- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  `Enable/disable fast access to FlexIO registers,`
  fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.
- uint32_t baudRate_Bps
  Baud rate in Bps.
- flexio_spi_clock_phase_t phase
  Clock phase.
- flexio_spi_data_bitcount_mode_t dataMode
  8bit or 16bit mode.

16.6.3.2.0.2 Field Documentation

16.6.3.2.0.2.1 bool flexio_spi_master_config_t::enableMaster

16.6.3.2.0.2.2 bool flexio_spi_master_config_t::enableInDoze

16.6.3.2.0.2.3 bool flexio_spi_master_config_t::enableInDebug

16.6.3.2.0.2.4 bool flexio_spi_master_config_t::enableFastAccess

16.6.3.2.0.2.5 uint32_t flexio_spi_master_config_t::baudRate_Bps

16.6.3.2.0.2.6 flexio_spi_clock_phase_t flexio_spi_master_config_t::phase

16.6.3.2.0.2.7 flexio_spi_data_bitcount_mode_t flexio_spi_master_config_t::dataMode

16.6.3.3 struct flexio_spi_slave_config_t

Data Fields

- bool enableSlave
  Enable/disable FlexIO SPI slave after configuration.
- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  ```
  Enable/disable fast access to FlexIO registers,
  ```
  fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.
- flexio_spi_clock_phase_t phase
  Clock phase.
- flexio_spi_data_bitcount_mode_t dataMode
  8bit or 16bit mode.

16.6.3.3.0.3 Field Documentation

16.6.3.3.0.3.1 bool flexio_spi_slave_config_t::enableSlave

16.6.3.3.0.3.2 bool flexio_spi_slave_config_t::enableInDoze

16.6.3.3.0.3.3 bool flexio_spi_slave_config_t::enableInDebug

16.6.3.3.0.3.4 bool flexio_spi_slave_config_t::enableFastAccess

16.6.3.3.0.3.5 flexio_spi_clock_phase_t flexio_spi_slave_config_t::phase

16.6.3.3.0.3.6 flexio_spi_data_bitcount_mode_t flexio_spi_slave_config_t::dataMode

16.6.3.4 struct flexio_spi_transfer_t

Data Fields

- uint8_t ∗ txData
  Send buffer.
- uint8_t ∗ rxData
  Receive buffer.
- size_t dataSize
  Transfer bytes.
- uint8_t flags
  FlexIO SPI control flag, MSB first or LSB first.

16.6.3.4.0.4 Field Documentation

16.6.3.4.0.4.1 uint8_t∗ flexio_spi_transfer_t::txData

16.6.3.4.0.4.2 uint8_t∗ flexio_spi_transfer_t::rxData

16.6.3.4.0.4.3 size_t flexio_spi_transfer_t::dataSize

16.6.3.4.0.4.4 uint8_t flexio_spi_transfer_t::flags

16.6.3.5 struct _flexio_spi_master_handle

typedef for flexio_spi_master_handle_t in advance.

Data Fields

- uint8_t ∗ txData
  Transfer buffer.
- uint8_t ∗ rxData
  Receive buffer.
- size_t transferSize
  Total bytes to be transferred.
- volatile size_t txRemainingBytes

Send data remaining in bytes.
- volatile size_t rxRemainingBytes
  Receive data remaining in bytes.
- volatile uint32_t state
  FlexIO SPI internal state.
- uint8_t bytePerFrame
  SPI mode, 2bytes or 1byte in a frame.
- flexio_spi_shift_direction_t direction
  Shift direction.
- flexio_spi_master_transfer_callback_t callback
  FlexIO SPI callback.
- void * userData
  Callback parameter.

16.6.3.5.0.5 Field Documentation

16.6.3.5.0.5.1 uint8_t* flexio_spi_master_handle_t::txData

16.6.3.5.0.5.2 uint8_t* flexio_spi_master_handle_t::rxData

16.6.3.5.0.5.3 size_t flexio_spi_master_handle_t::transferSize

16.6.3.5.0.5.4 volatile size_t flexio_spi_master_handle_t::txRemainingBytes

16.6.3.5.0.5.5 volatile size_t flexio_spi_master_handle_t::rxRemainingBytes

16.6.3.5.0.5.6 volatile uint32_t flexio_spi_master_handle_t::state

16.6.3.5.0.5.7 flexio_spi_shift_direction_t flexio_spi_master_handle_t::direction

16.6.3.5.0.5.8 flexio_spi_master_transfer_callback_t flexio_spi_master_handle_t::callback

16.6.3.5.0.5.9 void* flexio_spi_master_handle_t::userData

16.6.4 Macro Definition Documentation

16.6.4.1 #define FSL_FLEXIO_SPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

16.6.4.2 #define FLEXIO_SPI_DUMMYDATA (0xFFFFU)

16.6.5 Typedef Documentation

16.6.5.1 typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t

16.6.6 Enumeration Type Documentation

16.6.6.1 enum _flexio_spi_status

Enumerator

kStatus_FLEXIO_SPI_Busy FlexIO SPI is busy.
kStatus_FLEXIO_SPI_Idle SPI is idle.
kStatus_FLEXIO_SPI_Error FlexIO SPI error.

16.6.6.2 enum flexio_spi_clock_phase_t

Enumerator

kFLEXIO_SPI_ClockPhaseFirstEdge First edge on SPSCK occurs at the middle of the first cycle of a data transfer.

kFLEXIO_SPI_ClockPhaseSecondEdge First edge on SPSCK occurs at the start of the first cycle of a data transfer.

16.6.6.3 enum flexio_spi_shift_direction_t

Enumerator

kFLEXIO_SPI_MsbFirst Data transfers start with most significant bit.
kFLEXIO_SPI_LsbFirst Data transfers start with least significant bit.

16.6.6.4 enum flexio_spi_data_bitcount_mode_t

Enumerator

kFLEXIO_SPI_8BitMode 8-bit data transmission mode.
kFLEXIO_SPI_16BitMode 16-bit data transmission mode.

16.6.6.5 enum _flexio_spi_interrupt_enable

Enumerator

kFLEXIO_SPI_TxEmptyInterruptEnable Transmit buffer empty interrupt enable.
kFLEXIO_SPI_RxFullInterruptEnable Receive buffer full interrupt enable.

16.6.6.6 enum _flexio_spi_status_flags

Enumerator

kFLEXIO_SPI_TxBufferEmptyFlag Transmit buffer empty flag.
kFLEXIO_SPI_RxBufferFullFlag Receive buffer full flag.

16.6.6.7 enum _flexio_spi_dma_enable

Enumerator

kFLEXIO_SPI_TxDmaEnable Tx DMA request source.
kFLEXIO_SPI_RxDmaEnable Rx DMA request source.
kFLEXIO_SPI_DmaAllEnable All DMA request source.

16.6.6.8 enum _flexio_spi_transfer_flags

Enumerator

kFLEXIO_SPI_8bitMsb FlexIO SPI 8-bit MSB first.
kFLEXIO_SPI_8bitLsb FlexIO SPI 8-bit LSB first.
kFLEXIO_SPI_16bitMsb FlexIO SPI 16-bit MSB first.
kFLEXIO_SPI_16bitLsb FlexIO SPI 16-bit LSB first.

16.6.7 Function Documentation

16.6.7.1 void FLEXIO_SPI_MasterInit (FLEXIO_SPI_Type ∗ *base*, flexio_spi_master_config_t ∗ *masterConfig*, uint32_t *srcClock_Hz*)

The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_-MasterGetDefaultConfig().

Note

FlexIO SPI master only support CPOL = 0, which means clock inactive low.

Example

```
FLEXIO_SPI_Type spiDev = {
.flexioBase = FLEXIO,
.SDOPinIndex = 0,
.SDIPinIndex = 1,
.SCKPinIndex = 2,
.CSnPinIndex = 3,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_spi_master_config_t config = {
.enableMaster = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 500000,
.phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
.direction = kFLEXIO_SPI_MsbFirst,
.dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_MasterInit(&spiDev, &config, srcClock_Hz);
```

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
masterConfig	Pointer to the flexio_spi_master_config_t structure.
srcClock_Hz	FlexIO source clock in Hz.

16.6.7.2 void FLEXIO_SPI_MasterDeinit (FLEXIO_SPI_Type * *base*)

Parameters

base	Pointer to the FLEXIO_SPI_Type.

16.6.7.3 void FLEXIO_SPI_MasterGetDefaultConfig (flexio_spi_master_config_t * *masterConfig*)

The configuration can be used directly by calling the FLEXIO_SPI_MasterConfigure(). Example:

```
flexio_spi_master_config_t masterConfig;
FLEXIO_SPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

masterConfig	Pointer to the flexio_spi_master_config_t structure.

16.6.7.4 void FLEXIO_SPI_SlaveInit (FLEXIO_SPI_Type * *base*, flexio_spi_slave_config_t * *slaveConfig*)

The configuration structure can be filled by the user, or be set with default values by the FLEXIO_SPI_-SlaveGetDefaultConfig().

Note

Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. FlexIO SPI slave only support CPOL = 0, which means clock inactive low. Example

```
FLEXIO_SPI_Type spiDev = {
.flexioBase = FLEXIO,
.SDOPinIndex = 0,
.SDIPinIndex = 1,
.SCKPinIndex = 2,
.CSnPinIndex = 3,
.shifterIndex = {0,1},
.timerIndex = {0}
};
flexio_spi_slave_config_t config = {
.enableSlave = true,
.enableInDoze = false,
.enableInDebug = true,
.enableFastAccess = false,
.phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
.direction = kFLEXIO_SPI_MsbFirst,
.dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_SlaveInit(&spiDev, &config);
```

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
slaveConfig	Pointer to the flexio_spi_slave_config_t structure.

16.6.7.5 void FLEXIO_SPI_SlaveDeinit (FLEXIO_SPI_Type ∗ *base*)

Parameters

base	Pointer to the FLEXIO_SPI_Type.

16.6.7.6 void FLEXIO_SPI_SlaveGetDefaultConfig (flexio_spi_slave_config_t ∗ *slaveConfig*)

The configuration can be used directly for calling the FLEXIO_SPI_SlaveConfigure(). Example:

```
flexio_spi_slave_config_t slaveConfig;
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);
```

Parameters

slaveConfig	Pointer to the flexio_spi_slave_config_t structure.

16.6.7.7 uint32_t FLEXIO_SPI_GetStatusFlags (FLEXIO_SPI_Type ∗ *base*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.

Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- kFLEXIO_SPI_TxEmptyFlag
- kFLEXIO_SPI_RxEmptyFlag

16.6.7.8 void FLEXIO_SPI_ClearStatusFlags (FLEXIO_SPI_Type ∗ *base*, uint32_t *mask*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
mask	status flag The parameter can be any combination of the following values: • kFLEXIO_SPI_TxEmptyFlag • kFLEXIO_SPI_RxEmptyFlag

16.6.7.9 void FLEXIO_SPI_EnableInterrupts (FLEXIO_SPI_Type * *base*, uint32_t *mask*)

This function enables the FlexIO SPI interrupt.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
mask	interrupt source. The parameter can be any combination of the following values: • kFLEXIO_SPI_RxFullInterruptEnable • kFLEXIO_SPI_TxEmptyInterruptEnable

16.6.7.10 void FLEXIO_SPI_DisableInterrupts (FLEXIO_SPI_Type * *base*, uint32_t *mask*)

This function disables the FlexIO SPI interrupt.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
mask	interrupt source The parameter can be any combination of the following values: • kFLEXIO_SPI_RxFullInterruptEnable • kFLEXIO_SPI_TxEmptyInterruptEnable

16.6.7.11 void FLEXIO_SPI_EnableDMA (FLEXIO_SPI_Type * *base*, uint32_t *mask*, bool *enable*)

This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO_SPI_-TxEmptyFlag does/doesn't trigger the DMA request.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
mask	SPI DMA source.
enable	True means enable DMA, false means disable DMA.

16.6.7.12 static uint32_t FLEXIO_SPI_GetTxDataRegisterAddress (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*) [inline], [static]

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.

Returns

FlexIO SPI transmit data register address.

16.6.7.13 static uint32_t FLEXIO_SPI_GetRxDataRegisterAddress (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*) [inline], [static]

This function returns the SPI data register address, which is mainly used by DMA/eDMA.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.

Returns

FlexIO SPI receive data register address.

16.6.7.14 static void FLEXIO_SPI_Enable (FLEXIO_SPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to the FLEXIO_SPI_Type.
enable	True to enable, false to disable.

16.6.7.15 void FLEXIO_SPI_MasterSetBaudRate (FLEXIO_SPI_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClockHz*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
baudRate_Bps	Baud Rate needed in Hz.
srcClockHz	SPI source clock frequency in Hz.

16.6.7.16 static void FLEXIO_SPI_WriteData (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*, uint16_t *data*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.
data	8 bit/16 bit data.

16.6.7.17 static uint16_t FLEXIO_SPI_ReadData (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.

Returns

8 bit/16 bit data received.

16.6.7.18 void FLEXIO_SPI_WriteBlocking (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*, const uint8_t * *buffer*, size_t *size*)

Note

This function blocks using the polling method until all bytes have been sent.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.
buffer	The data bytes to send.
size	The number of data bytes to send.

16.6.7.19 void FLEXIO_SPI_ReadBlocking (FLEXIO_SPI_Type * *base*, flexio_spi_shift_direction_t *direction*, uint8_t * *buffer*, size_t *size*)

Note

This function blocks using the polling method until all bytes have been received.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
direction	Shift direction of MSB first or LSB first.
buffer	The buffer to store the received bytes.
size	The number of data bytes to be received.
direction	Shift direction of MSB first or LSB first.

16.6.7.20 void FLEXIO_SPI_MasterTransferBlocking (FLEXIO_SPI_Type * *base*, flexio_spi_transfer_t * *xfer*)

Note

This function blocks via polling until all bytes have been received.

Parameters

base	pointer to FLEXIO_SPI_Type structure
xfer	FlexIO SPI transfer structure, see flexio_spi_transfer_t.

16.6.7.21 status_t FLEXIO_SPI_MasterTransferCreateHandle (FLEXIO_SPI_Type * *base*, flexio_spi_master_handle_t * *handle*, flexio_spi_master_transfer_callback_t *callback*, void * *userData*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
callback	The callback function.
userData	The parameter of the callback function.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/ISR table out of range.

16.6.7.22 status_t FLEXIO_SPI_MasterTransferNonBlocking (FLEXIO_SPI_Type * *base*, flexio_spi_master_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_master_handle_t structure to store the transfer state.
xfer	FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	SPI is not idle, is running another transfer.

16.6.7.23 void FLEXIO_SPI_MasterTransferAbort (FLEXIO_SPI_Type * *base*, flexio_spi_master_handle_t * *handle*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

16.6.7.24 status_t FLEXIO_SPI_MasterTransferGetCount (FLEXIO_SPI_Type * *base*, flexio_spi_master_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_master_handle_t structure to store 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.

16.6.7.25 void FLEXIO_SPI_MasterTransferHandleIRQ (void * *spiType*, void * *spiHandle*)

Parameters

spiType	Pointer to the FLEXIO_SPI_Type structure.
spiHandle	Pointer to the flexio_spi_master_handle_t structure to store the transfer state.

16.6.7.26 status_t FLEXIO_SPI_SlaveTransferCreateHandle (FLEXIO_SPI_Type ∗ *base*, flexio_spi_slave_handle_t ∗ *handle*, flexio_spi_slave_transfer_callback_t *callback*, void ∗ *userData*)

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
callback	The callback function.
userData	The parameter of the callback function.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/ISR table out of range.

16.6.7.27 status_t FLEXIO_SPI_SlaveTransferNonBlocking (FLEXIO_SPI_Type * *base*, flexio_spi_slave_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

handle	Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.
base	Pointer to the FLEXIO_SPI_Type structure.
xfer	FlexIO SPI transfer structure. See flexio_spi_transfer_t.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	SPI is not idle; it is running another transfer.

16.6.7.28 static void FLEXIO_SPI_SlaveTransferAbort (FLEXIO_SPI_Type * *base*, flexio_spi_slave_handle_t * *handle*) [inline], [static]

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

16.6.7.29 static status_t FLEXIO_SPI_SlaveTransferGetCount (FLEXIO_SPI_Type * *base*, flexio_spi_slave_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

base	Pointer to the FLEXIO_SPI_Type structure.
handle	Pointer to the flexio_spi_slave_handle_t structure to store 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.

16.6.7.30 void FLEXIO_SPI_SlaveTransferHandleIRQ (void * *spiType*, void * *spiHandle*)

Parameters

spiType	Pointer to the FLEXIO_SPI_Type structure.
spiHandle	Pointer to the flexio_spi_slave_handle_t structure to store the transfer state.

16.6.8 FlexIO eDMA SPI Driver

16.6.8.1 Overview

Data Structures

- struct flexio_spi_master_edma_handle_t
  FlexIO SPI eDMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef
  flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t
  Slave handle is the same with master handle.
- typedef void(* flexio_spi_master_edma_transfer_callback_t)(FLEXIO_SPI_Type *base, flexio_-spi_master_edma_handle_t *handle, status_t status, void *userData)
  FlexIO SPI master callback for finished transmit.
- typedef void(* flexio_spi_slave_edma_transfer_callback_t)(FLEXIO_SPI_Type *base, flexio_spi-_slave_edma_handle_t *handle, status_t status, void *userData)
  FlexIO SPI slave callback for finished transmit.

eDMA Transactional

- status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA (FLEXIO_SPI_Type *base, flexio_-spi_master_edma_handle_t *handle, flexio_spi_master_edma_transfer_callback_t callback, void *userData, edma_handle_t *txHandle, edma_handle_t *rxHandle)
  Initializes the FlexIO SPI master eDMA handle.
- status_t FLEXIO_SPI_MasterTransferEDMA (FLEXIO_SPI_Type *base, flexio_spi_master_-edma_handle_t *handle, flexio_spi_transfer_t *xfer)
  Performs a non-blocking FlexIO SPI transfer using eDMA.
- void FLEXIO_SPI_MasterTransferAbortEDMA (FLEXIO_SPI_Type *base, flexio_spi_master_-edma_handle_t *handle)
  Aborts a FlexIO SPI transfer using eDMA.
- status_t FLEXIO_SPI_MasterTransferGetCountEDMA (FLEXIO_SPI_Type *base, flexio_spi_-master_edma_handle_t *handle, size_t *count)
  Gets the remaining bytes for FlexIO SPI eDMA transfer.
- static void FLEXIO_SPI_SlaveTransferCreateHandleEDMA (FLEXIO_SPI_Type *base, flexio-_spi_slave_edma_handle_t *handle, flexio_spi_slave_edma_transfer_callback_t callback, void *userData, edma_handle_t *txHandle, edma_handle_t *rxHandle)
  Initializes the FlexIO SPI slave eDMA handle.
- status_t FLEXIO_SPI_SlaveTransferEDMA (FLEXIO_SPI_Type *base, flexio_spi_slave_edma_-handle_t *handle, flexio_spi_transfer_t *xfer)
  Performs a non-blocking FlexIO SPI transfer using eDMA.
- static void FLEXIO_SPI_SlaveTransferAbortEDMA (FLEXIO_SPI_Type *base, flexio_spi_slave-_edma_handle_t *handle)
  Aborts a FlexIO SPI transfer using eDMA.

- static status_t FLEXIO_SPI_SlaveTransferGetCountEDMA (FLEXIO_SPI_Type ∗base, flexio_spi-_slave_edma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes to be transferred for FlexIO SPI eDMA.

16.6.8.2 Data Structure Documentation

16.6.8.2.1 struct _flexio_spi_master_edma_handle

typedef for flexio_spi_master_edma_handle_t in advance.

Data Fields

- size_t transferSize
  Total bytes to be transferred.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- bool txInProgress
  Send transfer in progress.
- bool rxInProgress
  Receive transfer in progress.
- edma_handle_t ∗ txHandle
  DMA handler for SPI send.
- edma_handle_t ∗ rxHandle
  DMA handler for SPI receive.
- flexio_spi_master_edma_transfer_callback_t callback
  Callback for SPI DMA transfer.
- void ∗ userData
  User Data for SPI DMA callback.

16.6.8.2.1.1 Field Documentation

16.6.8.2.1.1.1 size_t flexio_spi_master_edma_handle_t::transferSize

16.6.8.2.1.1.2 uint8_t flexio_spi_master_edma_handle_t::nbytes

16.6.8.3 Typedef Documentation

16.6.8.3.1 typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t

16.6.8.4 Function Documentation

16.6.8.4.1 status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_edma_handle_t * *handle*, flexio_spi_master_edma_transfer_-callback_t *callback*, void * *userData*, edma_handle_t * *txHandle*, edma_handle_t * *rxHandle*)

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

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
callback	SPI callback, NULL means no callback.
userData	callback function parameter.
txHandle	User requested eDMA handle for FlexIO SPI RX eDMA transfer.
rxHandle	User requested eDMA handle for FlexIO SPI TX eDMA transfer.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO SPI eDMA type/handle table out of range.

16.6.8.4.2 status_t FLEXIO_SPI_MasterTransferEDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_edma_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

Note

> This interface returns immediately after transfer initiates. Call FLEXIO_SPI_MasterGetTransfer-CountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_master_edma_handle_t structure to store the transfer state.
xfer	Pointer to FlexIO SPI transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	FlexIO SPI is not idle, is running another transfer.

16.6.8.4.3 void FLEXIO_SPI_MasterTransferAbortEDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_edma_handle_t * *handle*)

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI eDMA handle pointer.

16.6.8.4.4 status_t FLEXIO_SPI_MasterTransferGetCountEDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI eDMA handle pointer.
count	Number of bytes transferred so far by the non-blocking transaction.

16.6.8.4.5 static void FLEXIO_SPI_SlaveTransferCreateHandleEDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_edma_handle_t * *handle*, flexio_spi_slave_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *txHandle*, edma_handle_t * *rxHandle*) [inline], [static]

This function initializes the FlexIO SPI slave eDMA handle.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
callback	SPI callback, NULL means no callback.
userData	callback function parameter.
txHandle	User requested eDMA handle for FlexIO SPI TX eDMA transfer.
rxHandle	User requested eDMA handle for FlexIO SPI RX eDMA transfer.

16.6.8.4.6 status_t FLEXIO_SPI_SlaveTransferEDMA (FLEXIO_SPI_Type ∗ *base*, flexio_spi_slave_edma_handle_t ∗ *handle*, flexio_spi_transfer_t ∗ *xfer*)

Note

This interface returns immediately after transfer initiates. Call FLEXIO_SPI_SlaveGetTransfer-CountEDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.
xfer	Pointer to FlexIO SPI transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	FlexIO SPI is not idle, is running another transfer.

16.6.8.4.7 static void FLEXIO_SPI_SlaveTransferAbortEDMA (FLEXIO_SPI_Type ∗ *base*, flexio_spi_slave_edma_handle_t ∗ *handle*) [inline], [static]

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_edma_handle_t structure to store the transfer state.

16.6.8.4.8 static status_t FLEXIO_SPI_SlaveTransferGetCountEDMA (FLEXIO_SPI_Type * base, flexio_spi_slave_edma_handle_t * handle, size_t * count) [inline], [static]

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI eDMA handle pointer.
count	Number of bytes transferred so far by the non-blocking transaction.

16.6.9 FlexIO DMA SPI Driver

16.6.9.1 Overview

Data Structures

- struct flexio_spi_master_dma_handle_t
  FlexIO SPI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef
  flexio_spi_master_dma_handle_t flexio_spi_slave_dma_handle_t
  Slave handle is the same with master handle.
- typedef void(∗ flexio_spi_master_dma_transfer_callback_t)(FLEXIO_SPI_Type ∗base, flexio_spi_master_dma_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO SPI master callback for finished transmit.
- typedef void(∗ flexio_spi_slave_dma_transfer_callback_t)(FLEXIO_SPI_Type ∗base, flexio_spi_slave_dma_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO SPI slave callback for finished transmit.

DMA Transactional

- status_t FLEXIO_SPI_MasterTransferCreateHandleDMA (FLEXIO_SPI_Type ∗base, flexio_spi_master_dma_handle_t ∗handle, flexio_spi_master_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txHandle, dma_handle_t ∗rxHandle)
  Initializes the FLEXO SPI master DMA handle.
- status_t FLEXIO_SPI_MasterTransferDMA (FLEXIO_SPI_Type ∗base, flexio_spi_master_dma_handle_t ∗handle, flexio_spi_transfer_t ∗xfer)
  Performs a non-blocking FlexIO SPI transfer using DMA.
- void FLEXIO_SPI_MasterTransferAbortDMA (FLEXIO_SPI_Type ∗base, flexio_spi_master_dma_handle_t ∗handle)
  Aborts a FlexIO SPI transfer using DMA.
- status_t FLEXIO_SPI_MasterTransferGetCountDMA (FLEXIO_SPI_Type ∗base, flexio_spi_master_dma_handle_t ∗handle, size_t ∗count)
  Gets the remaining bytes for FlexIO SPI DMA transfer.
- static void FLEXIO_SPI_SlaveTransferCreateHandleDMA (FLEXIO_SPI_Type ∗base, flexio_spi_slave_dma_handle_t ∗handle, flexio_spi_slave_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txHandle, dma_handle_t ∗rxHandle)
  Initializes the FlexIO SPI slave DMA handle.
- status_t FLEXIO_SPI_SlaveTransferDMA (FLEXIO_SPI_Type ∗base, flexio_spi_slave_dma_handle_t ∗handle, flexio_spi_transfer_t ∗xfer)
  Performs a non-blocking FlexIO SPI transfer using DMA.
- static void FLEXIO_SPI_SlaveTransferAbortDMA (FLEXIO_SPI_Type ∗base, flexio_spi_slave_dma_handle_t ∗handle)
  Aborts a FlexIO SPI transfer using DMA.

- static status_t FLEXIO_SPI_SlaveTransferGetCountDMA (FLEXIO_SPI_Type *base, flexio_spi_-slave_dma_handle_t *handle, size_t *count)
  Gets the remaining bytes to be transferred for FlexIO SPI DMA.

16.6.9.2 Data Structure Documentation

16.6.9.2.1 struct _flexio_spi_master_dma_handle

typedef for flexio_spi_master_dma_handle_t in advance.

Data Fields

- size_t transferSize
  Total bytes to be transferred.
- bool txInProgress
  Send transfer in progress.
- bool rxInProgress
  Receive transfer in progress.
- dma_handle_t * txHandle
  DMA handler for SPI send.
- dma_handle_t * rxHandle
  DMA handler for SPI receive.
- flexio_spi_master_dma_transfer_callback_t callback
  Callback for SPI DMA transfer.
- void * userData
  User Data for SPI DMA callback.

16.6.9.2.1.1 Field Documentation

16.6.9.2.1.1.1 size_t flexio_spi_master_dma_handle_t::transferSize

16.6.9.3 Typedef Documentation

16.6.9.3.1 typedef flexio_spi_master_dma_handle_t flexio_spi_slave_dma_handle_t

16.6.9.4 Function Documentation

16.6.9.4.1 status_t FLEXIO_SPI_MasterTransferCreateHandleDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_dma_handle_t * *handle*, flexio_spi_master_dma_transfer_callback-_t *callback*, void * *userData*, dma_handle_t * *txHandle*, dma_handle_t * *rxHandle*)

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

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_master_dma_handle_t structure to store the transfer state.
callback	SPI callback, NULL means no callback.
userData	callback function parameter.
txHandle	User requested DMA handle for FlexIO SPI RX DMA transfer.
rxHandle	User requested DMA handle for FlexIO SPI TX DMA transfer.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO SPI DMA type/handle table out of range.

16.6.9.4.2 status_t FLEXIO_SPI_MasterTransferDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_dma_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

Note

> This interface returned immediately after transfer initiates. Call FLEXIO_SPI_MasterGetTransfer-CountDMA to poll the transfer status to check whether the FlexIO SPI transfer is finished.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_master_dma_handle_t structure to store the transfer state.
xfer	Pointer to FlexIO SPI transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	FlexIO SPI is not idle, is running another transfer.

16.6.9.4.3 void FLEXIO_SPI_MasterTransferAbortDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_dma_handle_t * *handle*)

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI DMA handle pointer.

16.6.9.4.4 status_t FLEXIO_SPI_MasterTransferGetCountDMA (FLEXIO_SPI_Type * *base*, flexio_spi_master_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI DMA handle pointer.
count	Number of bytes transferred so far by the non-blocking transaction.

16.6.9.4.5 static void FLEXIO_SPI_SlaveTransferCreateHandleDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_dma_handle_t * *handle*, flexio_spi_slave_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *txHandle*, dma_handle_t * *rxHandle*) [inline], [static]

This function initializes the FlexIO SPI slave DMA handle.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_dma_handle_t structure to store the transfer state.
callback	SPI callback, NULL means no callback.
userData	callback function parameter.
txHandle	User requested DMA handle for FlexIO SPI TX DMA transfer.
rxHandle	User requested DMA handle for FlexIO SPI RX DMA transfer.

16.6.9.4.6 status_t FLEXIO_SPI_SlaveTransferDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_dma_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

Note

This interface returns immediately after transfer initiates. Call FLEXIO_SPI_SlaveGetTransfer-CountDMA to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_dma_handle_t structure to store the transfer state.
xfer	Pointer to FlexIO SPI transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_FLEXIO_SPI_-Busy	FlexIO SPI is not idle, is running another transfer.

16.6.9.4.7 static void FLEXIO_SPI_SlaveTransferAbortDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_dma_handle_t * *handle*) [inline], [static]

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	Pointer to flexio_spi_slave_dma_handle_t structure to store the transfer state.

16.6.9.4.8 static status_t FLEXIO_SPI_SlaveTransferGetCountDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_dma_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

base	Pointer to FLEXIO_SPI_Type structure.
handle	FlexIO SPI DMA handle pointer.
count	Number of bytes transferred so far by the non-blocking transaction.

16.7 FlexIO UART Driver

16.7.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) function using the Flexible I/O.

FlexIO UART driver includes functional APIs and transactional APIs. Functional APIs target low-level APIs. Functional APIs can be used for the FlexIO UART initialization/configuration/operation for optimization/customization purpose. Using the functional APIs requires the knowledge of the FlexIO UART peripheral and how to organize functional APIs to meet the application requirements. All functional API use the FLEXIO_UART_Type * as the first parameter. FlexIO UART functional operation groups provide the functional APIs set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and also in the application if the code size and performance of transactional APIs satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the flexio_uart_handle_t as the second parameter. Initialize the handle by calling the FLEXIO_UART_TransferCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions FLEXIO_UART_SendNonBlocking() and FLEXIO_UART_ReceiveNonBlocking() set up an interrupt for data transfer. When the transfer is complete, the upper layer is notified through a callback function with the kStatus_FLEXIO_UART_TxIdle and kStatus_FLEXIO_UART_RxIdle status.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size through calling the FLEXIO_UART_InstallRingBuffer(). When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The function FLEXIO_UART_ReceiveNonBlocking() first gets data the from the ring buffer. If ring buffer does not have enough data, the function returns the data to the ring buffer and saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the statuskStatus_FLEXIO_UART_RxIdle status.

If the receive ring buffer is full, the upper layer is informed through a callback with status kStatus_FLEXIO_UART_RxRingBufferOverrun. In the callback function, the upper layer reads data from the ring buffer. If not, the oldest data is overwritten by the new data.

The ring buffer size is specified when calling the FLEXIO_UART_InstallRingBuffer. Note that one byte is reserved for the ring buffer maintenance. Create a handle as follows.

```
FLEXIO_UART_InstallRingBuffer(&uartDev, &handle, &ringBuffer, 32);
```

In this example, the buffer size is 32. However, only 31 bytes are used for saving data.

16.7.2 Typical use case

16.7.2.1 FlexIO UART send/receive using a polling method

```
uint8_t ch;
```

```
FLEXIO_UART_Type uartDev;
status_t result = kStatus_Success;
flexio_uart_user_config user_config;
FLEXIO_UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableUart = true;

uartDev.flexioBase = BOARD_FLEXIO_BASE;
uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
uartDev.shifterIndex[0] = 0U;
uartDev.shifterIndex[1] = 1U;
uartDev.timerIndex[0] = 0U;
uartDev.timerIndex[1] = 1U;

result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
//Check if configuration is correct.
if(result != kStatus_Success)
{
    return;
}
FLEXIO_UART_WriteBlocking(&uartDev, txbuff, sizeof(txbuff));

while(1)
{
    FLEXIO_UART_ReadBlocking(&uartDev, &ch, 1);
    FLEXIO_UART_WriteBlocking(&uartDev, &ch, 1);
}
```

16.7.2.2 FlexIO UART send/receive using an interrupt method

```
FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
      status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_FLEXIO_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    FLEXIO_UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableUart = true;
```

```
    uartDev.flexioBase = BOARD_FLEXIO_BASE;
    uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
    uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
    uartDev.shifterIndex[0] = 0U;
    uartDev.shifterIndex[1] = 1U;
    uartDev.timerIndex[0] = 0U;
    uartDev.timerIndex[1] = 1U;

    result = FLEXIO_UART_Init(&uartDev, &user_config, 120000000U);
    //Check if configuration is correct.
    if(result != kStatus_Success)
    {
        return;
    }

    FLEXIO_UART_TransferCreateHandle(&uartDev, &g_uartHandle,
      FLEXIO_UART_UserCallback, NULL);

    // Prepares to send.
    sendXfer.data = sendData;
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Sends out.
    FLEXIO_UART_SendNonBlocking(&uartDev, &g_uartHandle, &sendXfer);

    // Send finished.
    while (!txFinished)
    {
    }

    // Prepares to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receives.
    FLEXIO_UART_ReceiveNonBlocking(&uartDev, &g_uartHandle, &receiveXfer, NULL);

    // Receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

16.7.2.3 FlexIO UART receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE     32

FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
      status_t status, void *userData)
{
```

```
    userData = userData;

    if (kStatus_FLEXIO_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    FLEXIO_UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableUart = true;

    uartDev.flexioBase = BOARD_FLEXIO_BASE;
    uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
    uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
    uartDev.shifterIndex[0] = 0U;
    uartDev.shifterIndex[1] = 1U;
    uartDev.timerIndex[0] = 0U;
    uartDev.timerIndex[1] = 1U;

    result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
    //Check if configuration is correct.
    if(result != kStatus_Success)
    {
        return;
    }

    FLEXIO_UART_TransferCreateHandle(&uartDev, &g_uartHandle,
      FLEXIO_UART_UserCallback, NULL);
    FLEXIO_UART_InstallRingBuffer(&uartDev, &g_uartHandle, ringBuffer, RING_BUFFER_SIZE);

    // Receive is working in the background to the ring buffer.

    // Prepares to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = RX_DATA_SIZE;
    rxFinished = false;

    // Receives.
    FLEXIO_UART_ReceiveNonBlocking(&uartDev, &g_uartHandle, &receiveXfer, &bytesRead);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data. */
    {
        ;
    }
    else
    {
        if (bytesRead) /* Received some data, process first. */
        {
            ;
        }

        // Receive finished.
        while (!rxFinished)
        {
        }
    }

    // ...
}
```

16.7.2.4 FlexIO UART send/receive using a DMA method

```
FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
      status_t status, void *userData)
{
    userData = userData;

    if (kStatus_FLEXIO_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_FLEXIO_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    FLEXIO_UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableUart = true;

    uartDev.flexioBase = BOARD_FLEXIO_BASE;
    uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
    uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
    uartDev.shifterIndex[0] = 0U;
    uartDev.shifterIndex[1] = 1U;
    uartDev.timerIndex[0] = 0U;
    uartDev.timerIndex[1] = 1U;
    result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
    //Check if configuration is correct.
    if(result != kStatus_Success)
    {
        return;
    }

    /*Initializes the DMA for the example*/
    DMAMGR_Init();

    dma_request_source_tx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + uartDev.
      shifterIndex[0]);
    dma_request_source_rx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + uartDev.
      shifterIndex[1]);

    /* Requests DMA channels for transmit and receive. */
    DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_tx, 0, &g_uartTxDmaHandle);
    DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_rx, 1, &g_uartRxDmaHandle);

    FLEXIO_UART_TransferCreateHandleDMA(&uartDev, &g_uartHandle,
      FLEXIO_UART_UserCallback, NULL, &g_uartTxDmaHandle, &g_uartRxDmaHandle);
```

```
    // Prepares to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Sends out.
    FLEXIO_UART_SendDMA(&uartDev, &g_uartHandle, &sendXfer);

    // Send finished.
    while (!txFinished)
    {
    }

    // Prepares to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receives.
    FLEXIO_UART_ReceiveDMA(&uartDev, &g_uartHandle, &receiveXfer, NULL);

    // Receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

Modules

- FlexIO DMA UART Driver
- FlexIO eDMA UART Driver

Data Structures

- struct FLEXIO_UART_Type
  Define FlexIO UART access structure typedef. More...
- struct flexio_uart_config_t
  Define FlexIO UART user configuration structure. More...
- struct flexio_uart_transfer_t
  Define FlexIO UART transfer structure. More...
- struct flexio_uart_handle_t
  Define FLEXIO UART handle structure. More...

Typedefs

- typedef void(∗ flexio_uart_transfer_callback_t)(FLEXIO_UART_Type ∗base, flexio_uart_handle_t ∗handle, status_t status, void ∗userData)
  FlexIO UART transfer callback function.

Enumerations

- enum _flexio_uart_status {
  kStatus_FLEXIO_UART_TxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_UART, 0),
  kStatus_FLEXIO_UART_RxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_UART, 1),
  kStatus_FLEXIO_UART_TxIdle = MAKE_STATUS(kStatusGroup_FLEXIO_UART, 2),
  kStatus_FLEXIO_UART_RxIdle = MAKE_STATUS(kStatusGroup_FLEXIO_UART, 3),
  kStatus_FLEXIO_UART_ERROR = MAKE_STATUS(kStatusGroup_FLEXIO_UART, 4),
  kStatus_FLEXIO_UART_RxRingBufferOverrun,
  kStatus_FLEXIO_UART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_FLEXIO_UA-RT, 6) }
  Error codes for the UART driver.
- enum flexio_uart_bit_count_per_char_t {
  kFLEXIO_UART_7BitsPerChar = 7U,
  kFLEXIO_UART_8BitsPerChar = 8U,
  kFLEXIO_UART_9BitsPerChar = 9U }
  FlexIO UART bit count per char.
- enum _flexio_uart_interrupt_enable {
  kFLEXIO_UART_TxDataRegEmptyInterruptEnable = 0x1U,
  kFLEXIO_UART_RxDataRegFullInterruptEnable = 0x2U }
  Define FlexIO UART interrupt mask.
- enum _flexio_uart_status_flags {
  kFLEXIO_UART_TxDataRegEmptyFlag = 0x1U,
  kFLEXIO_UART_RxDataRegFullFlag = 0x2U,
  kFLEXIO_UART_RxOverRunFlag = 0x4U }
  Define FlexIO UART status mask.

Driver version

- #define FSL_FLEXIO_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
  FlexIO UART driver version 2.1.2.

Initialization and deinitialization

- status_t FLEXIO_UART_Init (FLEXIO_UART_Type ∗base, const flexio_uart_config_t ∗userConfig, uint32_t srcClock_Hz)
  Ungates the FlexIO clock, resets the FlexIO module, configures FlexIO UART hardware, and configures the FlexIO UART with FlexIO UART configuration.
- void FLEXIO_UART_Deinit (FLEXIO_UART_Type ∗base)
  Disables the FlexIO UART and gates the FlexIO clock.
- void FLEXIO_UART_GetDefaultConfig (flexio_uart_config_t ∗userConfig)
  Gets the default configuration to configure the FlexIO UART.

Status

- uint32_t FLEXIO_UART_GetStatusFlags (FLEXIO_UART_Type *base)
  Gets the FlexIO UART status flags.
- void FLEXIO_UART_ClearStatusFlags (FLEXIO_UART_Type *base, uint32_t mask)
  Gets the FlexIO UART status flags.

Interrupts

- void FLEXIO_UART_EnableInterrupts (FLEXIO_UART_Type *base, uint32_t mask)
  Enables the FlexIO UART interrupt.
- void FLEXIO_UART_DisableInterrupts (FLEXIO_UART_Type *base, uint32_t mask)
  Disables the FlexIO UART interrupt.

DMA Control

- static uint32_t FLEXIO_UART_GetTxDataRegisterAddress (FLEXIO_UART_Type *base)
  Gets the FlexIO UARt transmit data register address.
- static uint32_t FLEXIO_UART_GetRxDataRegisterAddress (FLEXIO_UART_Type *base)
  Gets the FlexIO UART receive data register address.
- static void FLEXIO_UART_EnableTxDMA (FLEXIO_UART_Type *base, bool enable)
  Enables/disables the FlexIO UART transmit DMA.
- static void FLEXIO_UART_EnableRxDMA (FLEXIO_UART_Type *base, bool enable)
  Enables/disables the FlexIO UART receive DMA.

Bus Operations

- static void FLEXIO_UART_Enable (FLEXIO_UART_Type *base, bool enable)
  Enables/disables the FlexIO UART module operation.
- static void FLEXIO_UART_WriteByte (FLEXIO_UART_Type *base, const uint8_t *buffer)
  Writes one byte of data.
- static void FLEXIO_UART_ReadByte (FLEXIO_UART_Type *base, uint8_t *buffer)
  Reads one byte of data.
- void FLEXIO_UART_WriteBlocking (FLEXIO_UART_Type *base, const uint8_t *txData, size_t txSize)
  Sends a buffer of data bytes.
- void FLEXIO_UART_ReadBlocking (FLEXIO_UART_Type *base, uint8_t *rxData, size_t rxSize)
  Receives a buffer of bytes.

Transactional

- status_t FLEXIO_UART_TransferCreateHandle (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, flexio_uart_transfer_callback_t callback, void *userData)
  Initializes the UART handle.

- void FLEXIO_UART_TransferStartRingBuffer (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void FLEXIO_UART_TransferStopRingBuffer (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- status_t FLEXIO_UART_TransferSendNonBlocking (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, flexio_uart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void FLEXIO_UART_TransferAbortSend (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle)
  Aborts the interrupt-driven data transmit.
- status_t FLEXIO_UART_TransferGetSendCount (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, size_t *count)
  Gets the number of bytes sent.
- status_t FLEXIO_UART_TransferReceiveNonBlocking (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, flexio_uart_transfer_t *xfer, size_t *receivedBytes)
  Receives a buffer of data using the interrupt method.
- void FLEXIO_UART_TransferAbortReceive (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle)
  Aborts the receive data which was using IRQ.
- status_t FLEXIO_UART_TransferGetReceiveCount (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, size_t *count)
  Gets the number of bytes received.
- void FLEXIO_UART_TransferHandleIRQ (void *uartType, void *uartHandle)
  FlexIO UART IRQ handler function.

16.7.3 Data Structure Documentation

16.7.3.1 struct FLEXIO_UART_Type

Data Fields

- FLEXIO_Type * flexioBase
  FlexIO base pointer.
- uint8_t TxPinIndex
  Pin select for UART_Tx.
- uint8_t RxPinIndex
  Pin select for UART_Rx.
- uint8_t shifterIndex [2]
  Shifter index used in FlexIO UART.
- uint8_t timerIndex [2]
  Timer index used in FlexIO UART.

16.7.3.1.0.1 Field Documentation

16.7.3.1.0.1.1 FLEXIO_Type* FLEXIO_UART_Type::flexioBase

16.7.3.1.0.1.2 uint8_t FLEXIO_UART_Type::TxPinIndex

16.7.3.1.0.1.3 uint8_t FLEXIO_UART_Type::RxPinIndex

16.7.3.1.0.1.4 uint8_t FLEXIO_UART_Type::shifterIndex[2]

16.7.3.1.0.1.5 uint8_t FLEXIO_UART_Type::timerIndex[2]

16.7.3.2 struct flexio_uart_config_t

Data Fields

- bool enableUart
  Enable/disable FlexIO UART TX & RX.
- bool enableInDoze
  Enable/disable FlexIO operation in doze mode.
- bool enableInDebug
  Enable/disable FlexIO operation in debug mode.
- bool enableFastAccess
  `Enable/disable fast access to FlexIO registers,`
  fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.
- uint32_t baudRate_Bps
  Baud rate in Bps.
- flexio_uart_bit_count_per_char_t bitCountPerChar
  number of bits, 7/8/9 -bit

16.7.3.2.0.2 Field Documentation

16.7.3.2.0.2.1 bool flexio_uart_config_t::enableUart

16.7.3.2.0.2.2 bool flexio_uart_config_t::enableFastAccess

16.7.3.2.0.2.3 uint32_t flexio_uart_config_t::baudRate_Bps

16.7.3.3 struct flexio_uart_transfer_t

Data Fields

- uint8_t * data
  Transfer buffer.
- size_t dataSize
  Transfer size.

16.7.3.4 struct _flexio_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.
- uint8_t ∗volatile rxData
  Address of remaining data to receive.
- volatile size_t rxDataSize
  Size of the remaining data to receive.
- size_t txDataSizeAll
  Total bytes to be sent.
- size_t rxDataSizeAll
  Total bytes to be received.
- 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.
- flexio_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.

16.7.3.4.0.3 Field Documentation

16.7.3.4.0.3.1 uint8_t* volatile flexio_uart_handle_t::txData

16.7.3.4.0.3.2 volatile size_t flexio_uart_handle_t::txDataSize

16.7.3.4.0.3.3 uint8_t* volatile flexio_uart_handle_t::rxData

16.7.3.4.0.3.4 volatile size_t flexio_uart_handle_t::rxDataSize

16.7.3.4.0.3.5 size_t flexio_uart_handle_t::txDataSizeAll

16.7.3.4.0.3.6 size_t flexio_uart_handle_t::rxDataSizeAll

16.7.3.4.0.3.7 uint8_t* flexio_uart_handle_t::rxRingBuffer

16.7.3.4.0.3.8 size_t flexio_uart_handle_t::rxRingBufferSize

16.7.3.4.0.3.9 volatile uint16_t flexio_uart_handle_t::rxRingBufferHead

16.7.3.4.0.3.10 volatile uint16_t flexio_uart_handle_t::rxRingBufferTail

16.7.3.4.0.3.11 flexio_uart_transfer_callback_t flexio_uart_handle_t::callback

16.7.3.4.0.3.12 void* flexio_uart_handle_t::userData

16.7.3.4.0.3.13 volatile uint8_t flexio_uart_handle_t::txState

16.7.4 Macro Definition Documentation

16.7.4.1 #define FSL_FLEXIO_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))

16.7.5 Typedef Documentation

16.7.5.1 typedef void(* flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)

16.7.6 Enumeration Type Documentation

16.7.6.1 enum _flexio_uart_status

Enumerator

kStatus_FLEXIO_UART_TxBusy Transmitter is busy.
kStatus_FLEXIO_UART_RxBusy Receiver is busy.
kStatus_FLEXIO_UART_TxIdle UART transmitter is idle.
kStatus_FLEXIO_UART_RxIdle UART receiver is idle.
kStatus_FLEXIO_UART_ERROR ERROR happens on UART.

kStatus_FLEXIO_UART_RxRingBufferOverrun UART RX software ring buffer overrun.
kStatus_FLEXIO_UART_RxHardwareOverrun UART RX receiver overrun.

16.7.6.2 enum flexio_uart_bit_count_per_char_t

Enumerator

kFLEXIO_UART_7BitsPerChar 7-bit data characters
kFLEXIO_UART_8BitsPerChar 8-bit data characters
kFLEXIO_UART_9BitsPerChar 9-bit data characters

16.7.6.3 enum _flexio_uart_interrupt_enable

Enumerator

kFLEXIO_UART_TxDataRegEmptyInterruptEnable Transmit buffer empty interrupt enable.
kFLEXIO_UART_RxDataRegFullInterruptEnable Receive buffer full interrupt enable.

16.7.6.4 enum _flexio_uart_status_flags

Enumerator

kFLEXIO_UART_TxDataRegEmptyFlag Transmit buffer empty flag.
kFLEXIO_UART_RxDataRegFullFlag Receive buffer full flag.
kFLEXIO_UART_RxOverRunFlag Receive buffer over run flag.

16.7.7 Function Documentation

16.7.7.1 status_t FLEXIO_UART_Init (FLEXIO_UART_Type * *base*, const flexio_uart_config_t * *userConfig*, uint32_t *srcClock_Hz*)

The configuration structure can be filled by the user or be set with default values by FLEXIO_UART_GetDefaultConfig().

Example

```
FLEXIO_UART_Type base = {
.flexioBase = FLEXIO,
.TxPinIndex = 0,
.RxPinIndex = 1,
.shifterIndex = {0,1},
.timerIndex = {0,1}
};
flexio_uart_config_t config = {
.enableInDoze = false,
```

```
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 115200U,
.bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
userConfig	Pointer to the flexio_uart_config_t structure.
srcClock_Hz	FlexIO source clock in Hz.

Return values

kStatus_Success	Configuration success
kStatus_InvalidArgument	Buadrate configuration out of range

16.7.7.2 void FLEXIO_UART_Deinit (FLEXIO_UART_Type ∗ *base*)

Note

After calling this API, call the FLEXO_UART_Init to use the FlexIO UART module.

Parameters

base	Pointer to FLEXIO_UART_Type structure

16.7.7.3 void FLEXIO_UART_GetDefaultConfig (flexio_uart_config_t ∗ *userConfig*)

The configuration can be used directly for calling the FLEXIO_UART_Init(). Example:

```
flexio_uart_config_t config;
FLEXIO_UART_GetDefaultConfig(&userConfig);
```

Parameters

userConfig	Pointer to the flexio_uart_config_t structure.

16.7.7.4 uint32_t FLEXIO_UART_GetStatusFlags (FLEXIO_UART_Type ∗ *base*)

Parameters

base	Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART status flags.

16.7.7.5 void FLEXIO_UART_ClearStatusFlags (FLEXIO_UART_Type ∗ *base,* uint32_t *mask*)

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
mask	Status flag. The parameter can be any combination of the following values: • kFLEXIO_UART_TxDataRegEmptyFlag • kFLEXIO_UART_RxEmptyFlag • kFLEXIO_UART_RxOverRunFlag

16.7.7.6 void FLEXIO_UART_EnableInterrupts (FLEXIO_UART_Type ∗ *base,* uint32_t *mask*)

This function enables the FlexIO UART interrupt.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
mask	Interrupt source.

16.7.7.7 void FLEXIO_UART_DisableInterrupts (FLEXIO_UART_Type ∗ *base,* uint32_t *mask*)

This function disables the FlexIO UART interrupt.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
mask	Interrupt source.

16.7.7.8 static uint32_t FLEXIO_UART_GetTxDataRegisterAddress (FLEXIO_UART_Type * *base*) [inline], [static]

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART transmit data register address.

16.7.7.9 static uint32_t FLEXIO_UART_GetRxDataRegisterAddress (FLEXIO_UART_Type * *base*) [inline], [static]

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART receive data register address.

16.7.7.10 static void FLEXIO_UART_EnableTxDMA (FLEXIO_UART_Type * *base*, bool *enable*) [inline], [static]

This function enables/disables the FlexIO UART Tx DMA, which means asserting the kFLEXIO_UART_TxDataRegEmptyFlag does/doesn't trigger the DMA request.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
enable	True to enable, false to disable.

16.7.7.11 static void FLEXIO_UART_EnableRxDMA (FLEXIO_UART_Type * *base*, bool *enable*) [inline], [static]

This function enables/disables the FlexIO UART Rx DMA, which means asserting kFLEXIO_UART_-RxDataRegFullFlag does/doesn't trigger the DMA request.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
enable	True to enable, false to disable.

16.7.7.12 static void FLEXIO_UART_Enable (FLEXIO_UART_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to the FLEXIO_UART_Type.
enable	True to enable, false to disable.

16.7.7.13 static void FLEXIO_UART_WriteByte (FLEXIO_UART_Type * *base*, const uint8_t * *buffer*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is put into the data register. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
buffer	The data bytes to send.

16.7.7.14 static void FLEXIO_UART_ReadByte (FLEXIO_UART_Type * *base*, uint8_t * *buffer*) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
buffer	The buffer to store the received bytes.

16.7.7.15 void FLEXIO_UART_WriteBlocking (FLEXIO_UART_Type * *base*, const uint8_t * *txData*, size_t *txSize*)

Note

This function blocks using the polling method until all bytes have been sent.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
txData	The data bytes to send.
txSize	The number of data bytes to send.

16.7.7.16 void FLEXIO_UART_ReadBlocking (FLEXIO_UART_Type * *base*, uint8_t * *rxData*, size_t *rxSize*)

Note

This function blocks using the polling method until all bytes have been received.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
rxData	The buffer to store the received bytes.
rxSize	The number of data bytes to be received.

16.7.7.17 status_t FLEXIO_UART_TransferCreateHandle (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, flexio_uart_transfer_callback_t *callback*, void * *userData*)

This function initializes the FlexIO UART handle, which can be used for other FlexIO UART transactional APIs. Call this API once to get the initialized handle.

The UART driver supports the "background" receiving, which means that users can set up a RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn't call the FLEXIO_UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as `ringBuffer`.

Parameters

base	to FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
callback	The callback function.
userData	The parameter of the callback function.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO type/handle/ISR table out of range.

16.7.7.18 void FLEXIO_UART_TransferStartRingBuffer (FLEXIO_UART_Type * *base*, flexio_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 is stored into the ring buffer even when the user doesn't call the UART_ReceiveNonBlocking() API. If there is already data received in the ring buffer, users can get the received data from the ring buffer directly.

Note

> When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
ringBuffer	Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
ringBufferSize	Size of the ring buffer.

16.7.7.19 void FLEXIO_UART_TransferStopRingBuffer (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.

16.7.7.20 status_t FLEXIO_UART_TransferSendNonBlocking (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, flexio_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 ISR, the FlexIO UART driver calls the callback function and passes the kStatus_FLEXIO_UART_TxIdle as status parameter.

Note

> The kStatus_FLEXIO_UART_TxIdle is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
xfer	FlexIO UART transfer structure. See flexio_uart_transfer_t.

Return values

kStatus_Success	Successfully starts the data transmission.
kStatus_UART_TxBusy	Previous transmission still not finished, data not written to the TX register.

16.7.7.21 void FLEXIO_UART_TransferAbortSend (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*)

This function aborts the interrupt-driven data sending. Get the remainBytes to find out how many bytes are still not sent out.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.

16.7.7.22 status_t FLEXIO_UART_TransferGetSendCount (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes sent driven by interrupt.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
count	Number of bytes sent so far by the non-blocking transaction.

Return values

kStatus_NoTransferIn-Progress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

16.7.7.23 status_t FLEXIO_UART_TransferReceiveNonBlocking (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, flexio_uart_transfer_t * *xfer*, size_t * *receivedBytes*)

This function receives data using the 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 ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in ring buffer is not enough to read, the receive request is saved by the UART driver. When 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, if the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer, the 5 bytes are copied to xfer->data. This function returns with the parameter `receivedBytes` set to 5. For the last 5 bytes, newly arrived data is saved from the xfer->data[5]. When 5 bytes are received, the UART driver notifies upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to xfer->data. When all data is received, the upper layer is notified.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
xfer	UART transfer structure. See flexio_uart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into the transmit queue.
kStatus_FLEXIO_UART-_RxBusy	Previous receive request is not finished.

16.7.7.24 void FLEXIO_UART_TransferAbortReceive (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*)

This function aborts the receive data which was using IRQ.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.

16.7.7.25 status_t FLEXIO_UART_TransferGetReceiveCount (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes received driven by interrupt.

Parameters

base	Pointer to the FLEXIO_UART_Type structure.
handle	Pointer to the flexio_uart_handle_t structure to store the transfer state.
count	Number of bytes received so far by the non-blocking transaction.

Return values

kStatus_NoTransferIn-Progress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

16.7.7.26 void FLEXIO_UART_TransferHandleIRQ (void * *uartType*, void * *uartHandle*)

This function processes the FlexIO UART transmit and receives the IRQ request.

Parameters

uartType	Pointer to the FLEXIO_UART_Type structure.
uartHandle	Pointer to the flexio_uart_handle_t structure to store the transfer state.

16.7.8 FlexIO eDMA UART Driver

16.7.8.1 Overview

Data Structures

- struct flexio_uart_edma_handle_t
  UART eDMA handle. More...

Typedefs

- typedef void(∗ flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

eDMA transactional

- status_t FLEXIO_UART_TransferCreateHandleEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, flexio_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 FLEXIO_UART_TransferSendEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, flexio_uart_transfer_t ∗xfer)
  Sends data using eDMA.
- status_t FLEXIO_UART_TransferReceiveEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, flexio_uart_transfer_t ∗xfer)
  Receives data using eDMA.
- void FLEXIO_UART_TransferAbortSendEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle)
  Aborts the sent data which using eDMA.
- void FLEXIO_UART_TransferAbortReceiveEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle)
  Aborts the receive data which using eDMA.
- status_t FLEXIO_UART_TransferGetSendCountEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, size_t ∗count)
  Gets the number of bytes sent out.
- status_t FLEXIO_UART_TransferGetReceiveCountEDMA (FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, size_t ∗count)
  Gets the number of bytes received.

16.7.8.2 Data Structure Documentation

16.7.8.2.1 struct _flexio_uart_edma_handle

Data Fields

- flexio_uart_edma_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- size_t txDataSizeAll
  Total bytes to be sent.
- size_t rxDataSizeAll
  Total bytes to be received.
- edma_handle_t * txEdmaHandle
  The eDMA TX channel used.
- edma_handle_t * rxEdmaHandle
  The eDMA RX channel used.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

16.7.8.2.1.1 Field Documentation

16.7.8.2.1.1.1 flexio_uart_edma_transfer_callback_t flexio_uart_edma_handle_t::callback

16.7.8.2.1.1.2 void∗ flexio_uart_edma_handle_t::userData

16.7.8.2.1.1.3 size_t flexio_uart_edma_handle_t::txDataSizeAll

16.7.8.2.1.1.4 size_t flexio_uart_edma_handle_t::rxDataSizeAll

16.7.8.2.1.1.5 edma_handle_t∗ flexio_uart_edma_handle_t::txEdmaHandle

16.7.8.2.1.1.6 edma_handle_t∗ flexio_uart_edma_handle_t::rxEdmaHandle

16.7.8.2.1.1.7 uint8_t flexio_uart_edma_handle_t::nbytes

16.7.8.2.1.1.8 volatile uint8_t flexio_uart_edma_handle_t::txState

16.7.8.3 Typedef Documentation

16.7.8.3.1 typedef void(∗ flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type ∗base, flexio_uart_edma_handle_t ∗handle, status_t status, void ∗userData)

16.7.8.4 Function Documentation

16.7.8.4.1 status_t FLEXIO_UART_TransferCreateHandleEDMA (FLEXIO_UART_Type ∗ *base*, flexio_uart_edma_handle_t ∗ *handle*, flexio_uart_edma_transfer_callback_t *callback*, void ∗ *userData*, edma_handle_t ∗ *txEdmaHandle*, edma_handle_t ∗ *rxEdmaHandle*)

Parameters

base	Pointer to FLEXIO_UART_Type.
handle	Pointer to flexio_uart_edma_handle_t structure.
callback	The callback function.
userData	The parameter of the callback function.
rxEdmaHandle	User requested DMA handle for RX DMA transfer.
txEdmaHandle	User requested DMA handle for TX DMA transfer.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO SPI eDMA type/handle table out of range.

16.7.8.4.2 status_t FLEXIO_UART_TransferSendEDMA (FLEXIO_UART_Type ∗ *base*, flexio_uart_edma_handle_t ∗ *handle*, flexio_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 out, the send callback function is called.

Parameters

base	Pointer to FLEXIO_UART_Type
handle	UART handle pointer.
xfer	UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_TxBusy	Previous transfer on going.

16.7.8.4.3 status_t FLEXIO_UART_TransferReceiveEDMA (FLEXIO_UART_Type ∗ *base*, flexio_uart_edma_handle_t ∗ *handle*, flexio_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	Pointer to FLEXIO_UART_Type
handle	Pointer to flexio_uart_edma_handle_t structure
xfer	UART eDMA transfer structure, see flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.

16.7.8.4.4 void FLEXIO_UART_TransferAbortSendEDMA (FLEXIO_UART_Type * *base*, flexio_uart_edma_handle_t * *handle*)

This function aborts sent data which using eDMA.

Parameters

base	Pointer to FLEXIO_UART_Type
handle	Pointer to flexio_uart_edma_handle_t structure

16.7.8.4.5 void FLEXIO_UART_TransferAbortReceiveEDMA (FLEXIO_UART_Type * *base*, flexio_uart_edma_handle_t * *handle*)

This function aborts the receive data which using eDMA.

Parameters

base	Pointer to FLEXIO_UART_Type
handle	Pointer to flexio_uart_edma_handle_t structure

16.7.8.4.6 status_t FLEXIO_UART_TransferGetSendCountEDMA (FLEXIO_UART_Type * *base*, flexio_uart_edma_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes sent out.

Parameters

base	Pointer to FLEXIO_UART_Type
handle	Pointer to flexio_uart_edma_handle_t structure
count	Number of bytes sent so far by the non-blocking transaction.

Return values

kStatus_NoTransferInProgress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

16.7.8.4.7 status_t FLEXIO_UART_TransferGetReceiveCountEDMA (FLEXIO_UART_Type * *base*, flexio_uart_edma_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes received.

Parameters

base	Pointer to FLEXIO_UART_Type
handle	Pointer to flexio_uart_edma_handle_t structure
count	Number of bytes received so far by the non-blocking transaction.

Return values

kStatus_NoTransferInProgress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

16.7.9 FlexIO DMA UART Driver

16.7.9.1 Overview

Data Structures

- struct flexio_uart_dma_handle_t
  UART DMA handle. More...

Typedefs

- typedef void(∗ flexio_uart_dma_transfer_callback_t)(FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

eDMA transactional

- status_t FLEXIO_UART_TransferCreateHandleDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, flexio_uart_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txDmaHandle, dma_handle_t ∗rxDmaHandle)
  Initializes the FLEXIO_UART handle which is used in transactional functions.
- status_t FLEXIO_UART_TransferSendDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, flexio_uart_transfer_t ∗xfer)
  Sends data using DMA.
- status_t FLEXIO_UART_TransferReceiveDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, flexio_uart_transfer_t ∗xfer)
  Receives data using DMA.
- void FLEXIO_UART_TransferAbortSendDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle)
  Aborts the sent data which using DMA.
- void FLEXIO_UART_TransferAbortReceiveDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle)
  Aborts the receive data which using DMA.
- status_t FLEXIO_UART_TransferGetSendCountDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, size_t ∗count)
  Gets the number of bytes sent out.
- status_t FLEXIO_UART_TransferGetReceiveCountDMA (FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, size_t ∗count)
  Gets the number of bytes received.

16.7.9.2 Data Structure Documentation

16.7.9.2.1 struct _flexio_uart_dma_handle

Data Fields

- flexio_uart_dma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  UART callback function parameter.
- size_t txDataSizeAll
  Total bytes to be sent.
- size_t rxDataSizeAll
  Total bytes to be received.
- 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.

16.7.9.2.1.1 Field Documentation

16.7.9.2.1.1.1 flexio_uart_dma_transfer_callback_t flexio_uart_dma_handle_t::callback

16.7.9.2.1.1.2 void∗ flexio_uart_dma_handle_t::userData

16.7.9.2.1.1.3 size_t flexio_uart_dma_handle_t::txDataSizeAll

16.7.9.2.1.1.4 size_t flexio_uart_dma_handle_t::rxDataSizeAll

16.7.9.2.1.1.5 dma_handle_t∗ flexio_uart_dma_handle_t::txDmaHandle

16.7.9.2.1.1.6 dma_handle_t∗ flexio_uart_dma_handle_t::rxDmaHandle

16.7.9.2.1.1.7 volatile uint8_t flexio_uart_dma_handle_t::txState

16.7.9.3 Typedef Documentation

16.7.9.3.1 typedef void(∗ flexio_uart_dma_transfer_callback_t)(FLEXIO_UART_Type ∗base, flexio_uart_dma_handle_t ∗handle, status_t status, void ∗userData)

16.7.9.4 Function Documentation

16.7.9.4.1 status_t FLEXIO_UART_TransferCreateHandleDMA (FLEXIO_UART_Type ∗ *base*, flexio_uart_dma_handle_t ∗ *handle*, flexio_uart_dma_transfer_callback_t *callback*, void ∗ *userData*, dma_handle_t ∗ *txDmaHandle*, dma_handle_t ∗ *rxDmaHandle*)

Parameters

base	Pointer to FLEXIO_UART_Type structure.
handle	Pointer to flexio_uart_dma_handle_t structure.
callback	FlexIO UART callback, NULL means no callback.
userData	User callback function data.
txDmaHandle	User requested DMA handle for TX DMA transfer.
rxDmaHandle	User requested DMA handle for RX DMA transfer.

Return values

kStatus_Success	Successfully create the handle.
kStatus_OutOfRange	The FlexIO UART DMA type/handle table out of range.

16.7.9.4.2 status_t FLEXIO_UART_TransferSendDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*, flexio_uart_transfer_t * *xfer*)

This function send data using DMA. This is non-blocking function, which returns right away. When all data is sent out, the send callback function is called.

Parameters

base	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure
xfer	FLEXIO_UART DMA transfer structure, see flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_TxBusy	Previous transfer on going.

16.7.9.4.3 status_t FLEXIO_UART_TransferReceiveDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*, flexio_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	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure
xfer	FLEXIO_UART DMA transfer structure, see flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_RxBusy	Previous transfer on going.

16.7.9.4.4 void FLEXIO_UART_TransferAbortSendDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*)

This function aborts the sent data which using DMA.

Parameters

base	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure

16.7.9.4.5 void FLEXIO_UART_TransferAbortReceiveDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*)

This function aborts the receive data which using DMA.

Parameters

base	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure

16.7.9.4.6 status_t FLEXIO_UART_TransferGetSendCountDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes sent out.

Parameters

base	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure
count	Number of bytes sent so far by the non-blocking transaction.

Return values

kStatus_NoTransferInProgress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

16.7.9.4.7 status_t FLEXIO_UART_TransferGetReceiveCountDMA (FLEXIO_UART_Type * *base*, flexio_uart_dma_handle_t * *handle*, size_t * *count*)

This function gets the number of bytes received.

Parameters

base	Pointer to FLEXIO_UART_Type structure
handle	Pointer to flexio_uart_dma_handle_t structure
count	Number of bytes received so far by the non-blocking transaction.

Return values

kStatus_NoTransferInProgress	transfer has finished or no transfer in progress.
kStatus_Success	Successfully return the count.

Chapter 17
FTM: FlexTimer Driver

17.1 Overview

The KSDK provides a driver for the FlexTimer Module (FTM) of Kinetis devices.

17.2 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.

17.2.1 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.

17.2.2 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.

17.2.3 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.

17.2.4 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.

17.2.5 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.

17.2.6 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.

17.3 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)
```

17.4 Typical use case

17.4.1 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);
    }
}
```

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)
  Gets the FTM Quad Decoder flags.
- static void FTM_SetQuadDecoderModuloValue (FTM_Type ∗base, uint32_t startValue, uint32_t overValue)
  Sets the modulo values for Quad Decoder.
- static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type ∗base)
  Gets the current Quad Decoder counter value.
- static void FTM_ClearQuadDecoderCounterValue (FTM_Type ∗base)
  Clears the current Quad Decoder counter value.

17.5 Data Structure Documentation

17.5.1 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.

17.5.1.0.7.1 Field Documentation

17.5.1.0.7.1.1 ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber

In combined mode, this represents the channel pair number.

17.5.1.0.7.1.2 ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level

17.5.1.0.7.1.3 uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

17.5.1.0.7.1.4 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

17.5.2 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.

17.5.3 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.

17.5.4 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; in other words, '0' indicates a fault; False: Fault polarity is active high.
- bool useFaultFilter
  True: Use the filtered fault signal; False: Use the direct path from fault input.

17.5.5 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.
- uint32_t deadTimeValue
  The dead time value deadTimeValue's available range is 0-1023 when register has DTVALEX, otherwise its available range is 0-63.
- 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.

17.5.5.0.7.2 Field Documentation

17.5.5.0.7.2.1 uint32_t ftm_config_t::pwmSyncMode

17.5.5.0.7.2.2 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.

17.5.5.0.7.2.3 uint32_t ftm_config_t::deadTimeValue

17.5.5.0.7.2.4 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.

17.6 Enumeration Type Documentation

17.6.1 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.

17.6.2 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.

17.6.3 enum ftm_pwm_mode_t

Enumerator

kFTM_EdgeAlignedPwm Edge-aligned PWM.
kFTM_CenterAlignedPwm Center-aligned PWM.
kFTM_CombinedPwm Combined PWM.

17.6.4 enum ftm_pwm_level_select_t

Enumerator

kFTM_NoPwmSignal No PWM output on pin.
kFTM_LowTrue Low true pulses.
kFTM_HighTrue High true pulses.

17.6.5 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.

17.6.6 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.

17.6.7 enum ftm_dual_edge_capture_mode_t

Enumerator

kFTM_OneShot One-shot capture mode.
kFTM_Continuous Continuous capture mode.

17.6.8 enum ftm_quad_decode_mode_t

Enumerator

kFTM_QuadPhaseEncode Phase A and Phase B encoding mode.
kFTM_QuadCountAndDir Count and direction encoding mode.

17.6.9 enum ftm_phase_polarity_t

Enumerator

kFTM_QuadPhaseNormal Phase input signal is not inverted.
kFTM_QuadPhaseInvert Phase input signal is inverted.

17.6.10 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.

17.6.11 enum ftm_clock_source_t

Enumerator

kFTM_SystemClock System clock selected.
kFTM_FixedClock Fixed frequency clock.
kFTM_ExternalClock External clock.

17.6.12 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.

17.6.13 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.

17.6.14 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.

17.6.15 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.

17.6.16 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.

17.6.17 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.

17.6.18 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.

17.6.19 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.

17.6.20 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.

17.7 Function Documentation

17.7.1 status_t FTM_Init (FTM_Type ∗ *base,* const ftm_config_t ∗ *config*)

Note

This API should be called at the beginning of the application which is 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.

17.7.2 void FTM_Deinit (FTM_Type * *base*)

Parameters

base	FTM peripheral base address

17.7.3 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.

17.7.4 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

17.7.5 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
currentPwm-Mode	The current PWM mode set during PWM setup
dutyCycle-Percent	New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

17.7.6 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.

17.7.7 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.

17.7.8 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.

17.7.9 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.

17.7.10 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

17.7.11 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

17.7.12 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

17.7.13 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

17.7.14 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

17.7.15 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

17.7.16 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.

17.7.17 static void FTM_StopTimer (FTM_Type * *base*) [inline], [static]

Parameters

base	FTM peripheral base address

17.7.18 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

17.7.19 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

17.7.20 static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

17.7.21 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

17.7.22 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

17.7.23 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

17.7.24 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

17.7.25 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

17.7.26 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

17.7.27 static uint32_t FTM_GetQuadDecoderFlags (FTM_Type * *base*) `[inline]`, `[static]`

Parameters

base	FTM peripheral base address.

Returns

Flag mask of FTM Quad Decoder, see _ftm_quad_decoder_flags.

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

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

Parameters

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

17.7.29 static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type * *base*) `[inline]`, `[static]`

Parameters

base	FTM peripheral base address.

Returns

Current quad Decoder counter value.

17.7.30 static void FTM_ClearQuadDecoderCounterValue (FTM_Type * *base*) [inline], [static]

The counter is set as the initial value.

Parameters

base	FTM peripheral base address.

17.7.31 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

17.7.32 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 18
GPIO: General-Purpose Input/Output Driver

18.1 Overview

Modules

- FGPIO Driver
- GPIO Driver

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, 1))
  GPIO driver version 2.1.1.

18.2 Data Structure Documentation

18.2.1 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 that 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.

18.3 Macro Definition Documentation

18.3.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

18.4 Enumeration Type Documentation

18.4.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.
kGPIO_DigitalOutput Set current pin as digital output.

18.5 GPIO Driver

18.5.1 Overview

The KSDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of Kinetis devices.

18.5.2 Typical use case

18.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

18.5.2.2 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.

18.5.3 Function Documentation

18.5.3.1 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

18.5.3.2 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.

18.5.3.3 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

18.5.3.4 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

18.5.3.5 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

18.5.3.6 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.

18.5.3.7 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.

18.5.3.8 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

18.6 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.

18.6.1 Typical use case

18.6.1.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
FGPIO_PinInit(FGPIO_LED, LED_PINNUM, &led_config);
```

18.6.1.2 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 19
I2C: Inter-Integrated Circuit Driver

19.1 Overview

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver
- I2C µCOS/II Driver
- I2C µCOS/III Driver

19.2 I2C Driver

19.2.1 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 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 knowing 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_MasterTransferNonBlocking() set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

19.2.2 Typical use case

19.2.2.1 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);
```

19.2.2.2 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;
```

19.2.2.3 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;
```

19.2.2.4 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;
```

19.2.2.5 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;
```

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),
  kStatus_I2C_Addr_Nak = MAKE_STATUS(kStatusGroup_I2C, 5) }
  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,
  kI2C_StopDetectFlag = I2C_FLT_STOPF_MASK << 8,
  kI2C_StartDetectFlag = I2C_FLT_STARTF_MASK << 8 }
  I2C peripheral flags.

- enum _i2c_interrupt_enable {
  kI2C_GlobalInterruptEnable = I2C_C1_IICIE_MASK,
  kI2C_StartStopDetectInterruptEnable = I2C_FLT_SSIE_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_SlaveStartEvent = 0x10U,
  kI2C_SlaveCompletionEvent = 0x20U,
  kI2C_SlaveGenaralcallEvent = 0x40U,
  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, uint32_t srcClock_Hz)
  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 void I2C_EnableDMA (I2C_Type ∗base, bool enable)
  Enables/disables the I2C DMA interrupt.
- 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, uint32_t flags)
  Performs a polling send transaction on the I2C bus.

- status_t I2C_MasterReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)
    Performs a polling receive transaction on the I2C bus.
- 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.

19.2.3 Data Structure Documentation

19.2.3.1 struct i2c_master_config_t

Data Fields

- bool enableMaster
    Enables the I2C peripheral at initialization time.
- bool enableHighDrive
    Controls the drive capability of the I2C pads.

- bool enableStopHold
  Controls the stop hold enable.
- uint32_t baudRate_Bps
  Baud rate configuration of I2C peripheral.
- uint8_t glitchFilterWidth
  Controls the width of the glitch.

19.2.3.1.0.3 Field Documentation

19.2.3.1.0.3.1 bool i2c_master_config_t::enableMaster

19.2.3.1.0.3.2 bool i2c_master_config_t::enableHighDrive

19.2.3.1.0.3.3 bool i2c_master_config_t::enableStopHold

19.2.3.1.0.3.4 uint32_t i2c_master_config_t::baudRate_Bps

19.2.3.1.0.3.5 uint8_t i2c_master_config_t::glitchFilterWidth

19.2.3.2 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 enableHighDrive
  Controls the drive capability of the I2C pads.
- bool enableBaudRateCtl
  Enables/disables independent slave baud rate on SCL in very fast I2C modes.
- uint16_t slaveAddress
  A slave address configuration.
- uint16_t upperAddress
  A maximum boundary slave address used in a range matching mode.
- i2c_slave_address_mode_t addressingMode
  An addressing mode configuration of i2c_slave_address_mode_config_t.
- uint32_t sclStopHoldTime_ns
  the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.

19.2.3.2.0.4 Field Documentation

19.2.3.2.0.4.1 bool i2c_slave_config_t::enableSlave

19.2.3.2.0.4.2 bool i2c_slave_config_t::enableGeneralCall

19.2.3.2.0.4.3 bool i2c_slave_config_t::enableWakeUp

19.2.3.2.0.4.4 bool i2c_slave_config_t::enableHighDrive

19.2.3.2.0.4.5 bool i2c_slave_config_t::enableBaudRateCtl

19.2.3.2.0.4.6 uint16_t i2c_slave_config_t::slaveAddress

19.2.3.2.0.4.7 uint16_t i2c_slave_config_t::upperAddress

19.2.3.2.0.4.8 i2c_slave_address_mode_t i2c_slave_config_t::addressingMode

19.2.3.2.0.4.9 uint32_t i2c_slave_config_t::sclStopHoldTime_ns

19.2.3.3 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.

19.2.3.3.0.5 Field Documentation

19.2.3.3.0.5.1 uint32_t i2c_master_transfer_t::flags

19.2.3.3.0.5.2 uint8_t i2c_master_transfer_t::slaveAddress

19.2.3.3.0.5.3 i2c_direction_t i2c_master_transfer_t::direction

19.2.3.3.0.5.4 uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

19.2.3.3.0.5.5 uint8_t i2c_master_transfer_t::subaddressSize

19.2.3.3.0.5.6 uint8_t* volatile i2c_master_transfer_t::data

19.2.3.3.0.5.7 volatile size_t i2c_master_transfer_t::dataSize

19.2.3.4 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.

19.2.3.4.0.6 Field Documentation

19.2.3.4.0.6.1 i2c_master_transfer_t i2c_master_handle_t::transfer

19.2.3.4.0.6.2 size_t i2c_master_handle_t::transferSize

19.2.3.4.0.6.3 uint8_t i2c_master_handle_t::state

19.2.3.4.0.6.4 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

19.2.3.4.0.6.5 void* i2c_master_handle_t::userData

19.2.3.5 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.

19.2.3.5.0.7 Field Documentation

19.2.3.5.0.7.1 i2c_slave_transfer_event_t i2c_slave_transfer_t::event

19.2.3.5.0.7.2 uint8_t* volatile i2c_slave_transfer_t::data

19.2.3.5.0.7.3 volatile size_t i2c_slave_transfer_t::dataSize

19.2.3.5.0.7.4 status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

19.2.3.5.0.7.5 size_t i2c_slave_transfer_t::transferredCount

19.2.3.6 struct _i2c_slave_handle

I2C slave handle typedef.

Data Fields

- volatile bool isBusy
  Indicates whether a transfer is busy.
- i2c_slave_transfer_t transfer
  I2C slave transfer copy.
- uint32_t eventMask
  A mask of enabled events.
- i2c_slave_transfer_callback_t callback
  A callback function called at the transfer event.
- void * userData
  A callback parameter passed to the callback.

19.2.3.6.0.8 Field Documentation

19.2.3.6.0.8.1 volatile bool i2c_slave_handle_t::isBusy

19.2.3.6.0.8.2 i2c_slave_transfer_t i2c_slave_handle_t::transfer

19.2.3.6.0.8.3 uint32_t i2c_slave_handle_t::eventMask

19.2.3.6.0.8.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

19.2.3.6.0.8.5 void* i2c_slave_handle_t::userData

19.2.4 Macro Definition Documentation

19.2.4.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

19.2.5 Typedef Documentation

19.2.5.1 typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)

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

19.2.6 Enumeration Type Documentation

19.2.6.1 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.
kStatus_I2C_Addr_Nak NAK received during the address probe.

19.2.6.2 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.
kI2C_StopDetectFlag I2C stop detect flag.
kI2C_StartDetectFlag I2C start detect flag.

19.2.6.3 enum _i2c_interrupt_enable

Enumerator

kI2C_GlobalInterruptEnable I2C global interrupt.
kI2C_StartStopDetectInterruptEnable I2C start&stop detect interrupt.

19.2.6.4 enum i2c_direction_t

Enumerator

kI2C_Write Master transmits to the slave.
kI2C_Read Master receives from the slave.

19.2.6.5 enum i2c_slave_address_mode_t

Enumerator

kI2C_Address7bit 7-bit addressing mode.
kI2C_RangeMatch Range address match addressing mode.

19.2.6.6 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.

19.2.6.7 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_SlaveStartEvent A start/repeated start was detected.
kI2C_SlaveCompletionEvent A stop was detected or finished transfer, completing the transfer.
kI2C_SlaveGenaralcallEvent Received the general call address after a start or repeated start.
kI2C_SlaveAllEvents A bit mask of all available events.

19.2.7 Function Documentation

19.2.7.1 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. This is an example.

```
* i2c_master_config_t config = {
* .enableMaster = true,
* .enableStopHold = false,
* .highDrive = false,
* .baudRate_Bps = 100000,
```

```
* .glitchFilterWidth = 0
* };
* I2C_MasterInit(I2C0, &config, 12000000U);
*
```

Parameters

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

19.2.7.2 void I2C_SlaveInit (I2C_Type ∗ *base*, const i2c_slave_config_t ∗ *slaveConfig*, uint32_t *srcClock_Hz*)

Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note

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

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

Parameters

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

19.2.7.3 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

19.2.7.4 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

19.2.7.5 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(). This is an example.

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

Parameters

masterConfig	A pointer to the master configuration structure.

19.2.7.6 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(). This is an example.

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

Parameters

slaveConfig	A pointer to the slave configuration structure.

19.2.7.7 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.

19.2.7.8 uint32_t I2C_MasterGetStatusFlags (I2C_Type ∗ *base*)

Parameters

base	I2C base pointer

Returns

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

19.2.7.9 static uint32_t I2C_SlaveGetStatusFlags (I2C_Type ∗ *base*) [inline], [static]

Parameters

base	I2C base pointer

Returns

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

19.2.7.10 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 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

19.2.7.11 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 can be any combination of the following values: • kI2C_StartDetectFlag (if available) • kI2C_StopDetectFlag (if available) • kI2C_ArbitrationLostFlag • kI2C_IntPendingFlagFlag

19.2.7.12 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

19.2.7.13 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

19.2.7.14 static void I2C_EnableDMA (I2C_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

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

19.2.7.15 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

19.2.7.16 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

19.2.7.17 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.

19.2.7.18 status_t I2C_MasterStop (I2C_Type ∗ *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

19.2.7.19 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.

19.2.7.20 status_t I2C_MasterWriteBlocking (I2C_Type ∗ *base,* const uint8_t ∗ *txBuff,* size_t *txSize,* uint32_t *flags*)

Parameters

base	The I2C peripheral base pointer.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.
flags	Transfer control flag to decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

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.

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

Note

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

Parameters

base	I2C peripheral base pointer.
rxBuff	The pointer to the data to store the received data.
rxSize	The length in bytes of the data to be received.
flags	Transfer control flag to decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

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

19.2.7.22 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.

19.2.7.23 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.

19.2.7.24 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.

19.2.7.25 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 parameter passed to the callback function.

19.2.7.26 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. The user needs to call I2C_MasterGetTransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

base	I2C base pointer.
handle	pointer to i2c_master_handle_t structure which stores the transfer state.
xfer	pointer to i2c_master_transfer_t structure.

Return values

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

19.2.7.27 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.

19.2.7.28 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

19.2.7.29 void I2C_MasterTransferHandleIRQ (I2C_Type ∗ *base*, void ∗ *i2cHandle*)

Parameters

base	I2C base pointer.
i2cHandle	pointer to i2c_master_handle_t structure.

19.2.7.30 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.

19.2.7.31 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.

19.2.7.32 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.

19.2.7.33 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.

19.2.7.34 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

19.3 I2C eDMA Driver

19.3.1 Overview

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.

19.3.2 Data Structure Documentation

19.3.2.1 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 nbytes
  eDMA minor byte transfer count initially configured.
- 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.

19.3.2.1.0.9 Field Documentation

19.3.2.1.0.9.1 i2c_master_transfer_t i2c_master_edma_handle_t::transfer

19.3.2.1.0.9.2 size_t i2c_master_edma_handle_t::transferSize

19.3.2.1.0.9.3 uint8_t i2c_master_edma_handle_t::nbytes

19.3.2.1.0.9.4 uint8_t i2c_master_edma_handle_t::state

19.3.2.1.0.9.5 edma_handle_t∗ i2c_master_edma_handle_t::dmaHandle

19.3.2.1.0.9.6 i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completion-Callback

19.3.2.1.0.9.7 void∗ i2c_master_edma_handle_t::userData

19.3.3 Typedef Documentation

19.3.3.1 typedef void(∗ i2c_master_edma_transfer_callback_t)(I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, status_t status, void ∗userData)

19.3.4 Function Documentation

19.3.4.1 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.

19.3.4.2 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_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

19.3.4.3 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.

19.3.4.4 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.

19.4 I2C DMA Driver

19.4.1 Overview

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.

19.4.2 Data Structure Documentation

19.4.2.1 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.

19.4.2.1.0.10 Field Documentation

19.4.2.1.0.10.1 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

19.4.2.1.0.10.2 size_t i2c_master_dma_handle_t::transferSize

19.4.2.1.0.10.3 uint8_t i2c_master_dma_handle_t::state

19.4.2.1.0.10.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

19.4.2.1.0.10.5 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

19.4.2.1.0.10.6 void* i2c_master_dma_handle_t::userData

19.4.3 Typedef Documentation

19.4.3.1 typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)

19.4.4 Function Documentation

19.4.4.1 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

19.4.4.2 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.

19.4.4.3 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.

19.4.4.4 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.

19.5 I2C FreeRTOS Driver

19.5.1 Overview

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 the I2C transfer.

19.5.2 Function Documentation

19.5.2.1 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	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.

19.5.2.2 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.

19.5.2.3 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.

19.6 I2C µCOS/II Driver

19.6.1 Overview

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.

19.6.2 Function Documentation

19.6.2.1 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.

19.6.2.2 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.

19.6.2.3 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.

19.7 I2C µCOS/III Driver

19.7.1 Overview

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.

19.7.2 Function Documentation

19.7.2.1 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 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.

19.7.2.2 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.

19.7.2.3 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.

Chapter 20
LLWU: Low-Leakage Wakeup Unit Driver

20.1 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.

20.2 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.

20.3 Internal wakeup modules configurations

Enables/disables the internal wakeup modules and gets the module flags. Internal modules are accessed by `moduleIndex`, which is started from 1. Numbers of external pins depend the on SoC configuration.

20.4 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.

Data Structures

- struct llwu_external_pin_filter_mode_t
  An external input pin filter control structure. More...

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.

Low-Leakage Wakeup Unit Control APIs

- void LLWU_SetExternalWakeupPinMode (LLWU_Type ∗base, uint32_t pinIndex, llwu_external_pin_mode_t pinMode)
  Sets the external input pin source mode.
- bool LLWU_GetExternalWakeupPinFlag (LLWU_Type ∗base, uint32_t pinIndex)
  Gets the external wakeup source flag.
- void LLWU_ClearExternalWakeupPinFlag (LLWU_Type ∗base, uint32_t pinIndex)
  Clears the external wakeup source flag.
- static void LLWU_EnableInternalModuleInterruptWakup (LLWU_Type ∗base, uint32_t moduleIndex, bool enable)
  Enables/disables the internal module source.
- static bool LLWU_GetInternalWakeupModuleFlag (LLWU_Type ∗base, uint32_t moduleIndex)
  Gets the external wakeup source flag.
- void LLWU_SetPinFilterMode (LLWU_Type ∗base, uint32_t filterIndex, llwu_external_pin_filter_mode_t filterMode)
  Sets the pin filter configuration.
- bool LLWU_GetPinFilterFlag (LLWU_Type ∗base, uint32_t filterIndex)
  Gets the pin filter configuration.
- void LLWU_ClearPinFilterFlag (LLWU_Type ∗base, uint32_t filterIndex)
  Clears the pin filter configuration.

20.5 Data Structure Documentation

20.5.1 struct llwu_external_pin_filter_mode_t

Data Fields

- uint32_t pinIndex
  A pin number.
- llwu_pin_filter_mode_t filterMode
  Filter mode.

20.6 Macro Definition Documentation

20.6.1 #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

20.7 Enumeration Type Documentation

20.7.1 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.

20.7.2 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.

20.8 Function Documentation

20.8.1 void LLWU_SetExternalWakeupPinMode (LLWU_Type * *base*, uint32_t *pinIndex*, llwu_external_pin_mode_t *pinMode*)

This function sets the external input pin source mode that is used as a wake up source.

Parameters

base	LLWU peripheral base address.
pinIndex	A pin index to be enabled as an external wakeup source starting from 1.
pinMode	A pin configuration mode defined in the llwu_external_pin_modes_t.

20.8.2 bool LLWU_GetExternalWakeupPinFlag (LLWU_Type * *base*, uint32_t *pinIndex*)

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

Parameters

base	LLWU peripheral base address.
pinIndex	A pin index, which starts from 1.

Returns

True if the specific pin is a wakeup source.

20.8.3 void LLWU_ClearExternalWakeupPinFlag (LLWU_Type * *base,* uint32_t *pinIndex*)

This function clears the external wakeup source flag for a specific pin.

Parameters

base	LLWU peripheral base address.
pinIndex	A pin index, which starts from 1.

20.8.4 static void LLWU_EnableInternalModuleInterruptWakup (LLWU_Type * *base*, uint32_t *moduleIndex*, bool *enable*) [inline], [static]

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

base	LLWU peripheral base address.
moduleIndex	A module index to be enabled as an internal wakeup source starting from 1.
enable	An enable or a disable setting

20.8.5 static bool LLWU_GetInternalWakeupModuleFlag (LLWU_Type * *base*, uint32_t *moduleIndex*) [inline], [static]

This function checks the external pin flag to detect whether the system is woken up by the specific pin.

Parameters

base	LLWU peripheral base address.
moduleIndex	A module index, which starts from 1.

Returns

True if the specific pin is a wake up source.

20.8.6 void LLWU_SetPinFilterMode (LLWU_Type * *base*, uint32_t *filterIndex*, llwu_external_pin_filter_mode_t *filterMode*)

This function sets the pin filter configuration.

Parameters

base	LLWU peripheral base address.
filterIndex	A pin filter index used to enable/disable the digital filter, starting from 1.
filterMode	A filter mode configuration

20.8.7 bool LLWU_GetPinFilterFlag (LLWU_Type * *base,* uint32_t *filterIndex*)

This function gets the pin filter flag.

Parameters

base	LLWU peripheral base address.
filterIndex	A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

20.8.8 void LLWU_ClearPinFilterFlag (LLWU_Type * *base,* uint32_t *filterIndex*)

This function clears the pin filter flag.

Parameters

base	LLWU peripheral base address.
filterIndex	A pin filter index to clear the flag, starting from 1.

Chapter 21
LMEM: Local Memory Controller Cache Control Driver

21.1 Overview

The KSDK provides a peripheral driver for the Local Memory Controller Cache Controller module of Kinetis devices.

21.2 Descriptions

The LMEM Cache peripheral driver allows the user to enable/disable the cache and to perform cache maintenance operations such as invalidate, push, and clear. These maintenance operations may be performed on the Processor Code (PC) bus or Both Processor Code (PC) and Processor System (PS) bus.

The Kinetis devices contain a Processor Code (PC) bus and a Processor System (PS) bus as follows. The Processor Code (PC) bus - a 32-bit address space bus with low-order addresses (0x0000_0000 through 0x1FFF_FFFF) used normally for code access. The Processor System (PS) bus - a 32-bit address space bus with high-order addresses (0x2000_0000 through 0xFFFF_FFFF) used normally for data accesses.

Some Kinetic MCU devices have caches available for the PC bus and PS bus, others may only have a PC bus cache, while some do not have PC or PS caches at all. See the appropriate Kinetis reference manual for cache availability.

Cache maintenance operations:

command	description
Invalidate	U
Push	P ush a cache entry if it is valid and modified, then clear the m
Clear	P ush a cache entry if it is valic

The above cache maintenance operations may be performed on the entire cache or on a line-basis. The peripheral driver API names distinguish between the two using the terms "All" or Line".

21.3 Function groups

21.3.1 Local Memory Processor Code Bus Cache Control

The invalidate command can be performed on the entire cache, one line, or multiple lines by calling LMEM_CodeCacheInvalidateAll(), LMEM_CodeCacheInvalidateLine(), and LMEM_CodeCacheInvalidateMultiLines().

The push command can be performed on the entire cache, one line, or multiple lines by calling LMEM_-CodeCachePushAll(), LMEM_CodeCachePushLine(), and LMEM_CodeCachePushMultiLines().

The clear command can be performed on the entire cache, one line, or multiple lines by calling LMEM_-CodeCacheClearAll(), LMEM_CodeCacheClearLine(), and LMEM_CodeCacheClearMultiLines().

Note that the parameter "address" must be supplied, which indicates the physical address of the line to perform the one line cache maintenance operation. In addition, the length of the number of bytes should be supplied for multiple line operation. The function determines if the length meets or exceeds 1/2 the cache size because the cache contains 2 WAYs, half of the cache is in WAY0 and the other half in W-AY1 and if so, performs a cache maintenance "all" operation which is faster than performing the cache maintenance on a line-basis.

Cache Demotion: Cache region demotion - Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The cache region demote function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state. To demote a cache region, call the LMEM_CodeCacheDemoteRegion().

Note that the address region assignment of the 16 subregions is device-specific and is detailed in the Chip Configuration part of the SoC Kinetis reference manual. The LMEM provides typedef enums for each of the 16 regions, starting with "kLMEM_CacheRegion0" and ending with "kLMEM_CacheRegion15". The parameter cacheMode is of type lmem_cache_mode_t. This provides typedef enums for each of the cache modes, such as "kLMEM_CacheNonCacheable", "kLMEM_CacheWriteThrough", and "kLMEM-_CacheWriteBack". Cache Enable and Disable: The cache enable function enables the PC bus cache and the write buffer. However, before enabling these, the function first performs an invalidate all. Call LMEM_EnableCodeCache() to enable a particular bus cache.

21.3.2 Local Memory Processor System Bus Cache Control

The invalidate command can be performed on the entire cache, one line, or multiple lines by calling LMEM_SystemCacheInvalidateAll(), LMEM_SystemCacheInvalidateLine(), and LMEM_SystemCache-InvalidateMultiLines().

The push command can be performed on the entire cache, one line, or multiple lines by calling LMEM_-SystemCachePushAll(), LMEM_SystemCachePushLine(), and LMEM_SystemCachePushMultiLines().

The clear command can be performed on the entire cache, one line, or multiple lines by calling LM-EM_SystemCacheClearAll(), LMEM_SystemCacheClearLine(), and LMEM_SystemCacheClearMulti-Lines().

Note that the parameter "address" must be supplied, which indicates the physical address of the line to perform the one line cache maintenance operation. In addition, the length of the number of bytes should be supplied for multiple lines operation. The function determines if the length meets or exceeds 1/2 the cache size because the cache contains 2 WAYs, half of the cache is in WAY0 and the other half in W-AY1 and if so, performs a cache maintenance "all" operation which is faster than performing the cache maintenance on a line-basis.

Cache Demotion: Cache region demotion - Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The cache region demote function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state. To demote a cache region, call the LMEM_SystemCacheDemoteRegion().

Note that the address region assignment of the 16 subregions is device-specific and is described in the Chip Configuration part of the Kinetis SoC reference manual. The LMEM provides typedef enumerations for each of the 16 regions, starting with "kLMEM_CacheRegion0" and ending with "kLMEM-_CacheRegion15". The parameter cacheMode is of type lmem_cache_mode_t. This provides typedef enumerations for each of the cache modes, such as "kLMEM_CacheNonCacheable", "kLMEM_Cache-WriteThrough", and "kLMEM_CacheWriteBack". Cache Enable and Disable: The cache enable function enables the PS bus cache and the write buffer. However, before enabling these, the function first performs an invalidate all. Call LMEM_EnableSystemCache() to enable a particular bus cache.

Macros

- #define LMEM_CACHE_LINE_SIZE (0x10U)
    Cache line is 16-bytes.
- #define LMEM_CACHE_SIZE_ONEWAY (4096U)
    Cache size is 4K-bytes one way.

Enumerations

- enum lmem_cache_mode_t {
  kLMEM_NonCacheable = 0x0U,
  kLMEM_CacheWriteThrough = 0x2U,
  kLMEM_CacheWriteBack = 0x3U }
    LMEM cache mode options.
- enum lmem_cache_region_t {
  kLMEM_CacheRegion15 = 0U,
  kLMEM_CacheRegion14,
  kLMEM_CacheRegion13,
  kLMEM_CacheRegion12,
  kLMEM_CacheRegion11,
  kLMEM_CacheRegion10,
  kLMEM_CacheRegion9,
  kLMEM_CacheRegion8,
  kLMEM_CacheRegion7,
  kLMEM_CacheRegion6,
  kLMEM_CacheRegion5,
  kLMEM_CacheRegion4,
  kLMEM_CacheRegion3,
  kLMEM_CacheRegion2,
  kLMEM_CacheRegion1,
  kLMEM_CacheRegion0 }
    LMEM cache regions.

- enum lmem_cache_line_command_t {
kLMEM_CacheLineSearchReadOrWrite = 0U,
kLMEM_CacheLineInvalidate,
kLMEM_CacheLinePush,
kLMEM_CacheLineClear }
LMEM cache line command.

Driver version

- #define FSL_LMEM_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
LMEM controller driver version 2.1.0.

Local Memory Processor Code Bus Cache Control

- void LMEM_EnableCodeCache (LMEM_Type ∗base, bool enable)
Enables/disables the processor code bus cache.
- static void LMEM_EnableCodeWriteBuffer (LMEM_Type ∗base, bool enable)
Enables/disables the processor code bus write buffer.
- void LMEM_CodeCacheInvalidateAll (LMEM_Type ∗base)
Invalidates the processor code bus cache.
- void LMEM_CodeCachePushAll (LMEM_Type ∗base)
Pushes all modified lines in the processor code bus cache.
- void LMEM_CodeCacheClearAll (LMEM_Type ∗base)
Clears the processor code bus cache.
- void LMEM_CodeCacheInvalidateLine (LMEM_Type ∗base, uint32_t address)
Invalidates a specific line in the processor code bus cache.
- void LMEM_CodeCacheInvalidateMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
Invalidates multiple lines in the processor code bus cache.
- void LMEM_CodeCachePushLine (LMEM_Type ∗base, uint32_t address)
Pushes a specific modified line in the processor code bus cache.
- void LMEM_CodeCachePushMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
Pushes multiple modified lines in the processor code bus cache.
- void LMEM_CodeCacheClearLine (LMEM_Type ∗base, uint32_t address)
Clears a specific line in the processor code bus cache.
- void LMEM_CodeCacheClearMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
Clears multiple lines in the processor code bus cache.
- status_t LMEM_CodeCacheDemoteRegion (LMEM_Type ∗base, lmem_cache_region_t region, lmem_cache_mode_t cacheMode)
Demotes the cache mode of a region in processor code bus cache.

Local Memory Processor System Bus Cache Control

- void LMEM_EnableSystemCache (LMEM_Type ∗base, bool enable)
Enables/disables the processor system bus cache.
- static void LMEM_EnableSystemWriteBuffer (LMEM_Type ∗base, bool enable)
Enables/disables the processor system bus write buffer.
- void LMEM_SystemCacheInvalidateAll (LMEM_Type ∗base)
Invalidates the processor system bus cache.
- void LMEM_SystemCachePushAll (LMEM_Type ∗base)

Pushes all modified lines in the processor system bus cache.
- void LMEM_SystemCacheClearAll (LMEM_Type ∗base)
  Clears the entire processor system bus cache.
- void LMEM_SystemCacheInvalidateLine (LMEM_Type ∗base, uint32_t address)
  Invalidates a specific line in the processor system bus cache.
- void LMEM_SystemCacheInvalidateMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
  Invalidates multiple lines in the processor system bus cache.
- void LMEM_SystemCachePushLine (LMEM_Type ∗base, uint32_t address)
  Pushes a specific modified line in the processor system bus cache.
- void LMEM_SystemCachePushMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
  Pushes multiple modified lines in the processor system bus cache.
- void LMEM_SystemCacheClearLine (LMEM_Type ∗base, uint32_t address)
  Clears a specific line in the processor system bus cache.
- void LMEM_SystemCacheClearMultiLines (LMEM_Type ∗base, uint32_t address, uint32_t length)
  Clears multiple lines in the processor system bus cache.
- status_t LMEM_SystemCacheDemoteRegion (LMEM_Type ∗base, lmem_cache_region_t region, lmem_cache_mode_t cacheMode)
  Demotes the cache mode of a region in the processor system bus cache.

21.4 Macro Definition Documentation

21.4.1 #define FSL_LMEM_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

21.4.2 #define LMEM_CACHE_LINE_SIZE (0x10U)

21.4.3 #define LMEM_CACHE_SIZE_ONEWAY (4096U)

21.5 Enumeration Type Documentation

21.5.1 enum lmem_cache_mode_t

Enumerator

kLMEM_NonCacheable Cache mode: non-cacheable.
kLMEM_CacheWriteThrough Cache mode: write-through.
kLMEM_CacheWriteBack Cache mode: write-back.

21.5.2 enum lmem_cache_region_t

Enumerator

kLMEM_CacheRegion15 Cache Region 15.
kLMEM_CacheRegion14 Cache Region 14.
kLMEM_CacheRegion13 Cache Region 13.

kLMEM_CacheRegion12 Cache Region 12.
kLMEM_CacheRegion11 Cache Region 11.
kLMEM_CacheRegion10 Cache Region 10.
kLMEM_CacheRegion9 Cache Region 9.
kLMEM_CacheRegion8 Cache Region 8.
kLMEM_CacheRegion7 Cache Region 7.
kLMEM_CacheRegion6 Cache Region 6.
kLMEM_CacheRegion5 Cache Region 5.
kLMEM_CacheRegion4 Cache Region 4.
kLMEM_CacheRegion3 Cache Region 3.
kLMEM_CacheRegion2 Cache Region 2.
kLMEM_CacheRegion1 Cache Region 1.
kLMEM_CacheRegion0 Cache Region 0.

21.5.3 enum lmem_cache_line_command_t

Enumerator

kLMEM_CacheLineSearchReadOrWrite Cache line search and read or write.
kLMEM_CacheLineInvalidate Cache line invalidate.
kLMEM_CacheLinePush Cache line push.
kLMEM_CacheLineClear Cache line clear.

21.6 Function Documentation

21.6.1 void LMEM_EnableCodeCache (LMEM_Type ∗ *base*, bool *enable*)

This function enables/disables the cache. The function first invalidates the entire cache and then enables/disables both the cache and write buffers.

Parameters

base	LMEM peripheral base address.
enable	The enable or disable flag. true - enable the code cache. false - disable the code cache.

21.6.2 static void LMEM_EnableCodeWriteBuffer (LMEM_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	LMEM peripheral base address.
enable	The enable or disable flag. true - enable the code bus write buffer. false - disable the code bus write buffer.

21.6.3 void LMEM_CodeCacheInvalidateAll (LMEM_Type * *base*)

This function invalidates the cache both ways, which means that it unconditionally clears valid bits and modifies bits of a cache entry.

Parameters

base	LMEM peripheral base address.

21.6.4 void LMEM_CodeCachePushAll (LMEM_Type * *base*)

This function pushes all modified lines in both ways in the entire cache. It pushes a cache entry if it is valid and modified and clears the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.

21.6.5 void LMEM_CodeCacheClearAll (LMEM_Type * *base*)

This function clears the entire cache and pushes (flushes) and invalidates the operation. Clear - Pushes a cache entry if it is valid and modified, then clears the valid and modified bits. If the entry is not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.

21.6.6 void LMEM_CodeCacheInvalidateLine (LMEM_Type * *base*, uint32_t *address*)

This function invalidates a specific line in the cache based on the physical address passed in by the user. Invalidate - Unconditionally clears valid and modified bits of a cache entry.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

21.6.7 void LMEM_CodeCacheInvalidateMultiLines (LMEM_Type ∗ *base,* uint32_t *address,* uint32_t *length*)

This function invalidates multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half the cache, the function performs an entire cache invalidate function, which is more efficient than invalidating the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Invalidate - Unconditionally clear valid and modified bits of a cache entry.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.8 void LMEM_CodeCachePushLine (LMEM_Type ∗ *base,* uint32_t *address*)

This function pushes a specific modified line based on the physical address passed in by the user. Push - Push a cache entry if it is valid and modified, then clear the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

21.6.9 void LMEM_CodeCachePushMultiLines (LMEM_Type * *base,* uint32_t *address,* uint32_t *length*)

This function pushes multiple modified lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half of the cache, the function performs an cache push function, which is more efficient than pushing the modified lines in the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Push - Push a cache entry if it is valid and modified, then clear the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.10 void LMEM_CodeCacheClearLine (LMEM_Type * *base,* uint32_t *address*)

This function clears a specific line based on the physical address passed in by the user. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If entry not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

21.6.11 void LMEM_CodeCacheClearMultiLines (LMEM_Type * *base,* uint32_t *address,* uint32_t *length*)

This function clears multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half the total amount of cache, the function performs a cache clear function which is more efficient than clearing the lines in the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If entry not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.12 status_t LMEM_CodeCacheDemoteRegion (LMEM_Type * *base*, lmem_cache_region_t *region*, lmem_cache_mode_t *cacheMode*)

This function allows the user to demote the cache mode of a region within the device's memory map. Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state which is the highest cache configure for each region. To maintain cache coherency, changes to the cache mode should be completed while the address space being changed is not being accessed or the cache is disabled. Before a cache mode change, this function completes a cache clear all command to push and invalidate any cache entries that may have changed.

Parameters

base	LMEM peripheral base address.
region	The desired region to demote of type lmem_cache_region_t.
cacheMode	The new, demoted cache mode of type lmem_cache_mode_t.

Returns

The execution result. kStatus_Success The cache demote operation is successful. kStatus_Fail The cache demote operation is failure.

21.6.13 void LMEM_EnableSystemCache (LMEM_Type * *base*, bool *enable*)

This function enables/disables the cache. It first invalidates the entire cache, then enables /disable both the cache and write buffer.

Parameters

base	LMEM peripheral base address.
The	enable or disable flag. true - enable the system cache. false - disable the system cache.

21.6.14 static void LMEM_EnableSystemWriteBuffer (LMEM_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	LMEM peripheral base address.
enable	The enable or disable flag. true - enable the system bus write buffer. false - disable the system bus write buffer.

21.6.15 void LMEM_SystemCacheInvalidateAll (LMEM_Type * *base*)

This function invalidates the entire cache both ways. Invalidate - Unconditionally clear valid and modify bits of a cache entry

Parameters

base	LMEM peripheral base address.

21.6.16 void LMEM_SystemCachePushAll (LMEM_Type * *base*)

This function pushes all modified lines in both ways (the entire cache). Push - Push a cache entry if it is valid and modified, then clear the modify bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.

21.6.17 void LMEM_SystemCacheClearAll (LMEM_Type * *base*)

This function clears the entire cache, which is a push (flush) and invalidate operation. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If the entry is not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.

21.6.18 void LMEM_SystemCacheInvalidateLine (LMEM_Type * *base*, uint32_t *address*)

This function invalidates a specific line in the cache based on the physical address passed in by the user. Invalidate - Unconditionally clears valid and modify bits of a cache entry.

Parameters

base	LMEM peripheral base address. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
address	The physical address of the cache line.

21.6.19 void LMEM_SystemCacheInvalidateMultiLines (LMEM_Type * *base*, uint32_t *address*, uint32_t *length*)

This function invalidates multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half of the cache, the function performs an entire cache invalidate function (which is more efficient than invalidating the cache line-by-line). Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Invalidate - Unconditionally clear valid and modify bits of a cache entry

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.20 void LMEM_SystemCachePushLine (LMEM_Type * *base*, uint32_t *address*)

This function pushes a specific modified line based on the physical address passed in by the user. Push - Push a cache entry if it is valid and modified, then clear the modify bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

21.6.21 void LMEM_SystemCachePushMultiLines (LMEM_Type * *base,* uint32_t *address,* uint32_t *length*)

This function pushes multiple modified lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half of the cache, the function performs an entire cache push function (which is more efficient than pushing the modified lines in the cache line-by-line). Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Push - Push a cache entry if it is valid and modified, then clear the modify bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.22 void LMEM_SystemCacheClearLine (LMEM_Type * *base,* uint32_t *address*)

This function clears a specific line based on the physical address passed in by the user. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If the entry is not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

21.6.23 void LMEM_SystemCacheClearMultiLines (LMEM_Type ∗ *base,* uint32_t *address,* uint32_t *length*)

This function clears multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half of the cache, the function performs an entire cache clear function (which is more efficient than clearing the lines in the cache line-by-line). Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If the entry is not valid or not modified, clear the valid bit.

Parameters

base	LMEM peripheral base address.
address	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
length	The length in bytes of the total amount of cache lines.

21.6.24 status_t LMEM_SystemCacheDemoteRegion (LMEM_Type ∗ *base,* lmem_cache_region_t *region,* lmem_cache_mode_t *cacheMode*)

This function allows the user to demote the cache mode of a region within the device's memory map. Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state which is the highest cache configure for each region. To maintain cache coherency, changes to the cache mode should be completed while the address space being changed is not being accessed or the cache is disabled. Before a cache mode change, this function completes a cache clear all command to push and invalidate any cache entries that may have changed.

Parameters

base	LMEM peripheral base address.
region	The desired region to demote of type lmem_cache_region_t.
cacheMode	The new, demoted cache mode of type lmem_cache_mode_t.

Returns

The execution result. kStatus_Success The cache demote operation is successful. kStatus_Fail The cache demote operation is failure.

Chapter 22
LPTMR: Low-Power Timer

22.1 Overview

The KSDK provides a driver for the Low-Power Timer (LPTMR) of Kinetis devices.

22.2 Function groups

The LPTMR driver supports operating the module as a time counter or as a pulse counter.

22.2.1 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.

22.2.2 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.

22.2.3 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.

22.2.4 Status

Provides functions to get and clear the LPTMR status.

22.2.5 Interrupt

Provides functions to enable/disable LPTMR interrupts and get the currently enabled interrupts.

22.3 Typical use case

22.3.1 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);
        }
    }
}
```

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.

22.4 Data Structure Documentation

22.4.1 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.

22.5 Enumeration Type Documentation

22.5.1 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.

22.5.2 enum lptmr_pin_polarity_t

Enumerator

kLPTMR_PinPolarityActiveHigh Pulse Counter input source is active-high.
kLPTMR_PinPolarityActiveLow Pulse Counter input source is active-low.

22.5.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.
kLPTMR_TimerModePulseCounter Pulse Counter mode.

22.5.4 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.

22.5.5 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.

22.5.6 enum lptmr_interrupt_enable_t

Enumerator

kLPTMR_TimerInterruptEnable Timer interrupt enable.

22.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

22.6 Function Documentation

22.6.1 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.

22.6.2 void LPTMR_Deinit (LPTMR_Type ∗ *base*)

Parameters

base	LPTMR peripheral base address

22.6.3 void LPTMR_GetDefaultConfig (lptmr_config_t ∗ *config*)

The default values are as follows.

```
*    config->timerMode = kLPTMR_TimerModeTimeCounter;
*    config->pinSelect = kLPTMR_PinSelectInput_0;
*    config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
*    config->enableFreeRunning = false;
*    config->bypassPrescaler = true;
*    config->prescalerClockSource = kLPTMR_PrescalerClock_1;
*    config->value = kLPTMR_Prescale_Glitch_0;
*
```

Parameters

config	A pointer to the LPTMR configuration structure.

22.6.4 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

22.6.5 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.

22.6.6 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

22.6.7 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

22.6.8 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.

22.6.9 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, which should be equal or greater than 1.

22.6.10 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

22.6.11 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

22.6.12 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 23
LPUART: Low Power UART Driver

23.1 Overview

Modules

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

23.2 LPUART Driver

23.2.1 Overview

The KSDK provides a peripheral driver for the Low Power UART (LPUART) module of Kinetis devices.

23.2.2 Typical use case

23.2.2.1 LPUART Operation

```
uint8_t ch;
LPUART_GetDefaultConfig(&user_config);
user_config.baudRate = 115200U;
config.enableTx = true;
config.enableRx = true;

LPUART_Init(LPUART1,&user_config,120000000U);

LPUART_WriteBlocking(LPUART1, txbuff, sizeof(txbuff) - 1);

while(1)
{
    LPUART_ReadBlocking(LPUART1, &ch, 1);
    LPUART_WriteBlocking(LPUART1, &ch, 1);
}
```

Data Structures

- struct lpuart_config_t
  LPUART configuration structure. More...
- struct lpuart_transfer_t
  LPUART transfer structure. More...
- struct lpuart_handle_t
  LPUART handle structure. More...

Typedefs

- typedef void(∗ lpuart_transfer_callback_t)(LPUART_Type ∗base, lpuart_handle_t ∗handle, status_t status, void ∗userData)
  LPUART transfer callback function.

Enumerations

- enum _lpuart_status {
kStatus_LPUART_TxBusy = MAKE_STATUS(kStatusGroup_LPUART, 0),
kStatus_LPUART_RxBusy = MAKE_STATUS(kStatusGroup_LPUART, 1),
kStatus_LPUART_TxIdle = MAKE_STATUS(kStatusGroup_LPUART, 2),
kStatus_LPUART_RxIdle = MAKE_STATUS(kStatusGroup_LPUART, 3),
kStatus_LPUART_TxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_LPUART, 4),
kStatus_LPUART_RxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_LPUART, 5),
kStatus_LPUART_FlagCannotClearManually = MAKE_STATUS(kStatusGroup_LPUART, 6),
kStatus_LPUART_Error = MAKE_STATUS(kStatusGroup_LPUART, 7),
kStatus_LPUART_RxRingBufferOverrun,
kStatus_LPUART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_LPUART, 9),
kStatus_LPUART_NoiseError = MAKE_STATUS(kStatusGroup_LPUART, 10),
kStatus_LPUART_FramingError = MAKE_STATUS(kStatusGroup_LPUART, 11),
kStatus_LPUART_ParityError = MAKE_STATUS(kStatusGroup_LPUART, 12),
kStatus_LPUART_BaudrateNotSupport }
  Error codes for the LPUART driver.
- enum lpuart_parity_mode_t {
kLPUART_ParityDisabled = 0x0U,
kLPUART_ParityEven = 0x2U,
kLPUART_ParityOdd = 0x3U }
  LPUART parity mode.
- enum lpuart_data_bits_t { kLPUART_EightDataBits = 0x0U }
  LPUART data bits count.
- enum lpuart_stop_bit_count_t {
kLPUART_OneStopBit = 0U,
kLPUART_TwoStopBit = 1U }
  LPUART stop bit count.
- enum _lpuart_interrupt_enable {
kLPUART_LinBreakInterruptEnable = (LPUART_BAUD_LBKDIE_MASK >> 8),
kLPUART_RxActiveEdgeInterruptEnable = (LPUART_BAUD_RXEDGIE_MASK >> 8),
kLPUART_TxDataRegEmptyInterruptEnable = (LPUART_CTRL_TIE_MASK),
kLPUART_TransmissionCompleteInterruptEnable = (LPUART_CTRL_TCIE_MASK),
kLPUART_RxDataRegFullInterruptEnable = (LPUART_CTRL_RIE_MASK),
kLPUART_IdleLineInterruptEnable = (LPUART_CTRL_ILIE_MASK),
kLPUART_RxOverrunInterruptEnable = (LPUART_CTRL_ORIE_MASK),
kLPUART_NoiseErrorInterruptEnable = (LPUART_CTRL_NEIE_MASK),
kLPUART_FramingErrorInterruptEnable = (LPUART_CTRL_FEIE_MASK),
kLPUART_ParityErrorInterruptEnable = (LPUART_CTRL_PEIE_MASK),
kLPUART_TxFifoOverflowInterruptEnable = (LPUART_FIFO_TXOFE_MASK >> 8),
kLPUART_RxFifoUnderflowInterruptEnable = (LPUART_FIFO_RXUFE_MASK >> 8) }
  LPUART interrupt configuration structure, default settings all disabled.
- enum _lpuart_flags {

kLPUART_TxDataRegEmptyFlag,
kLPUART_TransmissionCompleteFlag,
kLPUART_RxDataRegFullFlag,
kLPUART_IdleLineFlag = (LPUART_STAT_IDLE_MASK),
kLPUART_RxOverrunFlag = (LPUART_STAT_OR_MASK),
kLPUART_NoiseErrorFlag = (LPUART_STAT_NF_MASK),
kLPUART_FramingErrorFlag,
kLPUART_ParityErrorFlag = (LPUART_STAT_PF_MASK),
kLPUART_LinBreakFlag = (LPUART_STAT_LBKDIF_MASK),
kLPUART_RxActiveEdgeFlag,
kLPUART_RxActiveFlag,
kLPUART_DataMatch1Flag = LPUART_STAT_MA1F_MASK,
kLPUART_DataMatch2Flag = LPUART_STAT_MA2F_MASK,
kLPUART_NoiseErrorInRxDataRegFlag,
kLPUART_ParityErrorInRxDataRegFlag,
kLPUART_TxFifoEmptyFlag = (LPUART_FIFO_TXEMPT_MASK >> 16),
kLPUART_RxFifoEmptyFlag = (LPUART_FIFO_RXEMPT_MASK >> 16),
kLPUART_TxFifoOverflowFlag,
kLPUART_RxFifoUnderflowFlag }

LPUART status flags.

Driver version

- #define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))
  LPUART driver version 2.2.1.

Initialization and deinitialization

- status_t LPUART_Init (LPUART_Type ∗base, const lpuart_config_t ∗config, uint32_t srcClock_Hz)
  Initializes an LPUART instance with the user configuration structure and the peripheral clock.
- void LPUART_Deinit (LPUART_Type ∗base)
  Deinitializes a LPUART instance.
- void LPUART_GetDefaultConfig (lpuart_config_t ∗config)
  Gets the default configuration structure.
- status_t LPUART_SetBaudRate (LPUART_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the LPUART instance baudrate.

Status

- uint32_t LPUART_GetStatusFlags (LPUART_Type ∗base)
  Gets LPUART status flags.
- status_t LPUART_ClearStatusFlags (LPUART_Type ∗base, uint32_t mask)

Clears status flags with a provided mask.

Interrupts

- void LPUART_EnableInterrupts (LPUART_Type *base, uint32_t mask)
  Enables LPUART interrupts according to a provided mask.
- void LPUART_DisableInterrupts (LPUART_Type *base, uint32_t mask)
  Disables LPUART interrupts according to a provided mask.
- uint32_t LPUART_GetEnabledInterrupts (LPUART_Type *base)
  Gets enabled LPUART interrupts.
- static uint32_t LPUART_GetDataRegisterAddress (LPUART_Type *base)
  Gets the LPUART data register address.
- static void LPUART_EnableTxDMA (LPUART_Type *base, bool enable)
  Enables or disables the LPUART transmitter DMA request.
- static void LPUART_EnableRxDMA (LPUART_Type *base, bool enable)
  Enables or disables the LPUART receiver DMA.

Bus Operations

- static void LPUART_EnableTx (LPUART_Type *base, bool enable)
  Enables or disables the LPUART transmitter.
- static void LPUART_EnableRx (LPUART_Type *base, bool enable)
  Enables or disables the LPUART receiver.
- static void LPUART_WriteByte (LPUART_Type *base, uint8_t data)
  Writes to the transmitter register.
- static uint8_t LPUART_ReadByte (LPUART_Type *base)
  Reads the receiver register.
- void LPUART_WriteBlocking (LPUART_Type *base, const uint8_t *data, size_t length)
  Writes to the transmitter register using a blocking method.
- status_t LPUART_ReadBlocking (LPUART_Type *base, uint8_t *data, size_t length)
  Reads the receiver data register using a blocking method.

Transactional

- void LPUART_TransferCreateHandle (LPUART_Type *base, lpuart_handle_t *handle, lpuart_transfer_callback_t callback, void *userData)
  Initializes the LPUART handle.
- status_t LPUART_TransferSendNonBlocking (LPUART_Type *base, lpuart_handle_t *handle, lpuart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void LPUART_TransferStartRingBuffer (LPUART_Type *base, lpuart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void LPUART_TransferStopRingBuffer (LPUART_Type *base, lpuart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- void LPUART_TransferAbortSend (LPUART_Type *base, lpuart_handle_t *handle)
  Aborts the interrupt-driven data transmit.

- status_t LPUART_TransferGetSendCount (LPUART_Type ∗base, lpuart_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes that have been written to the LPUART transmitter register.
- status_t LPUART_TransferReceiveNonBlocking (LPUART_Type ∗base, lpuart_handle_t ∗handle, lpuart_transfer_t ∗xfer, size_t ∗receivedBytes)
  Receives a buffer of data using the interrupt method.
- void LPUART_TransferAbortReceive (LPUART_Type ∗base, lpuart_handle_t ∗handle)
  Aborts the interrupt-driven data receiving.
- status_t LPUART_TransferGetReceiveCount (LPUART_Type ∗base, lpuart_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes that have been received.
- void LPUART_TransferHandleIRQ (LPUART_Type ∗base, lpuart_handle_t ∗handle)
  LPUART IRQ handle function.
- void LPUART_TransferHandleErrorIRQ (LPUART_Type ∗base, lpuart_handle_t ∗handle)
  LPUART Error IRQ handle function.

23.2.3 Data Structure Documentation

23.2.3.1 struct lpuart_config_t

Data Fields

- uint32_t baudRate_Bps
  LPUART baud rate.
- lpuart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- lpuart_data_bits_t dataBitsCount
  Data bits count, eight (default), seven.
- bool isMsb
  Data bits order, LSB (default), MSB.
- lpuart_stop_bit_count_t stopBitCount
  Number of stop bits, 1 stop bit (default) or 2 stop bits.
- uint8_t txFifoWatermark
  TX FIFO watermark.
- uint8_t rxFifoWatermark
  RX FIFO watermark.
- bool enableTx
  Enable TX.
- bool enableRx
  Enable RX.

23.2.3.2 struct lpuart_transfer_t

Data Fields

- uint8_t ∗ data
  The buffer of data to be transfer.
- size_t dataSize

The byte count to be transfer.

23.2.3.2.0.11 Field Documentation

23.2.3.2.0.11.1 uint8_t* lpuart_transfer_t::data

23.2.3.2.0.11.2 size_t lpuart_transfer_t::dataSize

23.2.3.3 struct _lpuart_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.
- lpuart_transfer_callback_t callback
  Callback function.
- void * userData
  LPUART callback function parameter.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

23.2.3.3.0.12 Field Documentation

23.2.3.3.0.12.1 uint8_t∗ volatile lpuart_handle_t::txData

23.2.3.3.0.12.2 volatile size_t lpuart_handle_t::txDataSize

23.2.3.3.0.12.3 size_t lpuart_handle_t::txDataSizeAll

23.2.3.3.0.12.4 uint8_t∗ volatile lpuart_handle_t::rxData

23.2.3.3.0.12.5 volatile size_t lpuart_handle_t::rxDataSize

23.2.3.3.0.12.6 size_t lpuart_handle_t::rxDataSizeAll

23.2.3.3.0.12.7 uint8_t∗ lpuart_handle_t::rxRingBuffer

23.2.3.3.0.12.8 size_t lpuart_handle_t::rxRingBufferSize

23.2.3.3.0.12.9 volatile uint16_t lpuart_handle_t::rxRingBufferHead

23.2.3.3.0.12.10 volatile uint16_t lpuart_handle_t::rxRingBufferTail

23.2.3.3.0.12.11 lpuart_transfer_callback_t lpuart_handle_t::callback

23.2.3.3.0.12.12 void∗ lpuart_handle_t::userData

23.2.3.3.0.12.13 volatile uint8_t lpuart_handle_t::txState

23.2.3.3.0.12.14 volatile uint8_t lpuart_handle_t::rxState

23.2.4 Macro Definition Documentation

23.2.4.1 #define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))

23.2.5 Typedef Documentation

23.2.5.1 typedef void(∗ lpuart_transfer_callback_t)(LPUART_Type ∗base, lpuart_handle_t ∗handle, status_t status, void ∗userData)

23.2.6 Enumeration Type Documentation

23.2.6.1 enum _lpuart_status

Enumerator

kStatus_LPUART_TxBusy TX busy.
kStatus_LPUART_RxBusy RX busy.
kStatus_LPUART_TxIdle LPUART transmitter is idle.

kStatus_LPUART_RxIdle LPUART receiver is idle.
kStatus_LPUART_TxWatermarkTooLarge TX FIFO watermark too large.
kStatus_LPUART_RxWatermarkTooLarge RX FIFO watermark too large.
kStatus_LPUART_FlagCannotClearManually Some flag can't manually clear.
kStatus_LPUART_Error Error happens on LPUART.
kStatus_LPUART_RxRingBufferOverrun LPUART RX software ring buffer overrun.
kStatus_LPUART_RxHardwareOverrun LPUART RX receiver overrun.
kStatus_LPUART_NoiseError LPUART noise error.
kStatus_LPUART_FramingError LPUART framing error.
kStatus_LPUART_ParityError LPUART parity error.
kStatus_LPUART_BaudrateNotSupport Baudrate is not support in current clock source.

23.2.6.2 enum lpuart_parity_mode_t

Enumerator

kLPUART_ParityDisabled Parity disabled.
kLPUART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kLPUART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

23.2.6.3 enum lpuart_data_bits_t

Enumerator

kLPUART_EightDataBits Eight data bit.

23.2.6.4 enum lpuart_stop_bit_count_t

Enumerator

kLPUART_OneStopBit One stop bit.
kLPUART_TwoStopBit Two stop bits.

23.2.6.5 enum _lpuart_interrupt_enable

This structure contains the settings for all LPUART interrupt configurations.

Enumerator

kLPUART_LinBreakInterruptEnable LIN break detect.
kLPUART_RxActiveEdgeInterruptEnable Receive Active Edge.
kLPUART_TxDataRegEmptyInterruptEnable Transmit data register empty.

kLPUART_TransmissionCompleteInterruptEnable Transmission complete.
kLPUART_RxDataRegFullInterruptEnable Receiver data register full.
kLPUART_IdleLineInterruptEnable Idle line.
kLPUART_RxOverrunInterruptEnable Receiver Overrun.
kLPUART_NoiseErrorInterruptEnable Noise error flag.
kLPUART_FramingErrorInterruptEnable Framing error flag.
kLPUART_ParityErrorInterruptEnable Parity error flag.
kLPUART_TxFifoOverflowInterruptEnable Transmit FIFO Overflow.
kLPUART_RxFifoUnderflowInterruptEnable Receive FIFO Underflow.

23.2.6.6 enum _lpuart_flags

This provides constants for the LPUART status flags for use in the LPUART functions.

Enumerator

kLPUART_TxDataRegEmptyFlag Transmit data register empty flag, sets when transmit buffer is empty.
kLPUART_TransmissionCompleteFlag Transmission complete flag, sets when transmission activity complete.
kLPUART_RxDataRegFullFlag Receive data register full flag, sets when the receive data buffer is full.
kLPUART_IdleLineFlag Idle line detect flag, sets when idle line detected.
kLPUART_RxOverrunFlag Receive Overrun, sets when new data is received before data is read from receive register.
kLPUART_NoiseErrorFlag Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kLPUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kLPUART_ParityErrorFlag If parity enabled, sets upon parity error detection.
kLPUART_LinBreakFlag LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
kLPUART_RxActiveEdgeFlag Receive pin active edge interrupt flag, sets when active edge detected.
kLPUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.
kLPUART_DataMatch1Flag The next character to be read from LPUART_DATA matches MA1.
kLPUART_DataMatch2Flag The next character to be read from LPUART_DATA matches MA2.
kLPUART_NoiseErrorInRxDataRegFlag NOISY bit, sets if noise detected in current data word.
kLPUART_ParityErrorInRxDataRegFlag PARITYE bit, sets if noise detected in current data word.
kLPUART_TxFifoEmptyFlag TXEMPT bit, sets if transmit buffer is empty.
kLPUART_RxFifoEmptyFlag RXEMPT bit, sets if receive buffer is empty.
kLPUART_TxFifoOverflowFlag TXOF bit, sets if transmit buffer overflow occurred.
kLPUART_RxFifoUnderflowFlag RXUF bit, sets if receive buffer underflow occurred.

23.2.7 Function Documentation

23.2.7.1 status_t LPUART_Init (LPUART_Type * *base,* const lpuart_config_t * *config,* uint32_t *srcClock_Hz*)

This function configures the LPUART module with user-defined settings. Call the LPUART_GetDefault-Config() function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
* lpuart_config_t lpuartConfig;
* lpuartConfig.baudRate_Bps = 115200U;
* lpuartConfig.parityMode = kLPUART_ParityDisabled;
* lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
* lpuartConfig.isMsb = false;
* lpuartConfig.stopBitCount = kLPUART_OneStopBit;
* lpuartConfig.txFifoWatermark = 0;
* lpuartConfig.rxFifoWatermark = 1;
* LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
*
```

Parameters

base	LPUART peripheral base address.
config	Pointer to a user-defined configuration structure.
srcClock_Hz	LPUART clock source frequency in HZ.

Return values

kStatus_LPUART_-BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_Success	LPUART initialize succeed

23.2.7.2 void LPUART_Deinit (LPUART_Type * *base*)

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

base	LPUART peripheral base address.

23.2.7.3 void LPUART_GetDefaultConfig (lpuart_config_t * *config*)

This function initializes the LPUART configuration structure to a default value. The default values are-: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled;

lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

config	Pointer to a configuration structure.

23.2.7.4 status_t LPUART_SetBaudRate (LPUART_Type ∗ *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
*  LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
*
```

Parameters

base	LPUART peripheral base address.
baudRate_Bps	LPUART baudrate to be set.
srcClock_Hz	LPUART clock source frequency in HZ.

Return values

kStatus_LPUART_-BaudrateNotSupport	Baudrate is not supported in the current clock source.
kStatus_Success	Set baudrate succeeded.

23.2.7.5 uint32_t LPUART_GetStatusFlags (LPUART_Type ∗ *base*)

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators _lpuart_flags. To check for a specific status, compare the return value with enumerators in the _lpuart_flags. For example, to check whether the TX is empty:

```
*     if (kLPUART_TxDataRegEmptyFlag &
      LPUART_GetStatusFlags(LPUART1))
*     {
*         ...
*     }
*
```

Parameters

base	LPUART peripheral base address.

Returns

LPUART status flags which are ORed by the enumerators in the _lpuart_flags.

23.2.7.6 status_t LPUART_ClearStatusFlags (LPUART_Type ∗ *base*, uint32_t *mask*)

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only cleared or set by hardware are: kLPUART_TxDataRegEmptyFlag, kLPUART_TransmissionCompleteFlag, kLPUART_RxDataRegFullFlag, kLPUART_RxActiveFlag, kLPUART_NoiseErrorInRxDataRegFlag, kLPUART_ParityErrorInRxDataRegFlag, kLPUART_TxFifoEmptyFlag,kLPUART_RxFifoEmptyFlag Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

base	LPUART peripheral base address.
mask	the status flags to be cleared. The user can use the enumerators in the _lpuart_status_flag_t to do the OR operation and get the mask.

Returns

0 succeed, others failed.

Return values

kStatus_LPUART_FlagCannotClearManually	The flag can't be cleared by this function but it is cleared automatically by hardware.
kStatus_Success	Status in the mask are cleared.

23.2.7.7 void LPUART_EnableInterrupts (LPUART_Type ∗ *base*, uint32_t *mask*)

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the _lpuart_interrupt_enable. This examples shows how to enable TX empty interrupt and RX full interrupt:

```
*     LPUART_EnableInterrupts(LPUART1,
      kLPUART_TxDataRegEmptyInterruptEnable |
      kLPUART_RxDataRegFullInterruptEnable);
*
```

Parameters

base	LPUART peripheral base address.
mask	The interrupts to enable. Logical OR of _uart_interrupt_enable.

23.2.7.8 void LPUART_DisableInterrupts (LPUART_Type * *base,* uint32_t *mask*)

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See _lpuart_interrupt_enable. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
*     LPUART_DisableInterrupts(LPUART1,
      kLPUART_TxDataRegEmptyInterruptEnable |
      kLPUART_RxDataRegFullInterruptEnable);
*
```

Parameters

base	LPUART peripheral base address.
mask	The interrupts to disable. Logical OR of _lpuart_interrupt_enable.

23.2.7.9 uint32_t LPUART_GetEnabledInterrupts (LPUART_Type * *base*)

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators _lpuart_interrupt_enable. To check a specific interrupt enable status, compare the return value with enumerators in _lpuart_interrupt_enable. For example, to check whether the TX empty interrupt is enabled:

```
*     uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);
*
*     if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*     {
*         ...
*     }
*
```

Parameters

base	LPUART peripheral base address.

Returns

LPUART interrupt flags which are logical OR of the enumerators in _lpuart_interrupt_enable.

23.2.7.10 static uint32_t LPUART_GetDataRegisterAddress (LPUART_Type * *base*) [inline], [static]

This function returns the LPUART data register address, which is mainly used by the DMA/eDMA.

Parameters

base	LPUART peripheral base address.

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

23.2.7.11 static void LPUART_EnableTxDMA (LPUART_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the transmit data register empty flag, STAT[TDRE], to generate DMA requests.

Parameters

base	LPUART peripheral base address.
enable	True to enable, false to disable.

23.2.7.12 static void LPUART_EnableRxDMA (LPUART_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

base	LPUART peripheral base address.
enable	True to enable, false to disable.

23.2.7.13 static void LPUART_EnableTx (LPUART_Type ∗ *base*, bool *enable*) [inline], [static]

This function enables or disables the LPUART transmitter.

Parameters

base	LPUART peripheral base address.
enable	True to enable, false to disable.

23.2.7.14 static void LPUART_EnableRx (LPUART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the LPUART receiver.

Parameters

base	LPUART peripheral base address.
enable	True to enable, false to disable.

23.2.7.15 static void LPUART_WriteByte (LPUART_Type ∗ *base,* uint8_t *data*) [inline], [static]

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

base	LPUART peripheral base address.
data	Data write to the TX register.

23.2.7.16 static uint8_t LPUART_ReadByte (LPUART_Type ∗ *base*) [inline], [static]

This function reads data from the receiver register directly. The upper layer must ensure that the receiver register is full or that the RX FIFO has data before calling this function.

Parameters

base	LPUART peripheral base address.

Returns

Data read from data register.

23.2.7.17 void LPUART_WriteBlocking (LPUART_Type ∗ *base,* const uint8_t ∗ *data,* size_t *length*)

This function polls the transmitter register, waits for the register to be empty or for TX FIFO to have room, and writes data to the transmitter buffer.

Note

This function does not check whether all data has been sent out to the bus. Before disabling the transmitter, check the kLPUART_TransmissionCompleteFlag to ensure that the transmit is finished.

Parameters

base	LPUART peripheral base address.
data	Start address of the data to write.
length	Size of the data to write.

23.2.7.18 status_t LPUART_ReadBlocking (LPUART_Type ∗ *base*, uint8_t ∗ *data*, size_t *length*)

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the TX register.

Parameters

base	LPUART peripheral base address.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_LPUART_Rx-HardwareOverrun	Receiver overrun happened while receiving data.
kStatus_LPUART_Noise-Error	Noise error happened while receiving data.
kStatus_LPUART_-FramingError	Framing error happened while receiving data.
kStatus_LPUART_Parity-Error	Parity error happened while receiving data.
kStatus_Success	Successfully received all data.

23.2.7.19 void LPUART_TransferCreateHandle (LPUART_Type ∗ *base*, lpuart_handle_t ∗ *handle*, lpuart_transfer_callback_t *callback*, void ∗ *userData*)

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

The LPUART driver supports the "background" receiving, which means that user can set up an RX ring buffer optionally. Data received is stored into the ring buffer even when the user doesn't call the LPUART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly. The ring buffer is disabled if passing NULL as `ringBuffer`.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.
callback	Callback function.
userData	User data.

23.2.7.20 status_t LPUART_TransferSendNonBlocking (LPUART_Type ∗ *base*, lpuart_handle_t ∗ *handle*, lpuart_transfer_t ∗ *xfer*)

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the kStatus_LPUART_TxIdle as status parameter.

Note

> The kStatus_LPUART_TxIdle is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the T-X, check the kLPUART_TransmissionCompleteFlag to ensure that the transmit is finished.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.
xfer	LPUART transfer structure, see lpuart_transfer_t.

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_LPUART_TxBusy	Previous transmission still not finished, data not all written to the TX register.
kStatus_InvalidArgument	Invalid argument.

23.2.7.21 void LPUART_TransferStartRingBuffer (LPUART_Type ∗ *base*, lpuart_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 is 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 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	LPUART peripheral base address.
handle	LPUART handle pointer.
ringBuffer	Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
ringBufferSize	size of the ring buffer.

23.2.7.22 void LPUART_TransferStopRingBuffer (LPUART_Type ∗ *base*, lpuart_handle_t ∗ *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

23.2.7.23 void LPUART_TransferAbortSend (LPUART_Type ∗ *base*, lpuart_handle_t ∗ *handle*)

This function aborts the interrupt driven data sending. The user can get the remainBtyes to find out how many bytes are not sent out.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

23.2.7.24 status_t LPUART_TransferGetSendCount (LPUART_Type ∗ *base*, lpuart_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes that have been written to LPUART TX register by an interrupt method.

Parameters

base	LPUART peripheral base address.
handle	LPUART 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`;

23.2.7.25 status_t LPUART_TransferReceiveNonBlocking (LPUART_Type * *base*, lpuart_handle_t * *handle*, lpuart_transfer_t * *xfer*, size_t * *receivedBytes*)

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are 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 for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter kStatus_UART_RxIdle. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to xfer->data, which returns with the parameter `receivedBytes` set to 5. For the remaining 5 bytes, the newly arrived data is saved from xfer->data[5]. When 5 bytes are received, the LPUART 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 xfer->data. When all data is received, the upper layer is notified.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.
xfer	LPUART transfer structure, see #uart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into the transmit queue.
kStatus_LPUART_Rx-Busy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

23.2.7.26 void LPUART_TransferAbortReceive (LPUART_Type ∗ *base,* lpuart_handle_t ∗ *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

23.2.7.27 status_t LPUART_TransferGetReceiveCount (LPUART_Type ∗ *base,* lpuart_handle_t ∗ *handle,* uint32_t ∗ *count*)

This function gets the number of bytes that have been received.

Parameters

base	LPUART peripheral base address.
handle	LPUART 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`;

23.2.7.28 void LPUART_TransferHandleIRQ (LPUART_Type ∗ *base,* lpuart_handle_t ∗ *handle*)

This function handles the LPUART transmit and receive IRQ request.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

23.2.7.29 void LPUART_TransferHandleErrorIRQ (LPUART_Type * *base*, lpuart_handle_t * *handle*)

This function handles the LPUART error IRQ request.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

23.3 LPUART DMA Driver

23.3.1 Overview

Data Structures

- struct lpuart_dma_handle_t
  LPUART DMA handle. More...

Typedefs

- typedef void(∗ lpuart_dma_transfer_callback_t)(LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, status_t status, void ∗userData)
  LPUART transfer callback function.

EDMA transactional

- void LPUART_TransferCreateHandleDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, lpuart_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txDmaHandle, dma_-handle_t ∗rxDmaHandle)
  Initializes the LPUART handle which is used in transactional functions.
- status_t LPUART_TransferSendDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, lpuart_transfer_t ∗xfer)
  Sends data using DMA.
- status_t LPUART_TransferReceiveDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, lpuart_transfer_t ∗xfer)
  Receives data using DMA.
- void LPUART_TransferAbortSendDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle)
  Aborts the sent data using DMA.
- void LPUART_TransferAbortReceiveDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle)
  Aborts the received data using DMA.
- status_t LPUART_TransferGetSendCountDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes written to the LPUART TX register.
- status_t LPUART_TransferGetReceiveCountDMA (LPUART_Type ∗base, lpuart_dma_handle_-t ∗handle, uint32_t ∗count)
  Gets the number of received bytes.

23.3.2 Data Structure Documentation

23.3.2.1 struct _lpuart_dma_handle

Data Fields

- lpuart_dma_transfer_callback_t callback

Callback function.
- void ∗ userData
  LPUART 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.

23.3.2.1.0.13 Field Documentation

23.3.2.1.0.13.1 lpuart_dma_transfer_callback_t lpuart_dma_handle_t::callback

23.3.2.1.0.13.2 void∗ lpuart_dma_handle_t::userData

23.3.2.1.0.13.3 size_t lpuart_dma_handle_t::rxDataSizeAll

23.3.2.1.0.13.4 size_t lpuart_dma_handle_t::txDataSizeAll

23.3.2.1.0.13.5 dma_handle_t∗ lpuart_dma_handle_t::txDmaHandle

23.3.2.1.0.13.6 dma_handle_t∗ lpuart_dma_handle_t::rxDmaHandle

23.3.2.1.0.13.7 volatile uint8_t lpuart_dma_handle_t::txState

23.3.3 Typedef Documentation

23.3.3.1 typedef void(∗ lpuart_dma_transfer_callback_t)(LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, status_t status, void ∗userData)

23.3.4 Function Documentation

23.3.4.1 void LPUART_TransferCreateHandleDMA (LPUART_Type ∗ *base*, lpuart_dma_handle_t ∗ *handle*, lpuart_dma_transfer_callback_t *callback*, void ∗ *userData*, dma_handle_t ∗ *txDmaHandle*, dma_handle_t ∗ *rxDmaHandle*)

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_dma_handle_t structure.
callback	Callback function.
userData	User data.
txDmaHandle	User-requested DMA handle for TX DMA transfer.
rxDmaHandle	User-requested DMA handle for RX DMA transfer.

23.3.4.2 status_t LPUART_TransferSendDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*, lpuart_transfer_t * *xfer*)

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

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.
xfer	LPUART DMA transfer structure. See lpuart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_LPUART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

23.3.4.3 status_t LPUART_TransferReceiveDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*, lpuart_transfer_t * *xfer*)

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

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_dma_handle_t structure.
xfer	LPUART DMA transfer structure. See lpuart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_LPUART_Rx-Busy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

23.3.4.4 void LPUART_TransferAbortSendDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*)

This function aborts send data using DMA.

Parameters

base	LPUART peripheral base address
handle	Pointer to lpuart_dma_handle_t structure

23.3.4.5 void LPUART_TransferAbortReceiveDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*)

This function aborts the received data using DMA.

Parameters

base	LPUART peripheral base address
handle	Pointer to lpuart_dma_handle_t structure

23.3.4.6 status_t LPUART_TransferGetSendCountDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to LPUART TX register by DMA.

Parameters

base	LPUART peripheral base address.

handle	LPUART 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`;

23.3.4.7 status_t LPUART_TransferGetReceiveCountDMA (LPUART_Type ∗ *base,* lpuart_dma_handle_t ∗ *handle,* uint32_t ∗ *count*)

This function gets the number of received bytes.

Parameters

base	LPUART peripheral base address.
handle	LPUART 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`;

23.4 LPUART eDMA Driver

23.4.1 Overview

Data Structures

- struct lpuart_edma_handle_t
  LPUART eDMA handle. More...

Typedefs

- typedef void(∗ lpuart_edma_transfer_callback_t)(LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, status_t status, void ∗userData)
  LPUART transfer callback function.

eDMA transactional

- void LPUART_TransferCreateHandleEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, lpuart_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗txEdmaHandle, edma_handle_t ∗rxEdmaHandle)
  Initializes the LPUART handle which is used in transactional functions.
- status_t LPUART_SendEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, lpuart_transfer_t ∗xfer)
  Sends data using eDMA.
- status_t LPUART_ReceiveEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, lpuart_transfer_t ∗xfer)
  Receives data using eDMA.
- void LPUART_TransferAbortSendEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle)
  Aborts the sent data using eDMA.
- void LPUART_TransferAbortReceiveEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle)
  Aborts the received data using eDMA.
- status_t LPUART_TransferGetSendCountEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, uint32_t ∗count)
  Gets the number of bytes written to the LPUART TX register.
- status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, uint32_t ∗count)
  Gets the number of received bytes.

23.4.2 Data Structure Documentation

23.4.2.1 struct _lpuart_edma_handle

Data Fields

- lpuart_edma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  LPUART 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.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

23.4.2.1.0.14 Field Documentation

23.4.2.1.0.14.1 lpuart_edma_transfer_callback_t lpuart_edma_handle_t::callback

23.4.2.1.0.14.2 void* lpuart_edma_handle_t::userData

23.4.2.1.0.14.3 size_t lpuart_edma_handle_t::rxDataSizeAll

23.4.2.1.0.14.4 size_t lpuart_edma_handle_t::txDataSizeAll

23.4.2.1.0.14.5 edma_handle_t* lpuart_edma_handle_t::txEdmaHandle

23.4.2.1.0.14.6 edma_handle_t* lpuart_edma_handle_t::rxEdmaHandle

23.4.2.1.0.14.7 uint8_t lpuart_edma_handle_t::nbytes

23.4.2.1.0.14.8 volatile uint8_t lpuart_edma_handle_t::txState

23.4.3 Typedef Documentation

23.4.3.1 typedef void(* lpuart_edma_transfer_callback_t)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)

23.4.4 Function Documentation

23.4.4.1 void LPUART_TransferCreateHandleEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, lpuart_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *txEdmaHandle*, edma_handle_t * *rxEdmaHandle*)

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_edma_handle_t structure.
callback	Callback function.
userData	User data.
txEdmaHandle	User requested DMA handle for TX DMA transfer.
rxEdmaHandle	User requested DMA handle for RX DMA transfer.

23.4.4.2 status_t LPUART_SendEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, lpuart_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	LPUART peripheral base address.
handle	LPUART handle pointer.
xfer	LPUART eDMA transfer structure. See lpuart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_LPUART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

23.4.4.3 status_t LPUART_ReceiveEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, lpuart_transfer_t * *xfer*)

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_edma_handle_t structure.
xfer	LPUART eDMA transfer structure, see lpuart_transfer_t.

Return values

kStatus_Success	if succeed, others fail.
kStatus_LPUART_Rx-Busy	Previous transfer ongoing.
kStatus_InvalidArgument	Invalid argument.

23.4.4.4 void LPUART_TransferAbortSendEDMA (LPUART_Type ∗ *base*, lpuart_edma_handle_t ∗ *handle*)

This function aborts the sent data using eDMA.

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_edma_handle_t structure.

23.4.4.5 void LPUART_TransferAbortReceiveEDMA (LPUART_Type ∗ *base*, lpuart_edma_handle_t ∗ *handle*)

This function aborts the received data using eDMA.

Parameters

base	LPUART peripheral base address.
handle	Pointer to lpuart_edma_handle_t structure.

23.4.4.6 status_t LPUART_TransferGetSendCountEDMA (LPUART_Type ∗ *base*, lpuart_edma_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of bytes written to the LPUART TX register by DMA.

Parameters

base	LPUART peripheral base address.
handle	LPUART 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`;

23.4.4.7 status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type ∗ *base*, lpuart_edma_handle_t ∗ *handle*, uint32_t ∗ *count*)

This function gets the number of received bytes.

Parameters

base	LPUART peripheral base address.
handle	LPUART 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`;

23.5 LPUART µCOS/II Driver

23.5.1 Overview

Data Structures

- struct lpuart_rtos_config_t
  LPUART RTOS configuration structure. More...

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t ∗handle, lpuart_handle_t ∗t_handle, const lpuart_-rtos_config_t ∗cfg)
  Initializes an LPUART instance for operation in RTOS.
- int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗handle)
  Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int LPUART_RTOS_Send (lpuart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

23.5.2 Data Structure Documentation

23.5.2.1 struct lpuart_rtos_config_t

Data Fields

- LPUART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- lpuart_parity_mode_t parity
  Parity setting.
- lpuart_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.

23.5.3 Function Documentation

23.5.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t ∗ *handle,* lpuart_handle_t ∗ *t_handle,* const lpuart_rtos_config_t ∗ *cfg*)

Parameters

handle	The RTOS LPUART handle, the pointer to an allocated space for RTOS context.
lpuart_t_-handle	The pointer to an allocated space to store the transactional layer internal state.
cfg	The pointer to the parameters required to configure the LPUART after initialization.

Returns

0 succeed, others failed

23.5.3.2 int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗ *handle*)

This function deinitializes the LPUART module, sets all register values to the reset value, and releases the resources.

Parameters

handle	The RTOS LPUART handle.

23.5.3.3 int LPUART_RTOS_Send (lpuart_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 LPUART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

23.5.3.4 int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗ *handle,* uint8_t ∗ *buffer,* uint32_t *length,* size_t ∗ *received*)

This function receives data from LPUART. 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 LPUART 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.

23.6 LPUART µCOS/III Driver

23.6.1 Overview

Data Structures

- struct lpuart_rtos_config_t
  LPUART RTOS configuration structure. More...

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t ∗handle, lpuart_handle_t ∗t_handle, const lpuart_rtos_config_t ∗cfg)
  Initializes an LPUART instance for operation in RTOS.
- int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗handle)
  Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int LPUART_RTOS_Send (lpuart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

23.6.2 Data Structure Documentation

23.6.2.1 struct lpuart_rtos_config_t

Data Fields

- LPUART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- lpuart_parity_mode_t parity
  Parity setting.
- lpuart_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.

23.6.3 Function Documentation

23.6.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t ∗ *handle,* lpuart_handle_t ∗ *t_handle,* const lpuart_rtos_config_t ∗ *cfg*)

Parameters

handle	The RTOS LPUART handle, the pointer to allocated space for RTOS context.
lpuart_t_-handle	The pointer to allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the LPUART after initialization.

Returns

0 succeed, others failed

23.6.3.2 int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗ *handle*)

This function deinitializes the LPUART modulem, set all register value to reset value and releases the resources.

Parameters

handle	The RTOS LPUART handle.

23.6.3.3 int LPUART_RTOS_Send (lpuart_rtos_handle_t ∗ *handle,* const uint8_t ∗ *buffer,* uint32_t *length*)

This function sends data. It is synchronous API. If the HW buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS LPUART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

23.6.3.4 int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗ *handle,* uint8_t ∗ *buffer,* uint32_t *length,* size_t ∗ *received*)

It is a synchronous API.

This function receives data from LPUART. If any data is immediately available it will be returned imidiately and the number of bytes received.

Parameters

handle	The RTOS LPUART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to variable of size_t where the number of received data will be filled.

23.7 LPUART FreeRTOS Driver

23.7.1 Overview

Data Structures

- struct lpuart_rtos_config_t
  LPUART RTOS configuration structure. More...

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t ∗handle, lpuart_handle_t ∗t_handle, const lpuart_rtos_config_t ∗cfg)
  Initializes an LPUART instance for operation in RTOS.
- int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗handle)
  Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int LPUART_RTOS_Send (lpuart_rtos_handle_t ∗handle, const uint8_t ∗buffer, uint32_t length)
  Sends data in the background.
- int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗handle, uint8_t ∗buffer, uint32_t length, size_t ∗received)
  Receives data.

23.7.2 Data Structure Documentation

23.7.2.1 struct lpuart_rtos_config_t

Data Fields

- LPUART_Type ∗ base
  UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- lpuart_parity_mode_t parity
  Parity setting.
- lpuart_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.

23.7.3 Function Documentation

23.7.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t * *handle*, lpuart_handle_t * *t_handle*, const lpuart_rtos_config_t * *cfg*)

Parameters

handle	The RTOS LPUART handle, the pointer to an allocated space for RTOS context.
t_handle	The pointer to an allocated space to store the transactional layer internal state.
cfg	The pointer to the parameters required to configure the LPUART after initialization.

Returns

0 succeed, others failed

23.7.3.2 int LPUART_RTOS_Deinit (lpuart_rtos_handle_t ∗ *handle*)

This function deinitializes the LPUART module, sets all register value to the reset value, and releases the resources.

Parameters

handle	The RTOS LPUART handle.

23.7.3.3 int LPUART_RTOS_Send (lpuart_rtos_handle_t ∗ *handle,* const uint8_t ∗ *buffer,* uint32_t *length*)

This function sends data. It is an synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS LPUART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

23.7.3.4 int LPUART_RTOS_Receive (lpuart_rtos_handle_t ∗ *handle,* uint8_t ∗ *buffer,* uint32_t *length,* size_t ∗ *received*)

This function receives data from LPUART. It is an synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS LPUART 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.

Chapter 24
MPU: Memory Protection Unit

24.1 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 their appropriateness.

24.2 Initialization and Deinitialization

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 an 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);
```

24.3 Basic Control Operations

MPU can be enabled/disabled for the entire memory protection region by calling the MPU_Enable() function. 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.

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.
- #define MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX (3)
  define the maximum index of master with privileged rights.

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, 1))
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.

24.4 Data Structure Documentation

24.4.1 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.

24.4.1.0.0.15 Field Documentation

24.4.1.0.0.15.1 uint8_t mpu_hardware_info_t::hardwareRevisionLevel

24.4.1.0.0.15.2 uint8_t mpu_hardware_info_t::slavePortsNumbers

24.4.1.0.0.15.3 mpu_region_total_num_t mpu_hardware_info_t::regionsNumbers

24.4.2 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.
- uint8_t processorIdentification
  Access error processor identification.

24.4.2.0.0.16 Field Documentation

24.4.2.0.0.16.1 uint32_t mpu_access_err_info_t::master

24.4.2.0.0.16.2 mpu_err_attributes_t mpu_access_err_info_t::attributes

24.4.2.0.0.16.3 mpu_err_access_type_t mpu_access_err_info_t::accessType

24.4.2.0.0.16.4 mpu_err_access_control_t mpu_access_err_info_t::accessControl

24.4.2.0.0.16.5 uint32_t mpu_access_err_info_t::address

24.4.2.0.0.16.6 uint8_t mpu_access_err_info_t::processorIdentification

24.4.3 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.
- bool processIdentifierEnable
  Enables or disables process identifier.

24.4.3.0.0.17 Field Documentation

24.4.3.0.0.17.1 mpu_supervisor_access_rights_t mpu_rwxrights_master_access_control_t::superAccessRights

24.4.3.0.0.17.2 mpu_user_access_rights_t mpu_rwxrights_master_access_control_t::userAccessRights

24.4.3.0.0.17.3 bool mpu_rwxrights_master_access_control_t::processIdentifierEnable

24.4.4 struct mpu_rwrights_master_access_control_t

Data Fields

- bool writeEnable
  Enables or disables write permission.
- bool readEnable
  Enables or disables read permission.

24.4.4.0.0.18 Field Documentation

24.4.4.0.0.18.1 bool mpu_rwrights_master_access_control_t::writeEnable

24.4.4.0.0.18.2 bool mpu_rwrights_master_access_control_t::readEnable

24.4.5 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 that the 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 guarantees that the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

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.
- uint8_t processIdentifier
  Process identifier used when "processIdentifierEnable" set with true.
- uint8_t processIdMask
  Process identifier mask.

24.4.5.0.0.19 Field Documentation

24.4.5.0.0.19.1 uint32_t mpu_region_config_t::regionNum

24.4.5.0.0.19.2 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.

24.4.5.0.0.19.3 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.

24.4.5.0.0.19.4 mpu_rwxrights_master_access_control_t mpu_region_config_t::accessRights1[4]

24.4.5.0.0.19.5 mpu_rwrights_master_access_control_t mpu_region_config_t::accessRights2[4]

24.4.5.0.0.19.6 uint8_t mpu_region_config_t::processIdentifier

24.4.5.0.0.19.7 uint8_t mpu_region_config_t::processIdMask

The setting bit will ignore the same bit in process identifier.

24.4.6 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.

24.4.6.0.0.20 Field Documentation

24.4.6.0.0.20.1 mpu_region_config_t mpu_config_t::regionConfig

24.4.6.0.0.20.2 struct _mpu_config∗ mpu_config_t::next

24.5 Macro Definition Documentation

24.5.1 #define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

24.5.2 #define MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n) (n * 6)

24.5.3 #define MPU_REGION_RWXRIGHTS_MASTER_MASK(n) (0x1Fu << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))

24.5.4 #define MPU_REGION_RWXRIGHTS_MASTER_WIDTH 5

24.5.5 #define MPU_REGION_RWXRIGHTS_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_MASK(n))

24.5.6 #define MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n) (n * 6 + MPU_REGION_RWXRIGHTS_MASTER_WIDTH)

24.5.7 #define MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n) (0x1u << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))

24.5.8 #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))

24.5.9 #define MPU_REGION_RWRIGHTS_MASTER_SHIFT(n) ((n - FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_COUNT) * 2 + 24)

24.5.10 #define MPU_REGION_RWRIGHTS_MASTER_MASK(n) (0x3u << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))

24.5.11 #define MPU_REGION_RWRIGHTS_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWRIGHTS_MASTER_MASK(n))

24.5.12 #define MPU_SLAVE_PORT_NUM (4u)

24.5.13 #define MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX (3)

24.6 Enumeration Type Documentation

24.6.1 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.

24.6.2 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.

24.6.3 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.

24.6.4 enum mpu_err_access_type_t

Enumerator

kMPU_ErrTypeRead MPU error access type — read.
kMPU_ErrTypeWrite MPU error access type — write.

24.6.5 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.

24.6.6 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.

24.6.7 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.

24.7 Function Documentation

24.7.1 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.

24.7.2 void MPU_Deinit (MPU_Type ∗ *base*)

Parameters

base	MPU peripheral base address.

24.7.3 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.

24.7.4 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.

24.7.5 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".

24.7.6 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".

24.7.7 void MPU_SetRegionAddr (MPU_Type ∗ *base*, uint32_t *regionNum*, uint32_t *startAddr*, uint32_t *endAddr*)

Memory region start address. Note: bit0 ∼ 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 ∼ bit4 always be marked as 1 by MPU. The end address used by the 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.

24.7.8 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. Except the normal read and write rights, the execute rights are also allowed for these masters. The privilege rights masters normally range from bus masters 0 - 3. However, the maximum master number is device-specific. See the "MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX". 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 MPU_PRIVILEGED_RIGHTS_-MASTER_MAX_INDEX.
accessRights	The pointer to the MPU access rights configuration. See "mpu_rwxrights_master_-access_control_t".

24.7.9 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".

24.7.10 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.

24.7.11 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 25
PDB: Programmable Delay Block

25.1 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 the 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 events.

- 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.

25.2 Typical use case

25.2.1 Working as basic PDB 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);
}
```

25.2.2 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))
        {
        }
        // ...
    }
}
```

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 counter period.
- 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 the 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 pre-trigger in the PDB module.
- static void PDB_SetADCPreTriggerDelayValue (PDB_Type *base, uint32_t channel, uint32_t pre-Channel, uint32_t value)
  Sets the value for the 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 status flags.

DAC Interval Trigger

- void PDB_SetDACTriggerConfig (PDB_Type ∗base, uint32_t channel, pdb_dac_trigger_config_t ∗config)
  Configures the DAC trigger in the PDB module.
- static void PDB_SetDACTriggerIntervalValue (PDB_Type ∗base, uint32_t channel, uint32_t value)
  Sets the value for the DAC interval event.

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 the pulse out trigger.

25.3 Data Structure Documentation

25.3.1 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.

25.3.1.0.0.21 Field Documentation

25.3.1.0.0.21.1 pdb_load_value_mode_t pdb_config_t::loadValueMode

25.3.1.0.0.21.2 pdb_prescaler_divider_t pdb_config_t::prescalerDivider

25.3.1.0.0.21.3 pdb_divider_multiplication_factor_t pdb_config_t::dividerMultiplicationFactor

25.3.1.0.0.21.4 pdb_trigger_input_source_t pdb_config_t::triggerInputSource

25.3.1.0.0.21.5 bool pdb_config_t::enableContinuousMode

25.3.2 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.

25.3.2.0.0.22 Field Documentation

25.3.2.0.0.22.1 uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask

25.3.2.0.0.22.2 uint32_t pdb_adc_pretrigger_config_t::enableOutputMask

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

25.3.2.0.0.22.3 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.

25.3.3 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.

25.3.3.0.0.23 Field Documentation

25.3.3.0.0.23.1 bool pdb_dac_trigger_config_t::enableExternalTriggerInput

25.3.3.0.0.23.2 bool pdb_dac_trigger_config_t::enableIntervalTrigger

25.4 Macro Definition Documentation

25.4.1 #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

25.5 Enumeration Type Documentation

25.5.1 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.

25.5.2 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.

25.5.3 enum _pdb_interrupt_enable

Enumerator

kPDB_SequenceErrorInterruptEnable PDB sequence error interrupt enable.
kPDB_DelayInterruptEnable PDB delay interrupt enable.

25.5.4 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 the following operations.

- 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.

25.5.5 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.

25.5.6 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.

25.5.7 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.

25.6 Function Documentation

25.6.1 void PDB_Init (PDB_Type * *base*, const pdb_config_t * *config*)

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

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

base	PDB peripheral base address.
config	Pointer to the configuration structure. See "pdb_config_t".

25.6.2 void PDB_Deinit (PDB_Type * *base*)

Parameters

base	PDB peripheral base address.

25.6.3 void PDB_GetDefaultConfig (pdb_config_t * *config*)

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

```
*   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".

25.6.4 static void PDB_Enable (PDB_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the module or not.

25.6.5 static void PDB_DoSoftwareTrigger (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

25.6.6 static void PDB_DoLoadValues (PDB_Type * *base*) [inline], [static]

This function loads the counter values from the internal buffer. See "pdb_load_value_mode_t" about PD-B's load mode.

Parameters

base	PDB peripheral base address.

25.6.7 static void PDB_EnableDMA (PDB_Type ∗ *base,* bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the feature or not.

25.6.8 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".

25.6.9 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".

25.6.10 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".

25.6.11 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".

25.6.12 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.

25.6.13 static uint32_t PDB_GetCounterValue (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

Returns

PDB counter's current value.

25.6.14 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.

25.6.15 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 the configuration structure. See "pdb_adc_pretrigger_config_t".

25.6.16 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 set value.

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.

25.6.17 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".

25.6.18 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".

25.6.19 void PDB_SetDACTriggerConfig (PDB_Type * *base*, uint32_t *channel*, pdb_dac_trigger_config_t * *config*)

Parameters

base	PDB peripheral base address.
channel	Channel index for DAC instance.
config	Pointer to the configuration structure. See "pdb_dac_trigger_config_t".

25.6.20 static void PDB_SetDACTriggerIntervalValue (PDB_Type * *base*, uint32_t *channel*, uint32_t *value*) [inline], [static]

This fucntion sets the value for DAC interval event. DAC interval trigger triggers the DAC module to update the buffer when the DAC interval counter is equal to the set value.

Parameters

base	PDB peripheral base address.
channel	Channel index for DAC instance.
value	Setting value for the DAC interval event.

25.6.21 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	Whether the feature is enabled or not.

25.6.22 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 the 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 26
PIT: Periodic Interrupt Timer

26.1 Overview

The KSDK provides a driver for the Periodic Interrupt Timer (PIT) of Kinetis devices.

26.2 Function groups

The PIT driver supports operating the module as a time counter.

26.2.1 Initialization and deinitialization

The function PIT_Init() initializes the PIT with specified configurations. The function PIT_GetDefaultConfig() 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.

26.2.2 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. Users can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds.

26.2.3 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.

26.2.4 Status

Provides functions to get and clear the PIT status.

26.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

26.3 Typical use case

26.3.1 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;
        }
    }
}
```

Data Structures

- struct pit_config_t
  PIT configuration 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 operations.
- void PIT_Deinit (PIT_Type ∗base)
  Gates the PIT clock and disables the PIT module.
- static void PIT_GetDefaultConfig (pit_config_t ∗config)
  Fills in the PIT configuration structure with the default settings.
- static void PIT_SetTimerChainMode (PIT_Type ∗base, pit_chnl_t channel, bool enable)
  Enables or disables chaining a timer with the previous timer.

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.

26.4 Data Structure Documentation

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

Data Fields

- bool enableRunInDebug
  true: Timers run in debug mode; false: Timers stop in debug mode

26.5 Enumeration Type Documentation

26.5.1 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.

26.5.2 enum pit_interrupt_enable_t

Enumerator

kPIT_TimerInterruptEnable Timer interrupt enable.

26.5.3 enum pit_status_flags_t

Enumerator

kPIT_TimerFlag Timer flag.

26.6 Function Documentation

26.6.1 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 the user's PIT config structure

26.6.2 void PIT_Deinit (PIT_Type * *base*)

Parameters

base	PIT peripheral base address

26.6.3 static void PIT_GetDefaultConfig (pit_config_t * *config*) [inline], [static]

The default values are as follows.

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

Parameters

config	Pointer to the onfiguration structure.

26.6.4 static void PIT_SetTimerChainMode (PIT_Type ∗ *base*, pit_chnl_t *channel*, bool *enable*) [inline], [static]

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

Parameters

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

26.6.5 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

26.6.6 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

26.6.7 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

26.6.8 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

26.6.9 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

26.6.10 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

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

Parameters

base	PIT peripheral base address
channel	Timer channel number
count	Timer period in units of ticks

26.6.11 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

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec.

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

Current timer counting value in ticks

26.6.12 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.

26.6.13 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 27
PMC: Power Management Controller

27.1 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.

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...
- struct pmc_high_volt_detect_config_t
  High-voltage Detect Configuration Structure. More...
- struct pmc_bandgap_buffer_config_t
  Bandgap Buffer configuration. More...

Enumerations

- enum pmc_low_volt_detect_volt_select_t {
  kPMC_LowVoltDetectLowTrip = 0U,
  kPMC_LowVoltDetectHighTrip = 1U }
  Low-voltage Detect Voltage Select.
- enum pmc_low_volt_warning_volt_select_t {
  kPMC_LowVoltWarningLowTrip = 0U,
  kPMC_LowVoltWarningMid1Trip = 1U,
  kPMC_LowVoltWarningMid2Trip = 2U,
  kPMC_LowVoltWarningHighTrip = 3U }
  Low-voltage Warning Voltage Select.
- enum pmc_high_volt_detect_volt_select_t {
  kPMC_HighVoltDetectLowTrip = 0U,
  kPMC_HighVoltDetectHighTrip = 1U }
  High-voltage Detect Voltage Select.

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)

Configures the low-voltage detect setting.
- static bool PMC_GetLowVoltDetectFlag (PMC_Type ∗base)
  Gets the Low-voltage Detect Flag status.
- static void PMC_ClearLowVoltDetectFlag (PMC_Type ∗base)
  Acknowledges clearing the Low-voltage Detect flag.
- void PMC_ConfigureLowVoltWarning (PMC_Type ∗base, const pmc_low_volt_warning_config_t ∗config)
  Configures the low-voltage warning setting.
- static bool PMC_GetLowVoltWarningFlag (PMC_Type ∗base)
  Gets the Low-voltage Warning Flag status.
- static void PMC_ClearLowVoltWarningFlag (PMC_Type ∗base)
  Acknowledges the Low-voltage Warning flag.
- void PMC_ConfigureHighVoltDetect (PMC_Type ∗base, const pmc_high_volt_detect_config_t ∗config)
  Configures the high-voltage detect setting.
- static bool PMC_GetHighVoltDetectFlag (PMC_Type ∗base)
  Gets the High-voltage Detect Flag status.
- static void PMC_ClearHighVoltDetectFlag (PMC_Type ∗base)
  Acknowledges clearing the High-voltage Detect flag.
- void PMC_ConfigureBandgapBuffer (PMC_Type ∗base, const pmc_bandgap_buffer_config_t ∗config)
  Configures the PMC bandgap.
- static bool PMC_GetPeriphIOIsolationFlag (PMC_Type ∗base)
  Gets the acknowledge Peripherals and I/O pads isolation flag.
- static void PMC_ClearPeriphIOIsolationFlag (PMC_Type ∗base)
  Acknowledges the isolation flag to Peripherals and I/O pads.
- static bool PMC_IsRegulatorInRunRegulation (PMC_Type ∗base)
  Gets the regulator regulation status.

27.2 Data Structure Documentation

27.2.1 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.
- pmc_low_volt_detect_volt_select_t voltSelect
  Low-voltage detect trip point voltage selection.

27.2.2 struct pmc_low_volt_warning_config_t

Data Fields

- bool enableInt
  Enable interrupt when low-voltage warning.

- pmc_low_volt_warning_volt_select_t voltSelect
  Low-voltage warning trip point voltage selection.

27.2.3 struct pmc_high_volt_detect_config_t

Data Fields

- bool enableInt
  Enable interrupt when high-voltage detect.
- bool enableReset
  Enable system reset when high-voltage detect.
- pmc_high_volt_detect_volt_select_t voltSelect
  High-voltage detect trip point voltage selection.

27.2.4 struct pmc_bandgap_buffer_config_t

Data Fields

- bool enable
  Enable bandgap buffer.
- bool enableInLowPowerMode
  Enable bandgap buffer in low-power mode.

27.2.4.0.0.24 Field Documentation

27.2.4.0.0.24.1 bool pmc_bandgap_buffer_config_t::enable

27.2.4.0.0.24.2 bool pmc_bandgap_buffer_config_t::enableInLowPowerMode

27.3 Macro Definition Documentation

27.3.1 #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

27.4 Enumeration Type Documentation

27.4.1 enum pmc_low_volt_detect_volt_select_t

Enumerator

kPMC_LowVoltDetectLowTrip Low-trip point selected (VLVD = VLVDL)
kPMC_LowVoltDetectHighTrip High-trip point selected (VLVD = VLVDH)

27.4.2 enum pmc_low_volt_warning_volt_select_t

Enumerator

kPMC_LowVoltWarningLowTrip Low-trip point selected (VLVW = VLVW1)
kPMC_LowVoltWarningMid1Trip Mid 1 trip point selected (VLVW = VLVW2)
kPMC_LowVoltWarningMid2Trip Mid 2 trip point selected (VLVW = VLVW3)
kPMC_LowVoltWarningHighTrip High-trip point selected (VLVW = VLVW4)

27.4.3 enum pmc_high_volt_detect_volt_select_t

Enumerator

kPMC_HighVoltDetectLowTrip Low-trip point selected (VHVD = VHVDL)
kPMC_HighVoltDetectHighTrip High-trip point selected (VHVD = VHVDH)

27.5 Function Documentation

27.5.1 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, enables or disables the interrupt, enables or disables the system reset.

Parameters

base	PMC peripheral base address.
config	Low-voltage detect configuration structure.

27.5.2 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

27.5.3 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.

27.5.4 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 enabling or disabling the interrupt.

Parameters

base	PMC peripheral base address.
config	Low-voltage warning configuration structure.

27.5.5 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.

27.5.6 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.

27.5.7 void PMC_ConfigureHighVoltDetect (PMC_Type * *base*, const pmc_high_volt_detect_config_t * *config*)

This function configures the high-voltage detect setting, including the trip point voltage setting, enabling or disabling the interrupt, enabling or disabling the system reset.

Parameters

base	PMC peripheral base address.
config	High-voltage detect configuration structure.

27.5.8 static bool PMC_GetHighVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function reads the current HVDF status. If it returns 1, a low voltage event is detected.

Parameters

base	PMC peripheral base address.

Returns

Current high-voltage detect flag
- true: High-voltage detected
- false: High-voltage not detected

27.5.9 static void PMC_ClearHighVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function acknowledges the high-voltage detection errors (write 1 to clear HVDF).

Parameters

base	PMC peripheral base address.

27.5.10 void PMC_ConfigureBandgapBuffer (PMC_Type * *base*, const pmc_bandgap_buffer_config_t * *config*)

This function configures the PMC bandgap, including the drive select and behavior in low-power mode.

Parameters

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

27.5.11 static bool PMC_GetPeriphIOIsolationFlag (PMC_Type * *base*) [inline], [static]

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

base	PMC peripheral base address.
base	Base address for current PMC instance.

Returns

ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

27.5.12 static void PMC_ClearPeriphIOIsolationFlag (PMC_Type * *base*) [inline], [static]

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

base	PMC peripheral base address.

27.5.13 static bool PMC_IsRegulatorInRunRegulation (PMC_Type * *base*) [inline], [static]

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

Parameters

base	PMC peripheral base address.
base	Base address for current PMC instance.

Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

Chapter 28
PORT: Port Control and Interrupts

28.1 Overview

The KSDK provides a driver for the Port Control and Interrupts (PORT) module of Kinetis devices.

28.2 Typical configuration use case

28.2.1 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);
```

28.2.2 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);
```

Data Structures

- struct port_digital_filter_config_t
  PORT digital filter feature configuration definition. More...
- struct port_pin_config_t
  PORT pin configuration structure. More...

Enumerations

- enum _port_pull {
  kPORT_PullDisable = 0U,
  kPORT_PullDown = 2U,

kPORT_PullUp = 3U }
Internal resistor pull feature selection.
- enum _port_slew_rate {
kPORT_FastSlewRate = 0U,
kPORT_SlowSlewRate = 1U }
Slew rate selection.
- enum _port_open_drain_enable {
kPORT_OpenDrainDisable = 0U,
kPORT_OpenDrainEnable = 1U }
Open Drain feature enable/disable.
- enum _port_passive_filter_enable {
kPORT_PassiveFilterDisable = 0U,
kPORT_PassiveFilterEnable = 1U }
Passive filter feature enable/disable.
- enum _port_drive_strength {
kPORT_LowDriveStrength = 0U,
kPORT_HighDriveStrength = 1U }
Configures the drive strength.
- enum _port_lock_register {
kPORT_UnlockRegister = 0U,
kPORT_LockRegister = 1U }
Unlock/lock the pin control register field[15:0].
- enum port_mux_t {
kPORT_PinDisabledOrAnalog = 0U,
kPORT_MuxAsGpio = 1U,
kPORT_MuxAlt2 = 2U,
kPORT_MuxAlt3 = 3U,
kPORT_MuxAlt4 = 4U,
kPORT_MuxAlt5 = 5U,
kPORT_MuxAlt6 = 6U,
kPORT_MuxAlt7 = 7U,
kPORT_MuxAlt8 = 8U,
kPORT_MuxAlt9 = 9U,
kPORT_MuxAlt10 = 10U,
kPORT_MuxAlt11 = 11U,
kPORT_MuxAlt12 = 12U,
kPORT_MuxAlt13 = 13U,
kPORT_MuxAlt14 = 14U,
kPORT_MuxAlt15 = 15U }
Pin mux selection.
- enum port_interrupt_t {

kPORT_InterruptOrDMADisabled = 0x0U,
kPORT_DMARisingEdge = 0x1U,
kPORT_DMAFallingEdge = 0x2U,
kPORT_DMAEitherEdge = 0x3U,
kPORT_InterruptLogicZero = 0x8U,
kPORT_InterruptRisingEdge = 0x9U,
kPORT_InterruptFallingEdge = 0xAU,
kPORT_InterruptEitherEdge = 0xBU,
kPORT_InterruptLogicOne = 0xCU }
Configures the interrupt generation condition.
- enum port_digital_filter_clock_source_t {
kPORT_BusClock = 0U,
kPORT_LpoClock = 1U }
Digital filter clock source selection.

Driver version

- #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
Version 2.0.2.

Configuration

- static void PORT_SetPinConfig (PORT_Type ∗base, uint32_t pin, const port_pin_config_t ∗config)
Sets the port PCR register.
- static void PORT_SetMultiplePinsConfig (PORT_Type ∗base, uint32_t mask, const port_pin_config_t ∗config)
Sets the port PCR register for multiple pins.
- static void PORT_SetPinMux (PORT_Type ∗base, uint32_t pin, port_mux_t mux)
Configures the pin muxing.
- static void PORT_EnablePinsDigitalFilter (PORT_Type ∗base, uint32_t mask, bool enable)
Enables the digital filter in one port, each bit of the 32-bit register represents one pin.
- static void PORT_SetDigitalFilterConfig (PORT_Type ∗base, const port_digital_filter_config_t ∗config)
Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Interrupt

- static void PORT_SetPinInterruptConfig (PORT_Type ∗base, uint32_t pin, port_interrupt_t config)
Configures the port pin interrupt/DMA request.
- static uint32_t PORT_GetPinsInterruptFlags (PORT_Type ∗base)
Reads the whole port status flag.
- static void PORT_ClearPinsInterruptFlags (PORT_Type ∗base, uint32_t mask)
Clears the multiple pin interrupt status flag.

28.3 Data Structure Documentation

28.3.1 struct port_digital_filter_config_t

Data Fields

- uint32_t digitalFilterWidth
  Set digital filter width.
- port_digital_filter_clock_source_t clockSource
  Set digital filter clockSource.

28.3.2 struct port_pin_config_t

Data Fields

- uint16_t pullSelect: 2
  No-pull/pull-down/pull-up select.
- uint16_t slewRate: 1
  Fast/slow slew rate Configure.
- uint16_t passiveFilterEnable: 1
  Passive filter enable/disable.
- uint16_t openDrainEnable: 1
  Open drain enable/disable.
- uint16_t driveStrength: 1
  Fast/slow drive strength configure.
- uint16_t mux: 3
  Pin mux Configure.
- uint16_t lockRegister: 1
  Lock/unlock the PCR field[15:0].

28.4 Macro Definition Documentation

28.4.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

28.5 Enumeration Type Documentation

28.5.1 enum _port_pull

Enumerator

kPORT_PullDisable Internal pull-up/down resistor is disabled.
kPORT_PullDown Internal pull-down resistor is enabled.
kPORT_PullUp Internal pull-up resistor is enabled.

28.5.2 enum _port_slew_rate

Enumerator

kPORT_FastSlewRate Fast slew rate is configured.
kPORT_SlowSlewRate Slow slew rate is configured.

28.5.3 enum _port_open_drain_enable

Enumerator

kPORT_OpenDrainDisable Open drain output is disabled.
kPORT_OpenDrainEnable Open drain output is enabled.

28.5.4 enum _port_passive_filter_enable

Enumerator

kPORT_PassiveFilterDisable Passive input filter is disabled.
kPORT_PassiveFilterEnable Passive input filter is enabled.

28.5.5 enum _port_drive_strength

Enumerator

kPORT_LowDriveStrength Low-drive strength is configured.
kPORT_HighDriveStrength High-drive strength is configured.

28.5.6 enum _port_lock_register

Enumerator

kPORT_UnlockRegister Pin Control Register fields [15:0] are not locked.
kPORT_LockRegister Pin Control Register fields [15:0] are locked.

28.5.7 enum port_mux_t

Enumerator

kPORT_PinDisabledOrAnalog Corresponding pin is disabled, but is used as an analog pin.

kPORT_MuxAsGpio Corresponding pin is configured as GPIO.
kPORT_MuxAlt2 Chip-specific.
kPORT_MuxAlt3 Chip-specific.
kPORT_MuxAlt4 Chip-specific.
kPORT_MuxAlt5 Chip-specific.
kPORT_MuxAlt6 Chip-specific.
kPORT_MuxAlt7 Chip-specific.
kPORT_MuxAlt8 Chip-specific.
kPORT_MuxAlt9 Chip-specific.
kPORT_MuxAlt10 Chip-specific.
kPORT_MuxAlt11 Chip-specific.
kPORT_MuxAlt12 Chip-specific.
kPORT_MuxAlt13 Chip-specific.
kPORT_MuxAlt14 Chip-specific.
kPORT_MuxAlt15 Chip-specific.

28.5.8 enum port_interrupt_t

Enumerator

kPORT_InterruptOrDMADisabled Interrupt/DMA request is disabled.
kPORT_DMARisingEdge DMA request on rising edge.
kPORT_DMAFallingEdge DMA request on falling edge.
kPORT_DMAEitherEdge DMA request on either edge.
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.

28.5.9 enum port_digital_filter_clock_source_t

Enumerator

kPORT_BusClock Digital filters are clocked by the bus clock.
kPORT_LpoClock Digital filters are clocked by the 1 kHz LPO clock.

28.6 Function Documentation

28.6.1 static void PORT_SetPinConfig (PORT_Type * *base*, uint32_t *pin*, const port_pin_config_t * *config*) [inline], [static]

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

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*      kPORT_PullUp,
*      kPORT_FastSlewRate,
*      kPORT_PassiveFilterDisable,
*      kPORT_OpenDrainDisable,
*      kPORT_LowDriveStrength,
*      kPORT_MuxAsGpio,
*      kPORT_UnLockRegister,
* };
*
```

Parameters

base	PORT peripheral base pointer.
pin	PORT pin number.
config	PORT PCR register configuration structure.

28.6.2 static void PORT_SetMultiplePinsConfig (PORT_Type ∗ *base*, uint32_t *mask*, const port_pin_config_t ∗ *config*) [inline], [static]

This is an example to define input pins or output pins PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*      kPORT_PullUp ,
*      kPORT_PullEnable,
*      kPORT_FastSlewRate,
*      kPORT_PassiveFilterDisable,
*      kPORT_OpenDrainDisable,
*      kPORT_LowDriveStrength,
*      kPORT_MuxAsGpio,
*      kPORT_UnlockRegister,
* };
*
```

Parameters

base	PORT peripheral base pointer.
mask	PORT pin number macro.
config	PORT PCR register configuration structure.

28.6.3 static void PORT_SetPinMux (PORT_Type ∗ *base*, uint32_t *pin*, port_mux_t *mux*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
pin	PORT pin number.
mux	pin muxing slot selection. • kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function. • kPORT_MuxAsGpio : Set as GPIO. • kPORT_MuxAlt2 : chip-specific. • kPORT_MuxAlt3 : chip-specific. • kPORT_MuxAlt4 : chip-specific. • kPORT_MuxAlt5 : chip-specific. • kPORT_MuxAlt6 : chip-specific. • kPORT_MuxAlt7 : chip-specific. : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

28.6.4 static void PORT_EnablePinsDigitalFilter (PORT_Type ∗ *base,* uint32_t *mask,* bool *enable*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
mask	PORT pin number macro.

28.6.5 static void PORT_SetDigitalFilterConfig (PORT_Type ∗ *base,* const port_digital_filter_config_t ∗ *config*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
config	PORT digital filter configuration structure.

28.6.6 static void PORT_SetPinInterruptConfig (PORT_Type ∗ *base,* uint32_t *pin,* port_interrupt_t *config*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
pin	PORT pin number.
config	PORT pin interrupt configuration. • kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled. • kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit). • kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit). • kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit). • #kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit). • #kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit). • #kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit). • kPORT_InterruptLogicZero : Interrupt when logic zero. • kPORT_InterruptRisingEdge : Interrupt on rising edge. • kPORT_InterruptFallingEdge: Interrupt on falling edge. • kPORT_InterruptEitherEdge : Interrupt on either edge. • kPORT_InterruptLogicOne : Interrupt when logic one. • #kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit). • #kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

28.6.7 static uint32_t PORT_GetPinsInterruptFlags (PORT_Type * *base*) [inline], [static]

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

base	PORT peripheral base pointer.

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

28.6.8 static void PORT_ClearPinsInterruptFlags (PORT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
mask	PORT pin number macro.

Chapter 29
QSPI: Quad Serial Peripheral Interface Driver

29.1 Overview

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

QSPI driver includes functional APIs and EDMA transactional APIs.

Functional APIs are feature/property target low level APIs. Functional APIs can be used for QSPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the QSPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. QSPI functional operation groups provide the functional API set.

Transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the qspi_handle_t as the first parameter. Initialize the handle by calling the QSPI_TransferTxCreateHandleEDMA() or QSPI_TransferRxCreateHandleEDMA() API.

Transactional APIs support asynchronous transfer. This means that the functions QSPI_TransferSendEDMA() and QSPI_TransferReceiveEDMA() set up EDMA for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_QSPI_Idle status.

Modules

- QSPI eDMA Driver

Data Structures

- struct qspi_dqs_config_t
  DQS configure features. More...
- struct qspi_flash_timing_t
  Flash timing configuration. More...
- struct qspi_config_t
  QSPI configuration structure. More...
- struct qspi_flash_config_t
  External flash configuration items. More...
- struct qspi_transfer_t
  Transfer structure for QSPI. More...

Enumerations

- enum _status_t {
  kStatus_QSPI_Idle = MAKE_STATUS(kStatusGroup_QSPI, 0),
  kStatus_QSPI_Busy = MAKE_STATUS(kStatusGroup_QSPI, 1),
  kStatus_QSPI_Error = MAKE_STATUS(kStatusGroup_QSPI, 2) }
  Status structure of QSPI.
- enum qspi_read_area_t {
  kQSPI_ReadAHB = 0x0U,
  kQSPI_ReadIP }
  QSPI read data area, from IP FIFO or AHB buffer.
- enum qspi_command_seq_t {
  kQSPI_IPSeq = QuadSPI_SPTRCLR_IPPTRC_MASK,
  kQSPI_BufferSeq = QuadSPI_SPTRCLR_BFPTRC_MASK }
  QSPI command sequence type.
- enum qspi_fifo_t {
  kQSPI_TxFifo = QuadSPI_MCR_CLR_TXF_MASK,
  kQSPI_RxFifo = QuadSPI_MCR_CLR_RXF_MASK,
  kQSPI_AllFifo = QuadSPI_MCR_CLR_TXF_MASK | QuadSPI_MCR_CLR_RXF_MASK }
  QSPI buffer type.
- enum qspi_endianness_t {
  kQSPI_64BigEndian = 0x0U,
  kQSPI_32LittleEndian,
  kQSPI_32BigEndian,
  kQSPI_64LittleEndian }
  QSPI transfer endianess.
- enum _qspi_error_flags {
  kQSPI_DataLearningFail = QuadSPI_FR_DLPFF_MASK,
  kQSPI_TxBufferFill = QuadSPI_FR_TBFF_MASK,
  kQSPI_TxBufferUnderrun = QuadSPI_FR_TBUF_MASK,
  kQSPI_IllegalInstruction = QuadSPI_FR_ILLINE_MASK,
  kQSPI_RxBufferOverflow = QuadSPI_FR_RBOF_MASK,
  kQSPI_RxBufferDrain = QuadSPI_FR_RBDF_MASK,
  kQSPI_AHBSequenceError = QuadSPI_FR_ABSEF_MASK,
  kQSPI_AHBIllegalTransaction = QuadSPI_FR_AITEF_MASK,
  kQSPI_AHBIllegalBurstSize = QuadSPI_FR_AIBSEF_MASK,
  kQSPI_AHBBufferOverflow = QuadSPI_FR_ABOF_MASK,
  kQSPI_IPCommandUsageError = QuadSPI_FR_IUEF_MASK,
  kQSPI_IPCommandTriggerDuringAHBAccess = QuadSPI_FR_IPAEF_MASK,
  kQSPI_IPCommandTriggerDuringIPAccess = QuadSPI_FR_IPIEF_MASK,
  kQSPI_IPCommandTriggerDuringAHBGrant = QuadSPI_FR_IPGEF_MASK,
  kQSPI_IPCommandTransactionFinished = QuadSPI_FR_TFF_MASK,
  kQSPI_FlagAll = 0x8C83F8D1U }
  QSPI error flags.
- enum _qspi_flags {

kQSPI_DataLearningSamplePoint = QuadSPI_SR_DLPSMP_MASK,
kQSPI_TxBufferFull = QuadSPI_SR_TXFULL_MASK,
kQSPI_TxDMA = QuadSPI_SR_TXDMA_MASK,
kQSPI_TxWatermark = QuadSPI_SR_TXWA_MASK,
kQSPI_TxBufferEnoughData = QuadSPI_SR_TXEDA_MASK,
kQSPI_RxDMA = QuadSPI_SR_RXDMA_MASK,
kQSPI_RxBufferFull = QuadSPI_SR_RXFULL_MASK,
kQSPI_RxWatermark = QuadSPI_SR_RXWE_MASK,
kQSPI_AHB3BufferFull = QuadSPI_SR_AHB3FUL_MASK,
kQSPI_AHB2BufferFull = QuadSPI_SR_AHB2FUL_MASK,
kQSPI_AHB1BufferFull = QuadSPI_SR_AHB1FUL_MASK,
kQSPI_AHB0BufferFull = QuadSPI_SR_AHB0FUL_MASK,
kQSPI_AHB3BufferNotEmpty = QuadSPI_SR_AHB3NE_MASK,
kQSPI_AHB2BufferNotEmpty = QuadSPI_SR_AHB2NE_MASK,
kQSPI_AHB1BufferNotEmpty = QuadSPI_SR_AHB1NE_MASK,
kQSPI_AHB0BufferNotEmpty = QuadSPI_SR_AHB0NE_MASK,
kQSPI_AHBTransactionPending = QuadSPI_SR_AHBTRN_MASK,
kQSPI_AHBCommandPriorityGranted = QuadSPI_SR_AHBGNT_MASK,
kQSPI_AHBAccess = QuadSPI_SR_AHB_ACC_MASK,
kQSPI_IPAccess = QuadSPI_SR_IP_ACC_MASK,
kQSPI_Busy = QuadSPI_SR_BUSY_MASK,
kQSPI_StateAll = 0xEF897FE7U }

QSPI state bit.

- enum _qspi_interrupt_enable {
kQSPI_DataLearningFailInterruptEnable,
kQSPI_TxBufferFillInterruptEnable = QuadSPI_RSER_TBFIE_MASK,
kQSPI_TxBufferUnderrunInterruptEnable = QuadSPI_RSER_TBUIE_MASK,
kQSPI_IllegalInstructionInterruptEnable,
kQSPI_RxBufferOverflowInterruptEnable = QuadSPI_RSER_RBOIE_MASK,
kQSPI_RxBufferDrainInterruptEnable = QuadSPI_RSER_RBDIE_MASK,
kQSPI_AHBSequenceErrorInterruptEnable = QuadSPI_RSER_ABSEIE_MASK,
kQSPI_AHBIllegalTransactionInterruptEnable,
kQSPI_AHBIllegalBurstSizeInterruptEnable,
kQSPI_AHBBufferOverflowInterruptEnable = QuadSPI_RSER_ABOIE_MASK,
kQSPI_IPCommandUsageErrorInterruptEnable = QuadSPI_RSER_IUEIE_MASK,
kQSPI_IPCommandTriggerDuringAHBAccessInterruptEnable,
kQSPI_IPCommandTriggerDuringIPAccessInterruptEnable,
kQSPI_IPCommandTriggerDuringAHBGrantInterruptEnable,
kQSPI_IPCommandTransactionFinishedInterruptEnable,
kQSPI_AllInterruptEnable = 0x8C83F8D1U }

  QSPI interrupt enable.

- enum _qspi_dma_enable {
kQSPI_TxBufferFillDMAEnable = QuadSPI_RSER_TBFDE_MASK,
kQSPI_RxBufferDrainDMAEnable = QuadSPI_RSER_RBDDE_MASK,
kQSPI_AllDDMAEnable = QuadSPI_RSER_TBFDE_MASK | QuadSPI_RSER_RBDDE_MASK

}
QSPI DMA request flag.
- enum qspi_dqs_phrase_shift_t {
kQSPI_DQSNoPhraseShift = 0x0U,
kQSPI_DQSPhraseShift45Degree,
kQSPI_DQSPhraseShift90Degree,
kQSPI_DQSPhraseShift135Degree }
Phrase shift number for DQS mode.

Driver version

- #define FSL_QSPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
I2C driver version 2.0.1.

Initialization and deinitialization

- void QSPI_Init (QuadSPI_Type ∗base, qspi_config_t ∗config, uint32_t srcClock_Hz)
Initializes the QSPI module and internal state.
- void QSPI_GetDefaultQspiConfig (qspi_config_t ∗config)
Gets default settings for QSPI.
- void QSPI_Deinit (QuadSPI_Type ∗base)
Deinitializes the QSPI module.
- void QSPI_SetFlashConfig (QuadSPI_Type ∗base, qspi_flash_config_t ∗config)
Configures the serial flash parameter.
- void QSPI_SoftwareReset (QuadSPI_Type ∗base)
Software reset for the QSPI logic.
- static void QSPI_Enable (QuadSPI_Type ∗base, bool enable)
Enables or disables the QSPI module.

Status

- static uint32_t QSPI_GetStatusFlags (QuadSPI_Type ∗base)
Gets the state value of QSPI.
- static uint32_t QSPI_GetErrorStatusFlags (QuadSPI_Type ∗base)
Gets QSPI error status flags.
- static void QSPI_ClearErrorFlag (QuadSPI_Type ∗base, uint32_t mask)
Clears the QSPI error flags.

Interrupts

- static void QSPI_EnableInterrupts (QuadSPI_Type ∗base, uint32_t mask)
Enables the QSPI interrupts.
- static void QSPI_DisableInterrupts (QuadSPI_Type ∗base, uint32_t mask)
Disables the QSPI interrupts.

DMA Control

- static void QSPI_EnableDMA (QuadSPI_Type ∗base, uint32_t mask, bool enable)
Enables the QSPI DMA source.
- static uint32_t QSPI_GetTxDataRegisterAddress (QuadSPI_Type ∗base)

Gets the Tx data register address.
- uint32_t QSPI_GetRxDataRegisterAddress (QuadSPI_Type ∗base)
  Gets the Rx data register address used for DMA operation.

Bus Operations

- static void QSPI_SetIPCommandAddress (QuadSPI_Type ∗base, uint32_t addr)
  Sets the IP command address.
- static void QSPI_SetIPCommandSize (QuadSPI_Type ∗base, uint32_t size)
  Sets the IP command size.
- void QSPI_ExecuteIPCommand (QuadSPI_Type ∗base, uint32_t index)
  Executes IP commands located in LUT table.
- void QSPI_ExecuteAHBCommand (QuadSPI_Type ∗base, uint32_t index)
  Executes AHB commands located in LUT table.
- static void QSPI_EnableIPParallelMode (QuadSPI_Type ∗base, bool enable)
  Enables/disables the QSPI IP command parallel mode.
- static void QSPI_EnableAHBParallelMode (QuadSPI_Type ∗base, bool enable)
  Enables/disables the QSPI AHB command parallel mode.
- void QSPI_UpdateLUT (QuadSPI_Type ∗base, uint32_t index, uint32_t ∗cmd)
  Updates the LUT table.
- static void QSPI_ClearFifo (QuadSPI_Type ∗base, uint32_t mask)
  Clears the QSPI FIFO logic.
- static void QSPI_ClearCommandSequence (QuadSPI_Type ∗base, qspi_command_seq_t seq)
  @ brief Clears the command sequence for the IP/buffer command.
- void QSPI_SetReadDataArea (QuadSPI_Type ∗base, qspi_read_area_t area)
  @ brief Set the RX buffer readout area.
- void QSPI_WriteBlocking (QuadSPI_Type ∗base, uint32_t ∗buffer, size_t size)
  Sends a buffer of data bytes using a blocking method.
- static void QSPI_WriteData (QuadSPI_Type ∗base, uint32_t data)
  Writes data into FIFO.
- void QSPI_ReadBlocking (QuadSPI_Type ∗base, uint32_t ∗buffer, size_t size)
  Receives a buffer of data bytes using a blocking method.
- uint32_t QSPI_ReadData (QuadSPI_Type ∗base)
  Receives data from data FIFO.

Transactional

- static void QSPI_TransferSendBlocking (QuadSPI_Type ∗base, qspi_transfer_t ∗xfer)
  Writes data to the QSPI transmit buffer.
- static void QSPI_TransferReceiveBlocking (QuadSPI_Type ∗base, qspi_transfer_t ∗xfer)
  Reads data from the QSPI receive buffer in polling way.

29.2 Data Structure Documentation

29.2.1 struct qspi_dqs_config_t

Data Fields

- uint32_t portADelayTapNum
  Delay chain tap number selection for QSPI port A DQS.

- uint32_t portBDelayTapNum
  Delay chain tap number selection for QSPI port B DQS.
- qspi_dqs_phrase_shift_t shift
  Phase shift for internal DQS generation.
- bool enableDQSClkInverse
  Enable inverse clock for internal DQS generation.
- bool enableDQSPadLoopback
  Enable DQS loop back from DQS pad.
- bool enableDQSLoopback
  Enable DQS loop back.

29.2.2 struct qspi_flash_timing_t

Data Fields

- uint32_t dataHoldTime
  Serial flash data in hold time.
- uint32_t CSHoldTime
  Serial flash CS hold time in terms of serial flash clock cycles.
- uint32_t CSSetupTime
  Serial flash CS setup time in terms of serial flash clock cycles.

29.2.3 struct qspi_config_t

Data Fields

- uint32_t clockSource
  Clock source for QSPI module.
- uint32_t baudRate
  Serial flash clock baud rate.
- uint8_t txWatermark
  QSPI transmit watermark value.
- uint8_t rxWatermark
  QSPI receive watermark value.
- uint32_t AHBbufferSize [FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]
  AHB buffer size.
- uint8_t AHBbufferMaster [FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]
  AHB buffer master.
- bool enableAHBbuffer3AllMaster
  Is AHB buffer3 for all master.
- qspi_read_area_t area
  Which area Rx data readout.
- bool enableQspi
  Enable QSPI after initialization.

29.2.3.0.0.25 Field Documentation

29.2.3.0.0.25.1 uint8_t qspi_config_t::rxWatermark

29.2.3.0.0.25.2 uint32_t qspi_config_t::AHBbufferSize[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]

29.2.3.0.0.25.3 uint8_t qspi_config_t::AHBbufferMaster[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]

29.2.3.0.0.25.4 bool qspi_config_t::enableAHBbuffer3AllMaster

29.2.4 struct qspi_flash_config_t

Data Fields

- uint32_t flashA1Size
  Flash A1 size.
- uint32_t flashA2Size
  Flash A2 size.
- uint32_t flashB1Size
  Flash B1 size.
- uint32_t flashB2Size
  Flash B2 size.
- uint32_t lookuptable [FSL_FEATURE_QSPI_LUT_DEPTH]
  Flash command in LUT.
- uint32_t dataHoldTime
  Data line hold time.
- uint32_t CSHoldTime
  CS line hold time.
- uint32_t CSSetupTime
  CS line setup time.
- uint32_t cloumnspace
  Column space size.
- uint32_t dataLearnValue
  Data Learn value if enable data learn.
- qspi_endianness_t endian
  Flash data endianess.
- bool enableWordAddress
  If enable word address.

29.2.4.0.0.26 Field Documentation

29.2.4.0.0.26.1 uint32_t qspi_flash_config_t::dataHoldTime

29.2.4.0.0.26.2 qspi_endianness_t qspi_flash_config_t::endian

29.2.4.0.0.26.3 bool qspi_flash_config_t::enableWordAddress

29.2.5 struct qspi_transfer_t

Data Fields

- uint32_t * data
  Pointer to data to transmit.
- size_t dataSize
  Bytes to be transmit.

29.3 Macro Definition Documentation

29.3.1 #define FSL_QSPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

29.4 Enumeration Type Documentation

29.4.1 enum _status_t

Enumerator

kStatus_QSPI_Idle QSPI is in idle state.
kStatus_QSPI_Busy QSPI is busy.
kStatus_QSPI_Error Error occurred during QSPI transfer.

29.4.2 enum qspi_read_area_t

Enumerator

kQSPI_ReadAHB QSPI read from AHB buffer.
kQSPI_ReadIP QSPI read from IP FIFO.

29.4.3 enum qspi_command_seq_t

Enumerator

kQSPI_IPSeq IP command sequence.
kQSPI_BufferSeq Buffer command sequence.

29.4.4 enum qspi_fifo_t

Enumerator

kQSPI_TxFifo QSPI Tx FIFO.
kQSPI_RxFifo QSPI Rx FIFO.
kQSPI_AllFifo QSPI all FIFO, including Tx and Rx.

29.4.5 enum qspi_endianness_t

Enumerator

kQSPI_64BigEndian 64 bits big endian
kQSPI_32LittleEndian 32 bit little endian
kQSPI_32BigEndian 32 bit big endian
kQSPI_64LittleEndian 64 bit little endian

29.4.6 enum _qspi_error_flags

Enumerator

kQSPI_DataLearningFail Data learning pattern failure flag.
kQSPI_TxBufferFill Tx buffer fill flag.
kQSPI_TxBufferUnderrun Tx buffer underrun flag.
kQSPI_IllegalInstruction Illegal instruction error flag.
kQSPI_RxBufferOverflow Rx buffer overflow flag.
kQSPI_RxBufferDrain Rx buffer drain flag.
kQSPI_AHBSequenceError AHB sequence error flag.
kQSPI_AHBIllegalTransaction AHB illegal transaction error flag.
kQSPI_AHBIllegalBurstSize AHB illegal burst error flag.
kQSPI_AHBBufferOverflow AHB buffer overflow flag.
kQSPI_IPCommandUsageError IP command usage error flag.
kQSPI_IPCommandTriggerDuringAHBAccess IP command trigger during AHB access error.
kQSPI_IPCommandTriggerDuringIPAccess IP command trigger cannot be executed.
kQSPI_IPCommandTriggerDuringAHBGrant IP command trigger during AHB grant error.
kQSPI_IPCommandTransactionFinished IP command transaction finished flag.
kQSPI_FlagAll All error flag.

29.4.7 enum _qspi_flags

Enumerator

kQSPI_DataLearningSamplePoint Data learning sample point.

kQSPI_TxBufferFull Tx buffer full flag.
kQSPI_TxDMA Tx DMA is requested or running.
kQSPI_TxWatermark Tx buffer watermark available.
kQSPI_TxBufferEnoughData Tx buffer enough data available.
kQSPI_RxDMA Rx DMA is requesting or running.
kQSPI_RxBufferFull Rx buffer full.
kQSPI_RxWatermark Rx buffer watermark exceeded.
kQSPI_AHB3BufferFull AHB buffer 3 full.
kQSPI_AHB2BufferFull AHB buffer 2 full.
kQSPI_AHB1BufferFull AHB buffer 1 full.
kQSPI_AHB0BufferFull AHB buffer 0 full.
kQSPI_AHB3BufferNotEmpty AHB buffer 3 not empty.
kQSPI_AHB2BufferNotEmpty AHB buffer 2 not empty.
kQSPI_AHB1BufferNotEmpty AHB buffer 1 not empty.
kQSPI_AHB0BufferNotEmpty AHB buffer 0 not empty.
kQSPI_AHBTransactionPending AHB access transaction pending.
kQSPI_AHBCommandPriorityGranted AHB command priority granted.
kQSPI_AHBAccess AHB access.
kQSPI_IPAccess IP access.
kQSPI_Busy Module busy.
kQSPI_StateAll All flags.

29.4.8 enum _qspi_interrupt_enable

Enumerator

kQSPI_DataLearningFailInterruptEnable Data learning pattern failure interrupt enable.
kQSPI_TxBufferFillInterruptEnable Tx buffer fill interrupt enable.
kQSPI_TxBufferUnderrunInterruptEnable Tx buffer underrun interrupt enable.
kQSPI_IllegalInstructionInterruptEnable Illegal instruction error interrupt enable.
kQSPI_RxBufferOverflowInterruptEnable Rx buffer overflow interrupt enable.
kQSPI_RxBufferDrainInterruptEnable Rx buffer drain interrupt enable.
kQSPI_AHBSequenceErrorInterruptEnable AHB sequence error interrupt enable.
kQSPI_AHBIllegalTransactionInterruptEnable AHB illegal transaction error interrupt enable.
kQSPI_AHBIllegalBurstSizeInterruptEnable AHB illegal burst error interrupt enable.
kQSPI_AHBBufferOverflowInterruptEnable AHB buffer overflow interrupt enable.
kQSPI_IPCommandUsageErrorInterruptEnable IP command usage error interrupt enable.
kQSPI_IPCommandTriggerDuringAHBAccessInterruptEnable IP command trigger during AHB access error.
kQSPI_IPCommandTriggerDuringIPAccessInterruptEnable IP command trigger cannot be executed.
kQSPI_IPCommandTriggerDuringAHBGrantInterruptEnable IP command trigger during AHB grant error.

kQSPI_IPCommandTransactionFinishedInterruptEnable IP command transaction finished interrupt enable.
kQSPI_AllInterruptEnable All error interrupt enable.

29.4.9 enum _qspi_dma_enable

Enumerator

kQSPI_TxBufferFillDMAEnable Tx buffer fill DMA.
kQSPI_RxBufferDrainDMAEnable Rx buffer drain DMA.
kQSPI_AllDDMAEnable All DMA source.

29.4.10 enum qspi_dqs_phrase_shift_t

Enumerator

kQSPI_DQSNoPhraseShift No phase shift.
kQSPI_DQSPhraseShift45Degree Select 45 degree phase shift.
kQSPI_DQSPhraseShift90Degree Select 90 degree phase shift.
kQSPI_DQSPhraseShift135Degree Select 135 degree phase shift.

29.5 Function Documentation

29.5.1 void QSPI_Init (QuadSPI_Type * *base*, qspi_config_t * *config*, uint32_t *srcClock_Hz*)

This function enables the clock for QSPI and also configures the QSPI with the input configure parameters. Users should call this function before any QSPI operations.

Parameters

base	Pointer to QuadSPI Type.
config	QSPI configure structure.
srcClock_Hz	QSPI source clock frequency in Hz.

29.5.2 void QSPI_GetDefaultQspiConfig (qspi_config_t * *config*)

Parameters

config	QSPI configuration structure.

29.5.3 void QSPI_Deinit (QuadSPI_Type * *base*)

Clears the QSPI state and QSPI module registers.

Parameters

base	Pointer to QuadSPI Type.

29.5.4 void QSPI_SetFlashConfig (QuadSPI_Type * *base*, qspi_flash_config_t * *config*)

This function configures the serial flash relevant parameters, such as the size, command, and so on. The flash configuration value cannot have a default value. The user needs to configure it according to the QSPI features.

Parameters

base	Pointer to QuadSPI Type.
config	Flash configuration parameters.

29.5.5 void QSPI_SoftwareReset (QuadSPI_Type * *base*)

This function sets the software reset flags for both AHB and buffer domain and resets both AHB buffer and also IP FIFOs.

Parameters

base	Pointer to QuadSPI Type.

29.5.6 static void QSPI_Enable (QuadSPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
enable	True means enable QSPI, false means disable.

29.5.7 static uint32_t QSPI_GetStatusFlags (QuadSPI_Type ∗ *base*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.

Returns

status flag, use status flag to AND _qspi_flags could get the related status.

29.5.8 static uint32_t QSPI_GetErrorStatusFlags (QuadSPI_Type ∗ *base*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.

Returns

status flag, use status flag to AND _qspi_error_flags could get the related status.

29.5.9 static void QSPI_ClearErrorFlag (QuadSPI_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	Which kind of QSPI flags to be cleared, a combination of _qspi_error_flags.

29.5.10 static void QSPI_EnableInterrupts (QuadSPI_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	QSPI interrupt source.

29.5.11 static void QSPI_DisableInterrupts (QuadSPI_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	QSPI interrupt source.

29.5.12 static void QSPI_EnableDMA (QuadSPI_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	QSPI DMA source.
enable	True means enable DMA, false means disable.

29.5.13 static uint32_t QSPI_GetTxDataRegisterAddress (QuadSPI_Type * *base*) [inline], [static]

It is used for DMA operation.

Parameters

base	Pointer to QuadSPI Type.

Returns

QSPI Tx data register address.

29.5.14 uint32_t QSPI_GetRxDataRegisterAddress (QuadSPI_Type * *base*)

This function returns the Rx data register address or Rx buffer address according to the Rx read area settings.

Parameters

base	Pointer to QuadSPI Type.

Returns

QSPI Rx data register address.

29.5.15 static void QSPI_SetIPCommandAddress (QuadSPI_Type * *base,* uint32_t *addr*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
addr	IP command address.

29.5.16 static void QSPI_SetIPCommandSize (QuadSPI_Type * *base,* uint32_t *size*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
size	IP command size.

29.5.17 void QSPI_ExecuteIPCommand (QuadSPI_Type * *base,* uint32_t *index*)

Parameters

base	Pointer to QuadSPI Type.
index	IP command located in which LUT table index.

29.5.18 void QSPI_ExecuteAHBCommand (QuadSPI_Type * *base,* uint32_t *index*)

Parameters

base	Pointer to QuadSPI Type.
index	AHB command located in which LUT table index.

29.5.19 static void QSPI_EnableIPParallelMode (QuadSPI_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
enable	True means enable parallel mode, false means disable parallel mode.

29.5.20 static void QSPI_EnableAHBParallelMode (QuadSPI_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
enable	True means enable parallel mode, false means disable parallel mode.

29.5.21 void QSPI_UpdateLUT (QuadSPI_Type ∗ *base*, uint32_t *index*, uint32_t ∗ *cmd*)

Parameters

base	Pointer to QuadSPI Type.
index	Which LUT index needs to be located. It should be an integer divided by 4.
cmd	Command sequence array.

29.5.22 static void QSPI_ClearFifo (QuadSPI_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	Which kind of QSPI FIFO to be cleared.

29.5.23 static void QSPI_ClearCommandSequence (QuadSPI_Type * *base*, qspi_command_seq_t *seq*) [inline], [static]

This function can reset the command sequence.

Parameters

base	QSPI base address.
seq	Which command sequence need to reset, IP command, buffer command or both.

29.5.24 void QSPI_SetReadDataArea (QuadSPI_Type * *base*, qspi_read_area_t *area*)

This function can set the RX buffer readout, from AHB bus or IP Bus.

Parameters

base	QSPI base address.
area	QSPI Rx buffer readout area. AHB bus buffer or IP bus buffer.

29.5.25 void QSPI_WriteBlocking (QuadSPI_Type * *base*, uint32_t * *buffer*, size_t *size*)

Note

This function blocks via polling until all bytes have been sent.

Parameters

base	QSPI base pointer
buffer	The data bytes to send
size	The number of data bytes to send

29.5.26 static void QSPI_WriteData (QuadSPI_Type * *base*, uint32_t *data*) [inline], [static]

Parameters

base	QSPI base pointer
data	The data bytes to send

29.5.27 void QSPI_ReadBlocking (QuadSPI_Type * *base*, uint32_t * *buffer*, size_t *size*)

Note

This function blocks via polling until all bytes have been sent.

Parameters

base	QSPI base pointer
buffer	The data bytes to send
size	The number of data bytes to receive

29.5.28 uint32_t QSPI_ReadData (QuadSPI_Type * *base*)

Parameters

base	QSPI base pointer

Returns

The data in the FIFO.

29.5.29 static void QSPI_TransferSendBlocking (QuadSPI_Type * *base*, qspi_transfer_t * *xfer*) `[inline]`, `[static]`

This function writes a continuous data to the QSPI transmit FIFO. This function is a block function and can return only when finished. This function uses polling methods.

Parameters

base	Pointer to QuadSPI Type.
xfer	QSPI transfer structure.

29.5.30 static void QSPI_TransferReceiveBlocking (QuadSPI_Type * *base*, qspi_transfer_t * *xfer*) [inline], [static]

This function reads continuous data from the QSPI receive buffer/FIFO. This function is a blocking function and can return only when finished. This function uses polling methods.

Parameters

base	Pointer to QuadSPI Type.
xfer	QSPI transfer structure.

29.6 QSPI eDMA Driver

29.6.1 Overview

Data Structures

- struct qspi_edma_handle_t
  QSPI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ qspi_edma_callback_t)(QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, status_t status, void ∗userData)
  QSPI eDMA transfer callback function for finish and error.

eDMA Transactional

- void QSPI_TransferTxCreateHandleEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, qspi_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the QSPI handle for send which is used in transactional functions and set the callback.
- void QSPI_TransferRxCreateHandleEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, qspi_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the QSPI handle for receive which is used in transactional functions and set the callback.
- status_t QSPI_TransferSendEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, qspi_transfer_t ∗xfer)
  Transfers QSPI data using an eDMA non-blocking method.
- status_t QSPI_TransferReceiveEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, qspi_transfer_t ∗xfer)
  Receives data using an eDMA non-blocking method.
- void QSPI_TransferAbortSendEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle)
  Aborts the sent data using eDMA.
- void QSPI_TransferAbortReceiveEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle)
  Aborts the receive data using eDMA.
- status_t QSPI_TransferGetSendCountEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, size_t ∗count)
  Gets the transferred counts of send.
- status_t QSPI_TransferGetReceiveCountEDMA (QuadSPI_Type ∗base, qspi_edma_handle_t ∗handle, size_t ∗count)
  Gets the status of the receive transfer.

29.6.2 Data Structure Documentation

29.6.2.1 struct _qspi_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  eDMA handler for QSPI send
- size_t transferSize
  Bytes need to transfer.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- uint8_t count
  The transfer data count in a DMA request.
- uint32_t state
  Internal state for QSPI eDMA transfer.
- qspi_edma_callback_t callback
  Callback for users while transfer finish or error occurred.
- void * userData
  User callback parameter.

29.6.2.1.0.27 Field Documentation

29.6.2.1.0.27.1 size_t qspi_edma_handle_t::transferSize

29.6.2.1.0.27.2 uint8_t qspi_edma_handle_t::nbytes

29.6.3 Function Documentation

29.6.3.1 void QSPI_TransferTxCreateHandleEDMA (QuadSPI_Type * *base*, qspi_edma_handle_t * *handle*, qspi_edma_callback_t *callback*, void * *userData*, edma_handle_t * *dmaHandle*)

Parameters

base	QSPI peripheral base address
handle	Pointer to qspi_edma_handle_t structure
callback	QSPI callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested eDMA handle for eDMA transfer

29.6.3.2 void QSPI_TransferRxCreateHandleEDMA (QuadSPI_Type * *base*, qspi_edma_handle_t * *handle*, qspi_edma_callback_t *callback*, void * *userData*, edma_handle_t * *dmaHandle*)

Parameters

base	QSPI peripheral base address
handle	Pointer to qspi_edma_handle_t structure
callback	QSPI callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested eDMA handle for eDMA transfer

29.6.3.3 status_t QSPI_TransferSendEDMA (QuadSPI_Type * *base*, qspi_edma_handle_t * *handle*, qspi_transfer_t * *xfer*)

This function writes data to the QSPI transmit FIFO. This function is non-blocking.

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to qspi_edma_handle_t structure
xfer	QSPI transfer structure.

29.6.3.4 status_t QSPI_TransferReceiveEDMA (QuadSPI_Type * *base*, qspi_edma_handle_t * *handle*, qspi_transfer_t * *xfer*)

This function receive data from the QSPI receive buffer/FIFO. This function is non-blocking.

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to qspi_edma_handle_t structure
xfer	QSPI transfer structure.

29.6.3.5 void QSPI_TransferAbortSendEDMA (QuadSPI_Type * *base*, qspi_edma_handle_t * *handle*)

This function aborts the sent data using eDMA.

Parameters

base	QSPI peripheral base address.
handle	Pointer to qspi_edma_handle_t structure

29.6.3.6 void QSPI_TransferAbortReceiveEDMA (QuadSPI_Type ∗ *base*, qspi_edma_handle_t ∗ *handle*)

This function abort receive data which using eDMA.

Parameters

base	QSPI peripheral base address.
handle	Pointer to qspi_edma_handle_t structure

29.6.3.7 status_t QSPI_TransferGetSendCountEDMA (QuadSPI_Type ∗ *base*, qspi_edma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to qspi_edma_handle_t structure.
count	Bytes sent.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

29.6.3.8 status_t QSPI_TransferGetReceiveCountEDMA (QuadSPI_Type ∗ *base*, qspi_edma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to qspi_edma_handle_t structure
count	Bytes received.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

Chapter 30
RCM: Reset Control Module Driver

30.1 Overview

The KSDK provides a Peripheral driver for the Reset Control Module (RCM) module of Kinetis devices.

Data Structures

- struct rcm_reset_pin_filter_config_t
  Reset pin filter configuration. More...

Enumerations

- enum rcm_reset_source_t {
  kRCM_SourceWakeup = RCM_SRS0_WAKEUP_MASK,
  kRCM_SourceLvd = RCM_SRS0_LVD_MASK,
  kRCM_SourceLoc = RCM_SRS0_LOC_MASK,
  kRCM_SourceLol = RCM_SRS0_LOL_MASK,
  kRCM_SourceWdog = RCM_SRS0_WDOG_MASK,
  kRCM_SourcePin = RCM_SRS0_PIN_MASK,
  kRCM_SourcePor = RCM_SRS0_POR_MASK,
  kRCM_SourceJtag = RCM_SRS1_JTAG_MASK << 8U,
  kRCM_SourceLockup = RCM_SRS1_LOCKUP_MASK << 8U,
  kRCM_SourceSw = RCM_SRS1_SW_MASK << 8U,
  kRCM_SourceMdmap = RCM_SRS1_MDM_AP_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.
- enum rcm_boot_rom_config_t {
  kRCM_BootFlash = 0U,
  kRCM_BootRomCfg0 = 1U,
  kRCM_BootRomFopt = 2U,
  kRCM_BootRomBoth = 3U }
  Boot from ROM configuration.

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.
- static uint32_t RCM_GetStickyResetSources (RCM_Type *base)
  Gets the sticky reset source status.
- static void RCM_ClearStickyResetSources (RCM_Type *base, uint32_t sourceMasks)
  Clears the sticky reset source status.
- void RCM_ConfigureResetPinFilter (RCM_Type *base, const rcm_reset_pin_filter_config_t *config)
  Configures the reset pin filter.
- static rcm_boot_rom_config_t RCM_GetBootRomSource (RCM_Type *base)
  Gets the ROM boot source.
- static void RCM_ClearBootRomSource (RCM_Type *base)
  Clears the ROM boot source flag.
- void RCM_SetForceBootRomSource (RCM_Type *base, rcm_boot_rom_config_t config)
  Forces the boot from ROM.

30.2 Data Structure Documentation

30.2.1 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.

30.2.1.0.0.28 Field Documentation

30.2.1.0.0.28.1 bool rcm_reset_pin_filter_config_t::enableFilterInStop

30.2.1.0.0.28.2 rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait

30.2.1.0.0.28.3 uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount

30.3 Macro Definition Documentation

30.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

30.4 Enumeration Type Documentation

30.4.1 enum rcm_reset_source_t

Enumerator

kRCM_SourceWakeup Low-leakage wakeup reset.

kRCM_SourceLvd Low-voltage detect reset.
kRCM_SourceLoc Loss of clock reset.
kRCM_SourceLol Loss of lock reset.
kRCM_SourceWdog Watchdog reset.
kRCM_SourcePin External pin reset.
kRCM_SourcePor Power on reset.
kRCM_SourceJtag JTAG generated reset.
kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceMdmap MDM-AP system reset.
kRCM_SourceSackerr Parameter could get all reset flags.

30.4.2 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.

30.4.3 enum rcm_boot_rom_config_t

Enumerator

kRCM_BootFlash Boot from flash.
kRCM_BootRomCfg0 Boot from boot ROM due to BOOTCFG0.
kRCM_BootRomFopt Boot from boot ROM due to FOPT[7].
kRCM_BootRomBoth Boot from boot ROM due to both BOOTCFG0 and FOPT[7].

30.5 Function Documentation

30.5.1 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.

This is an example.

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
```

```
resetStatus = RCM_GetPreviousResetSources(RCM) &
      kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (
      kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

base	RCM peripheral base address.

Returns

All reset source status bit map.

30.5.2 static uint32_t RCM_GetStickyResetSources (RCM_Type ∗ *base*) [inline], [static]

This function gets the current reset source status that has not been cleared by software for a specific source.

This is an example.

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetStickyResetSources(RCM) &
      kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetStickyResetSources(RCM) & (
      kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

base	RCM peripheral base address.

Returns

All reset source status bit map.

30.5.3 static void RCM_ClearStickyResetSources (RCM_Type ∗ *base,* uint32_t *sourceMasks*) [inline], [static]

This function clears the sticky system reset flags indicated by source masks.

This is an example.

```
// Clears multiple reset sources.
RCM_ClearStickyResetSources(kRCM_SourceWdog |
      kRCM_SourcePin);
```

Parameters

base	RCM peripheral base address.
sourceMasks	reset source status bit map

30.5.4 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.

30.5.5 static rcm_boot_rom_config_t RCM_GetBootRomSource (RCM_Type * *base*) [inline], [static]

This function gets the ROM boot source during the last chip reset.

Parameters

base	RCM peripheral base address.

Returns

The ROM boot source.

30.5.6 static void RCM_ClearBootRomSource (RCM_Type * *base*) [inline], [static]

This function clears the ROM boot source flag.

Parameters

base	Register base address of RCM

30.5.7 void RCM_SetForceBootRomSource (RCM_Type * *base*, rcm_boot_rom_config_t *config*)

This function forces booting from ROM during all subsequent system resets.

Parameters

base	RCM peripheral base address.
config	Boot configuration.

Chapter 31
RTC: Real Time Clock

31.1 Overview

The KSDK provides a driver for the Real Time Clock (RTC) of Kinetis devices.

31.2 Function groups

The RTC driver supports operating the module as a time counter.

31.2.1 Initialization and deinitialization

The function RTC_Init() initializes the RTC with specified configurations. The function RTC_GetDefault-Config() gets the default configurations.

The function RTC_Deinit() disables the RTC timer and disables the module clock.

31.2.2 Set & Get Datetime

The function RTC_SetDatetime() sets the timer period in seconds. Users pass in the details in date & time format by using the below data structure.

```
typedef struct _rtc_datetime
{
    uint16_t year;
    uint8_t month;
    uint8_t day;
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
} rtc_datetime_t;
```

The function RTC_GetDatetime() reads the current timer value in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

31.2.3 Set & Get Alarm

The function RTC_SetAlarm() sets the alarm time period in seconds. Users pass in the details in date & time format by using the datetime data structure.

The function RTC_GetAlarm() reads the alarm time in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

31.2.4 Start & Stop timer

The function RTC_StartTimer() starts the RTC time counter.

The function RTC_StopTimer() stops the RTC time counter.

31.2.5 Status

Provides functions to get and clear the RTC status.

31.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

31.2.7 RTC Oscillator

Some SoC's allow control of the RTC oscillator through the RTC module.

The function RTC_SetOscCapLoad() allows the user to modify the capacitor load configuration of the RTC oscillator.

31.2.8 Monotonic Counter

Some SoC's have a 64-bit Monotonic counter available in the RTC module.

The function RTC_SetMonotonicCounter() writes a 64-bit to the counter.

The function RTC_GetMonotonicCounter() reads the monotonic counter and returns the 64-bit counter value to the user.

The function RTC_IncrementMonotonicCounter() increments the Monotonic Counter by one.

31.3 Typical use case

31.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
    uint32_t sec;
    uint32_t currSeconds;
    rtc_datetime_t date;
    rtc_config_t rtcConfig;

    /* Board pin, clock, debug console init */
```

```
BOARD_InitHardware();
/* Init RTC */
RTC_GetDefaultConfig(&rtcConfig);
RTC_Init(RTC, &rtcConfig);
/* Select RTC clock source */
BOARD_SetRtcClockSource();

PRINTF("RTC example: set up time to wake up an alarm\r\n");

/* Set a start date time and start RT */
date.year = 2014U;
date.month = 12U;
date.day = 25U;
date.hour = 19U;
date.minute = 0;
date.second = 0;

/* RTC time counter has to be stopped before setting the date & time in the TSR register */
RTC_StopTimer(RTC);

/* Set RTC time to default */
RTC_SetDatetime(RTC, &date);

/* Enable RTC alarm interrupt */
RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

/* Enable at the NVIC */
EnableIRQ(RTC_IRQn);

/* Start the RTC time counter */
RTC_StartTimer(RTC);

/* This loop will set the RTC alarm */
while (1)
{
    busyWait = true;
    /* Get date time */
    RTC_GetDatetime(RTC, &date);

    /* print default time */
    PRINTF("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
  year, date.month, date.day, date.hour,
           date.minute, date.second);

    /* Get alarm time from the user */
    sec = 0;
    PRINTF("Input the number of second to wait for alarm \r\n");
    PRINTF("The second must be positive value\r\n");
    while (sec < 1)
    {
        SCANF("%d", &sec);
    }

    /* Read the RTC seconds register to get current time in seconds */
    currSeconds = RTC->TSR;

    /* Add alarm seconds to current time */
    currSeconds += sec;

    /* Set alarm time in seconds */
    RTC->TAR = currSeconds;

    /* Get alarm time */
    RTC_GetAlarm(RTC, &date);

    /* Print alarm time */
    PRINTF("Alarm will occur at: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
  year, date.month, date.day,
```

```
                date.hour, date.minute, date.second);

        /* Wait until alarm occurs */
        while (busyWait)
        {
        }

        PRINTF("\r\n Alarm occurs !!!! ");
    }
}
```

Data Structures

- struct rtc_datetime_t
  Structure is used to hold the date and time. More...
- struct rtc_config_t
  RTC config structure. More...

Enumerations

- enum rtc_interrupt_enable_t {
  kRTC_TimeInvalidInterruptEnable = RTC_IER_TIIE_MASK,
  kRTC_TimeOverflowInterruptEnable = RTC_IER_TOIE_MASK,
  kRTC_AlarmInterruptEnable = RTC_IER_TAIE_MASK,
  kRTC_SecondsInterruptEnable = RTC_IER_TSIE_MASK }
  List of RTC interrupts.
- enum rtc_status_flags_t {
  kRTC_TimeInvalidFlag = RTC_SR_TIF_MASK,
  kRTC_TimeOverflowFlag = RTC_SR_TOF_MASK,
  kRTC_AlarmFlag = RTC_SR_TAF_MASK }
  List of RTC flags.
- enum rtc_osc_cap_load_t {
  kRTC_Capacitor_2p = RTC_CR_SC2P_MASK,
  kRTC_Capacitor_4p = RTC_CR_SC4P_MASK,
  kRTC_Capacitor_8p = RTC_CR_SC8P_MASK,
  kRTC_Capacitor_16p = RTC_CR_SC16P_MASK }
  List of RTC Oscillator capacitor load settings.

Functions

- static void RTC_SetOscCapLoad (RTC_Type *base, uint32_t capLoad)
  This function sets the specified capacitor configuration for the RTC oscillator.
- static void RTC_Reset (RTC_Type *base)
  Performs a software reset on the RTC module.

Driver version

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

Initialization and deinitialization

- void RTC_Init (RTC_Type ∗base, const rtc_config_t ∗config)
  Ungates the RTC clock and configures the peripheral for basic operation.
- static void RTC_Deinit (RTC_Type ∗base)
  Stops the timer and gate the RTC clock.
- void RTC_GetDefaultConfig (rtc_config_t ∗config)
  Fills in the RTC config struct with the default settings.

Current Time & Alarm

- status_t RTC_SetDatetime (RTC_Type ∗base, const rtc_datetime_t ∗datetime)
  Sets the RTC date and time according to the given time structure.
- void RTC_GetDatetime (RTC_Type ∗base, rtc_datetime_t ∗datetime)
  Gets the RTC time and stores it in the given time structure.
- status_t RTC_SetAlarm (RTC_Type ∗base, const rtc_datetime_t ∗alarmTime)
  Sets the RTC alarm time.
- void RTC_GetAlarm (RTC_Type ∗base, rtc_datetime_t ∗datetime)
  Returns the RTC alarm time.

Interrupt Interface

- static void RTC_EnableInterrupts (RTC_Type ∗base, uint32_t mask)
  Enables the selected RTC interrupts.
- static void RTC_DisableInterrupts (RTC_Type ∗base, uint32_t mask)
  Disables the selected RTC interrupts.
- static uint32_t RTC_GetEnabledInterrupts (RTC_Type ∗base)
  Gets the enabled RTC interrupts.

Status Interface

- static uint32_t RTC_GetStatusFlags (RTC_Type ∗base)
  Gets the RTC status flags.
- void RTC_ClearStatusFlags (RTC_Type ∗base, uint32_t mask)
  Clears the RTC status flags.

Timer Start and Stop

- static void RTC_StartTimer (RTC_Type ∗base)
  Starts the RTC time counter.
- static void RTC_StopTimer (RTC_Type ∗base)
  Stops the RTC time counter.

31.4 Data Structure Documentation

31.4.1 struct rtc_datetime_t

Data Fields

- uint16_t year

Range from 1970 to 2099.
- uint8_t month
  Range from 1 to 12.
- uint8_t day
  Range from 1 to 31 (depending on month).
- uint8_t hour
  Range from 0 to 23.
- uint8_t minute
  Range from 0 to 59.
- uint8_t second
  Range from 0 to 59.

31.4.1.0.0.29 Field Documentation

31.4.1.0.0.29.1 uint16_t rtc_datetime_t::year

31.4.1.0.0.29.2 uint8_t rtc_datetime_t::month

31.4.1.0.0.29.3 uint8_t rtc_datetime_t::day

31.4.1.0.0.29.4 uint8_t rtc_datetime_t::hour

31.4.1.0.0.29.5 uint8_t rtc_datetime_t::minute

31.4.1.0.0.29.6 uint8_t rtc_datetime_t::second

31.4.2 struct rtc_config_t

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

The config struct can be made const so it resides in flash

Data Fields

- bool wakeupSelect
  true: Wakeup pin outputs the 32 KHz clock; false:Wakeup pin used to wakeup the chip
- bool updateMode
  true: Registers can be written even when locked under certain conditions, false: No writes allowed when registers are locked
- bool supervisorAccess
  true: Non-supervisor accesses are allowed; false: Non-supervisor accesses are not supported
- uint32_t compensationInterval
  Compensation interval that is written to the CIR field in RTC TCR Register.
- uint32_t compensationTime
  Compensation time that is written to the TCR field in RTC TCR Register.

31.5 Enumeration Type Documentation

31.5.1 enum rtc_interrupt_enable_t

Enumerator

kRTC_TimeInvalidInterruptEnable Time invalid interrupt.
kRTC_TimeOverflowInterruptEnable Time overflow interrupt.
kRTC_AlarmInterruptEnable Alarm interrupt.
kRTC_SecondsInterruptEnable Seconds interrupt.

31.5.2 enum rtc_status_flags_t

Enumerator

kRTC_TimeInvalidFlag Time invalid flag.
kRTC_TimeOverflowFlag Time overflow flag.
kRTC_AlarmFlag Alarm flag.

31.5.3 enum rtc_osc_cap_load_t

Enumerator

kRTC_Capacitor_2p 2 pF capacitor load
kRTC_Capacitor_4p 4 pF capacitor load
kRTC_Capacitor_8p 8 pF capacitor load
kRTC_Capacitor_16p 16 pF capacitor load

31.6 Function Documentation

31.6.1 void RTC_Init (RTC_Type ∗ *base,* const rtc_config_t ∗ *config*)

This function issues a software reset if the timer invalid flag is set.

Note

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

Parameters

base	RTC peripheral base address
config	Pointer to the user's RTC configuration structure.

31.6.2 static void RTC_Deinit (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

31.6.3 void RTC_GetDefaultConfig (rtc_config_t * *config*)

The default values are as follows.

```
*    config->wakeupSelect = false;
*    config->updateMode = false;
*    config->supervisorAccess = false;
*    config->compensationInterval = 0;
*    config->compensationTime = 0;
*
```

Parameters

config	Pointer to the user's RTC configuration structure.

31.6.4 status_t RTC_SetDatetime (RTC_Type * *base*, const rtc_datetime_t * *datetime*)

The RTC counter must be stopped prior to calling this function because writes to the RTC seconds register fail if the RTC counter is running.

Parameters

base	RTC peripheral base address
datetime	Pointer to the structure where the date and time details are stored.

Returns

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

31.6.5 void RTC_GetDatetime (RTC_Type * *base*, rtc_datetime_t * *datetime*)

Parameters

base	RTC peripheral base address
datetime	Pointer to the structure where the date and time details are stored.

31.6.6 status_t RTC_SetAlarm (RTC_Type ∗ *base*, const rtc_datetime_t ∗ *alarmTime*)

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

base	RTC peripheral base address
alarmTime	Pointer to the structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the RTC alarm kStatus_InvalidArgument: Error because the alarm datetime format is incorrect kStatus_Fail: Error because the alarm time has already passed

31.6.7 void RTC_GetAlarm (RTC_Type ∗ *base*, rtc_datetime_t ∗ *datetime*)

Parameters

base	RTC peripheral base address
datetime	Pointer to the structure where the alarm date and time details are stored.

31.6.8 static void RTC_EnableInterrupts (RTC_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	RTC peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

31.6.9 static void RTC_DisableInterrupts (RTC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	RTC peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration rtc_-interrupt_enable_t

31.6.10 static uint32_t RTC_GetEnabledInterrupts (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

Returns

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

31.6.11 static uint32_t RTC_GetStatusFlags (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

Returns

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

31.6.12 void RTC_ClearStatusFlags (RTC_Type * *base,* uint32_t *mask*)

Parameters

base	RTC peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration rtc_-status_flags_t

31.6.13 static void RTC_StartTimer (RTC_Type * *base*) [inline], [static]

After calling this function, the timer counter increments once a second provided SR[TOF] or SR[TIF] are not set.

Parameters

base	RTC peripheral base address

31.6.14 static void RTC_StopTimer (RTC_Type * base) [inline], [static]

RTC's seconds register can be written to only when the timer is stopped.

Parameters

base	RTC peripheral base address

31.6.15 static void RTC_SetOscCapLoad (RTC_Type * base, uint32_t capLoad) [inline], [static]

Parameters

base	RTC peripheral base address
capLoad	Oscillator loads to enable. This is a logical OR of members of the enumeration rtc_-osc_cap_load_t

31.6.16 static void RTC_Reset (RTC_Type * base) [inline], [static]

This resets all RTC registers except for the SWR bit and the RTC_WAR and RTC_RAR registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

base	RTC peripheral base address

Chapter 32
SAI: Serial Audio Interface

32.1 Overview

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

SAI driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for SAI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the SAI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. SAI functional operation groups provide the functional API set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the sai_handle_t as the first parameter. Initialize the handle by calling the SAI_TransferTxCreateHandle() or SAI_TransferRxCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions SAI_TransferSendNonBlocking() and SAI_TransfferReceiveNonBlocking() set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_SAI_TxIdle and kStatus_SAI_RxIdle status.

32.2 Typical use case

32.2.1 SAI Send/receive using an interrupt method

```
sai_handle_t g_saiTxHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
volatile bool rxFinished;
const uint8_t sendData[] = [......];

void SAI_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_SAI_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...

    SAI_TxGetDefaultConfig(&user_config);
```

```
    SAI_TxInit(SAI0, &user_config);
    SAI_TransferTxCreateHandle(SAI0, &g_saiHandle, SAI_UserCallback, NULL);

    //Configure sai format
    SAI_TransferTxSetTransferFormat(SAI0, &g_saiHandle, mclkSource, mclk);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Send out.
    SAI_TransferSendNonBlocking(SAI0, &g_saiHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }

    // ...
}
```

32.2.2 SAI Send/receive using a DMA method

```
sai_handle_t g_saiHandle;
dma_handle_t g_saiTxDmaHandle;
dma_handle_t g_saiRxDmaHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
uint8_t sendData[] = ...;

void SAI_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_SAI_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...

    SAI_TxGetDefaultConfig(&user_config);
    SAI_TxInit(SAI0, &user_config);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SAI_TX_DMA_CHANNEL, SAI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SAI_TX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_CreateHandle(&g_saiTxDmaHandle, DMA0, SAI_TX_DMA_CHANNEL);

    SAI_TransferTxCreateHandleDMA(SAI0, &g_saiTxDmaHandle, SAI_UserCallback,
      NULL);

    // Prepares to send.
    sendXfer.data = sendData
```

```
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Sends out.
    SAI_TransferSendDMA(&g_saiHandle, &sendXfer);

    // Waits for send to complete.
    while (!txFinished)
    {
    }

    // ...
}
```

Modules

- SAI DMA Driver
- SAI eDMA Driver

Data Structures

- struct sai_config_t
  SAI user configuration structure. More...
- struct sai_transfer_format_t
  sai transfer format More...
- struct sai_transfer_t
  SAI transfer structure. More...
- struct sai_handle_t
  SAI handle structure. More...

Macros

- #define SAI_XFER_QUEUE_SIZE (4)
  SAI transfer queue size, user can refine it according to use case.

Typedefs

- typedef void(∗ sai_transfer_callback_t)(I2S_Type ∗base, sai_handle_t ∗handle, status_t status, void ∗userData)
  SAI transfer callback prototype.

Enumerations

- enum _sai_status_t {
  kStatus_SAI_TxBusy = MAKE_STATUS(kStatusGroup_SAI, 0),
  kStatus_SAI_RxBusy = MAKE_STATUS(kStatusGroup_SAI, 1),
  kStatus_SAI_TxError = MAKE_STATUS(kStatusGroup_SAI, 2),
  kStatus_SAI_RxError = MAKE_STATUS(kStatusGroup_SAI, 3),
  kStatus_SAI_QueueFull = MAKE_STATUS(kStatusGroup_SAI, 4),
  kStatus_SAI_TxIdle = MAKE_STATUS(kStatusGroup_SAI, 5),
  kStatus_SAI_RxIdle = MAKE_STATUS(kStatusGroup_SAI, 6) }
  SAI return status.

- enum sai_protocol_t {
  kSAI_BusLeftJustified = 0x0U,
  kSAI_BusRightJustified,
  kSAI_BusI2S,
  kSAI_BusPCMA,
  kSAI_BusPCMB }
  Define the SAI bus type.
- enum sai_master_slave_t {
  kSAI_Master = 0x0U,
  kSAI_Slave = 0x1U }
  Master or slave mode.
- enum sai_mono_stereo_t {
  kSAI_Stereo = 0x0U,
  kSAI_MonoLeft,
  kSAI_MonoRight }
  Mono or stereo audio format.
- enum sai_sync_mode_t {
  kSAI_ModeAsync = 0x0U,
  kSAI_ModeSync,
  kSAI_ModeSyncWithOtherTx,
  kSAI_ModeSyncWithOtherRx }
  Synchronous or asynchronous mode.
- enum sai_mclk_source_t {
  kSAI_MclkSourceSysclk = 0x0U,
  kSAI_MclkSourceSelect1,
  kSAI_MclkSourceSelect2,
  kSAI_MclkSourceSelect3 }
  Mater clock source.
- enum sai_bclk_source_t {
  kSAI_BclkSourceBusclk = 0x0U,
  kSAI_BclkSourceMclkDiv,
  kSAI_BclkSourceOtherSai0,
  kSAI_BclkSourceOtherSai1 }
  Bit clock source.
- enum _sai_interrupt_enable_t {
  kSAI_WordStartInterruptEnable,
  kSAI_SyncErrorInterruptEnable = I2S_TCSR_SEIE_MASK,
  kSAI_FIFOWarningInterruptEnable = I2S_TCSR_FWIE_MASK,
  kSAI_FIFOErrorInterruptEnable = I2S_TCSR_FEIE_MASK,
  kSAI_FIFORequestInterruptEnable = I2S_TCSR_FRIE_MASK }
  The SAI interrupt enable flag.
- enum _sai_dma_enable_t {
  kSAI_FIFOWarningDMAEnable = I2S_TCSR_FWDE_MASK,
  kSAI_FIFORequestDMAEnable = I2S_TCSR_FRDE_MASK }
  The DMA request sources.
- enum _sai_flags {

kSAI_WordStartFlag = I2S_TCSR_WSF_MASK,
kSAI_SyncErrorFlag = I2S_TCSR_SEF_MASK,
kSAI_FIFOErrorFlag = I2S_TCSR_FEF_MASK,
kSAI_FIFORequestFlag = I2S_TCSR_FRF_MASK,
kSAI_FIFOWarningFlag = I2S_TCSR_FWF_MASK }

The SAI status flag.

- enum sai_reset_type_t {
kSAI_ResetTypeSoftware = I2S_TCSR_SR_MASK,
kSAI_ResetTypeFIFO = I2S_TCSR_FR_MASK,
kSAI_ResetAll = I2S_TCSR_SR_MASK | I2S_TCSR_FR_MASK }

  The reset type.
- enum sai_fifo_packing_t {
kSAI_FifoPackingDisabled = 0x0U,
kSAI_FifoPacking8bit = 0x2U,
kSAI_FifoPacking16bit = 0x3U }

  The SAI packing mode The mode includes 8 bit and 16 bit packing.
- enum sai_sample_rate_t {
kSAI_SampleRate8KHz = 8000U,
kSAI_SampleRate11025Hz = 11025U,
kSAI_SampleRate12KHz = 12000U,
kSAI_SampleRate16KHz = 16000U,
kSAI_SampleRate22050Hz = 22050U,
kSAI_SampleRate24KHz = 24000U,
kSAI_SampleRate32KHz = 32000U,
kSAI_SampleRate44100Hz = 44100U,
kSAI_SampleRate48KHz = 48000U,
kSAI_SampleRate96KHz = 96000U }

  Audio sample rate.
- enum sai_word_width_t {
kSAI_WordWidth8bits = 8U,
kSAI_WordWidth16bits = 16U,
kSAI_WordWidth24bits = 24U,
kSAI_WordWidth32bits = 32U }

  Audio word width.

Driver version

- #define FSL_SAI_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

  Version 2.1.1.

Initialization and deinitialization

- void SAI_TxInit (I2S_Type ∗base, const sai_config_t ∗config)

  Initializes the SAI Tx peripheral.
- void SAI_RxInit (I2S_Type ∗base, const sai_config_t ∗config)

  Initializes the the SAI Rx peripheral.
- void SAI_TxGetDefaultConfig (sai_config_t ∗config)

Sets the SAI Tx configuration structure to default values.
- void SAI_RxGetDefaultConfig (sai_config_t ∗config)
    Sets the SAI Rx configuration structure to default values.
- void SAI_Deinit (I2S_Type ∗base)
    De-initializes the SAI peripheral.
- void SAI_TxReset (I2S_Type ∗base)
    Resets the SAI Tx.
- void SAI_RxReset (I2S_Type ∗base)
    Resets the SAI Rx.
- void SAI_TxEnable (I2S_Type ∗base, bool enable)
    Enables/disables the SAI Tx.
- void SAI_RxEnable (I2S_Type ∗base, bool enable)
    Enables/disables the SAI Rx.

Status

- static uint32_t SAI_TxGetStatusFlag (I2S_Type ∗base)
    Gets the SAI Tx status flag state.
- static void SAI_TxClearStatusFlags (I2S_Type ∗base, uint32_t mask)
    Clears the SAI Tx status flag state.
- static uint32_t SAI_RxGetStatusFlag (I2S_Type ∗base)
    Gets the SAI Tx status flag state.
- static void SAI_RxClearStatusFlags (I2S_Type ∗base, uint32_t mask)
    Clears the SAI Rx status flag state.

Interrupts

- static void SAI_TxEnableInterrupts (I2S_Type ∗base, uint32_t mask)
    Enables the SAI Tx interrupt requests.
- static void SAI_RxEnableInterrupts (I2S_Type ∗base, uint32_t mask)
    Enables the SAI Rx interrupt requests.
- static void SAI_TxDisableInterrupts (I2S_Type ∗base, uint32_t mask)
    Disables the SAI Tx interrupt requests.
- static void SAI_RxDisableInterrupts (I2S_Type ∗base, uint32_t mask)
    Disables the SAI Rx interrupt requests.

DMA Control

- static void SAI_TxEnableDMA (I2S_Type ∗base, uint32_t mask, bool enable)
    Enables/disables the SAI Tx DMA requests.
- static void SAI_RxEnableDMA (I2S_Type ∗base, uint32_t mask, bool enable)
    Enables/disables the SAI Rx DMA requests.
- static uint32_t SAI_TxGetDataRegisterAddress (I2S_Type ∗base, uint32_t channel)
    Gets the SAI Tx data register address.
- static uint32_t SAI_RxGetDataRegisterAddress (I2S_Type ∗base, uint32_t channel)
    Gets the SAI Rx data register address.

Bus Operations

- void SAI_TxSetFormat (I2S_Type ∗base, sai_transfer_format_t ∗format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)

Configures the SAI Tx audio format.
- void SAI_RxSetFormat (I2S_Type *base, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Rx audio format.
- void SAI_WriteBlocking (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
  Sends data using a blocking method.
- static void SAI_WriteData (I2S_Type *base, uint32_t channel, uint32_t data)
  Writes data into SAI FIFO.
- void SAI_ReadBlocking (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
  Receives data using a blocking method.
- static uint32_t SAI_ReadData (I2S_Type *base, uint32_t channel)
  Reads data from the SAI FIFO.

Transactional

- void SAI_TransferTxCreateHandle (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
  Initializes the SAI Tx handle.
- void SAI_TransferRxCreateHandle (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
  Initializes the SAI Rx handle.
- status_t SAI_TransferTxSetFormat (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Tx audio format.
- status_t SAI_TransferRxSetFormat (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Rx audio format.
- status_t SAI_TransferSendNonBlocking (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)
  Performs an interrupt non-blocking send transfer on SAI.
- status_t SAI_TransferReceiveNonBlocking (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)
  Performs an interrupt non-blocking receive transfer on SAI.
- status_t SAI_TransferGetSendCount (I2S_Type *base, sai_handle_t *handle, size_t *count)
  Gets a set byte count.
- status_t SAI_TransferGetReceiveCount (I2S_Type *base, sai_handle_t *handle, size_t *count)
  Gets a received byte count.
- void SAI_TransferAbortSend (I2S_Type *base, sai_handle_t *handle)
  Aborts the current send.
- void SAI_TransferAbortReceive (I2S_Type *base, sai_handle_t *handle)
  Aborts the the current IRQ receive.
- void SAI_TransferTxHandleIRQ (I2S_Type *base, sai_handle_t *handle)
  Tx interrupt handler.
- void SAI_TransferRxHandleIRQ (I2S_Type *base, sai_handle_t *handle)
  Tx interrupt handler.

32.3 Data Structure Documentation

32.3.1 struct sai_config_t

Data Fields

- sai_protocol_t protocol
  Audio bus protocol in SAI.
- sai_sync_mode_t syncMode
  SAI sync mode, control Tx/Rx clock sync.
- bool mclkOutputEnable
  Master clock output enable, true means master clock divider enabled.
- sai_mclk_source_t mclkSource
  Master Clock source.
- sai_bclk_source_t bclkSource
  Bit Clock source.
- sai_master_slave_t masterSlave
  Master or slave.

32.3.2 struct sai_transfer_format_t

Data Fields

- uint32_t sampleRate_Hz
  Sample rate of audio data.
- uint32_t bitWidth
  Data length of audio data, usually 8/16/24/32 bits.
- sai_mono_stereo_t stereo
  Mono or stereo.
- uint32_t masterClockHz
  Master clock frequency in Hz.
- uint8_t watermark
  Watermark value.
- uint8_t channel
  Data channel used in transfer.
- sai_protocol_t protocol
  Which audio protocol used.

32.3.2.0.0.30 Field Documentation

32.3.2.0.0.30.1 uint8_t sai_transfer_format_t::channel

32.3.3 struct sai_transfer_t

Data Fields

- uint8_t ∗ data
  Data start address to transfer.

- size_t dataSize
  Transfer size.

32.3.3.0.0.31 Field Documentation

32.3.3.0.0.31.1 uint8_t∗ sai_transfer_t::data

32.3.3.0.0.31.2 size_t sai_transfer_t::dataSize

32.3.4 struct _sai_handle

Data Fields

- uint32_t state
  Transfer status.
- sai_transfer_callback_t callback
  Callback function called at transfer event.
- void ∗ userData
  Callback parameter passed to callback function.
- uint8_t bitWidth
  Bit width for transfer, 8/16/24/32 bits.
- uint8_t channel
  Transfer channel.
- sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [SAI_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.
- uint8_t watermark
  Watermark value.

32.4 Macro Definition Documentation

32.4.1 #define SAI_XFER_QUEUE_SIZE (4)

32.5 Enumeration Type Documentation

32.5.1 enum _sai_status_t

Enumerator

kStatus_SAI_TxBusy SAI Tx is busy.
kStatus_SAI_RxBusy SAI Rx is busy.
kStatus_SAI_TxError SAI Tx FIFO error.
kStatus_SAI_RxError SAI Rx FIFO error.
kStatus_SAI_QueueFull SAI transfer queue is full.

kStatus_SAI_TxIdle SAI Tx is idle.
kStatus_SAI_RxIdle SAI Rx is idle.

32.5.2 enum sai_protocol_t

Enumerator

kSAI_BusLeftJustified Uses left justified format.
kSAI_BusRightJustified Uses right justified format.
kSAI_BusI2S Uses I2S format.
kSAI_BusPCMA Uses I2S PCM A format.
kSAI_BusPCMB Uses I2S PCM B format.

32.5.3 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.
kSAI_Slave Slave mode.

32.5.4 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.
kSAI_MonoLeft Only left channel have sound.
kSAI_MonoRight Only Right channel have sound.

32.5.5 enum sai_sync_mode_t

Enumerator

kSAI_ModeAsync Asynchronous mode.
kSAI_ModeSync Synchronous mode (with receiver or transmit)
kSAI_ModeSyncWithOtherTx Synchronous with another SAI transmit.
kSAI_ModeSyncWithOtherRx Synchronous with another SAI receiver.

32.5.6 enum sai_mclk_source_t

Enumerator

kSAI_MclkSourceSysclk Master clock from the system clock.
kSAI_MclkSourceSelect1 Master clock from source 1.
kSAI_MclkSourceSelect2 Master clock from source 2.
kSAI_MclkSourceSelect3 Master clock from source 3.

32.5.7 enum sai_bclk_source_t

Enumerator

kSAI_BclkSourceBusclk Bit clock using bus clock.
kSAI_BclkSourceMclkDiv Bit clock using master clock divider.
kSAI_BclkSourceOtherSai0 Bit clock from other SAI device.
kSAI_BclkSourceOtherSai1 Bit clock from other SAI device.

32.5.8 enum _sai_interrupt_enable_t

Enumerator

kSAI_WordStartInterruptEnable Word start flag, means the first word in a frame detected.
kSAI_SyncErrorInterruptEnable Sync error flag, means the sync error is detected.
kSAI_FIFOWarningInterruptEnable FIFO warning flag, means the FIFO is empty.
kSAI_FIFOErrorInterruptEnable FIFO error flag.
kSAI_FIFORequestInterruptEnable FIFO request, means reached watermark.

32.5.9 enum _sai_dma_enable_t

Enumerator

kSAI_FIFOWarningDMAEnable FIFO warning caused by the DMA request.
kSAI_FIFORequestDMAEnable FIFO request caused by the DMA request.

32.5.10 enum _sai_flags

Enumerator

kSAI_WordStartFlag Word start flag, means the first word in a frame detected.
kSAI_SyncErrorFlag Sync error flag, means the sync error is detected.

kSAI_FIFOErrorFlag FIFO error flag.
kSAI_FIFORequestFlag FIFO request flag.
kSAI_FIFOWarningFlag FIFO warning flag.

32.5.11 enum sai_reset_type_t

Enumerator

kSAI_ResetTypeSoftware Software reset, reset the logic state.
kSAI_ResetTypeFIFO FIFO reset, reset the FIFO read and write pointer.
kSAI_ResetAll All reset.

32.5.12 enum sai_fifo_packing_t

Enumerator

kSAI_FifoPackingDisabled Packing disabled.
kSAI_FifoPacking8bit 8 bit packing enabled
kSAI_FifoPacking16bit 16bit packing enabled

32.5.13 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000 Hz.
kSAI_SampleRate11025Hz Sample rate 11025 Hz.
kSAI_SampleRate12KHz Sample rate 12000 Hz.
kSAI_SampleRate16KHz Sample rate 16000 Hz.
kSAI_SampleRate22050Hz Sample rate 22050 Hz.
kSAI_SampleRate24KHz Sample rate 24000 Hz.
kSAI_SampleRate32KHz Sample rate 32000 Hz.
kSAI_SampleRate44100Hz Sample rate 44100 Hz.
kSAI_SampleRate48KHz Sample rate 48000 Hz.
kSAI_SampleRate96KHz Sample rate 96000 Hz.

32.5.14 enum sai_word_width_t

Enumerator

kSAI_WordWidth8bits Audio data width 8 bits.

kSAI_WordWidth16bits	Audio data width 16 bits.
kSAI_WordWidth24bits	Audio data width 24 bits.
kSAI_WordWidth32bits	Audio data width 32 bits.

32.6 Function Documentation

32.6.1 void SAI_TxInit (I2S_Type * *base*, const sai_config_t * *config*)

Ungates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by SAI_TxGetDefaultConfig().

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

base	SAI base pointer
config	SAI configuration structure.

32.6.2 void SAI_RxInit (I2S_Type * *base*, const sai_config_t * *config*)

Ungates the SAI clock, resets the module, and configures the SAI Rx with a configuration structure. The configuration structure can be custom filled or set with default values by SAI_RxGetDefaultConfig().

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAI module can cause a hard fault because the clock is not enabled.

Parameters

base	SAI base pointer
config	SAI configuration structure.

32.6.3 void SAI_TxGetDefaultConfig (sai_config_t * *config*)

This API initializes the configuration structure for use in SAI_TxConfig(). The initialized structure can remain unchanged in SAI_TxConfig(), or it can be modified before calling SAI_TxConfig(). This is an example.

```
sai_config_t config;
SAI_TxGetDefaultConfig(&config);
```

Parameters

config	pointer to master configuration structure

32.6.4 void SAI_RxGetDefaultConfig (sai_config_t * *config*)

This API initializes the configuration structure for use in SAI_RxConfig(). The initialized structure can remain unchanged in SAI_RxConfig() or it can be modified before calling SAI_RxConfig(). This is an example.

```
sai_config_t config;
SAI_RxGetDefaultConfig(&config);
```

Parameters

config	pointer to master configuration structure

32.6.5 void SAI_Deinit (I2S_Type * *base*)

This API gates the SAI clock. The SAI module can't operate unless SAI_TxInit or SAI_RxInit is called to enable the clock.

Parameters

base	SAI base pointer

32.6.6 void SAI_TxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Tx. After reset, clear the reset bit.

Parameters

base	SAI base pointer

32.6.7 void SAI_RxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Rx. After reset, clear the reset bit.

Parameters

base	SAI base pointer

32.6.8 void SAI_TxEnable (I2S_Type * *base,* bool *enable*)

Parameters

base	SAI base pointer
enable	True means enable SAI Tx, false means disable.

32.6.9 void SAI_RxEnable (I2S_Type * *base,* bool *enable*)

Parameters

base	SAI base pointer
enable	True means enable SAI Rx, false means disable.

32.6.10 static uint32_t SAI_TxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

base	SAI base pointer

Returns

SAI Tx status flag value. Use the Status Mask to get the status value needed.

32.6.11 static void SAI_TxClearStatusFlags (I2S_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	State mask. It can be a combination of the following source if defined: • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

32.6.12 static uint32_t SAI_RxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

base	SAI base pointer

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

32.6.13 static void SAI_RxClearStatusFlags (I2S_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	State mask. It can be a combination of the following sources if defined. • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

32.6.14 static void SAI_TxEnableInterrupts (I2S_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	interrupt source The parameter can be a combination of the following sources if defined. • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

32.6.15 static void SAI_RxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	interrupt source The parameter can be a combination of the following sources if defined. • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

32.6.16 static void SAI_TxDisableInterrupts (I2S_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	interrupt source The parameter can be a combination of the following sources if defined. • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

32.6.17 static void SAI_RxDisableInterrupts (I2S_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SAI base pointer
mask	interrupt source The parameter can be a combination of the following sources if defined. • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

32.6.18 static void SAI_TxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

base	SAI base pointer
mask	DMA source The parameter can be combination of the following sources if defined. • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
enable	True means enable DMA, false means disable DMA.

32.6.19 static void SAI_RxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

base	SAI base pointer
mask	DMA source The parameter can be a combination of the following sources if defined. • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
enable	True means enable DMA, false means disable DMA.

32.6.20 static uint32_t SAI_TxGetDataRegisterAddress (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

base	SAI base pointer.
channel	Which data channel used.

Returns

data register address.

32.6.21 static uint32_t SAI_RxGetDataRegisterAddress (I2S_Type * *base,* uint32_t *channel*) [inline], [static]

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

base	SAI base pointer.
channel	Which data channel used.

Returns

data register address.

32.6.22 void SAI_TxSetFormat (I2S_Type * *base,* sai_transfer_format_t * *format,* uint32_t *mclkSourceClockHz,* uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

base	SAI base pointer.
format	Pointer to the SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

32.6.23 void SAI_RxSetFormat (I2S_Type * *base,* sai_transfer_format_t * *format,* uint32_t *mclkSourceClockHz,* uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

base	SAI base pointer.
format	Pointer to the SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

32.6.24 void SAI_WriteBlocking (I2S_Type * *base,* uint32_t *channel,* uint32_t *bitWidth,* uint8_t * *buffer,* uint32_t *size*)

Note

This function blocks by polling until data is ready to be sent.

Parameters

base	SAI base pointer.
channel	Data channel used.
bitWidth	How many bits in an audio word; usually 8/16/24/32 bits.
buffer	Pointer to the data to be written.
size	Bytes to be written.

32.6.25 static void SAI_WriteData (I2S_Type * *base,* uint32_t *channel,* uint32_t *data*) [inline], [static]

Parameters

base	SAI base pointer.
channel	Data channel used.
data	Data needs to be written.

32.6.26 void SAI_ReadBlocking (I2S_Type * *base,* uint32_t *channel,* uint32_t *bitWidth,* uint8_t * *buffer,* uint32_t *size*)

Note

This function blocks by polling until data is ready to be sent.

Parameters

base	SAI base pointer.
channel	Data channel used.
bitWidth	How many bits in an audio word; usually 8/16/24/32 bits.
buffer	Pointer to the data to be read.
size	Bytes to be read.

32.6.27 static uint32_t SAI_ReadData (I2S_Type ∗ *base*, uint32_t *channel*) [inline], [static]

Parameters

base	SAI base pointer.
channel	Data channel used.

Returns

Data in SAI FIFO.

32.6.28 void SAI_TransferTxCreateHandle (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*, sai_transfer_callback_t *callback*, void ∗ *userData*)

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

base	SAI base pointer
handle	SAI handle pointer.
callback	Pointer to the user callback function.
userData	User parameter passed to the callback function

32.6.29 void SAI_TransferRxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_callback_t *callback*, void * *userData*)

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

base	SAI base pointer.
handle	SAI handle pointer.
callback	Pointer to the user callback function.
userData	User parameter passed to the callback function.

32.6.30 status_t SAI_TransferTxSetFormat (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*, sai_transfer_format_t ∗ *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

base	SAI base pointer.
handle	SAI handle pointer.
format	Pointer to the SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is the status_t.

32.6.31 status_t SAI_TransferRxSetFormat (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*, sai_transfer_format_t ∗ *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

base	SAI base pointer.
handle	SAI handle pointer.
format	Pointer to the SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

32.6.32 status_t SAI_TransferSendNonBlocking (I2S_Type ∗ *base,* sai_handle_t ∗ *handle,* sai_transfer_t ∗ *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_-SAI_Busy, the transfer is finished.

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure which stores the transfer state.
xfer	Pointer to the sai_transfer_t structure.

Return values

kStatus_Success	Successfully started the data receive.
kStatus_SAI_TxBusy	Previous receive still not finished.
kStatus_InvalidArgument	The input parameter is invalid.

32.6.33 status_t SAI_TransferReceiveNonBlocking (I2S_Type ∗ *base,* sai_handle_t ∗ *handle,* sai_transfer_t ∗ *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_RxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_-SAI_Busy, the transfer is finished.

Parameters

base	SAI base pointer
handle	Pointer to the sai_handle_t structure which stores the transfer state.
xfer	Pointer to the sai_transfer_t structure.

Return values

kStatus_Success	Successfully started the data receive.
kStatus_SAI_RxBusy	Previous receive still not finished.
kStatus_InvalidArgument	The input parameter is invalid.

32.6.34 status_t SAI_TransferGetSendCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure which stores the transfer state.
count	Bytes count sent.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

32.6.35 status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure which stores the transfer state.
count	Bytes count received.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

32.6.36 void SAI_TransferAbortSend (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*)

Note

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

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure which stores the transfer state.

32.6.37 void SAI_TransferAbortReceive (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*)

Note

This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

base	SAI base pointer
handle	Pointer to the sai_handle_t structure which stores the transfer state.

32.6.38 void SAI_TransferTxHandleIRQ (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*)

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure.

32.6.39 void SAI_TransferRxHandleIRQ (I2S_Type ∗ *base*, sai_handle_t ∗ *handle*)

Parameters

base	SAI base pointer.
handle	Pointer to the sai_handle_t structure.

32.7 SAI DMA Driver

32.7.1 Overview

Data Structures

- struct sai_dma_handle_t
  SAI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ sai_dma_callback_t)(I2S_Type ∗base, sai_dma_handle_t ∗handle, status_t status, void ∗userData)
  Define SAI DMA callback.

DMA Transactional

- void SAI_TransferTxCreateHandleDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_dma_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Initializes the SAI master DMA handle.
- void SAI_TransferRxCreateHandleDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_dma_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Initializes the SAI slave DMA handle.
- void SAI_TransferTxSetFormatDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_transfer_format_t ∗format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Tx audio format.
- void SAI_TransferRxSetFormatDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_transfer_format_t ∗format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Rx audio format.
- status_t SAI_TransferSendDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_transfer_t ∗xfer)
  Performs a non-blocking SAI transfer using DMA.
- status_t SAI_TransferReceiveDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, sai_transfer_t ∗xfer)
  Performs a non-blocking SAI transfer using DMA.
- void SAI_TransferAbortSendDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle)
  Aborts a SAI transfer using DMA.
- void SAI_TransferAbortReceiveDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle)
  Aborts a SAI transfer using DMA.
- status_t SAI_TransferGetSendCountDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, size_t ∗count)
  Gets byte count sent by SAI.
- status_t SAI_TransferGetReceiveCountDMA (I2S_Type ∗base, sai_dma_handle_t ∗handle, size_t ∗count)
  Gets byte count received by SAI.

32.7.2 Data Structure Documentation

32.7.2.1 struct _sai_dma_handle

Data Fields

- dma_handle_t * dmaHandle
  DMA handler for SAI send.
- uint8_t bytesPerFrame
  Bytes in a frame.
- uint8_t channel
  Which Data channel SAI use.
- uint32_t state
  SAI DMA transfer internal state.
- sai_dma_callback_t callback
  Callback for users while transfer finish or error occured.
- void * userData
  User callback parameter.
- sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [SAI_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.

32.7.2.1.0.32 Field Documentation

32.7.2.1.0.32.1 sai_transfer_t sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]

32.7.2.1.0.32.2 volatile uint8_t sai_dma_handle_t::queueUser

32.7.3 Function Documentation

32.7.3.1 void SAI_TransferTxCreateHandleDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_dma_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

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

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
base	SAI peripheral base address.
callback	Pointer to user callback function.
userData	User parameter passed to the callback function.
dmaHandle	DMA handle pointer, this handle shall be static allocated by users.

32.7.3.2 void SAI_TransferRxCreateHandleDMA (I2S_Type ∗ *base,* sai_dma_handle_t ∗ *handle,* sai_dma_callback_t *callback,* void ∗ *userData,* dma_handle_t ∗ *dmaHandle*)

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

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
base	SAI peripheral base address.
callback	Pointer to user callback function.
userData	User parameter passed to the callback function.
dmaHandle	DMA handle pointer, this handle shall be static allocated by users.

32.7.3.3 void SAI_TransferTxSetFormatDMA (I2S_Type ∗ *base,* sai_dma_handle_t ∗ *handle,* sai_transfer_format_t ∗ *format,* uint32_t *mclkSourceClockHz,* uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to the format.

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
format	Pointer to SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input arguments is invalid.

32.7.3.4 void SAI_TransferRxSetFormatDMA (I2S_Type ∗ *base*, sai_dma_handle_t ∗ *handle*, sai_transfer_format_t ∗ *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets eDMA parameter according to format.

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
format	Pointer to SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input arguments is invalid.

32.7.3.5 status_t SAI_TransferSendDMA (I2S_Type ∗ *base*, sai_dma_handle_t ∗ *handle*, sai_transfer_t ∗ *xfer*)

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetTransferStatus to poll the transfer status to check whether the SAI transfer finished.

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Successfully start the data receive.
kStatus_SAI_TxBusy	Previous receive still not finished.
kStatus_InvalidArgument	The input parameter is invalid.

32.7.3.6 status_t SAI_TransferReceiveDMA (I2S_Type ∗ *base,* sai_dma_handle_t ∗ *handle,* sai_transfer_t ∗ *xfer*)

Note

This interface returns immediately after transfer initiates. Call SAI_GetTransferStatus to poll the transfer status to check whether the SAI transfer is finished.

Parameters

base	SAI base pointer
handle	SAI DMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Successfully start the data receive.
kStatus_SAI_RxBusy	Previous receive still not finished.
kStatus_InvalidArgument	The input parameter is invalid.

32.7.3.7 void SAI_TransferAbortSendDMA (I2S_Type ∗ *base,* sai_dma_handle_t ∗ *handle*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.

32.7.3.8 void SAI_TransferAbortReceiveDMA (I2S_Type ∗ *base*, sai_dma_handle_t ∗ *handle*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.

32.7.3.9 status_t SAI_TransferGetSendCountDMA (I2S_Type ∗ *base*, sai_dma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
count	Bytes count sent by SAI.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

32.7.3.10 status_t SAI_TransferGetReceiveCountDMA (I2S_Type ∗ *base*, sai_dma_handle_t ∗ *handle*, size_t ∗ *count*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.
count	Bytes count received by SAI.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

32.8 SAI eDMA Driver

32.8.1 Overview

Data Structures

- struct sai_edma_handle_t
  SAI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ sai_edma_callback_t)(I2S_Type ∗base, sai_edma_handle_t ∗handle, status_t status, void ∗userData)
  SAI eDMA transfer callback function for finish and error.

eDMA Transactional

- void SAI_TransferTxCreateHandleEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the SAI eDMA handle.
- void SAI_TransferRxCreateHandleEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_edma_callback_t callback, void ∗userData, edma_handle_t ∗dmaHandle)
  Initializes the SAI Rx eDMA handle.
- void SAI_TransferTxSetFormatEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_transfer_format_t ∗format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Tx audio format.
- void SAI_TransferRxSetFormatEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_transfer_format_t ∗format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
  Configures the SAI Rx audio format.
- status_t SAI_TransferSendEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_transfer_t ∗xfer)
  Performs a non-blocking SAI transfer using DMA.
- status_t SAI_TransferReceiveEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, sai_transfer_t ∗xfer)
  Performs a non-blocking SAI receive using eDMA.
- void SAI_TransferAbortSendEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle)
  Aborts a SAI transfer using eDMA.
- void SAI_TransferAbortReceiveEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle)
  Aborts a SAI receive using eDMA.
- status_t SAI_TransferGetSendCountEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, size_t ∗count)
  Gets byte count sent by SAI.
- status_t SAI_TransferGetReceiveCountEDMA (I2S_Type ∗base, sai_edma_handle_t ∗handle, size_t ∗count)
  Gets byte count received by SAI.

32.8.2 Data Structure Documentation

32.8.2.1 struct _sai_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  DMA handler for SAI send.
- uint8_t nbytes
  eDMA minor byte transfer count initially configured.
- uint8_t bytesPerFrame
  Bytes in a frame.
- uint8_t channel
  Which data channel.
- uint8_t count
  The transfer data count in a DMA request.
- uint32_t state
  Internal state for SAI eDMA transfer.
- sai_edma_callback_t callback
  Callback for users while transfer finish or error occurs.
- void * userData
  User callback parameter.
- edma_tcd_t tcd [SAI_XFER_QUEUE_SIZE+1U]
  TCD pool for eDMA transfer.
- sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]
  Transfer queue storing queued transfer.
- size_t transferSize [SAI_XFER_QUEUE_SIZE]
  Data bytes need to transfer.
- volatile uint8_t queueUser
  Index for user to queue transfer.
- volatile uint8_t queueDriver
  Index for driver to get the transfer data and size.

32.8.2.1.0.33 Field Documentation

32.8.2.1.0.33.1 uint8_t sai_edma_handle_t::nbytes

32.8.2.1.0.33.2 edma_tcd_t sai_edma_handle_t::tcd[SAI_XFER_QUEUE_SIZE+1U]

32.8.2.1.0.33.3 sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]

32.8.2.1.0.33.4 volatile uint8_t sai_edma_handle_t::queueUser

32.8.3 Function Documentation

32.8.3.1 void SAI_TransferTxCreateHandleEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_edma_callback_t *callback*, void * *userData*, edma_handle_t * *dmaHandle*)

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

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.
base	SAI peripheral base address.
callback	Pointer to user callback function.
userData	User parameter passed to the callback function.
dmaHandle	eDMA handle pointer, this handle shall be static allocated by users.

32.8.3.2 void SAI_TransferRxCreateHandleEDMA (I2S_Type ∗ *base*, sai_edma_handle_t ∗ *handle*, sai_edma_callback_t *callback*, void ∗ *userData*, edma_handle_t ∗ *dmaHandle*)

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

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.
base	SAI peripheral base address.
callback	Pointer to user callback function.
userData	User parameter passed to the callback function.
dmaHandle	eDMA handle pointer, this handle shall be static allocated by users.

32.8.3.3 void SAI_TransferTxSetFormatEDMA (I2S_Type ∗ *base*, sai_edma_handle_t ∗ *handle*, sai_transfer_format_t ∗ *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

base	SAI base pointer.

handle	SAI eDMA handle pointer.
format	Pointer to SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If bit clock source is master clock, this value should equals to masterClockHz in format.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input argument is invalid.

32.8.3.4 void SAI_TransferRxSetFormatEDMA (I2S_Type ∗ *base,* sai_edma_handle_t ∗ *handle,* sai_transfer_format_t ∗ *format,* uint32_t *mclkSourceClockHz,* uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.
format	Pointer to SAI audio data format structure.
mclkSource-ClockHz	SAI master clock source frequency in Hz.
bclkSource-ClockHz	SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format.

Return values

kStatus_Success	Audio format set successfully.
kStatus_InvalidArgument	The input argument is invalid.

32.8.3.5 status_t SAI_TransferSendEDMA (I2S_Type ∗ *base,* sai_edma_handle_t ∗ *handle,* sai_transfer_t ∗ *xfer*)

Note

This interface returns immediately after the transfer initiates. Call SAI_GetTransferStatus to poll the transfer status and check whether the SAI transfer is finished.

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.
xfer	Pointer to the DMA transfer structure.

Return values

kStatus_Success	Start a SAI eDMA send successfully.
kStatus_InvalidArgument	The input argument is invalid.
kStatus_TxBusy	SAI is busy sending data.

32.8.3.6 status_t SAI_TransferReceiveEDMA (I2S_Type ∗ *base*, sai_edma_handle_t ∗ *handle*, sai_transfer_t ∗ *xfer*)

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetReceiveRemaining-Bytes to poll the transfer status and check whether the SAI transfer is finished.

Parameters

base	SAI base pointer
handle	SAI eDMA handle pointer.
xfer	Pointer to DMA transfer structure.

Return values

kStatus_Success	Start a SAI eDMA receive successfully.
kStatus_InvalidArgument	The input argument is invalid.
kStatus_RxBusy	SAI is busy receiving data.

32.8.3.7 void SAI_TransferAbortSendEDMA (I2S_Type ∗ *base*, sai_edma_handle_t ∗ *handle*)

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.

32.8.3.8 void SAI_TransferAbortReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

Parameters

base	SAI base pointer
handle	SAI eDMA handle pointer.

32.8.3.9 status_t SAI_TransferGetSendCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.
count	Bytes count sent by SAI.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is no non-blocking transaction in progress.

32.8.3.10 status_t SAI_TransferGetReceiveCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer
handle	SAI eDMA handle pointer.
count	Bytes count received by SAI.

Return values

kStatus_Success	Succeed get the transfer count.
kStatus_NoTransferInProgress	There is no non-blocking transaction in progress.

Chapter 33
SDHC: Secured Digital Host Controller Driver

33.1 Overview

The KSDK provides a peripheral driver for the Secured Digital Host Controller (SDHC) module of Kinetis devices.

33.2 Typical use case

33.2.1 SDHC Operation

```
/* Initializes the SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Fills state in the card driver. */
card->sdhcBase = BOARD_SDHC_BASEADDR;
card->sdhcSourceClock = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->sdhcTransfer = sdhc_transfer_function;

/* Initializes the card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

PRINTF("\r\nRead/Write/Erase the card continuously until it encounters error......\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);
```

Data Structures

- struct sdhc_adma2_descriptor_t
  Defines the ADMA2 descriptor structure. More...
- struct sdhc_capability_t

SDHC capability information. More...
- struct sdhc_transfer_config_t
  Card transfer configuration. More...
- struct sdhc_boot_config_t
  Data structure to configure the MMC boot feature. More...
- struct sdhc_config_t
  Data structure to initialize the SDHC. More...
- struct sdhc_data_t
  Card data descriptor. More...
- struct sdhc_command_t
  Card command descriptor. More...
- struct sdhc_transfer_t
  Transfer state. More...
- struct sdhc_transfer_callback_t
  SDHC callback functions. More...
- struct sdhc_handle_t
  SDHC handle. More...
- struct sdhc_host_t
  SDHC host descriptor. More...

Macros

- #define SDHC_MAX_BLOCK_COUNT (SDHC_BLKATTR_BLKCNT_MASK >> SDHC_BLKATTR_BLKCNT_SHIFT)
  Maximum block count can be set one time.
- #define SDHC_ADMA1_ADDRESS_ALIGN (4096U)
  The alignment size for ADDRESS filed in ADMA1's descriptor.
- #define SDHC_ADMA1_LENGTH_ALIGN (4096U)
  The alignment size for LENGTH field in ADMA1's descriptor.
- #define SDHC_ADMA2_ADDRESS_ALIGN (4U)
  The alignment size for ADDRESS field in ADMA2's descriptor.
- #define SDHC_ADMA2_LENGTH_ALIGN (4U)
  The alignment size for LENGTH filed in ADMA2's descriptor.
- #define SDHC_ADMA1_DESCRIPTOR_ADDRESS_SHIFT (12U)
  The bit shift for ADDRESS filed in ADMA1's descriptor.
- #define SDHC_ADMA1_DESCRIPTOR_ADDRESS_MASK (0xFFFFFU)
  The bit mask for ADDRESS field in ADMA1's descriptor.
- #define SDHC_ADMA1_DESCRIPTOR_LENGTH_SHIFT (12U)
  The bit shift for LENGTH filed in ADMA1's descriptor.
- #define SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK (0xFFFFU)
  The mask for LENGTH field in ADMA1's descriptor.
- #define SDHC_ADMA1_DESCRIPTOR_MAX_LENGTH_PER_ENTRY (SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK + 1U)
  The maximum value of LENGTH filed in ADMA1's descriptor.
- #define SDHC_ADMA2_DESCRIPTOR_LENGTH_SHIFT (16U)
  The bit shift for LENGTH field in ADMA2's descriptor.
- #define SDHC_ADMA2_DESCRIPTOR_LENGTH_MASK (0xFFFFU)
  The bit mask for LENGTH field in ADMA2's descriptor.
- #define SDHC_ADMA2_DESCRIPTOR_MAX_LENGTH_PER_ENTRY (SDHC_ADMA2_DESCRIPTOR_LENGTH_MASK)
  The maximum value of LENGTH field in ADMA2's descriptor.

Typedefs

- typedef uint32_t sdhc_adma1_descriptor_t
  Defines the adma1 descriptor structure.
- typedef status_t(∗ sdhc_transfer_function_t)(SDHC_Type ∗base, sdhc_transfer_t ∗content)
  SDHC transfer function.

Enumerations

- enum _sdhc_status {
  kStatus_SDHC_BusyTransferring = MAKE_STATUS(kStatusGroup_SDHC, 0U),
  kStatus_SDHC_PrepareAdmaDescriptorFailed = MAKE_STATUS(kStatusGroup_SDHC, 1U),
  kStatus_SDHC_SendCommandFailed = MAKE_STATUS(kStatusGroup_SDHC, 2U),
  kStatus_SDHC_TransferDataFailed = MAKE_STATUS(kStatusGroup_SDHC, 3U) }
  SDHC status.
- enum _sdhc_capability_flag {
  kSDHC_SupportAdmaFlag = SDHC_HTCAPBLT_ADMAS_MASK,
  kSDHC_SupportHighSpeedFlag = SDHC_HTCAPBLT_HSS_MASK,
  kSDHC_SupportDmaFlag = SDHC_HTCAPBLT_DMAS_MASK,
  kSDHC_SupportSuspendResumeFlag = SDHC_HTCAPBLT_SRS_MASK,
  kSDHC_SupportV330Flag = SDHC_HTCAPBLT_VS33_MASK,
  kSDHC_Support4BitFlag = (SDHC_HTCAPBLT_MBL_SHIFT << 0U),
  kSDHC_Support8BitFlag = (SDHC_HTCAPBLT_MBL_SHIFT << 1U) }
  Host controller capabilities flag mask.
- enum _sdhc_wakeup_event {
  kSDHC_WakeupEventOnCardInt = SDHC_PROCTL_WECINT_MASK,
  kSDHC_WakeupEventOnCardInsert = SDHC_PROCTL_WECINS_MASK,
  kSDHC_WakeupEventOnCardRemove = SDHC_PROCTL_WECRM_MASK,
  kSDHC_WakeupEventsAll }
  Wakeup event mask.
- enum _sdhc_reset {
  kSDHC_ResetAll = SDHC_SYSCTL_RSTA_MASK,
  kSDHC_ResetCommand = SDHC_SYSCTL_RSTC_MASK,
  kSDHC_ResetData = SDHC_SYSCTL_RSTD_MASK,
  kSDHC_ResetsAll = (kSDHC_ResetAll | kSDHC_ResetCommand | kSDHC_ResetData) }
  Reset type mask.
- enum _sdhc_transfer_flag {

kSDHC_EnableDmaFlag = SDHC_XFERTYP_DMAEN_MASK,
kSDHC_CommandTypeSuspendFlag = (SDHC_XFERTYP_CMDTYP(1U)),
kSDHC_CommandTypeResumeFlag = (SDHC_XFERTYP_CMDTYP(2U)),
kSDHC_CommandTypeAbortFlag = (SDHC_XFERTYP_CMDTYP(3U)),
kSDHC_EnableBlockCountFlag = SDHC_XFERTYP_BCEN_MASK,
kSDHC_EnableAutoCommand12Flag = SDHC_XFERTYP_AC12EN_MASK,
kSDHC_DataReadFlag = SDHC_XFERTYP_DTDSEL_MASK,
kSDHC_MultipleBlockFlag = SDHC_XFERTYP_MSBSEL_MASK,
kSDHC_ResponseLength136Flag = SDHC_XFERTYP_RSPTYP(1U),
kSDHC_ResponseLength48Flag = SDHC_XFERTYP_RSPTYP(2U),
kSDHC_ResponseLength48BusyFlag = SDHC_XFERTYP_RSPTYP(3U),
kSDHC_EnableCrcCheckFlag = SDHC_XFERTYP_CCCEN_MASK,
kSDHC_EnableIndexCheckFlag = SDHC_XFERTYP_CICEN_MASK,
kSDHC_DataPresentFlag = SDHC_XFERTYP_DPSEL_MASK }

Transfer flag mask.

- enum _sdhc_present_status_flag {
kSDHC_CommandInhibitFlag = SDHC_PRSSTAT_CIHB_MASK,
kSDHC_DataInhibitFlag = SDHC_PRSSTAT_CDIHB_MASK,
kSDHC_DataLineActiveFlag = SDHC_PRSSTAT_DLA_MASK,
kSDHC_SdClockStableFlag = SDHC_PRSSTAT_SDSTB_MASK,
kSDHC_WriteTransferActiveFlag = SDHC_PRSSTAT_WTA_MASK,
kSDHC_ReadTransferActiveFlag = SDHC_PRSSTAT_RTA_MASK,
kSDHC_BufferWriteEnableFlag = SDHC_PRSSTAT_BWEN_MASK,
kSDHC_BufferReadEnableFlag = SDHC_PRSSTAT_BREN_MASK,
kSDHC_CardInsertedFlag = SDHC_PRSSTAT_CINS_MASK,
kSDHC_CommandLineLevelFlag = SDHC_PRSSTAT_CLSL_MASK,
kSDHC_Data0LineLevelFlag = (1U << 24U),
kSDHC_Data1LineLevelFlag = (1U << 25U),
kSDHC_Data2LineLevelFlag = (1U << 26U),
kSDHC_Data3LineLevelFlag = (1U << 27U),
kSDHC_Data4LineLevelFlag = (1U << 28U),
kSDHC_Data5LineLevelFlag = (1U << 29U),
kSDHC_Data6LineLevelFlag = (1U << 30U),
kSDHC_Data7LineLevelFlag = (1U << 31U) }

Present status flag mask.

- enum _sdhc_interrupt_status_flag {

kSDHC_CommandCompleteFlag = SDHC_IRQSTAT_CC_MASK,
kSDHC_DataCompleteFlag = SDHC_IRQSTAT_TC_MASK,
kSDHC_BlockGapEventFlag = SDHC_IRQSTAT_BGE_MASK,
kSDHC_DmaCompleteFlag = SDHC_IRQSTAT_DINT_MASK,
kSDHC_BufferWriteReadyFlag = SDHC_IRQSTAT_BWR_MASK,
kSDHC_BufferReadReadyFlag = SDHC_IRQSTAT_BRR_MASK,
kSDHC_CardInsertionFlag = SDHC_IRQSTAT_CINS_MASK,
kSDHC_CardRemovalFlag = SDHC_IRQSTAT_CRM_MASK,
kSDHC_CardInterruptFlag = SDHC_IRQSTAT_CINT_MASK,
kSDHC_CommandTimeoutFlag = SDHC_IRQSTAT_CTOE_MASK,
kSDHC_CommandCrcErrorFlag = SDHC_IRQSTAT_CCE_MASK,
kSDHC_CommandEndBitErrorFlag = SDHC_IRQSTAT_CEBE_MASK,
kSDHC_CommandIndexErrorFlag = SDHC_IRQSTAT_CIE_MASK,
kSDHC_DataTimeoutFlag = SDHC_IRQSTAT_DTOE_MASK,
kSDHC_DataCrcErrorFlag = SDHC_IRQSTAT_DCE_MASK,
kSDHC_DataEndBitErrorFlag = SDHC_IRQSTAT_DEBE_MASK,
kSDHC_AutoCommand12ErrorFlag = SDHC_IRQSTAT_AC12E_MASK,
kSDHC_DmaErrorFlag = SDHC_IRQSTAT_DMAE_MASK,
kSDHC_CommandErrorFlag,
kSDHC_DataErrorFlag,
kSDHC_ErrorFlag = (kSDHC_CommandErrorFlag | kSDHC_DataErrorFlag | kSDHC_DmaErrorFlag),
kSDHC_DataFlag,
kSDHC_CommandFlag = (kSDHC_CommandErrorFlag | kSDHC_CommandCompleteFlag),
kSDHC_CardDetectFlag = (kSDHC_CardInsertionFlag | kSDHC_CardRemovalFlag),
kSDHC_AllInterruptFlags }

Interrupt status flag mask.

- enum _sdhc_auto_command12_error_status_flag {
kSDHC_AutoCommand12NotExecutedFlag = SDHC_AC12ERR_AC12NE_MASK,
kSDHC_AutoCommand12TimeoutFlag = SDHC_AC12ERR_AC12TOE_MASK,
kSDHC_AutoCommand12EndBitErrorFlag = SDHC_AC12ERR_AC12EBE_MASK,
kSDHC_AutoCommand12CrcErrorFlag = SDHC_AC12ERR_AC12CE_MASK,
kSDHC_AutoCommand12IndexErrorFlag = SDHC_AC12ERR_AC12IE_MASK,
kSDHC_AutoCommand12NotIssuedFlag = SDHC_AC12ERR_CNIBAC12E_MASK }

  Auto CMD12 error status flag mask.
- enum _sdhc_adma_error_status_flag {
kSDHC_AdmaLenghMismatchFlag = SDHC_ADMAES_ADMALME_MASK,
kSDHC_AdmaDescriptorErrorFlag = SDHC_ADMAES_ADMADCE_MASK }

  ADMA error status flag mask.
- enum sdhc_adma_error_state_t {
kSDHC_AdmaErrorStateStopDma = 0x00U,
kSDHC_AdmaErrorStateFetchDescriptor = 0x01U,
kSDHC_AdmaErrorStateChangeAddress = 0x02U,
kSDHC_AdmaErrorStateTransferData = 0x03U }

  ADMA error state.
- enum _sdhc_force_event {

kSDHC_ForceEventAutoCommand12NotExecuted = SDHC_FEVT_AC12NE_MASK,
kSDHC_ForceEventAutoCommand12Timeout = SDHC_FEVT_AC12TOE_MASK,
kSDHC_ForceEventAutoCommand12CrcError = SDHC_FEVT_AC12CE_MASK,
kSDHC_ForceEventEndBitError = SDHC_FEVT_AC12EBE_MASK,
kSDHC_ForceEventAutoCommand12IndexError = SDHC_FEVT_AC12IE_MASK,
kSDHC_ForceEventAutoCommand12NotIssued = SDHC_FEVT_CNIBAC12E_MASK,
kSDHC_ForceEventCommandTimeout = SDHC_FEVT_CTOE_MASK,
kSDHC_ForceEventCommandCrcError = SDHC_FEVT_CCE_MASK,
kSDHC_ForceEventCommandEndBitError = SDHC_FEVT_CEBE_MASK,
kSDHC_ForceEventCommandIndexError = SDHC_FEVT_CIE_MASK,
kSDHC_ForceEventDataTimeout = SDHC_FEVT_DTOE_MASK,
kSDHC_ForceEventDataCrcError = SDHC_FEVT_DCE_MASK,
kSDHC_ForceEventDataEndBitError = SDHC_FEVT_DEBE_MASK,
kSDHC_ForceEventAutoCommand12Error = SDHC_FEVT_AC12E_MASK,
kSDHC_ForceEventCardInt = SDHC_FEVT_CINT_MASK,
kSDHC_ForceEventDmaError = SDHC_FEVT_DMAE_MASK,
kSDHC_ForceEventsAll }

Force event mask.

- enum sdhc_data_bus_width_t {
  kSDHC_DataBusWidth1Bit = 0U,
  kSDHC_DataBusWidth4Bit = 1U,
  kSDHC_DataBusWidth8Bit = 2U }

  Data transfer width.
- enum sdhc_endian_mode_t {
  kSDHC_EndianModeBig = 0U,
  kSDHC_EndianModeHalfWordBig = 1U,
  kSDHC_EndianModeLittle = 2U }

  Endian mode.
- enum sdhc_dma_mode_t {
  kSDHC_DmaModeNo = 0U,
  kSDHC_DmaModeAdma1 = 1U,
  kSDHC_DmaModeAdma2 = 2U }

  DMA mode.
- enum _sdhc_sdio_control_flag {
  kSDHC_StopAtBlockGapFlag = 0x01,
  kSDHC_ReadWaitControlFlag = 0x02,
  kSDHC_InterruptAtBlockGapFlag = 0x04,
  kSDHC_ExactBlockNumberReadFlag = 0x08 }

  SDIO control flag mask.
- enum sdhc_boot_mode_t {
  kSDHC_BootModeNormal = 0U,
  kSDHC_BootModeAlternative = 1U }

  MMC card boot mode.
- enum sdhc_command_type_t {

kSDHC_CommandTypeNormal = 0U,
kSDHC_CommandTypeSuspend = 1U,
kSDHC_CommandTypeResume = 2U,
kSDHC_CommandTypeAbort = 3U }
The command type.
- enum sdhc_response_type_t {
kSDHC_ResponseTypeNone = 0U,
kSDHC_ResponseTypeR1 = 1U,
kSDHC_ResponseTypeR1b = 2U,
kSDHC_ResponseTypeR2 = 3U,
kSDHC_ResponseTypeR3 = 4U,
kSDHC_ResponseTypeR4 = 5U,
kSDHC_ResponseTypeR5 = 6U,
kSDHC_ResponseTypeR5b = 7U,
kSDHC_ResponseTypeR6 = 8U,
kSDHC_ResponseTypeR7 = 9U }
The command response type.
- enum _sdhc_adma1_descriptor_flag {
kSDHC_Adma1DescriptorValidFlag = (1U << 0U),
kSDHC_Adma1DescriptorEndFlag = (1U << 1U),
kSDHC_Adma1DescriptorInterrupFlag = (1U << 2U),
kSDHC_Adma1DescriptorActivity1Flag = (1U << 4U),
kSDHC_Adma1DescriptorActivity2Flag = (1U << 5U),
kSDHC_Adma1DescriptorTypeNop = (kSDHC_Adma1DescriptorValidFlag),
kSDHC_Adma1DescriptorTypeTransfer,
kSDHC_Adma1DescriptorTypeLink,
kSDHC_Adma1DescriptorTypeSetLength }
The mask for the control/status field in ADMA1 descriptor.
- enum _sdhc_adma2_descriptor_flag {
kSDHC_Adma2DescriptorValidFlag = (1U << 0U),
kSDHC_Adma2DescriptorEndFlag = (1U << 1U),
kSDHC_Adma2DescriptorInterruptFlag = (1U << 2U),
kSDHC_Adma2DescriptorActivity1Flag = (1U << 4U),
kSDHC_Adma2DescriptorActivity2Flag = (1U << 5U),
kSDHC_Adma2DescriptorTypeNop = (kSDHC_Adma2DescriptorValidFlag),
kSDHC_Adma2DescriptorTypeReserved,
kSDHC_Adma2DescriptorTypeTransfer,
kSDHC_Adma2DescriptorTypeLink }
ADMA1 descriptor control and status mask.

Driver version

- #define FSL_SDHC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 1U))
Driver version 2.1.1.

Initialization and deinitialization

- void SDHC_Init (SDHC_Type ∗base, const sdhc_config_t ∗config)
  SDHC module initialization function.
- void SDHC_Deinit (SDHC_Type ∗base)
  Deinitializes the SDHC.
- bool SDHC_Reset (SDHC_Type ∗base, uint32_t mask, uint32_t timeout)
  Resets the SDHC.

DMA Control

- status_t SDHC_SetAdmaTableConfig (SDHC_Type ∗base, sdhc_dma_mode_t dmaMode, uint32_t ∗table, uint32_t tableWords, const uint32_t ∗data, uint32_t dataBytes)
  Sets the ADMA descriptor table configuration.

Interrupts

- static void SDHC_EnableInterruptStatus (SDHC_Type ∗base, uint32_t mask)
  Enables the interrupt status.
- static void SDHC_DisableInterruptStatus (SDHC_Type ∗base, uint32_t mask)
  Disables the interrupt status.
- static void SDHC_EnableInterruptSignal (SDHC_Type ∗base, uint32_t mask)
  Enables the interrupt signal corresponding to the interrupt status flag.
- static void SDHC_DisableInterruptSignal (SDHC_Type ∗base, uint32_t mask)
  Disables the interrupt signal corresponding to the interrupt status flag.

Status

- static uint32_t SDHC_GetInterruptStatusFlags (SDHC_Type ∗base)
  Gets the current interrupt status.
- static void SDHC_ClearInterruptStatusFlags (SDHC_Type ∗base, uint32_t mask)
  Clears a specified interrupt status.
- static uint32_t SDHC_GetAutoCommand12ErrorStatusFlags (SDHC_Type ∗base)
  Gets the status of auto command 12 error.
- static uint32_t SDHC_GetAdmaErrorStatusFlags (SDHC_Type ∗base)
  Gets the status of the ADMA error.
- static uint32_t SDHC_GetPresentStatusFlags (SDHC_Type ∗base)
  Gets a present status.

Bus Operations

- void SDHC_GetCapability (SDHC_Type ∗base, sdhc_capability_t ∗capability)
  Gets the capability information.
- static void SDHC_EnableSdClock (SDHC_Type ∗base, bool enable)
  Enables or disables the SD bus clock.
- uint32_t SDHC_SetSdClock (SDHC_Type ∗base, uint32_t srcClock_Hz, uint32_t busClock_Hz)
  Sets the SD bus clock frequency.
- bool SDHC_SetCardActive (SDHC_Type ∗base, uint32_t timeout)
  Sends 80 clocks to the card to set it to the active state.
- static void SDHC_SetDataBusWidth (SDHC_Type ∗base, sdhc_data_bus_width_t width)
  Sets the data transfer width.

- void SDHC_SetTransferConfig (SDHC_Type ∗base, const sdhc_transfer_config_t ∗config)
  Sets the card transfer-related configuration.
- static uint32_t SDHC_GetCommandResponse (SDHC_Type ∗base, uint32_t index)
  Gets the command response.
- static void SDHC_WriteData (SDHC_Type ∗base, uint32_t data)
  Fills the the data port.
- static uint32_t SDHC_ReadData (SDHC_Type ∗base)
  Retrieves the data from the data port.
- static void SDHC_EnableWakeupEvent (SDHC_Type ∗base, uint32_t mask, bool enable)
  Enables or disables a wakeup event in low-power mode.
- static void SDHC_EnableCardDetectTest (SDHC_Type ∗base, bool enable)
  Enables or disables the card detection level for testing.
- static void SDHC_SetCardDetectTestLevel (SDHC_Type ∗base, bool high)
  Sets the card detection test level.
- void SDHC_EnableSdioControl (SDHC_Type ∗base, uint32_t mask, bool enable)
  Enables or disables the SDIO card control.
- static void SDHC_SetContinueRequest (SDHC_Type ∗base)
  Restarts a transaction which has stopped at the block GAP for the SDIO card.
- void SDHC_SetMmcBootConfig (SDHC_Type ∗base, const sdhc_boot_config_t ∗config)
  Configures the MMC boot feature.
- static void SDHC_SetForceEvent (SDHC_Type ∗base, uint32_t mask)
  Forces generating events according to the given mask.

Transactional

- status_t SDHC_TransferBlocking (SDHC_Type ∗base, uint32_t ∗admaTable, uint32_t admaTableWords, sdhc_transfer_t ∗transfer)
  Transfers the command/data using a blocking method.
- void SDHC_TransferCreateHandle (SDHC_Type ∗base, sdhc_handle_t ∗handle, const sdhc_transfer_callback_t ∗callback, void ∗userData)
  Creates the SDHC handle.
- status_t SDHC_TransferNonBlocking (SDHC_Type ∗base, sdhc_handle_t ∗handle, uint32_t ∗admaTable, uint32_t admaTableWords, sdhc_transfer_t ∗transfer)
  Transfers the command/data using an interrupt and an asynchronous method.
- void SDHC_TransferHandleIRQ (SDHC_Type ∗base, sdhc_handle_t ∗handle)
  IRQ handler for the SDHC.

33.3 Data Structure Documentation

33.3.1 struct sdhc_adma2_descriptor_t

Data Fields

- uint32_t attribute
  The control and status field.
- const uint32_t ∗ address
  The address field.

33.3.2 struct sdhc_capability_t

Defines a structure to save the capability information of SDHC.

Data Fields

- uint32_t specVersion
  Specification version.
- uint32_t vendorVersion
  Vendor version.
- uint32_t maxBlockLength
  Maximum block length united as byte.
- uint32_t maxBlockCount
  Maximum block count can be set one time.
- uint32_t flags
  Capability flags to indicate the support information(_sdhc_capability_flag)

33.3.3 struct sdhc_transfer_config_t

Define structure to configure the transfer-related command index/argument/flags and data block size/data block numbers. This structure needs to be filled each time a command is sent to the card.

Data Fields

- size_t dataBlockSize
  Data block size.
- uint32_t dataBlockCount
  Data block count.
- uint32_t commandArgument
  Command argument.
- uint32_t commandIndex
  Command index.
- uint32_t flags
  Transfer flags(_sdhc_transfer_flag)

33.3.4 struct sdhc_boot_config_t

Data Fields

- uint32_t ackTimeoutCount
  Timeout value for the boot ACK.
- sdhc_boot_mode_t bootMode
  Boot mode selection.
- uint32_t blockCount

Stop at block gap value of automatic mode.
- bool enableBootAck
  Enable or disable boot ACK.
- bool enableBoot
  Enable or disable fast boot.
- bool enableAutoStopAtBlockGap
  Enable or disable auto stop at block gap function in boot period.

33.3.4.0.0.34 Field Documentation

33.3.4.0.0.34.1 uint32_t sdhc_boot_config_t::ackTimeoutCount

The available range is 0 ∼ 15.

33.3.4.0.0.34.2 sdhc_boot_mode_t sdhc_boot_config_t::bootMode

33.3.4.0.0.34.3 uint32_t sdhc_boot_config_t::blockCount

Available range is 0 ∼ 65535.

33.3.5 struct sdhc_config_t

Data Fields

- bool cardDetectDat3
  Enable DAT3 as card detection pin.
- sdhc_endian_mode_t endianMode
  Endian mode.
- sdhc_dma_mode_t dmaMode
  DMA mode.
- uint32_t readWatermarkLevel
  Watermark level for DMA read operation.
- uint32_t writeWatermarkLevel
  Watermark level for DMA write operation.

33.3.5.0.0.35 Field Documentation

33.3.5.0.0.35.1 uint32_t sdhc_config_t::readWatermarkLevel

Available range is 1 ∼ 128.

33.3.5.0.0.35.2 uint32_t sdhc_config_t::writeWatermarkLevel

Available range is 1 ∼ 128.

33.3.6 struct sdhc_data_t

Defines a structure to contain data-related attribute. 'enableIgnoreError' is used for the case that upper card driver want to ignore the error event to read/write all the data not to stop read/write immediately when error event happen for example bus testing procedure for MMC card.

Data Fields

- bool enableAutoCommand12
  Enable auto CMD12.
- bool enableIgnoreError
  Enable to ignore error event to read/write all the data.
- size_t blockSize
  Block size.
- uint32_t blockCount
  Block count.
- uint32_t ∗ rxData
  Buffer to save data read.
- const uint32_t ∗ txData
  Data buffer to write.

33.3.7 struct sdhc_command_t

Define card command-related attribute.

Data Fields

- uint32_t index
  Command index.
- uint32_t argument
  Command argument.
- sdhc_command_type_t type
  Command type.
- sdhc_response_type_t responseType
  Command response type.
- uint32_t response [4U]
  Response for this command.

33.3.8 struct sdhc_transfer_t

Data Fields

- sdhc_data_t ∗ data
  Data to transfer.

- sdhc_command_t ∗ command
  Command to send.

33.3.9 struct sdhc_transfer_callback_t

Data Fields

- void(∗ CardInserted)(void)
  Card inserted occurs when DAT3/CD pin is for card detect.
- void(∗ CardRemoved)(void)
  Card removed occurs.
- void(∗ SdioInterrupt)(void)
  SDIO card interrupt occurs.
- void(∗ SdioBlockGap)(void)
  SDIO card stopped at block gap occurs.
- void(∗ TransferComplete)(SDHC_Type ∗base, sdhc_handle_t ∗handle, status_t status, void ∗userData)
  Transfer complete callback.

33.3.10 struct _sdhc_handle

SDHC handle typedef.

Defines the structure to save the SDHC state information and callback function. The detailed interrupt status when sending a command or transfering data can be obtained from the interruptFlags field by using the mask defined in sdhc_interrupt_flag_t.

Note

All the fields except interruptFlags and transferredWords must be allocated by the user.

Data Fields

- sdhc_data_t ∗volatile data
  Data to transfer.
- sdhc_command_t ∗volatile command
  Command to send.
- volatile uint32_t interruptFlags
  Interrupt flags of last transaction.
- volatile uint32_t transferredWords
  Words transferred by DATAPORT way.
- sdhc_transfer_callback_t callback
  Callback function.
- void ∗ userData
  Parameter for transfer complete callback.

33.3.11 struct sdhc_host_t

Data Fields

- SDHC_Type ∗ base
  SDHC peripheral base address.
- uint32_t sourceClock_Hz
  SDHC source clock frequency united in Hz.
- sdhc_config_t config
  SDHC configuration.
- sdhc_capability_t capability
  SDHC capability information.
- sdhc_transfer_function_t transfer
  SDHC transfer function.

33.4 Macro Definition Documentation

33.4.1 #define FSL_SDHC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 1U))

33.5 Typedef Documentation

33.5.1 typedef uint32_t sdhc_adma1_descriptor_t

33.5.2 typedef status_t(∗ sdhc_transfer_function_t)(SDHC_Type ∗base, sdhc_transfer_t ∗content)

33.6 Enumeration Type Documentation

33.6.1 enum _sdhc_status

Enumerator

kStatus_SDHC_BusyTransferring Transfer is on-going.
kStatus_SDHC_PrepareAdmaDescriptorFailed Set DMA descriptor failed.
kStatus_SDHC_SendCommandFailed Send command failed.
kStatus_SDHC_TransferDataFailed Transfer data failed.

33.6.2 enum _sdhc_capability_flag

Enumerator

kSDHC_SupportAdmaFlag Support ADMA.
kSDHC_SupportHighSpeedFlag Support high-speed.
kSDHC_SupportDmaFlag Support DMA.
kSDHC_SupportSuspendResumeFlag Support suspend/resume.

kSDHC_SupportV330Flag Support voltage 3.3V.
kSDHC_Support4BitFlag Support 4 bit mode.
kSDHC_Support8BitFlag Support 8 bit mode.

33.6.3 enum _sdhc_wakeup_event

Enumerator

kSDHC_WakeupEventOnCardInt Wakeup on card interrupt.
kSDHC_WakeupEventOnCardInsert Wakeup on card insertion.
kSDHC_WakeupEventOnCardRemove Wakeup on card removal.
kSDHC_WakeupEventsAll All wakeup events.

33.6.4 enum _sdhc_reset

Enumerator

kSDHC_ResetAll Reset all except card detection.
kSDHC_ResetCommand Reset command line.
kSDHC_ResetData Reset data line.
kSDHC_ResetsAll All reset types.

33.6.5 enum _sdhc_transfer_flag

Enumerator

kSDHC_EnableDmaFlag Enable DMA.
kSDHC_CommandTypeSuspendFlag Suspend command.
kSDHC_CommandTypeResumeFlag Resume command.
kSDHC_CommandTypeAbortFlag Abort command.
kSDHC_EnableBlockCountFlag Enable block count.
kSDHC_EnableAutoCommand12Flag Enable auto CMD12.
kSDHC_DataReadFlag Enable data read.
kSDHC_MultipleBlockFlag Multiple block data read/write.
kSDHC_ResponseLength136Flag 136 bit response length
kSDHC_ResponseLength48Flag 48 bit response length
kSDHC_ResponseLength48BusyFlag 48 bit response length with busy status
kSDHC_EnableCrcCheckFlag Enable CRC check.
kSDHC_EnableIndexCheckFlag Enable index check.
kSDHC_DataPresentFlag Data present flag.

33.6.6 enum _sdhc_present_status_flag

Enumerator

kSDHC_CommandInhibitFlag Command inhibit.
kSDHC_DataInhibitFlag Data inhibit.
kSDHC_DataLineActiveFlag Data line active.
kSDHC_SdClockStableFlag SD bus clock stable.
kSDHC_WriteTransferActiveFlag Write transfer active.
kSDHC_ReadTransferActiveFlag Read transfer active.
kSDHC_BufferWriteEnableFlag Buffer write enable.
kSDHC_BufferReadEnableFlag Buffer read enable.
kSDHC_CardInsertedFlag Card inserted.
kSDHC_CommandLineLevelFlag Command line signal level.
kSDHC_Data0LineLevelFlag Data0 line signal level.
kSDHC_Data1LineLevelFlag Data1 line signal level.
kSDHC_Data2LineLevelFlag Data2 line signal level.
kSDHC_Data3LineLevelFlag Data3 line signal level.
kSDHC_Data4LineLevelFlag Data4 line signal level.
kSDHC_Data5LineLevelFlag Data5 line signal level.
kSDHC_Data6LineLevelFlag Data6 line signal level.
kSDHC_Data7LineLevelFlag Data7 line signal level.

33.6.7 enum _sdhc_interrupt_status_flag

Enumerator

kSDHC_CommandCompleteFlag Command complete.
kSDHC_DataCompleteFlag Data complete.
kSDHC_BlockGapEventFlag Block gap event.
kSDHC_DmaCompleteFlag DMA interrupt.
kSDHC_BufferWriteReadyFlag Buffer write ready.
kSDHC_BufferReadReadyFlag Buffer read ready.
kSDHC_CardInsertionFlag Card inserted.
kSDHC_CardRemovalFlag Card removed.
kSDHC_CardInterruptFlag Card interrupt.
kSDHC_CommandTimeoutFlag Command timeout error.
kSDHC_CommandCrcErrorFlag Command CRC error.
kSDHC_CommandEndBitErrorFlag Command end bit error.
kSDHC_CommandIndexErrorFlag Command index error.
kSDHC_DataTimeoutFlag Data timeout error.
kSDHC_DataCrcErrorFlag Data CRC error.
kSDHC_DataEndBitErrorFlag Data end bit error.
kSDHC_AutoCommand12ErrorFlag Auto CMD12 error.

kSDHC_DmaErrorFlag DMA error.
kSDHC_CommandErrorFlag Command error.
kSDHC_DataErrorFlag Data error.
kSDHC_ErrorFlag All error.
kSDHC_DataFlag Data interrupts.
kSDHC_CommandFlag Command interrupts.
kSDHC_CardDetectFlag Card detection interrupts.
kSDHC_AllInterruptFlags All flags mask.

33.6.8 enum _sdhc_auto_command12_error_status_flag

Enumerator

kSDHC_AutoCommand12NotExecutedFlag Not executed error.
kSDHC_AutoCommand12TimeoutFlag Timeout error.
kSDHC_AutoCommand12EndBitErrorFlag End bit error.
kSDHC_AutoCommand12CrcErrorFlag CRC error.
kSDHC_AutoCommand12IndexErrorFlag Index error.
kSDHC_AutoCommand12NotIssuedFlag Not issued error.

33.6.9 enum _sdhc_adma_error_status_flag

Enumerator

kSDHC_AdmaLenghMismatchFlag Length mismatch error.
kSDHC_AdmaDescriptorErrorFlag Descriptor error.

33.6.10 enum sdhc_adma_error_state_t

This state is the detail state when ADMA error has occurred.

Enumerator

kSDHC_AdmaErrorStateStopDma Stop DMA.
kSDHC_AdmaErrorStateFetchDescriptor Fetch descriptor.
kSDHC_AdmaErrorStateChangeAddress Change address.
kSDHC_AdmaErrorStateTransferData Transfer data.

33.6.11 enum _sdhc_force_event

Enumerator

kSDHC_ForceEventAutoCommand12NotExecuted Auto CMD12 not executed error.
kSDHC_ForceEventAutoCommand12Timeout Auto CMD12 timeout error.
kSDHC_ForceEventAutoCommand12CrcError Auto CMD12 CRC error.
kSDHC_ForceEventEndBitError Auto CMD12 end bit error.
kSDHC_ForceEventAutoCommand12IndexError Auto CMD12 index error.
kSDHC_ForceEventAutoCommand12NotIssued Auto CMD12 not issued error.
kSDHC_ForceEventCommandTimeout Command timeout error.
kSDHC_ForceEventCommandCrcError Command CRC error.
kSDHC_ForceEventCommandEndBitError Command end bit error.
kSDHC_ForceEventCommandIndexError Command index error.
kSDHC_ForceEventDataTimeout Data timeout error.
kSDHC_ForceEventDataCrcError Data CRC error.
kSDHC_ForceEventDataEndBitError Data end bit error.
kSDHC_ForceEventAutoCommand12Error Auto CMD12 error.
kSDHC_ForceEventCardInt Card interrupt.
kSDHC_ForceEventDmaError Dma error.
kSDHC_ForceEventsAll All force event flags mask.

33.6.12 enum sdhc_data_bus_width_t

Enumerator

kSDHC_DataBusWidth1Bit 1-bit mode
kSDHC_DataBusWidth4Bit 4-bit mode
kSDHC_DataBusWidth8Bit 8-bit mode

33.6.13 enum sdhc_endian_mode_t

Enumerator

kSDHC_EndianModeBig Big endian mode.
kSDHC_EndianModeHalfWordBig Half word big endian mode.
kSDHC_EndianModeLittle Little endian mode.

33.6.14 enum sdhc_dma_mode_t

Enumerator

kSDHC_DmaModeNo No DMA.

kSDHC_DmaModeAdma1 ADMA1 is selected.
kSDHC_DmaModeAdma2 ADMA2 is selected.

33.6.15 enum _sdhc_sdio_control_flag

Enumerator

kSDHC_StopAtBlockGapFlag Stop at block gap.
kSDHC_ReadWaitControlFlag Read wait control.
kSDHC_InterruptAtBlockGapFlag Interrupt at block gap.
kSDHC_ExactBlockNumberReadFlag Exact block number read.

33.6.16 enum sdhc_boot_mode_t

Enumerator

kSDHC_BootModeNormal Normal boot.
kSDHC_BootModeAlternative Alternative boot.

33.6.17 enum sdhc_command_type_t

Enumerator

kSDHC_CommandTypeNormal Normal command.
kSDHC_CommandTypeSuspend Suspend command.
kSDHC_CommandTypeResume Resume command.
kSDHC_CommandTypeAbort Abort command.

33.6.18 enum sdhc_response_type_t

Define the command response type from card to host controller.

Enumerator

kSDHC_ResponseTypeNone Response type: none.
kSDHC_ResponseTypeR1 Response type: R1.
kSDHC_ResponseTypeR1b Response type: R1b.
kSDHC_ResponseTypeR2 Response type: R2.
kSDHC_ResponseTypeR3 Response type: R3.
kSDHC_ResponseTypeR4 Response type: R4.

kSDHC_ResponseTypeR5 Response type: R5.
kSDHC_ResponseTypeR5b Response type: R5b.
kSDHC_ResponseTypeR6 Response type: R6.
kSDHC_ResponseTypeR7 Response type: R7.

33.6.19 enum _sdhc_adma1_descriptor_flag

Enumerator

kSDHC_Adma1DescriptorValidFlag Valid flag.
kSDHC_Adma1DescriptorEndFlag End flag.
kSDHC_Adma1DescriptorInterrupFlag Interrupt flag.
kSDHC_Adma1DescriptorActivity1Flag Activity 1 flag.
kSDHC_Adma1DescriptorActivity2Flag Activity 2 flag.
kSDHC_Adma1DescriptorTypeNop No operation.
kSDHC_Adma1DescriptorTypeTransfer Transfer data.
kSDHC_Adma1DescriptorTypeLink Link descriptor.
kSDHC_Adma1DescriptorTypeSetLength Set data length.

33.6.20 enum _sdhc_adma2_descriptor_flag

Enumerator

kSDHC_Adma2DescriptorValidFlag Valid flag.
kSDHC_Adma2DescriptorEndFlag End flag.
kSDHC_Adma2DescriptorInterruptFlag Interrupt flag.
kSDHC_Adma2DescriptorActivity1Flag Activity 1 mask.
kSDHC_Adma2DescriptorActivity2Flag Activity 2 mask.
kSDHC_Adma2DescriptorTypeNop No operation.
kSDHC_Adma2DescriptorTypeReserved Reserved.
kSDHC_Adma2DescriptorTypeTransfer Transfer type.
kSDHC_Adma2DescriptorTypeLink Link type.

33.7 Function Documentation

33.7.1 void SDHC_Init (SDHC_Type ∗ *base*, const sdhc_config_t ∗ *config*)

Configures the SDHC according to the user configuration.

Example:

```
sdhc_config_t config;
config.cardDetectDat3 = false;
config.endianMode = kSDHC_EndianModeLittle;
```

```
config.dmaMode = kSDHC_DmaModeAdma2;
config.readWatermarkLevel = 128U;
config.writeWatermarkLevel = 128U;
SDHC_Init(SDHC, &config);
```

Parameters

base	SDHC peripheral base address.
config	SDHC configuration information.

Return values

kStatus_Success	Operate successfully.

33.7.2 void SDHC_Deinit (SDHC_Type ∗ *base*)

Parameters

base	SDHC peripheral base address.

33.7.3 bool SDHC_Reset (SDHC_Type ∗ *base*, uint32_t *mask*, uint32_t *timeout*)

Parameters

base	SDHC peripheral base address.
mask	The reset type mask(_sdhc_reset).
timeout	Timeout for reset.

Return values

true	Reset successfully.
false	Reset failed.

33.7.4 status_t SDHC_SetAdmaTableConfig (SDHC_Type ∗ *base*, sdhc_dma_mode_t *dmaMode*, uint32_t ∗ *table*, uint32_t *tableWords*, const uint32_t ∗ *data*, uint32_t *dataBytes*)

Parameters

base	SDHC peripheral base address.
dmaMode	DMA mode.
table	ADMA table address.
tableWords	ADMA table buffer length united as Words.
data	Data buffer address.
dataBytes	Data length united as bytes.

Return values

kStatus_OutOfRange	ADMA descriptor table length isn't enough to describe data.
kStatus_Success	Operate successfully.

33.7.5 static void SDHC_EnableInterruptStatus (SDHC_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	Interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.6 static void SDHC_DisableInterruptStatus (SDHC_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	The interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.7 static void SDHC_EnableInterruptSignal (SDHC_Type ∗ *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	The interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.8 static void SDHC_DisableInterruptSignal (SDHC_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	The interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.9 static uint32_t SDHC_GetInterruptStatusFlags (SDHC_Type * *base*) [inline], [static]

Parameters

base	SDHC peripheral base address.

Returns

Current interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.10 static void SDHC_ClearInterruptStatusFlags (SDHC_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	The interrupt status flags mask(_sdhc_interrupt_status_flag).

33.7.11 static uint32_t SDHC_GetAutoCommand12ErrorStatusFlags (SDHC_Type * *base*) [inline], [static]

Parameters

base	SDHC peripheral base address.

Returns

Auto command 12 error status flags mask(_sdhc_auto_command12_error_status_flag).

33.7.12 static uint32_t SDHC_GetAdmaErrorStatusFlags (SDHC_Type * *base*) [inline], [static]

Parameters

base	SDHC peripheral base address.

Returns

ADMA error status flags mask(_sdhc_adma_error_status_flag).

33.7.13 static uint32_t SDHC_GetPresentStatusFlags (SDHC_Type * *base*) [inline], [static]

This function gets the present SDHC's status except for an interrupt status and an error status.

Parameters

base	SDHC peripheral base address.

Returns

Present SDHC's status flags mask(_sdhc_present_status_flag).

33.7.14 void SDHC_GetCapability (SDHC_Type * *base*, sdhc_capability_t * *capability*)

Parameters

base	SDHC peripheral base address.
capability	Structure to save capability information.

33.7.15 static void SDHC_EnableSdClock (SDHC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SDHC peripheral base address.
enable	True to enable, false to disable.

33.7.16 uint32_t SDHC_SetSdClock (SDHC_Type * *base*, uint32_t *srcClock_Hz*, uint32_t *busClock_Hz*)

Parameters

base	SDHC peripheral base address.
srcClock_Hz	SDHC source clock frequency united in Hz.
busClock_Hz	SD bus clock frequency united in Hz.

Returns

The nearest frequency of busClock_Hz configured to SD bus.

33.7.17 bool SDHC_SetCardActive (SDHC_Type * *base*, uint32_t *timeout*)

This function must be called each time the card is inserted to ensure that the card can receive the command correctly.

Parameters

base	SDHC peripheral base address.
timeout	Timeout to initialize card.

Return values

true	Set card active successfully.
false	Set card active failed.

33.7.18 static void SDHC_SetDataBusWidth (SDHC_Type ∗ *base*, sdhc_data_bus_width_t *width*) [inline], [static]

Parameters

base	SDHC peripheral base address.
width	Data transfer width.

33.7.19 void SDHC_SetTransferConfig (SDHC_Type ∗ *base*, const sdhc_transfer_config_t ∗ *config*)

This function fills the card transfer-related command argument/transfer flag/data size. The command and data are sent by SDHC after calling this function.

Example:

```
sdhc_transfer_config_t transferConfig;
transferConfig.dataBlockSize = 512U;
transferConfig.dataBlockCount = 2U;
transferConfig.commandArgument = 0x01AAU;
transferConfig.commandIndex = 8U;
transferConfig.flags |= (kSDHC_EnableDmaFlag |
      kSDHC_EnableAutoCommand12Flag |
      kSDHC_MultipleBlockFlag);
SDHC_SetTransferConfig(SDHC, &transferConfig);
```

Parameters

base	SDHC peripheral base address.
config	Command configuration structure.

33.7.20 static uint32_t SDHC_GetCommandResponse (SDHC_Type ∗ *base*, uint32_t *index*) [inline], [static]

Parameters

base	SDHC peripheral base address.
index	The index of response register, range from 0 to 3.

Returns

Response register transfer.

33.7.21 static void SDHC_WriteData (SDHC_Type * *base*, uint32_t *data*) [inline], [static]

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

base	SDHC peripheral base address.
data	The data about to be sent.

33.7.22 static uint32_t SDHC_ReadData (SDHC_Type * *base*) [inline], [static]

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

base	SDHC peripheral base address.

Returns

The data has been read.

33.7.23 static void SDHC_EnableWakeupEvent (SDHC_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	Wakeup events mask(_sdhc_wakeup_event).
enable	True to enable, false to disable.

33.7.24 static void SDHC_EnableCardDetectTest (SDHC_Type ∗ *base*, bool *enable*) [inline], [static]

Parameters

base	SDHC peripheral base address.
enable	True to enable, false to disable.

33.7.25 static void SDHC_SetCardDetectTestLevel (SDHC_Type ∗ *base*, bool *high*) [inline], [static]

This function sets the card detection test level to indicate whether the card is inserted into the SDHC when DAT[3]/ CD pin is selected as a card detection pin. This function can also assert the pin logic when DAT[3]/CD pin is selected as the card detection pin.

Parameters

base	SDHC peripheral base address.
high	True to set the card detect level to high.

33.7.26 void SDHC_EnableSdioControl (SDHC_Type ∗ *base*, uint32_t *mask*, bool *enable*)

Parameters

base	SDHC peripheral base address.
mask	SDIO card control flags mask(_sdhc_sdio_control_flag).
enable	True to enable, false to disable.

33.7.27 static void SDHC_SetContinueRequest (SDHC_Type * base) [inline], [static]

Parameters

base	SDHC peripheral base address.

33.7.28 void SDHC_SetMmcBootConfig (SDHC_Type * *base*, const sdhc_boot_config_t * *config*)

Example:

```
sdhc_boot_config_t config;
config.ackTimeoutCount = 4;
config.bootMode = kSDHC_BootModeNormal;
config.blockCount = 5;
config.enableBootAck = true;
config.enableBoot = true;
config.enableAutoStopAtBlockGap = true;
SDHC_SetMmcBootConfig(SDHC, &config);
```

Parameters

base	SDHC peripheral base address.
config	The MMC boot configuration information.

33.7.29 static void SDHC_SetForceEvent (SDHC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SDHC peripheral base address.
mask	The force events mask(_sdhc_force_event).

33.7.30 status_t SDHC_TransferBlocking (SDHC_Type * *base*, uint32_t * *admaTable*, uint32_t *admaTableWords*, sdhc_transfer_t * *transfer*)

This function waits until the command response/data is received or the SDHC encounters an error by polling the status flag. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

Note

There is no need to call the API 'SDHC_TransferCreateHandle' when calling this API.

Parameters

base	SDHC peripheral base address.
admaTable	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
admaTable-Words	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
transfer	Transfer content.

Return values

kStatus_InvalidArgument	Argument is invalid.
kStatus_SDHC_Prepare-AdmaDescriptorFailed	Prepare ADMA descriptor failed.
kStatus_SDHC_Send-CommandFailed	Send command failed.
kStatus_SDHC_Transfer-DataFailed	Transfer data failed.
kStatus_Success	Operate successfully.

33.7.31 void SDHC_TransferCreateHandle (SDHC_Type ∗ *base*, sdhc_handle_t ∗ *handle*, const sdhc_transfer_callback_t ∗ *callback*, void ∗ *userData*)

Parameters

base	SDHC peripheral base address.
handle	SDHC handle pointer.
callback	Structure pointer to contain all callback functions.
userData	Callback function parameter.

33.7.32 status_t SDHC_TransferNonBlocking (SDHC_Type ∗ *base*, sdhc_handle_t ∗ *handle*, uint32_t ∗ *admaTable*, uint32_t *admaTableWords*, sdhc_transfer_t ∗ *transfer*)

This function sends a command and data and returns immediately. It doesn't wait the transfer complete or encounter an error. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

Note

Call the API 'SDHC_TransferCreateHandle' when calling this API.

Parameters

base	SDHC peripheral base address.
handle	SDHC handle.
admaTable	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
admaTable-Words	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
transfer	Transfer content.

Return values

kStatus_InvalidArgument	Argument is invalid.
kStatus_SDHC_Busy-Transferring	Busy transferring.
kStatus_SDHC_Prepare-AdmaDescriptorFailed	Prepare ADMA descriptor failed.
kStatus_Success	Operate successfully.

33.7.33 void SDHC_TransferHandleIRQ (SDHC_Type ∗ *base,* sdhc_handle_t ∗ *handle*)

This function deals with the IRQs on the given host controller.

Parameters

base	SDHC peripheral base address.
handle	SDHC handle.

Chapter 34
SDRAMC: Synchronous DRAM Controller Driver

34.1 Overview

The KSDK provides a peripheral driver for the Synchronous DRAM Controller block of Kinetis devices.

The SDRAM controller commands include the initialization MRS command, precharge command, enter/exit self-refresh command, and enable/disable auto-refresh command. Use the SDRAMC_SendCommand() to send these commands to SDRAM to initialize it. The SDRAMC_EnableWriteProtect() is provided to enable/disable the write protection. The SDRAMC_EnableOperateValid() is provided to enable/disable the operation valid.

34.2 Typical use case

This example shows how to use the SDRAM Controller driver to initialize the external 16 bit port-size 8-column SDRAM chip. Initialize the SDRAM controller and run the initialization sequence. The external SDRAM is initialized and the SDRAM read and write is available.

First, initialize the SDRAM Controller.

```
sdramc_config_t config;
uint32_t clockSrc;

// SDRAM refresh timing configuration.
clockSrc = CLOCK_GetFreq(kCLOCK_BusClk);
sdramc_refresh_config_t refConfig =
{
   kSDRAMC_RefreshThreeClocks,
   15625,    // SDRAM: 4096 rows/ 64ms.
   clockSrc,
};
// SDRAM controller configuration.
sdramc_blockctl_config_t ctlConfig =
{
    kSDRAMC_Block0,
    kSDRAMC_PortSize16Bit,
    kSDRAMC_Commandbit19,
    kSDRAMC_LatencyOne,
    SDRAM_START_ADDRESS,
    0x7c0000,
};

config.refreshConfig = &refConfig;
config.blockConfig = &ctlConfig;
config.numBlockConfig = 1;

// SDRAM controller initialization.
SDRAMC_Init(base, &config);
```

Then, run the initialization sequence.

```
// Issues a PALL command.
```

```
SDRAMC_SendCommand(base, whichBlock, kSDRAMC_PrechargeCommand);

// Accesses an SDRAM location.
*(uint8_t *)(SDRAM_START_ADDRESS) = SDRAM_COMMAND_ACCESSVALUE;

// Enables the refresh.
SDRAMC_SendCommand(base, whichBlock,
      kSDRAMC_AutoRefreshEnableCommand);

// Waits for 8 refresh cycles less than one microsecond.
delay;

// Issues the MSR command.
SDRAMC_SendCommand(base, whichBlock, kSDRAMC_ImrsCommand);

// Puts the correct value on the SDRAM address bus for the SDRAM mode register.
addr = ....;

// Set MRS register.
mrsAddr = (uint8_t *)(SDRAM_START_ADDRESS + addr);
*mrsAddr = SDRAM_COMMAND_ACCESSVALUE;
```

Data Structures

- struct sdramc_blockctl_config_t
  SDRAM controller block control configuration structure. More...
- struct sdramc_refresh_config_t
  SDRAM controller refresh timing configuration structure. More...
- struct sdramc_config_t
  SDRAM controller configuration structure. More...

Enumerations

- enum sdramc_refresh_time_t {
  kSDRAMC_RefreshThreeClocks = 0x0U,
  kSDRAMC_RefreshSixClocks,
  kSDRAMC_RefreshNineClocks }
  SDRAM controller auto-refresh timing.
- enum sdramc_latency_t {
  kSDRAMC_LatencyZero = 0x0U,
  kSDRAMC_LatencyOne,
  kSDRAMC_LatencyTwo,
  kSDRAMC_LatencyThree }
  Setting latency for SDRAM controller timing specifications.
- enum sdramc_command_bit_location_t {
  kSDRAMC_Commandbit17 = 0x0U,
  kSDRAMC_Commandbit18,
  kSDRAMC_Commandbit19,
  kSDRAMC_Commandbit20,
  kSDRAMC_Commandbit21,
  kSDRAMC_Commandbit22,
  kSDRAMC_Commandbit23,
  kSDRAMC_Commandbit24 }
  SDRAM controller command bit location.

- enum sdramc_command_t {
  kSDRAMC_ImrsCommand = 0x0U,
  kSDRAMC_PrechargeCommand,
  kSDRAMC_SelfrefreshEnterCommand,
  kSDRAMC_SelfrefreshExitCommand,
  kSDRAMC_AutoRefreshEnableCommand,
  kSDRAMC_AutoRefreshDisableCommand }
  SDRAM controller command.
- enum sdramc_port_size_t {
  kSDRAMC_PortSize32Bit = 0x0U,
  kSDRAMC_PortSize8Bit,
  kSDRAMC_PortSize16Bit }
  SDRAM port size.
- enum sdramc_block_selection_t {
  kSDRAMC_Block0 = 0x0U,
  kSDRAMC_Block1 }
  SDRAM controller block selection.

Driver version

- #define FSL_SDRAMC_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  SDRAMC driver version 2.1.0.

SDRAM Controller Initialization and De-initialization

- void SDRAMC_Init (SDRAM_Type ∗base, sdramc_config_t ∗configure)
  Initializes the SDRAM controller.
- void SDRAMC_Deinit (SDRAM_Type ∗base)
  Deinitializes the SDRAM controller module and gates the clock.

SDRAM Controller Basic Operation

- void SDRAMC_SendCommand (SDRAM_Type ∗base, sdramc_block_selection_t block, sdramc_command_t command)
  Sends the SDRAM command.
- static void SDRAMC_EnableWriteProtect (SDRAM_Type ∗base, sdramc_block_selection_t block, bool enable)
  Enables/disables the write protection.
- static void SDRAMC_EnableOperateValid (SDRAM_Type ∗base, sdramc_block_selection_t block, bool enable)
  Enables/disables the valid operation.

34.3 Data Structure Documentation

34.3.1 struct sdramc_blockctl_config_t

Data Fields

- sdramc_block_selection_t block
  The block number.
- sdramc_port_size_t portSize
  The port size of the associated SDRAM block.
- sdramc_command_bit_location_t location
  The command bit location.
- sdramc_latency_t latency
  The latency for some timing specifications.
- uint32_t address
  The base address of the SDRAM block.
- uint32_t addressMask
  The base address mask of the SDRAM block.

34.3.1.0.0.36 Field Documentation

34.3.1.0.0.36.1 sdramc_block_selection_t sdramc_blockctl_config_t::block

34.3.1.0.0.36.2 sdramc_port_size_t sdramc_blockctl_config_t::portSize

34.3.1.0.0.36.3 sdramc_command_bit_location_t sdramc_blockctl_config_t::location

34.3.1.0.0.36.4 sdramc_latency_t sdramc_blockctl_config_t::latency

34.3.1.0.0.36.5 uint32_t sdramc_blockctl_config_t::address

34.3.1.0.0.36.6 uint32_t sdramc_blockctl_config_t::addressMask

34.3.2 struct sdramc_refresh_config_t

Data Fields

- sdramc_refresh_time_t refreshTime
  Trc:The number of bus clocks inserted between a REF and next ACTIVE command.
- uint32_t sdramRefreshRow
  The SDRAM refresh time each row: ns/row.
- uint32_t busClock_Hz
  The bus clock for SDRAMC.

34.3.2.0.0.37 Field Documentation

34.3.2.0.0.37.1 sdramc_refresh_time_t sdramc_refresh_config_t::refreshTime

34.3.2.0.0.37.2 uint32_t sdramc_refresh_config_t::sdramRefreshRow

34.3.2.0.0.37.3 uint32_t sdramc_refresh_config_t::busClock_Hz

34.3.3 struct sdramc_config_t

Defines a configure structure and uses the SDRAMC_Configure() function to make necessary initializations.

Data Fields

- sdramc_refresh_config_t ∗ refreshConfig
  Refresh timing configure structure pointer.
- sdramc_blockctl_config_t ∗ blockConfig
  Block configure structure pointer.
- uint8_t numBlockConfig
  SDRAM block numbers for configuration.

34.3.3.0.0.38 Field Documentation

34.3.3.0.0.38.1 sdramc_refresh_config_t∗ sdramc_config_t::refreshConfig

34.3.3.0.0.38.2 sdramc_blockctl_config_t∗ sdramc_config_t::blockConfig

If both SDRAM blocks are used, use the two continuous blockConfig.

34.3.3.0.0.38.3 uint8_t sdramc_config_t::numBlockConfig

34.4 Macro Definition Documentation

34.4.1 #define FSL_SDRAMC_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

34.5 Enumeration Type Documentation

34.5.1 enum sdramc_refresh_time_t

Enumerator

kSDRAMC_RefreshThreeClocks The refresh timing with three bus clocks.
kSDRAMC_RefreshSixClocks The refresh timing with six bus clocks.
kSDRAMC_RefreshNineClocks The refresh timing with nine bus clocks.

34.5.2 enum sdramc_latency_t

The latency setting affects the following SDRAM timing specifications:

- trcd: SRAS assertion to SCAS assertion
- tcasl: SCAS assertion to data out
- tras: ACTV command to Precharge command
- trp: Precharge command to ACTV command
- trwl, trdl: Last data input to Precharge command
- tep: Last data out to Precharge command
  The details of the latency setting and timing specifications are shown in the following table list.
  latency trcd: tcasl tras trp trwl,trdl tep
  0 1 bus clock 1 bus clock 2 bus clocks 1 bus clock 1 bus clock 1 bus clock
  1 2 bus clock 2 bus clock 4 bus clocks 2 bus clock 1 bus clock 1 bus clock
  2 3 bus clock 3 bus clock 6 bus clocks 3 bus clock 1 bus clock 1 bus clock
  3 3 bus clock 3 bus clock 6 bus clocks 3 bus clock 1 bus clock 1 bus clock

Enumerator

kSDRAMC_LatencyZero Latency 0.
kSDRAMC_LatencyOne Latency 1.
kSDRAMC_LatencyTwo Latency 2.
kSDRAMC_LatencyThree Latency 3.

34.5.3 enum sdramc_command_bit_location_t

Enumerator

kSDRAMC_Commandbit17 Command bit location is bit 17.
kSDRAMC_Commandbit18 Command bit location is bit 18.
kSDRAMC_Commandbit19 Command bit location is bit 19.
kSDRAMC_Commandbit20 Command bit location is bit 20.
kSDRAMC_Commandbit21 Command bit location is bit 21.
kSDRAMC_Commandbit22 Command bit location is bit 22.
kSDRAMC_Commandbit23 Command bit location is bit 23.
kSDRAMC_Commandbit24 Command bit location is bit 24.

34.5.4 enum sdramc_command_t

Enumerator

kSDRAMC_ImrsCommand Initiate MRS command.
kSDRAMC_PrechargeCommand Initiate precharge command.
kSDRAMC_SelfrefreshEnterCommand Enter self-refresh command.

kSDRAMC_SelfrefreshExitCommand Exit self-refresh command.
kSDRAMC_AutoRefreshEnableCommand Enable Auto refresh command.
kSDRAMC_AutoRefreshDisableCommand Disable Auto refresh command.

34.5.5 enum sdramc_port_size_t

Enumerator

kSDRAMC_PortSize32Bit 32-Bit port size.
kSDRAMC_PortSize8Bit 8-Bit port size.
kSDRAMC_PortSize16Bit 16-Bit port size.

34.5.6 enum sdramc_block_selection_t

Enumerator

kSDRAMC_Block0 Select SDRAM block 0.
kSDRAMC_Block1 Select SDRAM block 1.

34.6 Function Documentation

34.6.1 void SDRAMC_Init (SDRAM_Type ∗ *base*, sdramc_config_t ∗ *configure*)

This function ungates the SDRAM controller clock and initializes the SDRAM controller. This function must be called before calling any other SDRAM controller driver functions. Example

```
sdramc_refresh_config_t refreshConfig;
sdramc_blockctl_config_t blockConfig;
sdramc_config_t config;

refreshConfig.refreshTime  = kSDRAM_RefreshThreeClocks;
refreshConfig.sdramRefreshRow = 15625;
refreshConfig.busClock = 60000000;

blockConfig.block = kSDRAMC_Block0;
blockConfig.portSize = kSDRAMC_PortSize16Bit;
blockConfig.location = kSDRAMC_Commandbit19;
blockConfig.latency = kSDRAMC_RefreshThreeClocks;
blockConfig.address = SDRAM_START_ADDRESS;
blockConfig.addressMask = 0x7c0000;

config.refreshConfig = &refreshConfig,
config.blockConfig = &blockConfig,
config.totalBlocks = 1;

SDRAMC_Init(SDRAM, &config);
```

Parameters

base	SDRAM controller peripheral base address.
configure	The SDRAM configuration structure pointer.

34.6.2 void SDRAMC_Deinit (SDRAM_Type * *base*)

This function gates the SDRAM controller clock. As a result, the SDRAM controller module doesn't work after calling this function.

Parameters

base	SDRAM controller peripheral base address.

34.6.3 void SDRAMC_SendCommand (SDRAM_Type * *base*, sdramc_block_selection_t *block*, sdramc_command_t *command*)

This function sends commands to SDRAM. The commands are precharge command, initialization MR-S command, auto-refresh enable/disable command, and self-refresh enter/exit commands. Note that the self-refresh enter/exit commands are all blocks setting and "block" is ignored. Ensure to set the correct "block" when send other commands.

Parameters

base	SDRAM controller peripheral base address.
block	The block selection.
command	The SDRAM command, see "sdramc_command_t". kSDRAMC_ImrsCommand - Initialize MRS command kSDRAMC_PrechargeCommand - Initialize precharge command kSDRAMC_SelfrefreshEnterCommand - Enter self-refresh command kSDRAMC_SelfrefreshExitCommand - Exit self-refresh command kSDRAMC_AutoRefreshEnableCommand - Enable auto refresh command kSDRAMC_AutoRefreshDisableCommand - Disable auto refresh command

34.6.4 static void SDRAMC_EnableWriteProtect (SDRAM_Type * *base*, sdramc_block_selection_t *block*, bool *enable*) [inline], [static]

Parameters

base	SDRAM peripheral base address.
block	The block which is selected.
enable	True enable write protection, false disable write protection.

34.6.5 static void SDRAMC_EnableOperateValid (SDRAM_Type * *base*, sdramc_block_selection_t *block*, bool *enable*) [inline], [static]

Parameters

base	SDRAM peripheral base address.
block	The block which is selected.
enable	True enable the valid operation; false disable the valid operation.

Chapter 35
SIM: System Integration Module Driver

35.1 Overview

The KSDK provides a peripheral driver for the System Integration Module (SIM) of Kinetis devices.

Data Structures

- struct sim_uid_t
  Unique ID. More...

Enumerations

- enum _sim_usb_volt_reg_enable_mode {
  kSIM_UsbVoltRegEnable = SIM_SOPT1_USBREGEN_MASK,
  kSIM_UsbVoltRegEnableInLowPower = SIM_SOPT1_USBVSTBY_MASK,
  kSIM_UsbVoltRegEnableInStop = SIM_SOPT1_USBSSTBY_MASK,
  kSIM_UsbVoltRegEnableInAllModes }
  USB voltage regulator enable setting.
- enum _sim_flash_mode {
  kSIM_FlashDisableInWait = SIM_FCFG1_FLASHDOZE_MASK,
  kSIM_FlashDisable = SIM_FCFG1_FLASHDIS_MASK }
  Flash enable mode.

Functions

- void SIM_SetUsbVoltRegulatorEnableMode (uint32_t mask)
  Sets the USB voltage regulator setting.
- void SIM_GetUniqueId (sim_uid_t *uid)
  Gets the unique identification register value.
- static void SIM_SetFlashMode (uint8_t mode)
  Sets the flash enable mode.

Driver version

- #define FSL_SIM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Driver version 2.0.0.

35.2 Data Structure Documentation

35.2.1 struct sim_uid_t

Data Fields

- uint32_t MH

UIDMH.
- uint32_t ML
  UIDML.
- uint32_t L
  UIDL.

35.2.1.0.0.39 Field Documentation

35.2.1.0.0.39.1 uint32_t sim_uid_t::MH

35.2.1.0.0.39.2 uint32_t sim_uid_t::ML

35.2.1.0.0.39.3 uint32_t sim_uid_t::L

35.3 Enumeration Type Documentation

35.3.1 enum _sim_usb_volt_reg_enable_mode

Enumerator

kSIM_UsbVoltRegEnable Enable voltage regulator.
kSIM_UsbVoltRegEnableInLowPower Enable voltage regulator in VLPR/VLPW modes.
kSIM_UsbVoltRegEnableInStop Enable voltage regulator in STOP/VLPS/LLS/VLLS modes.
kSIM_UsbVoltRegEnableInAllModes Enable voltage regulator in all power modes.

35.3.2 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.
kSIM_FlashDisable Disable flash in normal mode.

35.4 Function Documentation

35.4.1 void SIM_SetUsbVoltRegulatorEnableMode (uint32_t *mask*)

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP/-VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of _sim_usb_volt_reg_enable_mode. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable | kSIM_UsbVoltRegEnableInLow-Power);

Parameters

mask	USB voltage regulator enable setting.

35.4.2 void SIM_GetUniqueId (sim_uid_t * *uid*)

Parameters

uid	Pointer to the structure to save the UID value.

35.4.3 static void SIM_SetFlashMode (uint8_t *mode*) [inline], [static]

Parameters

mode	The mode to set; see _sim_flash_mode for mode details.

Chapter 36
Smart Card

36.1 Overview

The Kinetis SDK provides Peripheral drivers for the UART-ISO7816 and EMVSIM modules of Kinetis devices.

Smart Card driver provides the necessary functions to access and control integrated circuit cards. The driver controls communication modules (UART/EMVSIM) and handles special ICC sequences, such as the activation/deactivation (using EMVSIM IP or external interface chip). The Smart Card driver consists of two IPs (SmartCard_Uart and SmartCard_EmvSim drivers) and three PHY drivers (smartcard_phy-_emvsim, smartcard_phy_tda8035, and smartcard_phy_gpio drivers). These drivers can be combined, which means that the Smart Card driver wraps one IP (transmission) and one PHY (interface) driver.

The driver provides asynchronous functions to communicate with the Integrated Circuit Card (ICC). The driver contains RTOS adaptation layers which use semaphores as synchronization objects of synchronous transfers. The RTOS driver support also provides protection for multithreading.

36.2 SmartCard Driver Initialization

The Smart Card Driver is initialized by calling the SMARTCARD_Init() and SMARTCARD_PHY_Init() functions. The Smart Card Driver initialization configuration structure requires these settings:

- Smart Card voltage class
- Smart Card Interface options such as the RST, IRQ, CLK pins, and so on.

The driver also supports user callbacks for assertion/de-assertion Smart Card events and transfer finish event. This feature is useful to detect the card presence or for handling transfer events i.e., in RTOS. The user should initialize the Smart Card driver, which consist of IP and PHY drivers.

36.3 SmartCard Call diagram

Because the call diagram is complex, the detailed use of the Smart Card driver is not described in this part. For details about using the Smart Card driver, see the Smart Card driver example which describes a simple use case.

PHY driver

The Smart Card interface driver is initialized by calling the function SMARTCARD_PHY_Init(). During the initialization phase, Smart Card clock is configured and all hardware pins for IC handling are configured.

Modules

- Smart Card EMVSIM Driver

- Smart Card FreeRTOS Driver
- Smart Card PHY EMVSIM Driver
- Smart Card PHY GPIO Driver
- Smart Card PHY TDA8035 Driver
- Smart Card UART Driver
- Smart Card µCOS/II Driver
- Smart Card µCOS/III Driver

Data Structures

- struct smartcard_card_params_t
  Defines card-specific parameters for Smart card driver. More...
- struct smartcard_timers_state_t
  Smart card defines the state of the EMV timers in the Smart card driver. More...
- struct smartcard_interface_config_t
  Defines user specified configuration of Smart card interface. More...
- struct smartcard_xfer_t
  Defines user transfer structure used to initialize transfer. More...
- struct smartcard_context_t
  Runtime state of the Smart card driver. More...

Macros

- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES (42000u)
  Smart card global define which specify number of clock cycles until initial 'TS' character has to be received.
- #define SMARTCARD_EMV_ATR_DURATION_ETU (20150u)
  Smart card global define which specify number of clock cycles during which ATR string has to be received.
- #define SMARTCARD_TS_DIRECT_CONVENTION (0x3Bu)
  Smart card specification initial TS character definition of direct convention.
- #define SMARTCARD_TS_INVERSE_CONVENTION (0x3Fu)
  Smart card specification initial TS character definition of inverse convention.

Typedefs

- typedef void(∗ smartcard_interface_callback_t)(void ∗smartcardContext, void ∗param)
  Smart card interface interrupt callback function type.
- typedef void(∗ smartcard_transfer_callback_t)(void ∗smartcardContext, void ∗param)
  Smart card transfer interrupt callback function type.
- typedef void(∗ smartcard_time_delay_t)(uint32_t us)
  Time Delay function used to passive waiting using RTOS [us].

Enumerations

- enum smartcard_status_t {
  kStatus_SMARTCARD_Success = MAKE_STATUS(kStatusGroup_SMARTCARD, 0),
  kStatus_SMARTCARD_TxBusy = MAKE_STATUS(kStatusGroup_SMARTCARD, 1),
  kStatus_SMARTCARD_RxBusy = MAKE_STATUS(kStatusGroup_SMARTCARD, 2),
  kStatus_SMARTCARD_NoTransferInProgress = MAKE_STATUS(kStatusGroup_SMARTCAR-

D, 3),
kStatus_SMARTCARD_Timeout = MAKE_STATUS(kStatusGroup_SMARTCARD, 4),
kStatus_SMARTCARD_Initialized,
kStatus_SMARTCARD_PhyInitialized,
kStatus_SMARTCARD_CardNotActivated = MAKE_STATUS(kStatusGroup_SMARTCARD, 7),
kStatus_SMARTCARD_InvalidInput,
kStatus_SMARTCARD_OtherError = MAKE_STATUS(kStatusGroup_SMARTCARD, 9) }
    Smart card Error codes.
- enum smartcard_control_t
    Control codes for the Smart card protocol timers and misc.
- enum smartcard_card_voltage_class_t
    Defines Smart card interface voltage class values.
- enum smartcard_transfer_state_t
    Defines Smart card I/O transfer states.
- enum smartcard_reset_type_t
    Defines Smart card reset types.
- enum smartcard_transport_type_t
    Defines Smart card transport protocol types.
- enum smartcard_parity_type_t
    Defines Smart card data parity types.
- enum smartcard_card_convention_t
    Defines data Convention format.
- enum smartcard_interface_control_t
    Defines Smart card interface IC control types.
- enum smartcard_direction_t
    Defines transfer direction.

Driver version

- #define FSL_SMARTCARD_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))
    Smart card driver version 2.1.2.

36.4 Data Structure Documentation

36.4.1 struct smartcard_card_params_t

Data Fields

- uint16_t Fi
    4 bits Fi - clock rate conversion integer
- uint8_t fMax
    Maximum Smart card frequency in MHz.
- uint8_t WI
    8 bits WI - work wait time integer
- uint8_t Di
    4 bits DI - baud rate divisor
- uint8_t BWI
    4 bits BWI - block wait time integer
- uint8_t CWI

4 bits CWI - character wait time integer
- uint8_t BGI
  4 bits BGI - block guard time integer
- uint8_t GTN
  8 bits GTN - extended guard time integer
- uint8_t IFSC
  Indicates IFSC value of the card.
- uint8_t modeNegotiable
  Indicates if the card acts in negotiable or a specific mode.
- uint8_t currentD
  4 bits DI - current baud rate divisor
- uint8_t status
  Indicates smart card status.
- bool t0Indicated
  Indicates ff T=0 indicated in TD1 byte.
- bool t1Indicated
  Indicates if T=1 indicated in TD2 byte.
- bool atrComplete
  Indicates whether the ATR received from the card was complete or not.
- bool atrValid
  Indicates whether the ATR received from the card was valid or not.
- bool present
  Indicates if a smart card is present.
- bool active
  Indicates if the smart card is activated.
- bool faulty
  Indicates whether smart card/interface is faulty.
- smartcard_card_convention_t convention
  Card convention, kSMARTCARD_DirectConvention for direct convention, kSMARTCARD_InverseConvention for inverse convention.

36.4.1.0.0.40 Field Documentation

36.4.1.0.0.40.1 uint8_t smartcard_card_params_t::modeNegotiable

36.4.2 struct smartcard_timers_state_t

Data Fields

- volatile bool adtExpired
  Indicates whether ADT timer expired.
- volatile bool wwtExpired
  Indicates whether WWT timer expired.
- volatile bool cwtExpired
  Indicates whether CWT timer expired.
- volatile bool bwtExpired
  Indicates whether BWT timer expired.
- volatile bool initCharTimerExpired
  `Indicates whether reception timer`
  for initialization character (TS) after the RST has expired

36.4.3 struct smartcard_interface_config_t

Data Fields

- uint32_t smartCardClock
  Smart card interface clock [Hz].
- uint32_t clockToResetDelay
  Indicates clock to RST apply delay [smart card clock cycles].
- uint8_t clockModule
  Smart card clock module number.
- uint8_t clockModuleChannel
  Smart card clock module channel number.
- uint8_t clockModuleSourceClock
  Smart card clock module source clock [e.g., BusClk].
- smartcard_card_voltage_class_t vcc
  Smart card voltage class.
- uint8_t controlPort
  Smart card PHY control port instance.
- uint8_t controlPin
  Smart card PHY control pin instance.
- uint8_t irqPort
  Smart card PHY Interrupt port instance.
- uint8_t irqPin
  Smart card PHY Interrupt pin instance.
- uint8_t resetPort
  Smart card reset port instance.
- uint8_t resetPin
  Smart card reset pin instance.
- uint8_t vsel0Port
  Smart card PHY Vsel0 control port instance.
- uint8_t vsel0Pin
  Smart card PHY Vsel0 control pin instance.
- uint8_t vsel1Port
  Smart card PHY Vsel1 control port instance.
- uint8_t vsel1Pin
  Smart card PHY Vsel1 control pin instance.
- uint8_t dataPort
  Smart card PHY data port instance.
- uint8_t dataPin
  Smart card PHY data pin instance.
- uint8_t dataPinMux
  Smart card PHY data pin mux option.
- uint8_t tsTimerId
  Numerical identifier of the External HW timer for Initial character detection.

36.4.4 struct smartcard_xfer_t

Data Fields

- smartcard_direction_t direction
  Direction of communication.
- uint8_t * buff
  The buffer of data.
- size_t size
  The number of transferred units.

36.4.4.0.0.41 Field Documentation

36.4.4.0.0.41.1 smartcard_direction_t smartcard_xfer_t::direction

(RX/TX)

36.4.4.0.0.41.2 uint8_t* smartcard_xfer_t::buff

36.4.4.0.0.41.3 size_t smartcard_xfer_t::size

36.4.5 struct smartcard_context_t

Data Fields

- void * base
  Smart card module base address.
- smartcard_direction_t direction
  Direction of communication.
- uint8_t * xBuff
  The buffer of data being transferred.
- volatile size_t xSize
  The number of bytes to be transferred.
- volatile bool xIsBusy
  True if there is an active transfer.
- uint8_t txFifoEntryCount
  Number of data word entries in transmit FIFO.
- smartcard_interface_callback_t interfaceCallback
  Callback to invoke after interface IC raised interrupt.
- smartcard_transfer_callback_t transferCallback
  Callback to invoke after transfer event occur.
- void * interfaceCallbackParam
  Interface callback parameter pointer.
- void * transferCallbackParam
  Transfer callback parameter pointer.
- smartcard_time_delay_t timeDelay
  Function which handles time delay defined by user or RTOS.
- smartcard_reset_type_t resetType
  Indicates whether a Cold reset or Warm reset was requested.

- smartcard_transport_type_t tType
  Indicates current transfer protocol (T0 or T1)
- volatile smartcard_transfer_state_t transferState
  Indicates the current transfer state.
- smartcard_timers_state_t timersState
  Indicates the state of different protocol timers used in driver.
- smartcard_card_params_t cardParams
  Smart card parameters(ATR and current) and interface slots states(ATR and current)
- uint8_t IFSD
  Indicates the terminal IFSD.
- smartcard_parity_type_t parity
  Indicates current parity even/odd.
- volatile bool rxtCrossed
  Indicates whether RXT thresholds has been crossed.
- volatile bool txtCrossed
  Indicates whether TXT thresholds has been crossed.
- volatile bool wtxRequested
  Indicates whether WTX has been requested or not.
- volatile bool parityError
  Indicates whether a parity error has been detected.
- uint8_t statusBytes [2]
  Used to store Status bytes SW1, SW2 of the last executed card command response.
- smartcard_interface_config_t interfaceConfig
  Smart card interface configuration structure.

36.4.5.0.0.42 Field Documentation

36.4.5.0.0.42.1 smartcard_direction_t smartcard_context_t::direction

(RX/TX)

36.4.5.0.0.42.2 uint8_t* smartcard_context_t::xBuff

36.4.5.0.0.42.3 volatile size_t smartcard_context_t::xSize

36.4.5.0.0.42.4 volatile bool smartcard_context_t::xIsBusy

36.4.5.0.0.42.5 uint8_t smartcard_context_t::txFifoEntryCount

36.4.5.0.0.42.6 smartcard_interface_callback_t smartcard_context_t::interfaceCallback

36.4.5.0.0.42.7 smartcard_transfer_callback_t smartcard_context_t::transferCallback

36.4.5.0.0.42.8 void* smartcard_context_t::interfaceCallbackParam

36.4.5.0.0.42.9 void* smartcard_context_t::transferCallbackParam

36.4.5.0.0.42.10 smartcard_time_delay_t smartcard_context_t::timeDelay

36.4.5.0.0.42.11 smartcard_reset_type_t smartcard_context_t::resetType

36.5 Enumeration Type Documentation

36.5.1 enum smartcard_status_t

Enumerator

kStatus_SMARTCARD_Success Transfer ends successfully.
kStatus_SMARTCARD_TxBusy Transmit in progress.
kStatus_SMARTCARD_RxBusy Receiving in progress.
kStatus_SMARTCARD_NoTransferInProgress No transfer in progress.
kStatus_SMARTCARD_Timeout Transfer ends with time-out.
kStatus_SMARTCARD_Initialized Smart card driver is already initialized.
kStatus_SMARTCARD_PhyInitialized Smart card PHY drive is already initialized.
kStatus_SMARTCARD_CardNotActivated Smart card is not activated.
kStatus_SMARTCARD_InvalidInput Function called with invalid input arguments.
kStatus_SMARTCARD_OtherError Some other error occur.

36.5.2 enum smartcard_control_t

36.5.3 enum smartcard_direction_t

36.6 Smart Card PHY TDA8035 Driver

36.6.1 Overview

The Smart Card interface TDA8035 driver handles the external interface chip TDA8035 which supports all necessary functions to control the ICC. These functions involve PHY pin initialization, ICC voltage selection and activation, ICC clock generation, ICC card detection, and activation/deactivation sequences.

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  Smart card definition which specifies the adjustment number of clock cycles during which an ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  Smart card definition which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.
- #define SMARTCARD_TDA8035_STATUS_PRES (0x01u)
  Masks for TDA8035 status register.
- #define SMARTCARD_TDA8035_STATUS_ACTIVE (0x02u)
  Smart card PHY TDA8035 Smart card active status.
- #define SMARTCARD_TDA8035_STATUS_FAULTY (0x04u)
  Smart card PHY TDA8035 Smart card faulty status.
- #define SMARTCARD_TDA8035_STATUS_CARD_REMOVED (0x08u)
  Smart card PHY TDA8035 Smart card removed status.
- #define SMARTCARD_TDA8035_STATUS_CARD_DEACTIVATED (0x10u)
  Smart card PHY TDA8035 Smart card deactivated status.

Functions

- void SMARTCARD_PHY_TDA8035_GetDefaultConfig (smartcard_interface_config_t ∗config)
  Fills in the configuration structure with default values.
- status_t SMARTCARD_PHY_TDA8035_Init (void ∗base, smartcard_interface_config_t const ∗config, uint32_t srcClock_Hz)
  Initializes a Smart card interface instance.
- void SMARTCARD_PHY_TDA8035_Deinit (void ∗base, smartcard_interface_config_t ∗config)
  De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.
- status_t SMARTCARD_PHY_TDA8035_Activate (void ∗base, smartcard_context_t ∗context, smartcard_reset_type_t resetType)
  Activates the Smart card IC.
- status_t SMARTCARD_PHY_TDA8035_Deactivate (void ∗base, smartcard_context_t ∗context)
  De-activates the Smart card IC.
- status_t SMARTCARD_PHY_TDA8035_Control (void ∗base, smartcard_context_t ∗context, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card interface IC.
- void SMARTCARD_PHY_TDA8035_IRQHandler (void ∗base, smartcard_context_t ∗context)
  Smart card interface IC IRQ ISR.

36.6.2 Macro Definition Documentation

36.6.2.1 #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)

36.6.2.2 #define SMARTCARD_TDA8035_STATUS_PRES (0x01u)

Smart card PHY TDA8035 Smart card present status

36.6.3 Function Documentation

36.6.3.1 void SMARTCARD_PHY_TDA8035_GetDefaultConfig (smartcard_interface_config_t * *config*)

Parameters

config	The Smart card user configuration structure which contains configuration structure of type smartcard_interface_config_t. Function fill in members: clockToResetDelay = 42000, vcc = kSmartcardVoltageClassB3_3V, with default values.

36.6.3.2 status_t SMARTCARD_PHY_TDA8035_Init (void * *base*, smartcard_interface_config_t const * *config*, uint32_t *srcClock_Hz*)

Parameters

base	The Smart card peripheral base address.
config	The user configuration structure of type smartcard_interface_config_t. The user can call to fill out configuration structure function SMARTCARD_PHY_TDA8035_GetDefaultConfig().
srcClock_Hz	Smart card clock generation module source clock.

Return values

kStatus_SMARTCARD_Success	or kStatus_SMARTCARD_OtherError for an error.

36.6.3.3 void SMARTCARD_PHY_TDA8035_Deinit (void * *base*, smartcard_interface_config_t * *config*)

Parameters

base	The Smart card peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t.

36.6.3.4 status_t SMARTCARD_PHY_TDA8035_Activate (void * *base*, smartcard_context_t * *context*, smartcard_reset_type_t *resetType*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.6.3.5 status_t SMARTCARD_PHY_TDA8035_Deactivate (void * *base*, smartcard_context_t * *context*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.6.3.6 status_t SMARTCARD_PHY_TDA8035_Control (void * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
control	A interface command type.
param	Integer value specific to control type

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.6.3.7 void SMARTCARD_PHY_TDA8035_IRQHandler (void * *base*, smartcard_context_t * *context*)

Parameters

base	The Smart card peripheral module base address.
context	The Smart card context pointer.

36.7 Smart Card PHY EMVSIM Driver

36.7.1 Overview

The Smart Card interface EMVSIM driver handles the EMVSIM peripheral, which covers all necessary functions to control the ICC. These functions are ICC clock setup, ICC voltage turning on/off, ICC card detection, activation/deactivation, and ICC reset sequences. The EMVSIM peripheral covers all features of interface ICC chips.

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  Smart card define which specifies the adjustment number of clock cycles during which an ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  Smart card define which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.

Functions

- void SMARTCARD_PHY_EMVSIM_GetDefaultConfig (smartcard_interface_config_t ∗config)
  Fills in the smartcardInterfaceConfig structure with default values.
- status_t SMARTCARD_PHY_EMVSIM_Init (EMVSIM_Type ∗base, const smartcard_interface_config_t ∗config, uint32_t srcClock_Hz)
  Configures a Smart card interface.
- void SMARTCARD_PHY_EMVSIM_Deinit (EMVSIM_Type ∗base, const smartcard_interface_config_t ∗config)
  De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.
- status_t SMARTCARD_PHY_EMVSIM_Activate (EMVSIM_Type ∗base, smartcard_context_t ∗context, smartcard_reset_type_t resetType)
  Activates the Smart card IC.
- status_t SMARTCARD_PHY_EMVSIM_Deactivate (EMVSIM_Type ∗base, smartcard_context_t ∗context)
  De-activates the Smart card IC.
- status_t SMARTCARD_PHY_EMVSIM_Control (EMVSIM_Type ∗base, smartcard_context_t ∗context, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card interface IC.

36.7.2 Macro Definition Documentation

36.7.2.1 #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)

36.7.3 Function Documentation

36.7.3.1 void SMARTCARD_PHY_EMVSIM_GetDefaultConfig (smartcard_interface_config_t ∗ *config*)

Parameters

config	The user configuration structure of type smartcard_interface_config_t. Function fill in members: clockToResetDelay = 42000, vcc = kSmartcardVoltageClassB3_3V, with default values.

36.7.3.2 status_t SMARTCARD_PHY_EMVSIM_Init (EMVSIM_Type ∗ *base*, const smartcard_interface_config_t ∗ *config*, uint32_t *srcClock_Hz*)

Parameters

base	The Smart card peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t. The user is responsible to fill out the members of this structure and to pass the pointer of this structure into this function or call SMARTCARD_PHY_EMVSIMInitUserConfig-Default to fill out structure with default values.
srcClock_Hz	Smart card clock generation module source clock.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.7.3.3 void SMARTCARD_PHY_EMVSIM_Deinit (EMVSIM_Type ∗ *base*, const smartcard_interface_config_t ∗ *config*)

Parameters

base	Smart card peripheral module base address.
config	Smart card configuration structure.

36.7.3.4 status_t SMARTCARD_PHY_EMVSIM_Activate (EMVSIM_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_reset_type_t *resetType*)

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a Smart card driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.7.3.5 status_t SMARTCARD_PHY_EMVSIM_Deactivate (EMVSIM_Type * *base*, smartcard_context_t * *context*)

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a Smart card driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.7.3.6 status_t SMARTCARD_PHY_EMVSIM_Control (EMVSIM_Type * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a Smart card driver context structure.
control	A interface command type.

param	Integer value specific to control type

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError for an error.

36.8 Smart Card PHY GPIO Driver

36.8.1 Overview

The Smart Card interface GPIO driver handles the GPIO and FTM/TPM peripheral for clock generation, which covers all necessary functions to control the ICC. These functions are ICC clock setup, ICC voltage turning on/off, activation/deactivation, and ICC reset sequences. This driver doesn't support the ICC pin short circuit protection and an emergency deactivation.

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  Smart card define which specifies the adjustment number of clock cycles during which an ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  Smart card define which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.

Functions

- void SMARTCARD_PHY_GPIO_GetDefaultConfig (smartcard_interface_config_t ∗config)
  Fills in the configuration structure with default values.
- status_t SMARTCARD_PHY_GPIO_Init (UART_Type ∗base, smartcard_interface_config_t const ∗config, uint32_t srcClock_Hz)
  Initializes a Smart card interface instance.
- void SMARTCARD_PHY_GPIO_Deinit (UART_Type ∗base, smartcard_interface_config_t ∗config)
  De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.
- status_t SMARTCARD_PHY_GPIO_Activate (UART_Type ∗base, smartcard_context_t ∗context, smartcard_reset_type_t resetType)
  Activates the Smart card IC.
- status_t SMARTCARD_PHY_GPIO_Deactivate (UART_Type ∗base, smartcard_context_t ∗context)
  De-activates the Smart card IC.
- status_t SMARTCARD_PHY_GPIO_Control (UART_Type ∗base, smartcard_context_t ∗context, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card interface IC.

36.8.2 Macro Definition Documentation

36.8.2.1 #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)

36.8.3 Function Documentation

36.8.3.1 void SMARTCARD_PHY_GPIO_GetDefaultConfig (smartcard_interface_config_t ∗ *config*)

Parameters

config	The Smart card user configuration structure which contains configuration structure of type smartcard_interface_config_t. Function fill in members: clockToResetDelay = 42000, vcc = kSmartcardVoltageClassB3_3V, with default values.

36.8.3.2 status_t SMARTCARD_PHY_GPIO_Init (UART_Type ∗ *base*, smartcard_interface_config_t const ∗ *config*, uint32_t *srcClock_Hz*)

Parameters

base	The Smart card peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t. Call to fill out configuration structure function SMARTCARD_PHY_GPIO_GetDefaultConfig().
srcClock_Hz	Smart card clock generation module source clock.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

36.8.3.3 void SMARTCARD_PHY_GPIO_Deinit (UART_Type ∗ *base*, smartcard_interface_config_t ∗ *config*)

Parameters

base	The Smart card peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t.

36.8.3.4 status_t SMARTCARD_PHY_GPIO_Activate (UART_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_reset_type_t *resetType*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

36.8.3.5 status_t SMARTCARD_PHY_GPIO_Deactivate (UART_Type ∗ *base*, smartcard_context_t ∗ *context*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

36.8.3.6 status_t SMARTCARD_PHY_GPIO_Control (UART_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
control	An interface command type.
param	Integer value specific to the control type.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

36.9 Smart Card UART Driver

36.9.1 Overview

The Smart Card UART driver uses a standard UART peripheral which supports the ISO-7816 standard. The driver supports transmission functionality in the CPU mode. The driver also supports non-blocking (asynchronous) type of data transfers. The blocking (synchronous) transfer is supported only by the RTOS adaptation layer.

Macros

- #define SMARTCARD_EMV_RX_NACK_THRESHOLD (5u)
  EMV RX NACK interrupt generation threshold.
- #define SMARTCARD_EMV_TX_NACK_THRESHOLD (4u)
  EMV TX NACK interrupt generation threshold.
- #define SMARTCARD_EMV_RX_TO_TX_GUARD_TIME_T0 (0x0u)
  EMV TX & RX GUART TIME default value.

Functions

- void SMARTCARD_UART_GetDefaultConfig (smartcard_card_params_t ∗cardParams)
  Fills in the smartcard_card_params structure with default values according to the EMV 4.3 specification.
- status_t SMARTCARD_UART_Init (UART_Type ∗base, smartcard_context_t ∗context, uint32_t srcClock_Hz)
  Initializes a UART peripheral for the Smart card/ISO-7816 operation.
- void SMARTCARD_UART_Deinit (UART_Type ∗base)
  This function disables the UART interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates UART clock in SIM.
- int32_t SMARTCARD_UART_GetTransferRemainingBytes (UART_Type ∗base, smartcard_context_t ∗context)
  Returns whether the previous UART transfer has finished.
- status_t SMARTCARD_UART_AbortTransfer (UART_Type ∗base, smartcard_context_t ∗context)
  Terminates an asynchronous UART transfer early.
- status_t SMARTCARD_UART_TransferNonBlocking (UART_Type ∗base, smartcard_context_t ∗context, smartcard_xfer_t ∗xfer)
  Transfers data using interrupts.
- status_t SMARTCARD_UART_Control (UART_Type ∗base, smartcard_context_t ∗context, smartcard_control_t control, uint32_t param)
  Controls the UART module per different user requests.
- void SMARTCARD_UART_IRQHandler (UART_Type ∗base, smartcard_context_t ∗context)
  Interrupt handler for UART.
- void SMARTCARD_UART_ErrIRQHandler (UART_Type ∗base, smartcard_context_t ∗context)
  Error interrupt handler for UART.
- void SMARTCARD_UART_TSExpiryCallback (UART_Type ∗base, smartcard_context_t ∗context)
  Handles initial TS character timer time-out event.

36.9.2 Function Documentation

36.9.2.1 void SMARTCARD_UART_GetDefaultConfig (smartcard_card_params_t * *cardParams*)

Parameters

cardParams	The configuration structure of type smartcard_interface_config_t. Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.

36.9.2.2 status_t SMARTCARD_UART_Init (UART_Type * *base*, smartcard_context_t * *context*, uint32_t *srcClock_Hz*)

This function un-gates the UART clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core, and initializes the driver context.

Parameters

base	The UART peripheral base address.
context	A pointer to a smart card driver context structure.
srcClock_Hz	Smart card clock generation module source clock.

Returns

An error code or kStatus_SMARTCARD_Success.

36.9.2.3 void SMARTCARD_UART_Deinit (UART_Type * *base*)

Parameters

base	The UART peripheral base address.

36.9.2.4 int32_t SMARTCARD_UART_GetTransferRemainingBytes (UART_Type * *base*, smartcard_context_t * *context*)

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred by reading xSize of smart card context structure.

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

Returns

The number of bytes not transferred.

36.9.2.5 status_t SMARTCARD_UART_AbortTransfer (UART_Type ∗ *base*, smartcard_context_t ∗ *context*)

During an async UART transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

Return values

kStatus_SMARTCARD_Success	The transfer abort was successful.
kStatus_SMARTCARD_NoTransmitInProgress	No transmission is currently in progress.

36.9.2.6 status_t SMARTCARD_UART_TransferNonBlocking (UART_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_xfer_t ∗ *xfer*)

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling non-blocking (asynchronous) transfer function, the application must get the transfer status to check if transmit is completed or not.

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

xfer	A pointer to Smart card transfer structure where the linked buffers and sizes are stored.

Returns

An error code or kStatus_SMARTCARD_Success.

36.9.2.7 status_t SMARTCARD_UART_Control (UART_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_control_t *control*, uint32_t *param*)

Parameters

base	The UART peripheral base address.
context	A pointer to a smart card driver context structure.
control	Smart card command type.
param	Integer value specific to a control command.

return An kStatus_SMARTCARD_OtherError in case of error return kStatus_SMARTCARD_Success in success

36.9.2.8 void SMARTCARD_UART_IRQHandler (UART_Type ∗ *base*, smartcard_context_t ∗ *context*)

This handler uses the buffers stored in the smartcard_context_t structures to transfer data. The Smart card driver requires this function to call when the UART interrupt occurs.

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

36.9.2.9 void SMARTCARD_UART_ErrIRQHandler (UART_Type ∗ *base*, smartcard_context_t ∗ *context*)

This function handles error conditions during a transfer.

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

36.9.2.10 void SMARTCARD_UART_TSExpiryCallback (UART_Type * *base*, smartcard_context_t * *context*)

Parameters

base	The UART peripheral base address.
context	A pointer to a Smart card driver context structure.

36.10 Smart Card EMVSIM Driver

36.10.1 Overview

The SmartCard EMVSIM driver covers the transmission functionality in the CPU mode. The driver supports non-blocking (asynchronous) type of data transfers. The blocking (synchronous) transfer is supported only by the RTOS adaptation layer.

Macros

- #define SMARTCARD_EMV_RX_NACK_THRESHOLD (5u)
  EMV RX NACK interrupt generation threshold.
- #define SMARTCARD_EMV_TX_NACK_THRESHOLD (5u)
  EMV TX NACK interrupt generation threshold.
- #define SMARTCARD_WWT_ADJUSTMENT (160u)
  Smart card Word Wait Timer adjustment value.
- #define SMARTCARD_CWT_ADJUSTMENT (3u)
  Smart card Character Wait Timer adjustment value.

Enumerations

- enum emvsim_gpc_clock_select_t {
  kEMVSIM_GPCClockDisable = 0u,
  kEMVSIM_GPCCardClock = 1u,
  kEMVSIM_GPCRxClock = 2u,
  kEMVSIM_GPCTxClock = 3u }
  General Purpose Counter clock selections.
- enum emvsim_presence_detect_edge_t {
  kEMVSIM_DetectOnFallingEdge = 0u,
  kEMVSIM_DetectOnRisingEdge = 1u }
  EMVSIM card presence detection edge control.
- enum emvsim_presence_detect_status_t {
  kEMVSIM_DetectPinIsLow = 0u,
  kEMVSIM_DetectPinIsHigh = 1u }
  EMVSIM card presence detection status.

Smart card EMVSIM Driver

- void SMARTCARD_EMVSIM_GetDefaultConfig (smartcard_card_params_t ∗cardParams)
  Fills in the smartcard_card_params structure with default values according to the EMV 4.3 specification.
- status_t SMARTCARD_EMVSIM_Init (EMVSIM_Type ∗base, smartcard_context_t ∗context, uint32_t srcClock_Hz)
  Initializes an EMVSIM peripheral for the Smart card/ISO-7816 operation.
- void SMARTCARD_EMVSIM_Deinit (EMVSIM_Type ∗base)

This function disables the EMVSIM interrupts, disables the transmitter and receiver, flushes the FIFOs, and gates EMVSIM clock in SIM.

- int32_t SMARTCARD_EMVSIM_GetTransferRemainingBytes (EMVSIM_Type ∗base, smartcard_context_t ∗context)
  Returns whether the previous EMVSIM transfer has finished.
- status_t SMARTCARD_EMVSIM_AbortTransfer (EMVSIM_Type ∗base, smartcard_context_t ∗context)
  Terminates an asynchronous EMVSIM transfer early.
- status_t SMARTCARD_EMVSIM_TransferNonBlocking (EMVSIM_Type ∗base, smartcard_context_t ∗context, smartcard_xfer_t ∗xfer)
  Transfer data using interrupts.
- status_t SMARTCARD_EMVSIM_Control (EMVSIM_Type ∗base, smartcard_context_t ∗context, smartcard_control_t control, uint32_t param)
  Controls the EMVSIM module per different user request.
- void SMARTCARD_EMVSIM_IRQHandler (EMVSIM_Type ∗base, smartcard_context_t ∗context)
  Handles EMVSIM module interrupts.

36.10.2 Enumeration Type Documentation

36.10.2.1 enum emvsim_gpc_clock_select_t

Enumerator

kEMVSIM_GPCClockDisable Disabled.
kEMVSIM_GPCCardClock Card clock.
kEMVSIM_GPCRxClock Receive clock.
kEMVSIM_GPCTxClock Transmit ETU clock.

36.10.2.2 enum emvsim_presence_detect_edge_t

Enumerator

kEMVSIM_DetectOnFallingEdge Presence detected on the falling edge.
kEMVSIM_DetectOnRisingEdge Presence detected on the rising edge.

36.10.2.3 enum emvsim_presence_detect_status_t

Enumerator

kEMVSIM_DetectPinIsLow Presence detected pin is logic low.
kEMVSIM_DetectPinIsHigh Presence detected pin is logic high.

36.10.3 Function Documentation

36.10.3.1 void SMARTCARD_EMVSIM_GetDefaultConfig (smartcard_card_params_t * *cardParams*)

Parameters

cardParams	The configuration structure of type smartcard_interface_config_t. Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.

36.10.3.2 status_t SMARTCARD_EMVSIM_Init (EMVSIM_Type ∗ *base*, smartcard_context_t ∗ *context*, uint32_t *srcClock_Hz*)

This function un-gates the EMVSIM clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core and, initializes the driver context.

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to the smart card driver context structure.
srcClock_Hz	Smart card clock generation module source clock.

Returns

An error code or kStatus_SMARTCARD_Success.

36.10.3.3 void SMARTCARD_EMVSIM_Deinit (EMVSIM_Type ∗ *base*)

Parameters

base	The EMVSIM module base address.

36.10.3.4 int32_t SMARTCARD_EMVSIM_GetTransferRemainingBytes (EMVSIM_Type ∗ *base*, smartcard_context_t ∗ *context*)

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred.

Parameters

base	The EMVSIM module base address.
context	A pointer to a smart card driver context structure.

Returns

The number of bytes not transferred.

36.10.3.5 status_t SMARTCARD_EMVSIM_AbortTransfer (EMVSIM_Type * *base*, smartcard_context_t * *context*)

During an async EMVSIM transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

base	The EMVSIM peripheral address.
context	A pointer to a smart card driver context structure.

Return values

kStatus_SMARTCARD_Success	The transmit abort was successful.
kStatus_SMARTCARD_NoTransmitInProgress	No transmission is currently in progress.

36.10.3.6 status_t SMARTCARD_EMVSIM_TransferNonBlocking (EMVSIM_Type * *base*, smartcard_context_t * *context*, smartcard_xfer_t * *xfer*)

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling the non-blocking (asynchronous) transfer function, the application must get the transfer status to check if the transmit is completed or not.

Parameters

base	The EMVSIM peripheral base address.

context	A pointer to a smart card driver context structure.
xfer	A pointer to the smart card transfer structure where the linked buffers and sizes are stored.

Returns

An error code or kStatus_SMARTCARD_Success.

36.10.3.7 status_t SMARTCARD_EMVSIM_Control (EMVSIM_Type ∗ *base*, smartcard_context_t ∗ *context*, smartcard_control_t *control*, uint32_t *param*)

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a smart card driver context structure.
control	Control type.
param	Integer value of specific to control command.

return kStatus_SMARTCARD_Success in success. return kStatus_SMARTCARD_OtherError in case of error.

36.10.3.8 void SMARTCARD_EMVSIM_IRQHandler (EMVSIM_Type ∗ *base*, smartcard_context_t ∗ *context*)

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a smart card driver context structure.

36.11 Smart Card FreeRTOS Driver

36.11.1 Overview

Data Structures

- struct rtos_smartcard_context_t
  Runtime RTOS Smart card driver context. More...

Macros

- #define RTOS_SMARTCARD_COMPLETE 0x1u
  Smart card RTOS transfer complete flag.
- #define RTOS_SMARTCARD_TIMEOUT 0x2u
  Smart card RTOS transfer time-out flag.
- #define SMARTCARD_Control(base, context, control, param) SMARTCARD_EMVSIM_Control(base, context, control, param)
  Common Smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_EMVSIM_TransferNonBlocking(base, context, xfer)
  Common Smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_EMVSIM_Init(base, context, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_EMVSIM_Deinit(base)
  Common Smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_EMVSIM_GetTransferRemainingBytes(base, context)
  Common Smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_EMVSIM_GetDefaultConfig(cardParams)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Activate(base, context, resetType) SMARTCARD_PHY_EMVSIM_Activate(base, context, resetType)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deactivate(base, context) SMARTCARD_PHY_EMVSIM_Deactivate(base, context)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Control(base, context, control, param) SMARTCARD_PHY_EMVSIM_Control(base, context, control, param)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Init(base, config, sourceClockHz) SMARTCARD_PHY_EMVSIM_Init(base, config, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deinit(base, config) SMARTCARD_PHY_EMVSIM_Deinit(base, config)
  *Common Smart card API macro *

- #define SMARTCARD_PHY_GetDefaultConfig(config) SMARTCARD_PHY_EMVSIM_GetDefaultConfig(config)
  Common Smart card API macro.

Functions

- int SMARTCARD_RTOS_Init (void ∗base, rtos_smartcard_context_t ∗ctx, uint32_t sourceClockHz)
  Initializes a Smart card (EMVSIM/UART) peripheral for Smart card/ISO-7816 operation.
- int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t ∗ctx)
  This function disables the Smart card (EMVSIM/UART) interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates Smart card clock in SIM.
- int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t ∗ctx, smartcard_xfer_t ∗xfer)
  Transfers data using interrupts.
- int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t ∗ctx)
  Waits until the transfer is finished.
- int SMARTCARD_RTOS_Control (rtos_smartcard_context_t ∗ctx, smartcard_control_t control, uint32_t param)
  Controls the Smart card module per different user requests.
- int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t ∗ctx, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card module as per different user request.
- int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t ∗ctx, smartcard_reset_type_t resetType)
  Activates the Smart card interface.
- int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t ∗ctx)
  Deactivates the Smart card interface.

36.11.2 Data Structure Documentation

36.11.2.1 struct rtos_smartcard_context_t

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- SemaphoreHandle_t x_event
  RTOS synchronization object.
- smartcard_context_t x_context
  transactional layer state
- OS_EVENT ∗ x_sem
  RTOS unique access assurance object.
- OS_FLAG_GRP ∗ x_event
  RTOS synchronization object.
- OS_SEM x_sem
  RTOS unique access assurance object.
- OS_FLAG_GRP x_event

RTOS synchronization object.

36.11.2.1.0.43 Field Documentation

36.11.2.1.0.43.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

36.11.3 Macro Definition Documentation

36.11.3.1 #define SMARTCARD_Control(*base, context, control, param*) SMARTCARD_EMVSIM_Control(base, context, control, param)

Common Smart card API macro

36.11.4 Function Documentation

36.11.4.1 int SMARTCARD_RTOS_Init (void * *base*, rtos_smartcard_context_t * *ctx*, uint32_t *sourceClockHz*)

Also initialize Smart card PHY interface.

This function ungates the Smart card clock, initializes the module to EMV default settings, configures the IRQ state structure, and enables the module-level interrupt to the core. Initializes RTOS synchronization objects and context.

Parameters

base	The Smart card peripheral base address.
ctx	The Smart card RTOS structure.
sourceClockHz	Smart card clock generation module source clock.

Returns

A zero in success or error code.

36.11.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

It also deactivates Smart card PHY interface, stops Smart card clocks, and frees all synchronization objects allocated in the RTOS Smart card context.

Parameters

ctx	The Smart card RTOS state.

Returns

A zero in success or error code.

36.11.4.3 int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t * *ctx*, smartcard_xfer_t * *xfer*)

A blocking (also known as synchronous) function means that the function returns after the transfer is done. User can cancel this transfer by calling the function AbortTransfer.

Parameters

ctx	A pointer to the RTOS Smart card driver context.
xfer	Smart card transfer structure.

Returns

A zero in success or error code.

36.11.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t * *ctx*)

Task waits on a transfer finish event. Don't initialize the transfer. Instead, wait for a transfer callback. This function can be used while waiting on an initial TS character.

Parameters

ctx	A pointer to the RTOS Smart card driver context.

Returns

A zero in success or error code.

36.11.4.5 int SMARTCARD_RTOS_Control (rtos_smartcard_context_t * *ctx*, smartcard_control_t *control*, uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type.
param	Integer value to control the command.

Returns

A zero in success or error code.

36.11.4.6 int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t * *ctx*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value to control the command.

Returns

A zero in success or error code.

36.11.4.7 int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t * *ctx*, smartcard_reset_type_t *resetType*)

Parameters

ctx	The Smart card RTOS driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Returns

A zero in success or error code.

36.11.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The Smart card RTOS driver context structure.

Returns

A zero in success or error code.

36.12 Smart Card µCOS/II Driver

36.12.1 Overview

Data Structures

- struct rtos_smartcard_context_t
  Runtime RTOS Smart card driver context. More...

Macros

- #define RTOS_SMARTCARD_COMPLETE 0x1u
  Smart card RTOS transfer complete flag.
- #define RTOS_SMARTCARD_TIMEOUT 0x2u
  Smart card RTOS transfer time-out flag.
- #define SMARTCARD_Control(base, context, control, param) SMARTCARD_EMVSIM_-Control(base, context, control, 0)
  Common Smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_EMVSIM_TransferNon-Blocking(base, context, xfer)
  Common Smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_EMVSIM_Init(base, context, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_EMVSIM_Deinit(base)
  Common Smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_EMVSIM_-GetTransferRemainingBytes(base, context)
  Common Smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_EMVSIM_GetDefault-Config(cardParams)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Activate(base, context, resetType) SMARTCARD_PHY_EMVSIM-_Activate(base, context, resetType)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deactivate(base, context) SMARTCARD_PHY_EMVSIM_-Deactivate(base, context)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Control(base, context, control, param) SMARTCARD_PHY_EMV-SIM_Control(base, context, control, param)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Init(base, config, sourceClockHz) SMARTCARD_PHY_EMVSIM-_Init(base, config, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deinit(base, config) SMARTCARD_PHY_EMVSIM_Deinit(base, config)
  *Common Smart card API macro *

- #define SMARTCARD_PHY_GetDefaultConfig(config) SMARTCARD_PHY_EMVSIM_GetDefaultConfig(config)
  Common Smart card API macro.

Functions

- int SMARTCARD_RTOS_Init (void ∗base, rtos_smartcard_context_t ∗ctx, uint32_t sourceClockHz)
  Initializes a Smart card (EMVSIM/UART) peripheral for the Smart card/ISO-7816 operation.
- int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t ∗ctx)
  This function disables the Smart card (EMVSIM/UART) interrupts, the transmitter and receiver, flushes the FIFOs (for modules that support FIFOs), gates the Smart card clock in SIM, deactivates the Smart card PHY interface, stops Smart card clocks, and frees all synchronization objects allocated in RTOS Smart card context.
- int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t ∗ctx, smartcard_xfer_t ∗xfer)
  Transfers data using interrupts.
- int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t ∗ctx)
  Waits until the transfer is finished.
- int SMARTCARD_RTOS_Control (rtos_smartcard_context_t ∗ctx, smartcard_control_t control, uint32_t param)
  Controls the Smart card module per different user requests.
- int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t ∗ctx, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card module per different user requests.
- int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t ∗ctx, smartcard_reset_type_t resetType)
  Activates the Smart card interface.
- int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t ∗ctx)
  Deactivates the Smart card interface.

36.12.2 Data Structure Documentation

36.12.2.1 struct rtos_smartcard_context_t

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- SemaphoreHandle_t x_event
  RTOS synchronization object.
- smartcard_context_t x_context
  transactional layer state
- OS_EVENT ∗ x_sem
  RTOS unique access assurance object.
- OS_FLAG_GRP ∗ x_event
  RTOS synchronization object.
- OS_SEM x_sem

RTOS unique access assurance object.

- OS_FLAG_GRP x_event

    RTOS synchronization object.

36.12.2.1.0.44 Field Documentation

36.12.2.1.0.44.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

36.12.3 Macro Definition Documentation

36.12.3.1 #define SMARTCARD_Control(*base*, *context*, *control*, *param*) SMARTCARD_EMVSIM_Control(base, context, control, 0)

Common Smart card API macro

36.12.4 Function Documentation

36.12.4.1 int SMARTCARD_RTOS_Init (void * *base*, rtos_smartcard_context_t * *ctx*, uint32_t *sourceClockHz*)

Also initialize Smart card PHY interface .

This function ungates the Smart card clock, initializes the module to EMV default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and initializes the RTOS synchronization objects and context.

Parameters

base	The Smart card peripheral base address.
ctx	The Smart card RTOS structure.
sourceClockHz	Smart card clock generation module source clock.

Returns

A zero in success or error code.

36.12.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The Smart card RTOS state.

Returns

A zero in success or error code.

36.12.4.3 int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t ∗ *ctx*, smartcard_xfer_t ∗ *xfer*)

A blocking (also known as synchronous) function means that the function returns after the transfer is done. Cancel this transfer by calling the function AbortTransfer.

Parameters

ctx	A pointer to the RTOS Smart card driver context.
xfer	Smart card transfer structure.

Returns

A zero in success or error code.

36.12.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t ∗ *ctx*)

Task waits on the transfer finish event. Don't initialize transfer. Instead, wait for a transfer callback. This function can be used while waiting on an initial TS character.

Parameters

ctx	A pointer to the RTOS Smart card driver context.

Returns

A zero in success or error code.

36.12.4.5 int SMARTCARD_RTOS_Control (rtos_smartcard_context_t ∗ *ctx*, smartcard_control_t *control*, uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value specific to a control command.

Returns

A zero in success or error code.

36.12.4.6 int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t ∗ *ctx*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value specific to a control command.

Returns

A zero in success or error code.

36.12.4.7 int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t ∗ *ctx*, smartcard_reset_type_t *resetType*)

Parameters

ctx	The Smart card RTOS driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Returns

A zero in success or error code.

36.12.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t ∗ *ctx*)

Parameters

ctx	The Smart card RTOS driver context structure.

Returns

A zero in success or error code.

36.13 Smart Card µCOS/III Driver

36.13.1 Overview

Data Structures

- struct rtos_smartcard_context_t
  Runtime RTOS Smart card driver context. More...

Macros

- #define RTOS_SMARTCARD_COMPLETE 0x1u
  Smart card RTOS transfer complete flag.
- #define RTOS_SMARTCARD_TIMEOUT 0x2u
  Smart card RTOS transfer time-out flag.
- #define SMARTCARD_Control(base, context, control, param) SMARTCARD_EMVSIM_Control(base, context, control, 0)
  Common Smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_EMVSIM_TransferNonBlocking(base, context, xfer)
  Common Smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_EMVSIM_Init(base, context, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_EMVSIM_Deinit(base)
  Common Smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_EMVSIM_GetTransferRemainingBytes(base, context)
  Common Smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_EMVSIM_GetDefaultConfig(cardParams)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Activate(base, context, resetType) SMARTCARD_PHY_EMVSIM_Activate(base, context, resetType)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deactivate(base, context) SMARTCARD_PHY_EMVSIM_Deactivate(base, context)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Control(base, context, control, param) SMARTCARD_PHY_EMVSIM_Control(base, context, control, param)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Init(base, config, sourceClockHz) SMARTCARD_PHY_EMVSIM_Init(base, config, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_PHY_Deinit(base, config) SMARTCARD_PHY_EMVSIM_Deinit(base, config)
  *Common Smart card API macro *

- #define SMARTCARD_PHY_GetDefaultConfig(config) SMARTCARD_PHY_EMVSIM_GetDefaultConfig(config)
  Common Smart card API macro.

Functions

- int SMARTCARD_RTOS_Init (void ∗base, rtos_smartcard_context_t ∗ctx, uint32_t sourceClockHz)
  Initializes a Smart card (EMVSIM/UART) peripheral for the Smart card/ISO-7816 operation, and initializes the Smart card PHY interface.
- int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t ∗ctx)
  This function disables the Smart card (EMVSIM/UART) interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs), gates the Smart card clock in SIM, deactivates the Smart card PHY interface, stops the Smart card clocks, and frees all synchronization objects allocated in the RTOS Smart card context.
- int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t ∗ctx, smartcard_xfer_t ∗xfer)
  Transfers data using interrupts.
- int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t ∗ctx)
  Waits until transfer is finished.
- int SMARTCARD_RTOS_Control (rtos_smartcard_context_t ∗ctx, smartcard_control_t control, uint32_t param)
  Controls the Smart card module per different user requests.
- int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t ∗ctx, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card module per different user requests.
- int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t ∗ctx, smartcard_reset_type_t resetType)
  Activates the Smart card interface.
- int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t ∗ctx)
  Deactivates the Smart card interface.

36.13.2 Data Structure Documentation

36.13.2.1 struct rtos_smartcard_context_t

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- SemaphoreHandle_t x_event
  RTOS synchronization object.
- smartcard_context_t x_context
  transactional layer state
- OS_EVENT ∗ x_sem
  RTOS unique access assurance object.
- OS_FLAG_GRP ∗ x_event
  RTOS synchronization object.

- OS_SEM x_sem
  RTOS unique access assurance object.
- OS_FLAG_GRP x_event
  RTOS synchronization object.

36.13.2.1.0.45 Field Documentation

36.13.2.1.0.45.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

36.13.3 Macro Definition Documentation

36.13.3.1 #define SMARTCARD_Control(*base,* *context,* *control,* *param*) SMARTCARD_EMVSIM_Control(base, context, control, 0)

Common Smart card API macro

36.13.4 Function Documentation

36.13.4.1 int SMARTCARD_RTOS_Init (void * *base,* rtos_smartcard_context_t * *ctx,* uint32_t *sourceClockHz*)

This function ungates the Smart card clock, initializes the module to EMV default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and initializes RTOS synchronization objects and context.

Parameters

base	The Smart card peripheral base address.
ctx	The Smart card RTOS structure.
sourceClockHz	Smart card clock generation module source clock.

Returns

A zero in success or error code.

36.13.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The Smart card RTOS state.

Returns

A zero in success or error code.

36.13.4.3 int SMARTCARD_RTOS_Transfer (rtos_smartcard_context_t ∗ *ctx*, smartcard_xfer_t ∗ *xfer*)

A blocking (also known as synchronous) function means that the function returns after the transfer is done. Cancel this transfer by calling the function AbortTransfer.

Parameters

ctx	A pointer to the RTOS Smart card driver context.
xfer	Smart card transfer structure.

Returns

A zero in success or error code.

36.13.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t ∗ *ctx*)

Task waits on the transfer finish event. Don't initialize transfer. Instead, wait for the transfer callback. This function can be used while waiting on an initial TS character.

Parameters

ctx	A pointer to the RTOS Smart card driver context.

Returns

A zero in success or error code.

36.13.4.5 int SMARTCARD_RTOS_Control (rtos_smartcard_context_t ∗ *ctx*, smartcard_control_t *control*, uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type.
param	Integer value specific to a control command.

Returns

A zero in success or error code.

36.13.4.6 int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t ∗ *ctx,* smartcard_interface_control_t *control,* uint32_t *param*)

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type.
param	Integer value specific to a control command.

Returns

A zero in success or error code.

36.13.4.7 int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t ∗ *ctx,* smartcard_reset_type_t *resetType*)

Parameters

ctx	The Smart card RTOS driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Returns

A zero in success or error code.

36.13.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t ∗ *ctx*)

Parameters

ctx	The Smart card RTOS driver context structure.

Returns

A zero in success or error code.

Chapter 37
SMC: System Mode Controller Driver

37.1 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.

37.2 Typical use case

37.2.1 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. Pre and post functions are used for setting the modes. The pre functions and post functions are used as follows.

1. Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function SMC_SetPowerModeStop to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function S-MC_SetPowerModeStop. As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMAS-K. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.
2. Disable/enable the flash speculation. When entering stop modes, the flash speculation might be interrupted. As a result, pre functions disable the flash speculation and post functions enable it.

```
SMC_PreEnterStopModes();

/* Enable the wakeup interrupt here. */

SMC_SetPowerModeStop(SMC, kSMC_PartialStop);

SMC_PostExitStopModes();
```

Data Structures

- struct smc_power_mode_lls_config_t
  SMC Low-Leakage Stop power mode configuration. More...
- struct smc_power_mode_vlls_config_t
  SMC Very Low-Leakage Stop power mode configuration. More...

Enumerations

- enum smc_power_mode_protection_t {
kSMC_AllowPowerModeVlls = SMC_PMPROT_AVLLS_MASK,
kSMC_AllowPowerModeLls = SMC_PMPROT_ALLS_MASK,
kSMC_AllowPowerModeVlp = SMC_PMPROT_AVLP_MASK,
kSMC_AllowPowerModeHsrun = SMC_PMPROT_AHSRUN_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,
kSMC_PowerStateLls = 0x01U << 5U,
kSMC_PowerStateVlls = 0x01U << 6U,
kSMC_PowerStateHsrun = 0x01U << 7U }
  Power Modes in PMSTAT.
- enum smc_run_mode_t {
kSMC_RunNormal = 0U,
kSMC_RunVlpr = 2U,
kSMC_Hsrun = 3U }
  Run mode definition.
- enum smc_stop_mode_t {
kSMC_StopNormal = 0U,
kSMC_StopVlps = 2U,
kSMC_StopLls = 3U,
kSMC_StopVlls = 4U }
  Stop mode definition.
- enum smc_stop_submode_t {
kSMC_StopSub0 = 0U,
kSMC_StopSub1 = 1U,
kSMC_StopSub2 = 2U,
kSMC_StopSub3 = 3U }
  VLLS/LLS stop sub 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, 3))
  SMC driver version 2.0.3.

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.
- void SMC_PreEnterStopModes (void)
  Prepares to enter stop modes.
- void SMC_PostExitStopModes (void)
  Recovers after wake up from stop modes.
- static void SMC_PreEnterWaitModes (void)
  Prepares to enter wait modes.
- static void SMC_PostExitWaitModes (void)
  Recovers after wake up from stop modes.
- status_t SMC_SetPowerModeRun (SMC_Type ∗base)
  Configures the system to RUN power mode.
- status_t SMC_SetPowerModeHsrun (SMC_Type ∗base)
  Configures the system to HSRUN 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.
- status_t SMC_SetPowerModeLls (SMC_Type ∗base, const smc_power_mode_lls_config_t ∗config)
  Configures the system to LLS power mode.
- status_t SMC_SetPowerModeVlls (SMC_Type ∗base, const smc_power_mode_vlls_config_t ∗config)
  Configures the system to VLLS power mode.

37.3 Data Structure Documentation

37.3.1 struct smc_power_mode_lls_config_t

Data Fields

- smc_stop_submode_t subMode
  Low-leakage Stop sub-mode.
- bool enableLpoClock
  Enable LPO clock in LLS mode.

37.3.2 struct smc_power_mode_vlls_config_t

Data Fields

- smc_stop_submode_t subMode
  Very Low-leakage Stop sub-mode.
- bool enablePorDetectInVlls0
  Enable Power on reset detect in VLLS mode.
- bool enableRam2InVlls2
  Enable RAM2 power in VLLS2.
- bool enableLpoClock
  Enable LPO clock in VLLS mode.

37.4 Macro Definition Documentation

37.4.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

37.5 Enumeration Type Documentation

37.5.1 enum smc_power_mode_protection_t

Enumerator

kSMC_AllowPowerModeVlls Allow Very-low-leakage Stop Mode.
kSMC_AllowPowerModeLls Allow Low-leakage Stop Mode.
kSMC_AllowPowerModeVlp Allow Very-Low-power Mode.
kSMC_AllowPowerModeHsrun Allow High-speed Run mode.
kSMC_AllowPowerModeAll Allow all power mode.

37.5.2 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
kSMC_PowerStateLls 0010_0000 - Current power mode is LLS
kSMC_PowerStateVlls 0100_0000 - Current power mode is VLLS
kSMC_PowerStateHsrun 1000_0000 - Current power mode is HSRUN

37.5.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal Normal RUN mode.
kSMC_RunVlpr Very-low-power RUN mode.
kSMC_Hsrun High-speed Run mode (HSRUN).

37.5.4 enum smc_stop_mode_t

Enumerator

kSMC_StopNormal Normal STOP mode.
kSMC_StopVlps Very-low-power STOP mode.
kSMC_StopLls Low-leakage Stop mode.
kSMC_StopVlls Very-low-leakage Stop mode.

37.5.5 enum smc_stop_submode_t

Enumerator

kSMC_StopSub0 Stop submode 0, for VLLS0/LLS0.
kSMC_StopSub1 Stop submode 1, for VLLS1/LLS1.
kSMC_StopSub2 Stop submode 2, for VLLS2/LLS2.
kSMC_StopSub3 Stop submode 3, for VLLS3/LLS3.

37.5.6 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.

37.5.7 enum _smc_status

Enumerator

kStatus_SMC_StopAbort Entering Stop mode is abort.

37.6 Function Documentation

37.6.1 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.

37.6.2 static smc_power_state_t SMC_GetPowerModeState (SMC_Type * *base*) [inline], [static]

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc_power_state_t for information about the power status.

Parameters

base	SMC peripheral base address.

Returns

Current power mode status.

37.6.3 void SMC_PreEnterStopModes (void)

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

37.6.4 void SMC_PostExitStopModes (void)

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with SMC-_PreEnterStopModes.

37.6.5 static void SMC_PreEnterWaitModes (void) [inline], [static]

This function should be called before entering WAIT/VLPW modes.

37.6.6 static void SMC_PostExitWaitModes (void) [inline], [static]

This function should be called after wake up from WAIT/VLPW modes. It is used with SMC_PreEnter-WaitModes.

37.6.7 status_t SMC_SetPowerModeRun (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.8 status_t SMC_SetPowerModeHsrun (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.9 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.10 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.

37.6.11 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.12 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.13 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

37.6.14 status_t SMC_SetPowerModeLls (SMC_Type * *base*, const smc_power_mode_lls_config_t * *config*)

Parameters

base	SMC peripheral base address.
config	The LLS power mode configuration structure

Returns

SMC configuration error code.

37.6.15 status_t SMC_SetPowerModeVlls (SMC_Type * *base*, const smc_power_mode_vlls_config_t * *config*)

Parameters

base	SMC peripheral base address.
config	The VLLS power mode configuration structure.

Returns

SMC configuration error code.

Chapter 38
TPM: Timer PWM Module

38.1 Overview

The KSDK provides a driver for the Timer PWM Module (TPM) of Kinetis devices.

The KSDK TPM driver supports the generation of PWM signals, input capture, and output compare modes. On some SoCs, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decoder modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoCs.

The function TPM_Init() initializes the TPM with a specified configurations. The function TPM_GetDefaultConfig() gets the default configurations. On some SoCs, the initialization function issues a software reset to reset the TPM internal logic. The initialization function configures the TPM's behavior when it receives a trigger input and its operation in doze and debug modes.

The function TPM_Deinit() disables the TPM counter and turns off the module clock.

The function TPM_SetupPwm() sets up TPM channels for the PWM output. The function can set up the PWM signal properties for multiple channels. Each channel has its own tpm_chnl_pwm_signal_param_t structure that is used to specify the output signals duty cycle and level-mode. However, the same PWM period and PWM mode is applied to all channels requesting a PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 where 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle). When generating a combined PWM signal, the channel number passed refers to a channel pair number, for example 0 refers to channel 0 and 1, 1 refers to channels 2 and 3.

The function TPM_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular TPM channel.

The function TPM_UpdateChnlEdgeLevelSelect() updates the level select bits of a particular TPM channel. This can be used to disable the PWM output when making changes to the PWM signal.

The function TPM_SetupInputCapture() sets up a TPM channel for input capture. The user can specify the capture edge.

The function TPM_SetupDualEdgeCapture() can be used to measure the pulse width of a signal. This is available only for certain SoCs. A channel pair is used during the capture with the input signal coming through a channel that can be configured. The user can specify the capture edge for each channel and any filter value to be used when processing the input signal.

The function TPM_SetupOutputCompare() sets up a TPM channel for output comparison. The user can specify the channel output on a successful comparison and a comparison value.

The function TPM_SetupQuadDecode() sets up TPM channels 0 and 1 for quad decode, which is available only for certain SoCs. The user can specify the quad decode mode, polarity, and filter properties for each input signal.

The function TPM_SetupFault() sets up the properties for each fault, which is available only for certain SoCs. 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.

Provides functions to get and clear the TPM status.

Provides functions to enable/disable TPM interrupts and get current enabled interrupts.

38.2 Typical use case

38.2.1 PWM output

Output the PWM signal on 2 TPM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether the LED is brighter or dimmer. */
    tpm_config_t tpmInfo;
    uint8_t updatedDutycycle = 0U;
    tpm_chnl_pwm_signal_param_t tpmParam[2];

    /* Configures the TPM parameters with frequency 24 kHZ. */
    tpmParam[0].chnlNumber = (tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL;
    tpmParam[0].level = kTPM_LowTrue;
    tpmParam[0].dutyCyclePercent = 0U;

    tpmParam[1].chnlNumber = (tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL;
    tpmParam[1].level = kTPM_LowTrue;
    tpmParam[1].dutyCyclePercent = 0U;

    /* Board pin, clock, and debug console initialization. */
    BOARD_InitHardware();

    TPM_GetDefaultConfig(&tpmInfo);
    /* Initializes the TPM module. */
    TPM_Init(BOARD_TPM_BASEADDR, &tpmInfo);

    TPM_SetupPwm(BOARD_TPM_BASEADDR, tpmParam, 2U,
      kTPM_EdgeAlignedPwm, 24000U, TPM_SOURCE_CLOCK);
    TPM_StartTimer(BOARD_TPM_BASEADDR, kTPM_SystemClock);
    while (1)
    {
        /* Delays to see the change of LED brightness. */
        delay();

        if (brightnessUp)
        {
            /* Increases a duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases a duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }
```

```
        /* Starts PWM mode with an updated duty cycle. */
        TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
      tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
                               updatedDutycycle);
        TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
      tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
                               updatedDutycycle);
    }
}
```

Data Structures

- struct tpm_chnl_pwm_signal_param_t
  Options to configure a TPM channel's PWM signal. More...
- struct tpm_dual_edge_capture_param_t
  TPM dual edge capture parameters. More...
- struct tpm_phase_params_t
  TPM quadrature decode phase parameters. More...
- struct tpm_config_t
  TPM config structure. More...

Enumerations

- enum tpm_chnl_t {
  kTPM_Chnl_0 = 0U,
  kTPM_Chnl_1,
  kTPM_Chnl_2,
  kTPM_Chnl_3,
  kTPM_Chnl_4,
  kTPM_Chnl_5,
  kTPM_Chnl_6,
  kTPM_Chnl_7 }
  List of TPM channels.
- enum tpm_pwm_mode_t {
  kTPM_EdgeAlignedPwm = 0U,
  kTPM_CenterAlignedPwm,
  kTPM_CombinedPwm }
  TPM PWM operation modes.
- enum tpm_pwm_level_select_t {
  kTPM_NoPwmSignal = 0U,
  kTPM_LowTrue,
  kTPM_HighTrue }
  TPM PWM output pulse mode: high-true, low-true or no output.
- enum tpm_trigger_select_t
  Trigger options available.
- enum tpm_trigger_source_t {
  kTPM_TriggerSource_External = 0U,
  kTPM_TriggerSource_Internal }
  Trigger source options available.
- enum tpm_output_compare_mode_t {

kTPM_NoOutputSignal = (1U << TPM_CnSC_MSA_SHIFT),
kTPM_ToggleOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_SHIFT)),
kTPM_ClearOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SHIFT)),
kTPM_SetOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (3U << TPM_CnSC_ELSA_SHIFT)),
kTPM_HighPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_SHIFT)),
kTPM_LowPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SHIFT)) }
TPM output compare modes.

- enum tpm_input_capture_edge_t {
kTPM_RisingEdge = (1U << TPM_CnSC_ELSA_SHIFT),
kTPM_FallingEdge = (2U << TPM_CnSC_ELSA_SHIFT),
kTPM_RiseAndFallEdge = (3U << TPM_CnSC_ELSA_SHIFT) }
TPM input capture edge.
- enum tpm_quad_decode_mode_t {
kTPM_QuadPhaseEncode = 0U,
kTPM_QuadCountAndDir }
TPM quadrature decode modes.
- enum tpm_phase_polarity_t {
kTPM_QuadPhaseNormal = 0U,
kTPM_QuadPhaseInvert }
TPM quadrature phase polarities.
- enum tpm_clock_source_t {
kTPM_SystemClock = 1U,
kTPM_ExternalClock }
TPM clock source selection.
- enum tpm_clock_prescale_t {
kTPM_Prescale_Divide_1 = 0U,
kTPM_Prescale_Divide_2,
kTPM_Prescale_Divide_4,
kTPM_Prescale_Divide_8,
kTPM_Prescale_Divide_16,
kTPM_Prescale_Divide_32,
kTPM_Prescale_Divide_64,
kTPM_Prescale_Divide_128 }
TPM prescale value selection for the clock source.
- enum tpm_interrupt_enable_t {

kTPM_Chnl0InterruptEnable = (1U << 0),
kTPM_Chnl1InterruptEnable = (1U << 1),
kTPM_Chnl2InterruptEnable = (1U << 2),
kTPM_Chnl3InterruptEnable = (1U << 3),
kTPM_Chnl4InterruptEnable = (1U << 4),
kTPM_Chnl5InterruptEnable = (1U << 5),
kTPM_Chnl6InterruptEnable = (1U << 6),
kTPM_Chnl7InterruptEnable = (1U << 7),
kTPM_TimeOverflowInterruptEnable = (1U << 8) }
List of TPM interrupts.
- enum tpm_status_flags_t {
kTPM_Chnl0Flag = (1U << 0),
kTPM_Chnl1Flag = (1U << 1),
kTPM_Chnl2Flag = (1U << 2),
kTPM_Chnl3Flag = (1U << 3),
kTPM_Chnl4Flag = (1U << 4),
kTPM_Chnl5Flag = (1U << 5),
kTPM_Chnl6Flag = (1U << 6),
kTPM_Chnl7Flag = (1U << 7),
kTPM_TimeOverflowFlag = (1U << 8) }
List of TPM flags.

Driver version

- #define FSL_TPM_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))
Version 2.0.2.

Initialization and deinitialization

- void TPM_Init (TPM_Type ∗base, const tpm_config_t ∗config)
Ungates the TPM clock and configures the peripheral for basic operation.
- void TPM_Deinit (TPM_Type ∗base)
Stops the counter and gates the TPM clock.
- void TPM_GetDefaultConfig (tpm_config_t ∗config)
Fill in the TPM config struct with the default settings.

Channel mode operations

- status_t TPM_SetupPwm (TPM_Type ∗base, const tpm_chnl_pwm_signal_param_t ∗chnlParams, uint8_t numOfChnls, tpm_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
Configures the PWM signal parameters.
- void TPM_UpdatePwmDutycycle (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_pwm_mode_t currentPwmMode, uint8_t dutyCyclePercent)
Update the duty cycle of an active PWM signal.
- void TPM_UpdateChnlEdgeLevelSelect (TPM_Type ∗base, tpm_chnl_t chnlNumber, uint8_t level)
Update the edge level selection for a channel.
- void TPM_SetupInputCapture (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_input_capture_edge_t captureMode)

Enables capturing an input signal on the channel using the function parameters.
- void TPM_SetupOutputCompare (TPM_Type ∗base, tpm_chnl_t chnlNumber, tpm_output_compare_mode_t compareMode, uint32_t compareValue)
  Configures the TPM to generate timed pulses.
- void TPM_SetupDualEdgeCapture (TPM_Type ∗base, tpm_chnl_t chnlPairNumber, const tpm_dual_edge_capture_param_t ∗edgeParam, uint32_t filterValue)
  Configures the dual edge capture mode of the TPM.
- void TPM_SetupQuadDecode (TPM_Type ∗base, const tpm_phase_params_t ∗phaseAParams, const tpm_phase_params_t ∗phaseBParams, tpm_quad_decode_mode_t quadMode)
  Configures the parameters and activates the quadrature decode mode.

Interrupt Interface

- void TPM_EnableInterrupts (TPM_Type ∗base, uint32_t mask)
  Enables the selected TPM interrupts.
- void TPM_DisableInterrupts (TPM_Type ∗base, uint32_t mask)
  Disables the selected TPM interrupts.
- uint32_t TPM_GetEnabledInterrupts (TPM_Type ∗base)
  Gets the enabled TPM interrupts.

Status Interface

- static uint32_t TPM_GetStatusFlags (TPM_Type ∗base)
  Gets the TPM status flags.
- static void TPM_ClearStatusFlags (TPM_Type ∗base, uint32_t mask)
  Clears the TPM status flags.

Timer Start and Stop

- static void TPM_StartTimer (TPM_Type ∗base, tpm_clock_source_t clockSource)
  Starts the TPM counter.
- static void TPM_StopTimer (TPM_Type ∗base)
  Stops the TPM counter.

38.3 Data Structure Documentation

38.3.1 struct tpm_chnl_pwm_signal_param_t

Data Fields

- tpm_chnl_t chnlNumber
  TPM channel to configure.
- tpm_pwm_level_select_t level
  PWM output active level select.
- uint8_t dutyCyclePercent
  PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...
- uint8_t firstEdgeDelayPercent
  Used only in combined PWM mode to generate asymmetrical PWM.

38.3.1.0.0.46 Field Documentation

38.3.1.0.0.46.1 tpm_chnl_t tpm_chnl_pwm_signal_param_t::chnlNumber

In combined mode (available in some SoC's, this represents the channel pair number

38.3.1.0.0.46.2 uint8_t tpm_chnl_pwm_signal_param_t::dutyCyclePercent

100=always active signal (100% duty cycle)

38.3.1.0.0.46.3 uint8_t tpm_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 percentage of the PWM period

38.3.2 struct tpm_dual_edge_capture_param_t

Note

This mode is available only on some SoC's.

Data Fields

- bool enableSwap
  true: Use channel n+1 input, channel n input is ignored; false: Use channel n input, channel n+1 input is ignored
- tpm_input_capture_edge_t currChanEdgeMode
  Input capture edge select for channel n.
- tpm_input_capture_edge_t nextChanEdgeMode
  Input capture edge select for channel n+1.

38.3.3 struct tpm_phase_params_t

Data Fields

- uint32_t phaseFilterVal
  Filter value, filter is disabled when the value is zero.
- tpm_phase_polarity_t phasePolarity
  Phase polarity.

38.3.4 struct tpm_config_t

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

The config struct can be made const so it resides in flash

Data Fields

- tpm_clock_prescale_t prescale
  Select TPM clock prescale value.
- bool useGlobalTimeBase
  true: Use of an external global time base is enabled; false: disabled
- tpm_trigger_select_t triggerSelect
  Input trigger to use for controlling the counter operation.
- tpm_trigger_source_t triggerSource
  Decides if we use external or internal trigger.
- bool enableDoze
  true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode
- bool enableDebugMode
  true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode
- bool enableReloadOnTrigger
  true: TPM counter is reloaded on trigger; false: TPM counter not reloaded
- bool enableStopOnOverflow
  true: TPM counter stops after overflow; false: TPM counter continues running after overflow
- bool enableStartOnTrigger
  true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately
- bool enablePauseOnTrigger
  true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running

38.3.4.0.0.47 Field Documentation

38.3.4.0.0.47.1 tpm_trigger_source_t tpm_config_t::triggerSource

38.4 Enumeration Type Documentation

38.4.1 enum tpm_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

kTPM_Chnl_0 TPM channel number 0.
kTPM_Chnl_1 TPM channel number 1.
kTPM_Chnl_2 TPM channel number 2.
kTPM_Chnl_3 TPM channel number 3.
kTPM_Chnl_4 TPM channel number 4.
kTPM_Chnl_5 TPM channel number 5.
kTPM_Chnl_6 TPM channel number 6.
kTPM_Chnl_7 TPM channel number 7.

38.4.2 enum tpm_pwm_mode_t

Enumerator

kTPM_EdgeAlignedPwm Edge aligned PWM.
kTPM_CenterAlignedPwm Center aligned PWM.
kTPM_CombinedPwm Combined PWM.

38.4.3 enum tpm_pwm_level_select_t

Enumerator

kTPM_NoPwmSignal No PWM output on pin.
kTPM_LowTrue Low true pulses.
kTPM_HighTrue High true pulses.

38.4.4 enum tpm_trigger_select_t

This is used for both internal & external trigger sources (external option available in certain SoC's)

Note

The actual trigger options available is SoC-specific.

38.4.5 enum tpm_trigger_source_t

Note

This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal triger.

Enumerator

kTPM_TriggerSource_External Use external trigger input.
kTPM_TriggerSource_Internal Use internal trigger.

38.4.6 enum tpm_output_compare_mode_t

Enumerator

kTPM_NoOutputSignal No channel output when counter reaches CnV.

kTPM_ToggleOnMatch Toggle output.
kTPM_ClearOnMatch Clear output.
kTPM_SetOnMatch Set output.
kTPM_HighPulseOutput Pulse output high.
kTPM_LowPulseOutput Pulse output low.

38.4.7 enum tpm_input_capture_edge_t

Enumerator

kTPM_RisingEdge Capture on rising edge only.
kTPM_FallingEdge Capture on falling edge only.
kTPM_RiseAndFallEdge Capture on rising or falling edge.

38.4.8 enum tpm_quad_decode_mode_t

Note

This mode is available only on some SoC's.

Enumerator

kTPM_QuadPhaseEncode Phase A and Phase B encoding mode.
kTPM_QuadCountAndDir Count and direction encoding mode.

38.4.9 enum tpm_phase_polarity_t

Enumerator

kTPM_QuadPhaseNormal Phase input signal is not inverted.
kTPM_QuadPhaseInvert Phase input signal is inverted.

38.4.10 enum tpm_clock_source_t

Enumerator

kTPM_SystemClock System clock.
kTPM_ExternalClock External clock.

38.4.11 enum tpm_clock_prescale_t

Enumerator

kTPM_Prescale_Divide_1 Divide by 1.
kTPM_Prescale_Divide_2 Divide by 2.
kTPM_Prescale_Divide_4 Divide by 4.
kTPM_Prescale_Divide_8 Divide by 8.
kTPM_Prescale_Divide_16 Divide by 16.
kTPM_Prescale_Divide_32 Divide by 32.
kTPM_Prescale_Divide_64 Divide by 64.
kTPM_Prescale_Divide_128 Divide by 128.

38.4.12 enum tpm_interrupt_enable_t

Enumerator

kTPM_Chnl0InterruptEnable Channel 0 interrupt.
kTPM_Chnl1InterruptEnable Channel 1 interrupt.
kTPM_Chnl2InterruptEnable Channel 2 interrupt.
kTPM_Chnl3InterruptEnable Channel 3 interrupt.
kTPM_Chnl4InterruptEnable Channel 4 interrupt.
kTPM_Chnl5InterruptEnable Channel 5 interrupt.
kTPM_Chnl6InterruptEnable Channel 6 interrupt.
kTPM_Chnl7InterruptEnable Channel 7 interrupt.
kTPM_TimeOverflowInterruptEnable Time overflow interrupt.

38.4.13 enum tpm_status_flags_t

Enumerator

kTPM_Chnl0Flag Channel 0 flag.
kTPM_Chnl1Flag Channel 1 flag.
kTPM_Chnl2Flag Channel 2 flag.
kTPM_Chnl3Flag Channel 3 flag.
kTPM_Chnl4Flag Channel 4 flag.
kTPM_Chnl5Flag Channel 5 flag.
kTPM_Chnl6Flag Channel 6 flag.
kTPM_Chnl7Flag Channel 7 flag.
kTPM_TimeOverflowFlag Time overflow flag.

38.5 Function Documentation

38.5.1 void TPM_Init (TPM_Type * *base,* const tpm_config_t * *config*)

Note

This API should be called at the beginning of the application using the TPM driver.

Parameters

base	TPM peripheral base address
config	Pointer to user's TPM config structure.

38.5.2 void TPM_Deinit (TPM_Type ∗ *base*)

Parameters

base	TPM peripheral base address

38.5.3 void TPM_GetDefaultConfig (tpm_config_t ∗ *config*)

The default values are:

```
*     config->prescale = kTPM_Prescale_Divide_1;
*     config->useGlobalTimeBase = false;
*     config->dozeEnable = false;
*     config->dbgMode = false;
*     config->enableReloadOnTrigger = false;
*     config->enableStopOnOverflow = false;
*     config->enableStartOnTrigger = false;
*#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
*     config->enablePauseOnTrigger = false;
*#endif
*     config->triggerSelect = kTPM_Trigger_Select_0;
*#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
*     config->triggerSource = kTPM_TriggerSource_External;
*#endif
*
```

Parameters

config	Pointer to user's TPM config structure.

38.5.4 status_t TPM_SetupPwm (TPM_Type ∗ *base*, const tpm_chnl_pwm_signal_param_t ∗ *chnlParams*, uint8_t *numOfChnls*, tpm_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*)

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

base	TPM peripheral base address
chnlParams	Array of PWM channel parameters to configure the channel(s)
numOfChnls	Number of channels to configure, this should be the size of the array passed in
mode	PWM operation mode, options available in enumeration tpm_pwm_mode_t
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	TPM counter clock in Hz

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

38.5.5 void TPM_UpdatePwmDutycycle (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, tpm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)

Parameters

base	TPM peripheral base address
chnlNumber	The channel number. In combined mode, this represents the channel pair number
currentPwm-Mode	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)

38.5.6 void TPM_UpdateChnlEdgeLevelSelect (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, uint8_t *level*)

Parameters

base	TPM peripheral base address

chnlNumber	The channel number
level	The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field.

38.5.7 void TPM_SetupInputCapture (TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, tpm_input_capture_edge_t *captureMode*)

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

base	TPM peripheral base address
chnlNumber	The channel number
captureMode	Specifies which edge to capture

38.5.8 void TPM_SetupOutputCompare (TPM_Type ∗ *base*, tpm_chnl_t *chnlNumber*, tpm_output_compare_mode_t *compareMode*, uint32_t *compareValue*)

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

Parameters

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

38.5.9 void TPM_SetupDualEdgeCapture (TPM_Type ∗ *base*, tpm_chnl_t *chnlPairNumber*, const tpm_dual_edge_capture_param_t ∗ *edgeParam*, uint32_t *filterValue*)

This function allows to measure a pulse width of the signal on the input of channel of a channel pair. The filter function is disabled if the filterVal argument passed is zero.

Parameters

base	TPM peripheral base address
chnlPair-Number	The TPM 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.

38.5.10 void TPM_SetupQuadDecode (TPM_Type * *base*, const tpm_phase_params_t * *phaseAParams*, const tpm_phase_params_t * *phaseBParams*, tpm_quad_decode_mode_t *quadMode*)

Parameters

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

38.5.11 void TPM_EnableInterrupts (TPM_Type * *base*, uint32_t *mask*)

Parameters

base	TPM peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration tpm_-interrupt_enable_t

38.5.12 void TPM_DisableInterrupts (TPM_Type * *base*, uint32_t *mask*)

Parameters

base	TPM peripheral base address
mask	The interrupts to disable. This is a logical OR of members of the enumeration tpm_-interrupt_enable_t

38.5.13 uint32_t TPM_GetEnabledInterrupts (TPM_Type * *base*)

Parameters

base	TPM peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration tpm_interrupt_enable-_t

38.5.14 static uint32_t TPM_GetStatusFlags (TPM_Type * *base*) [inline], [static]

Parameters

base	TPM peripheral base address

Returns

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

38.5.15 static void TPM_ClearStatusFlags (TPM_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	TPM peripheral base address

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

38.5.16 static void TPM_StartTimer (TPM_Type * *base*, tpm_clock_source_t *clockSource*) [inline], [static]

Parameters

base	TPM peripheral base address
clockSource	TPM clock source; once clock source is set the counter will start running

38.5.17 static void TPM_StopTimer (TPM_Type * *base*) [inline], [static]

Parameters

base	TPM peripheral base address

Chapter 39
TRNG: True Random Number Generator

39.1 Overview

The KSDK provides a peripheral driver for the True Random Number Generator (TRNG) module of Kinetis devices.

The True Random Number Generator is a 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.

39.2 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.

39.3 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");
    }
}
```

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.

39.4 Data Structure Documentation

39.4.1 struct trng_statistical_check_limit_t

Used by trng_config_t.

Data Fields

- uint32_t maximum
  Maximum limit.
- uint32_t minimum
  Minimum limit.

39.4.1.0.0.48 Field Documentation

39.4.1.0.0.48.1 uint32_t trng_statistical_check_limit_t::maximum

39.4.1.0.0.48.2 uint32_t trng_statistical_check_limit_t::minimum

39.4.2 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.

39.4.2.0.0.49 Field Documentation

39.4.2.0.0.49.1 bool trng_config_t::lock

39.4.2.0.0.49.2 trng_clock_mode_t trng_config_t::clockMode

39.4.2.0.0.49.3 trng_ring_osc_div_t trng_config_t::ringOscDiv

39.4.2.0.0.49.4 trng_sample_mode_t trng_config_t::sampleMode

39.4.2.0.0.49.5 uint16_t trng_config_t::entropyDelay

Defines the length (in system clocks) of each Entropy sample taken.

39.4.2.0.0.49.6 uint16_t trng_config_t::sampleSize

Defines the total number of Entropy samples that will be taken during Entropy generation.

39.4.2.0.0.49.7 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.

39.4.2.0.0.49.8 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.

39.4.2.0.0.49.9 uint8_t trng_config_t::longRunMaxLimit

39.4.2.0.0.49.10 trng_statistical_check_limit_t trng_config_t::monobitLimit

39.4.2.0.0.49.11 trng_statistical_check_limit_t trng_config_t::runBit1Limit

39.4.2.0.0.49.12 trng_statistical_check_limit_t trng_config_t::runBit2Limit

39.4.2.0.0.49.13 trng_statistical_check_limit_t trng_config_t::runBit3Limit

39.4.2.0.0.49.14 trng_statistical_check_limit_t trng_config_t::runBit4Limit

39.4.2.0.0.49.15 trng_statistical_check_limit_t trng_config_t::runBit5Limit

39.4.2.0.0.49.16 trng_statistical_check_limit_t trng_config_t::runBit6PlusLimit

39.4.2.0.0.49.17 trng_statistical_check_limit_t trng_config_t::pokerLimit

39.4.2.0.0.49.18 trng_statistical_check_limit_t trng_config_t::frequencyCountLimit

39.5 Macro Definition Documentation

39.5.1 #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

39.6 Enumeration Type Documentation

39.6.1 enum trng_sample_mode_t

Used by trng_config_t.

Enumerator

kTRNG_SampleModeVonNeumann Use von Neumann data in both Entropy shifter and Statistical Checker.

kTRNG_SampleModeRaw Use raw data into both Entropy shifter and Statistical Checker.

kTRNG_SampleModeVonNeumannRaw Use von Neumann data in Entropy shifter. Use raw data into Statistical Checker.

39.6.2 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.

39.6.3 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.

39.7 Function Documentation

39.7.1 status_t TRNG_GetDefaultConfig (trng_config_t * *userConfig*)

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

```
*     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.

39.7.2 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 the initialization configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

39.7.3 void TRNG_Deinit (TRNG_Type ∗ *base*)

This function shuts down the TRNG.

Parameters

base	TRNG base address.

39.7.4 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 40
VREF: Voltage Reference Driver

40.1 Overview

The KSDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of Kinetis devices.

The Voltage Reference(VREF) supplies an accurate 1.2 V voltage output that can be trimmed in 0.5 mV steps. VREF can be used in applications to provide a reference voltage to external devices and to internal analog peripherals, such as the ADC, DAC, or CMP. The voltage reference has operating modes that provide different levels of supply rejection and power consumption.

To configure the VREF driver, configure vref_config_t structure in one of two ways.

1. Use the VREF_GetDefaultConfig() function.
2. Set the parameter in the vref_config_t structure.

To initialize the VREF driver, call the VREF_Init() function and pass a pointer to the vref_config_t structure.

To de-initialize the VREF driver, call the VREF_Deinit() function.

40.2 Typical use case and example

This example shows how to generate a reference voltage by using the VREF module.

```
vref_config_t vrefUserConfig;
VREF_GetDefaultConfig(&vrefUserConfig); /* Gets a default configuration. */
VREF_Init(VREF, &vrefUserConfig);       /* Initializes and configures the VREF module */

/* Do something */

VREF_Deinit(VREF); /* De-initializes the VREF module */
```

Data Structures

- struct vref_config_t
  The description structure for the VREF module. More...

Enumerations

- enum vref_buffer_mode_t {
  kVREF_ModeBandgapOnly = 0U,
  kVREF_ModeHighPowerBuffer = 1U,
  kVREF_ModeLowPowerBuffer = 2U }
  VREF modes.

Driver version

- #define FSL_VREF_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  Version 2.1.0.

VREF functional operation

- void VREF_Init (VREF_Type ∗base, const vref_config_t ∗config)
  Enables the clock gate and configures the VREF module according to the configuration structure.
- void VREF_Deinit (VREF_Type ∗base)
  Stops and disables the clock for the VREF module.
- void VREF_GetDefaultConfig (vref_config_t ∗config)
  Initializes the VREF configuration structure.
- void VREF_SetTrimVal (VREF_Type ∗base, uint8_t trimValue)
  Sets a TRIM value for the reference voltage.
- static uint8_t VREF_GetTrimVal (VREF_Type ∗base)
  Reads the value of the TRIM meaning output voltage.

40.3 Data Structure Documentation

40.3.1 struct vref_config_t

Data Fields

- vref_buffer_mode_t bufferMode
  Buffer mode selection.

40.4 Macro Definition Documentation

40.4.1 #define FSL_VREF_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

40.5 Enumeration Type Documentation

40.5.1 enum vref_buffer_mode_t

Enumerator

kVREF_ModeBandgapOnly Bandgap on only, for stabilization and startup.
kVREF_ModeHighPowerBuffer High-power buffer mode enabled.
kVREF_ModeLowPowerBuffer Low-power buffer mode enabled.

40.6 Function Documentation

40.6.1 void VREF_Init (VREF_Type ∗ *base*, const vref_config_t ∗ *config*)

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up vref_config_t parameters and how to call the VREF_Init function by passing in these parameters. This is an example.

```
*   vref_config_t vrefConfig;
*   vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
*   vrefConfig.enableExternalVoltRef = false;
*   vrefConfig.enableLowRef = false;
*   VREF_Init(VREF, &vrefConfig);
*
```

Parameters

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

40.6.2 void VREF_Deinit (VREF_Type ∗ *base*)

This function should be called to shut down the module. This is an example.

```
*   vref_config_t vrefUserConfig;
*   VREF_Init(VREF);
*   VREF_GetDefaultConfig(&vrefUserConfig);
*   ...
*   VREF_Deinit(VREF);
*
```

Parameters

base	VREF peripheral address.

40.6.3 void VREF_GetDefaultConfig (vref_config_t ∗ *config*)

This function initializes the VREF configuration structure to default values. This is an example.

```
*   vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
*   vrefConfig->enableExternalVoltRef = false;
*   vrefConfig->enableLowRef = false;
*
```

Parameters

config	Pointer to the initialization structure.

40.6.4 void VREF_SetTrimVal (VREF_Type * *base*, uint8_t *trimValue*)

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

Parameters

base	VREF peripheral address.
trimValue	Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

40.6.5 static uint8_t VREF_GetTrimVal (VREF_Type * *base*) [inline], [static]

This function gets the TRIM value from the TRM register.

Parameters

base	VREF peripheral address.

Returns

Six-bit value of trim setting.

Chapter 41
WDOG: Watchdog Timer Driver

41.1 Overview

The KSDK provides a peripheral driver for the Watchdog module (WDOG) of Kinetis devices.

41.2 Typical use case

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG_Init(wdog_base,&config);
```

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 the WDOG configuration 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 the 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)
  Enables the WDOG interrupt.
- static void WDOG_DisableInterrupts (WDOG_Type ∗base, uint32_t mask)
  Disables the WDOG interrupt.
- uint32_t WDOG_GetStatusFlags (WDOG_Type ∗base)
  Gets the WDOG all status flags.
- void WDOG_ClearStatusFlags (WDOG_Type ∗base, uint32_t mask)
  Clears the WDOG flag.
- static void WDOG_SetTimeoutValue (WDOG_Type ∗base, uint32_t timeoutCount)
  Sets 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.

41.3 Data Structure Documentation

41.3.1 struct wdog_work_mode_t

Data Fields

- bool enableWait
  Enables or disables WDOG in wait mode.
- bool enableStop
  Enables or disables WDOG in stop mode.
- bool enableDebug
  Enables or disables WDOG in debug mode.

41.3.2 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.

41.3.3 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.

41.4 Macro Definition Documentation

41.4.1 #define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

41.5 Enumeration Type Documentation

41.5.1 enum wdog_clock_source_t

Enumerator

kWDOG_LpoClockSource WDOG clock sourced from LPO.
kWDOG_AlternateClockSource WDOG clock sourced from alternate clock source.

41.5.2 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.

41.5.3 enum wdog_test_mode_t

Enumerator

kWDOG_QuickTest Selects quick test.
kWDOG_ByteTest Selects byte test.

41.5.4 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.

41.5.5 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.

41.5.6 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.

41.6 Function Documentation

41.6.1 void WDOG_GetDefaultConfig (wdog_config_t * *config*)

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

```
*   wdogConfig->enableWdog = true;
*   wdogConfig->clockSource = kWDOG_LpoClockSource;
*   wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
*   wdogConfig->workMode.enableWait = true;
*   wdogConfig->workMode.enableStop = false;
*   wdogConfig->workMode.enableDebug = false;
*   wdogConfig->enableUpdate = true;
*   wdogConfig->enableInterrupt = false;
*   wdogConfig->enableWindowMode = false;
*   wdogConfig->windowValue = 0;
*   wdogConfig->timeoutValue = 0xFFFFU;
*
```

Parameters

config	Pointer to the WDOG configuration structure.

See Also

wdog_config_t

41.6.2 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. To reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in the configuration.

This is an example.

```
*   wdog_config_t config;
*   WDOG_GetDefaultConfig(&config);
*   config.timeoutValue = 0x7ffU;
*   config.enableUpdate = true;
*   WDOG_Init(wdog_base,&config);
*
```

Parameters

base	WDOG peripheral base address
config	The configuration of WDOG

41.6.3 void WDOG_Deinit (WDOG_Type ∗ *base*)

This function shuts down the WDOG. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which indicates that the register update is enabled.

41.6.4 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. Ensure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

This is an example.

```
*   wdog_test_config_t test_config;
*   test_config.testMode = kWDOG_QuickTest;
*   test_config.timeoutValue = 0xfffffu;
*   WDOG_SetTestModeConfig(wdog_base, &test_config);
*
```

Parameters

base	WDOG peripheral base address
config	The functional test configuration of WDOG

41.6.5 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

41.6.6 static void WDOG_Disable (WDOG_Type * *base*) [inline], [static]

This function writes a value into the WDOG_STCTRLH register to disable the WDOG. It is a write-once register. Ensure that the WCT window is still open and that register has not been written to in this WCT while the function is called.

Parameters

base	WDOG peripheral base address

41.6.7 static void WDOG_EnableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function writes a value into the WDOG_STCTRLH register to enable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to enable The parameter can be combination of the following source if defined. • kWDOG_InterruptEnable

41.6.8 static void WDOG_DisableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function writes a value into the WDOG_STCTRLH register to disable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to disable The parameter can be combination of the following source if defined. • kWDOG_InterruptEnable

41.6.9 uint32_t WDOG_GetStatusFlags (WDOG_Type * *base*)

This function gets all status flags.

This is an example for getting the Running Flag.

```
*   uint32_t status;
*   status = WDOG_GetStatusFlags (wdog_base) &
      kWDOG_RunningFlag;
*
```

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.

41.6.10 void WDOG_ClearStatusFlags (WDOG_Type * *base,* uint32_t *mask*)

This function clears the WDOG status flag.

This is an example for clearing the timeout (interrupt) flag.

```
*   WDOG_ClearStatusFlags(wdog_base,kWDOG_TimeoutFlag);
*
```

Parameters

base	WDOG peripheral base address
mask	The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag

41.6.11 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 writes a value into WDOG_TOVALH and WDOG_TOVALL registers which are wirte-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

base	WDOG peripheral base address
timeoutCount	WDOG timeout value; count of WDOG clock tick.

41.6.12 static void WDOG_SetWindowValue (WDOG_Type ∗ *base,* uint32_t *windowValue*) [inline], [static]

This function sets the WDOG window value. This function writes a value into WDOG_WINH and W-DOG_WINL registers which are wirte-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

base	WDOG peripheral base address
windowValue	WDOG window value.

41.6.13 static void WDOG_Unlock (WDOG_Type ∗ *base*) [inline], [static]

This function unlocks the WDOG register written. Before starting the unlock sequence and following con-gfiguration, disable the global interrupts. Otherwise, an interrupt may invalidate the unlocking sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

base	WDOG peripheral base address

41.6.14 void WDOG_Refresh (WDOG_Type * *base*)

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

base	WDOG peripheral base address

41.6.15 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.

41.6.16 static void WDOG_ClearResetCount (WDOG_Type * *base*) [inline], [static]

This function clears the WDOG reset count value.

Parameters

base	WDOG peripheral base address

Chapter 42
Clock Driver

42.1 Overview

The KSDK provides APIs for Kinetis devices clock operation.

42.2 Get frequency

A centralized function CLOCK_GetFreq gets different clock type frequencies by passing a clock name. For example, pass a kCLOCK_CoreSysClk to get the core clock and pass a kCLOCK_BusClk to get the bus clock. Additionally, there are separate functions to get the frequency. For example, use CLOCK-_GetCoreSysClkFreq to get the core clock frequency and CLOCK_GetBusClkFreq to get the bus clock frequency. Using these functions reduces the image size.

42.3 External clock frequency

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

The upper layer must set these values correctly. For example, after OSC0(SYSOSC) is initialized using CLOCK_InitOsc0 or CLOCK_InitSysOsc, the upper layer should call the CLOCK_SetXtal0Freq. Otherwise, the clock frequency get functions may not receive valid values. This is useful for multicore platforms where only one core calls CLOCK_InitOsc0 to initialize OSC0 and other cores call CLOCK_SetXtal0-Freq.

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 FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0
  Configure whether driver controls clock.
- #define MCG_INTERNAL_IRC_48M 48000000U
  IRC48M clock frequency in Hz.
- #define DMAMUX_CLOCKS
  Clock ip name array for DMAMUX.
- #define RTC_CLOCKS
  Clock ip name array for RTC.
- #define SAI_CLOCKS
  Clock ip name array for SAI.
- #define PORT_CLOCKS
  Clock ip name array for PORT.
- #define FLEXBUS_CLOCKS
  Clock ip name array for FLEXBUS.
- #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 EMVSIM_CLOCKS
  Clock ip name array for EMVSIM.
- #define QSPI_CLOCKS
  Clock ip name array for QSPI.
- #define LPTMR_CLOCKS
  Clock ip name array for LPTMR.
- #define SDHC_CLOCKS
  Clock ip name array for SDHC.
- #define FTM_CLOCKS
  Clock ip name array for FTM.
- #define EDMA_CLOCKS
  Clock ip name array for EDMA.
- #define LPUART_CLOCKS
  Clock ip name array for LPUART.
- #define DAC_CLOCKS
  Clock ip name array for DAC.
- #define ADC16_CLOCKS
  Clock ip name array for ADC16.
- #define SDRAM_CLOCKS
  Clock ip name array for SDRAM.
- #define TRNG_CLOCKS
  Clock ip name array for TRNG.
- #define MPU_CLOCKS
  Clock ip name array for MPU.
- #define FLEXIO_CLOCKS

Clock ip name array for FLEXIO.
- #define VREF_CLOCKS
  Clock ip name array for VREF.
- #define CMT_CLOCKS
  Clock ip name array for CMT.
- #define TPM_CLOCKS
  Clock ip name array for TPM.
- #define TSI_CLOCKS
  Clock ip name array for TSI.
- #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 FTF_CLOCKS
  Clock ip name array for FTF.
- #define CMP_CLOCKS
  Clock ip name array for CMP.
- #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_usb_src_t {
  kCLOCK_UsbSrcPll0 = SIM_SOPT2_USBSRC(1U) | SIM_SOPT2_PLLFLLSEL(1U),
  kCLOCK_UsbSrcIrc48M = SIM_SOPT2_USBSRC(1U) | SIM_SOPT2_PLLFLLSEL(3U),
  kCLOCK_UsbSrcExt = SIM_SOPT2_USBSRC(0U) }
  USB clock source definition.
- 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,
kMCG_OscselIrc }
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_RtcOscLostFlag = (1U << 4U),
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_SetSdhc0Clock (uint32_t src)
  Set SDHC0 clock source.
- static void CLOCK_SetEmvsimClock (uint32_t src)
  Set EMVSIM clock source.
- static void CLOCK_SetLpuartClock (uint32_t src)
  Set LPUART clock source.
- static void CLOCK_SetTpmClock (uint32_t src)
  Set TPM clock source.
- static void CLOCK_SetFlexio0Clock (uint32_t src)
  Set FLEXIO 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, uint32_t divValue, uint32_t fracValue)
  Set PLLFLLSEL clock source.
- static void CLOCK_SetClkOutClock (uint32_t src)
  Set CLKOUT source.
- static void CLOCK_SetRtcClkOutClock (uint32_t src)
  Set RTC_CLKOUT source.
- bool CLOCK_EnableUsbfs0Clock (clock_usb_src_t src, uint32_t freq)
  Enable USB FS clock.
- static void CLOCK_DisableUsbfs0Clock (void)
  Disable USB FS clock.
- 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_GetPlatClkFreq (void)
  Get the platform clock frequency.

- uint32_t CLOCK_GetBusClkFreq (void)
  Get the bus clock frequency.
- uint32_t CLOCK_GetFlexBusClkFreq (void)
  Get the flexbus 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.
- static void CLOCK_SetFllExtRefDiv (uint8_t frdiv)
  Set the FLL external reference clock divider value.
- 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_SetRtcOscMonitorMode (mcg_monitor_mode_t mode)
  Sets the RTC OSC 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.

42.4 Data Structure Documentation

42.4.1 struct sim_clock_config_t

Data Fields

- uint8_t pllFllSel
  PLL/FLL/IRC48M selection.

- uint8_t pllFllDiv
  PLLFLLSEL clock divider divisor.
- uint8_t pllFllFrac
  PLLFLLSEL clock divider fraction.
- uint8_t er32kSrc
  ERCLK32K source selection.
- uint32_t clkdiv1
  SIM_CLKDIV1.

42.4.1.0.0.50 Field Documentation

42.4.1.0.0.50.1 uint8_t sim_clock_config_t::pllFllSel

42.4.1.0.0.50.2 uint8_t sim_clock_config_t::pllFllDiv

42.4.1.0.0.50.3 uint8_t sim_clock_config_t::pllFllFrac

42.4.1.0.0.50.4 uint8_t sim_clock_config_t::er32kSrc

42.4.1.0.0.50.5 uint32_t sim_clock_config_t::clkdiv1

42.4.2 struct oscer_config_t

Data Fields

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

42.4.2.0.0.51 Field Documentation

42.4.2.0.0.51.1 uint8_t oscer_config_t::enableMode

OR'ed value of _oscer_enable_mode.

42.4.2.0.0.51.2 uint8_t oscer_config_t::erclkDiv

42.4.3 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.

42.4.3.0.0.52 Field Documentation

42.4.3.0.0.52.1 uint32_t osc_config_t::freq

42.4.3.0.0.52.2 uint8_t osc_config_t::capLoad

42.4.3.0.0.52.3 osc_mode_t osc_config_t::workMode

42.4.3.0.0.52.4 oscer_config_t osc_config_t::oscerConfig

42.4.4 struct mcg_pll_config_t

Data Fields

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

42.4.4.0.0.53 Field Documentation

42.4.4.0.0.53.1 uint8_t mcg_pll_config_t::enableMode

OR'ed value of _mcg_pll_enable_mode.

42.4.4.0.0.53.2 uint8_t mcg_pll_config_t::prdiv

42.4.4.0.0.53.3 uint8_t mcg_pll_config_t::vdiv

42.4.5 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_oscsel_t oscsel
  OSC select MCG_C7[OSCSEL].
- mcg_pll_config_t pll0Config
  MCGPLL0CLK configuration.

42.4.5.0.0.54 Field Documentation

42.4.5.0.0.54.1 mcg_mode_t mcg_config_t::mcgMode

42.4.5.0.0.54.2 uint8_t mcg_config_t::irclkEnableMode

42.4.5.0.0.54.3 mcg_irc_mode_t mcg_config_t::ircs

42.4.5.0.0.54.4 uint8_t mcg_config_t::fcrdiv

42.4.5.0.0.54.5 uint8_t mcg_config_t::frdiv

42.4.5.0.0.54.6 mcg_drs_t mcg_config_t::drs

42.4.5.0.0.54.7 mcg_dmx32_t mcg_config_t::dmx32

42.4.5.0.0.54.8 mcg_oscsel_t mcg_config_t::oscsel

42.4.5.0.0.54.9 mcg_pll_config_t mcg_config_t::pll0Config

42.5 Macro Definition Documentation

42.5.1 #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0

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

Note

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

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

42.5.3 #define MCG_INTERNAL_IRC_48M 48000000U

42.5.4 #define DMAMUX_CLOCKS

Value:

```
{                  \
        kCLOCK_Dmamux0 \
    }
```

42.5.5 #define RTC_CLOCKS

Value:

```
{               \
        kCLOCK_Rtc0 \
    }
```

42.5.6 #define SAI_CLOCKS

Value:

```
{               \
        kCLOCK_Sai0 \
    }
```

42.5.7 #define PORT_CLOCKS

Value:

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

42.5.8 #define FLEXBUS_CLOCKS

Value:

```
{                   \
        kCLOCK_Flexbus0 \
    }
```

42.5.9 #define EWM_CLOCKS

Value:

```
{               \
        kCLOCK_Ewm0 \
    }
```

42.5.10 #define PIT_CLOCKS

Value:

```
{               \
        kCLOCK_Pit0 \
    }
```

42.5.11 #define DSPI_CLOCKS

Value:

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

42.5.12 #define EMVSIM_CLOCKS

Value:

```
{                                  \
        kCLOCK_Emvsim0, kCLOCK_Emvsim1 \
    }
```

42.5.13 #define QSPI_CLOCKS

Value:

```
{                \
        kCLOCK_Qspi0 \
    }
```

42.5.14 #define LPTMR_CLOCKS

Value:

```
{                               \
        kCLOCK_Lptmr0, kCLOCK_Lptmr1 \
    }
```

42.5.15 #define SDHC_CLOCKS

Value:

```
{                \
        kCLOCK_Sdhc0 \
    }
```

42.5.16 #define FTM_CLOCKS

Value:

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

42.5.17 #define EDMA_CLOCKS

Value:

```
{               \
        kCLOCK_Dma0 \
    }
```

42.5.18 #define LPUART_CLOCKS

Value:

```
{                                                                                    \
        kCLOCK_Lpuart0, kCLOCK_Lpuart1, kCLOCK_Lpuart2, kCLOCK_Lpuart3, kCLOCK_Lpuart4 \
    }
```

42.5.19 #define DAC_CLOCKS

Value:

```
{               \
        kCLOCK_Dac0 \
    }
```

42.5.20 #define ADC16_CLOCKS

Value:

```
{                \
        kCLOCK_Adc0  \
    }
```

42.5.21 #define SDRAM_CLOCKS

Value:

```
{                  \
        kCLOCK_Sdramc0 \
    }
```

42.5.22 #define TRNG_CLOCKS

Value:

```
{                \
        kCLOCK_Trng0 \
    }
```

42.5.23 #define MPU_CLOCKS

Value:

```
{                \
        kCLOCK_Mpu0 \
    }
```

42.5.24 #define FLEXIO_CLOCKS

Value:

```
{                   \
        kCLOCK_Flexio0 \
    }
```

42.5.25 #define VREF_CLOCKS

Value:

```
{                 \
        kCLOCK_Vref0 \
    }
```

42.5.26 #define CMT_CLOCKS

Value:

```
{                \
        kCLOCK_Cmt0 \
    }
```

42.5.27 #define TPM_CLOCKS

Value:

```
{                                              \
        kCLOCK_IpInvalid, kCLOCK_Tpm1, kCLOCK_Tpm2 \
    }
```

42.5.28 #define TSI_CLOCKS

Value:

```
{            \
        kCLOCK_Tsi0 \
    }
```

42.5.29 #define CRC_CLOCKS

Value:

```
{            \
        kCLOCK_Crc0 \
    }
```

42.5.30 #define I2C_CLOCKS

Value:

```
{                                                    \
        kCLOCK_I2c0, kCLOCK_I2c1, kCLOCK_I2c2, kCLOCK_I2c3 \
    }
```

42.5.31 #define PDB_CLOCKS

Value:

```
{            \
        kCLOCK_Pdb0 \
    }
```

42.5.32 #define FTF_CLOCKS

Value:

```
{            \
        kCLOCK_Ftf0 \
    }
```

42.5.33 #define CMP_CLOCKS

Value:

```
{                          \
        kCLOCK_Cmp0, kCLOCK_Cmp1 \
    }
```

42.5.34 #define SYS_CLK kCLOCK_CoreSysClk

42.6 Enumeration Type Documentation

42.6.1 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.

42.6.2 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcPll0 Use PLL0.
kCLOCK_UsbSrcIrc48M Use IRC48M.
kCLOCK_UsbSrcExt Use USB_CLKIN.

42.6.3 enum clock_ip_name_t

42.6.4 enum osc_mode_t

Enumerator

kOSC_ModeExt Use an external clock.
kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

42.6.5 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

42.6.6 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.
kOSC_ErClkEnableInStop Enable in stop mode.

42.6.7 enum mcg_fll_src_t

Enumerator

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

42.6.8 enum mcg_irc_mode_t

Enumerator

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

42.6.9 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.

42.6.10 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.

42.6.11 enum mcg_pll_ref_src_t

Enumerator

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

42.6.12 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.

42.6.13 enum mcg_atm_select_t

Enumerator

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

42.6.14 enum mcg_oscsel_t

Enumerator

kMCG_OscselOsc Selects System Oscillator (OSCCLK)
kMCG_OscselRtc Selects 32 kHz RTC Oscillator.
kMCG_OscselIrc Selects 48 MHz IRC Oscillator.

42.6.15 enum mcg_pll_clk_select_t

Enumerator

kMCG_PllClkSelPll0 PLL0 output clock is selected.

42.6.16 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.

42.6.17 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.

42.6.18 enum _mcg_status_flags_t

Enumerator

kMCG_Osc0LostFlag OSC0 lost.
kMCG_Osc0InitFlag OSC0 crystal initialized.

kMCG_RtcOscLostFlag RTC OSC lost.
kMCG_Pll0LostFlag PLL0 lost.
kMCG_Pll0LockFlag PLL0 locked.

42.6.19 enum _mcg_irclk_enable_mode

Enumerator

kMCG_IrclkEnable MCGIRCLK enable.
kMCG_IrclkEnableInStop MCGIRCLK enable in stop mode.

42.6.20 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.

42.6.21 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.

42.7 Function Documentation

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

Parameters

name	Which clock to enable, see clock_ip_name_t.

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

Parameters

name	Which clock to disable, see clock_ip_name_t.

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

Parameters

src	The value to set ERCLK32K clock source.

42.7.4 static void CLOCK_SetSdhc0Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set SDHC0 clock source.

42.7.5 static void CLOCK_SetEmvsimClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set EMVSIM clock source.

42.7.6 static void CLOCK_SetLpuartClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set LPUART clock source.

42.7.7 static void CLOCK_SetTpmClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set TPM clock source.

42.7.8 static void CLOCK_SetFlexio0Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set FLEXIO clock source.

42.7.9 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.

42.7.10 static void CLOCK_SetPllFllSelClock (uint32_t *src*, uint32_t *divValue*, uint32_t *fracValue*) [inline], [static]

Parameters

src	The value to set PLLFLLSEL clock source.

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

Parameters

src	The value to set CLKOUT source.

42.7.12 static void CLOCK_SetRtcClkOutClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set RTC_CLKOUT source.

42.7.13 bool CLOCK_EnableUsbfs0Clock (clock_usb_src_t *src*, uint32_t *freq*)

Parameters

src	USB FS clock source.
freq	The frequency specified by src.

Return values

true	The clock is set successfully.
false	The clock source is invalid to get proper USB FS clock.

42.7.14 static void CLOCK_DisableUsbfs0Clock (void) [inline], [static]

Disable USB FS clock.

42.7.15 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.

42.7.16 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

42.7.17 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Clock frequency in Hz.

42.7.18 uint32_t CLOCK_GetPlatClkFreq (void)

Returns

Clock frequency in Hz.

42.7.19 uint32_t CLOCK_GetBusClkFreq (void)

Returns

Clock frequency in Hz.

42.7.20 uint32_t CLOCK_GetFlexBusClkFreq (void)

Returns

Clock frequency in Hz.

42.7.21 uint32_t CLOCK_GetFlashClkFreq (void)

Returns

Clock frequency in Hz.

42.7.22 uint32_t CLOCK_GetPllFllSelClkFreq (void)

Returns

Clock frequency in Hz.

42.7.23 uint32_t CLOCK_GetEr32kClkFreq (void)

Returns

Clock frequency in Hz.

42.7.24 uint32_t CLOCK_GetOsc0ErClkUndivFreq (void)

Returns

Clock frequency in Hz.

42.7.25 uint32_t CLOCK_GetOsc0ErClkFreq (void)

Returns

Clock frequency in Hz.

42.7.26 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.

42.7.27 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.

42.7.28 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.

42.7.29 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.

42.7.30 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.

42.7.31 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.

42.7.32 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.

42.7.33 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.

42.7.34 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.

42.7.35 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.

42.7.36 static void CLOCK_SetFllExtRefDiv (uint8_t *frdiv*) `[inline]`, `[static]`

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

Parameters

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

42.7.37 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.

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

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

42.7.39 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.

42.7.40 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.

42.7.41 void CLOCK_SetRtcOscMonitorMode (mcg_monitor_mode_t *mode*)

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

Parameters

mode	Monitor mode to set.

42.7.42 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.

42.7.43 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.

42.7.44 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.

42.7.45 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.

42.7.46 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);
```

42.7.47 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.

42.7.48 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

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

Parameters

freq	The XTAL0/EXTAL0 input clock frequency in Hz.

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

Parameters

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

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

42.7.52 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.

42.7.53 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.

42.7.54 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.

42.7.55 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.

42.7.56 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.

42.7.57 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.

42.7.58 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.

42.7.59 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);

42.7.60 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.

42.7.61 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.

42.7.62 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.

42.7.63 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.

42.7.64 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.

42.7.65 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_SourceUsed	Could not change MCGIRCLK setting.
kStatus_Success	Switched to the target mode successfully.

42.7.66 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_ModeUnreachable	Could not switch to the target mode.
kStatus_Success	Switched to the target mode successfully.

42.7.67 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.

42.7.68 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.

42.8 Variable Documentation

42.8.1 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.

42.8.2 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.

42.9 Multipurpose Clock Generator (MCG)

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

42.9.1 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.

42.9.1.1 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 C-LOCK_GetOutClkFreq(), CLOCK_GetInternalRefClkFreq(), CLOCK_GetFixedFreqClkFreq(), CLOC-K_GetFllFreq(), CLOCK_GetPll0Freq(), CLOCK_GetPll1Freq(), and CLOCK_GetExtPllFreq(). These functions get the clock frequency based on the current MCG registers.

42.9.1.2 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 M-CGPLLCLK 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.

42.9.1.3 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.

42.9.1.4 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.

42.9.1.5 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.

42.9.1.6 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 use case.

42.9.2 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. Enable the corresponding clock before using it as a clock source.

42.9.2.1 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)

42.9.2.2 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)

42.9.2.3 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)

42.9.2.4 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)

42.9.2.5 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)

42.9.2.6 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 43
Debug Console

43.1 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.

43.2 Function groups

43.2.1 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);
```

43.2.2 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 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. In other words, 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 */
```

43.3 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

43.4 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 can 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.

43.4.1 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.

43.4.2 Guide Semihosting for Keil µVision

NOTE: Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

Step 1: Prepare code

Remove function fputc and fgetc is used to support KEIL in "fsl_debug_console.c" and 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. Select "Target" tab and not select "Use MicroLIB".
3. Select “Debug” tab, select “J-Link/J-Trace Cortex” and click “Setting button”.
4. 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.

43.4.3 Guide Semihosting for KDS

NOTE: After the setting 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 clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

43.4.4 Guide Semihosting for ATL

NOTE: J-Link has to be used to enable semihosting.

Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
  /* Implement your write code here. This is used by puts and printf. */
  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 J-Link".
2. In tab "Debugger" set up as follows.
    - JTAG mode must be selected

- SWV tracing must be enabled
- Enter the Core Clock frequency, which is hardware 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. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a 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 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 to enable SWV trace recoding. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, 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.

43.4.5 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". Set up as follows.
   - "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 "${CMAKE_EXE_LINKER_FLA-GS_RELEASE} –specs=nano.specs")"
   to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_R-ELEASE} –specs=rdimon.specs")"
   Replace paragraph
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -fno-common")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -ffunction-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -fdata-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -ffreestanding")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -fno-builtin")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -mthumb")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -mapcs")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} --gc-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -static")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -z")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-G} muldefs")
   To
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-

G} --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) After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 44
Notification Framework

44.1 Overview

This section describes the programming interface of the Notifier driver.

44.2 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.

These are the steps for the configuration transition.

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 a "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 switches 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 example shows how 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, which indicates 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)
  Creates a Notifier handle.
- status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)
  Switches the configuration according to a pre-defined structure.
- uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)
  This function returns the last failed notification callback.

44.3 Data Structure Documentation

44.3.1 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.

44.3.1.0.0.55 Field Documentation

44.3.1.0.0.55.1 notifier_user_config_t∗ notifier_notification_block_t::targetConfig

44.3.1.0.0.55.2 notifier_policy_t notifier_notification_block_t::policy

44.3.1.0.0.55.3 notifier_notification_type_t notifier_notification_block_t::notifyType

44.3.2 struct notifier_callback_config_t

This structure holds the configuration of callbacks. Callbacks of this type are expected to be statically allocated. This structure contains the 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.

44.3.2.0.0.56 Field Documentation

44.3.2.0.0.56.1 notifier_callback_t notifier_callback_config_t::callback

44.3.2.0.0.56.2 notifier_callback_type_t notifier_callback_config_t::callbackType

44.3.2.0.0.56.3 void∗ notifier_callback_config_t::callbackData

44.3.3 struct notifier_handle_t

Notifier handle structure. Contains data necessary for the 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.

44.3.3.0.0.57 Field Documentation

44.3.3.0.0.57.1 notifier_user_config_t∗∗ notifier_handle_t::configsTable

44.3.3.0.0.57.2 uint8_t notifier_handle_t::configsNumber

44.3.3.0.0.57.3 notifier_callback_config_t∗ notifier_handle_t::callbacksTable

44.3.3.0.0.57.4 uint8_t notifier_handle_t::callbacksNumber

44.3.3.0.0.57.5 uint8_t notifier_handle_t::errorCallbackIndex

44.3.3.0.0.57.6 uint8_t notifier_handle_t::currentConfigIndex

44.3.3.0.0.57.7 notifier_user_function_t notifier_handle_t::userFunction

44.3.3.0.0.57.8 void∗ notifier_handle_t::userData

44.4 Typedef Documentation

44.4.1 typedef void notifier_user_config_t

Reference of the user defined configuration is stored in an array; the notifier switches between these configurations based on this array.

44.4.2 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.

44.4.3 typedef status_t(∗ notifier_callback_t)(notifier_notification_block_t ∗notify, void ∗data)

Declaration of a callback. It is common for registered callbacks. Reference to function of this type is part of the 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, the type of the notification is passed as a parameter along with the reference to the target configuration structure (see notifier_notification_block_t) and any data passed during the callback registration. When notified before the configuration switch, depending on the configuration switch policy (see notifier_policy_t), the callback may deny the execution of the user function by returning an error code different than 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.

44.5 Enumeration Type Documentation

44.5.1 enum _notifier_status

Used as return value of Notifier functions.

Enumerator

kStatus_NOTIFIER_ErrorNotificationBefore An error occurs during send "BEFORE" notification.

kStatus_NOTIFIER_ErrorNotificationAfter An error occurs during send "AFTER" notification.

44.5.2 enum notifier_policy_t

Defines whether the 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 The user function is executed regardless of the results.

44.5.3 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.

44.5.4 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 the 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 an unsuccessful attempt to switch configuration
- After a successful configuration switch

Enumerator

kNOTIFIER_CallbackBefore Callback handles BEFORE notification.

kNOTIFIER_CallbackAfter Callback handles AFTER notification.

kNOTIFIER_CallbackBeforeAfter Callback handles BEFORE and AFTER notification.

44.6 Function Documentation

44.6.1 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 the 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 the callbacks array.
userFunction	User function.
userData	User data passed to user function.

Returns

An error Code or kStatus_Success.

44.6.2 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 an agreement is required, if any callback returns an error code, 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 returns an error code, an error code indicating 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.

44.6.3 uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t * *notifierHandle*)

This function returns an index of the last callback that failed during the configuration switch while the last NOTIFIER_SwitchConfig() was called. If the last NOTIFIER_SwitchConfig() call ended successfully value equal to callbacks number is returned. The returned value represents an index in the array of static call-backs.

Parameters

notifierHandle	Pointer to the notifier handle

Returns

Callback Index of the last failed callback or value equal to callbacks count.

Chapter 45
Shell

45.1 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.

45.2 Function groups

45.2.1 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>> ");
```

45.2.2 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.

45.2.3 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 configures 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.

45.3 Data Structure Documentation

45.3.1 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.

45.3.2 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.

45.3.2.0.0.58 Field Documentation

45.3.2.0.0.58.1 const char* shell_command_context_t::pcCommand

For example "help". It must be all lower case.

45.3.2.0.0.58.2 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".

45.3.2.0.0.58.3 const cmd_function_t shell_command_context_t::pFuncCallBack

45.3.2.0.0.58.4 uint8_t shell_command_context_t::cExpectedNumberOfParameters

45.3.3 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.

45.4 Macro Definition Documentation

45.4.1 #define SHELL_USE_HISTORY (0U)

45.4.2 #define SHELL_SEARCH_IN_HIST (1U)

45.4.3 #define SHELL_USE_FILE_STREAM (0U)

45.4.4 #define SHELL_AUTO_COMPLETE (1U)

45.4.5 #define SHELL_BUFFER_SIZE (64U)

45.4.6 #define SHELL_MAX_ARGS (8U)

45.4.7 #define SHELL_HIST_MAX (3U)

45.4.8 #define SHELL_MAX_CMD (6U)

45.5 Typedef Documentation

45.5.1 typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)

45.5.2 typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)

45.5.3 typedef int(* printf_data_t)(const char *format,...)

45.5.4 typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)

45.6 Enumeration Type Documentation

45.6.1 enum fun_key_status_t

Enumerator

kSHELL_Normal Normal key.
kSHELL_Special Special key.
kSHELL_Function Function key.

45.7 Function Documentation

45.7.1 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. This is an 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

45.7.2 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

45.7.3 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 46
Smartcard_phy_ncn8025_driver

46.1 Overview

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  Smart card definition which specifies the adjustment number of clock cycles during which an ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  Smart card definition which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.
- #define SMARTCARD_NCN8025_STATUS_PRES (0x01u)
  Masks for NCN8025 status register.
- #define SMARTCARD_NCN8025_STATUS_ACTIVE (0x02u)
  Smart card phy NCN8025 Smart card active status.
- #define SMARTCARD_NCN8025_STATUS_FAULTY (0x04u)
  Smart card phy NCN8025 Smart card faulty status.
- #define SMARTCARD_NCN8025_STATUS_CARD_REMOVED (0x08u)
  Smart card phy NCN8025 Smart card removed status.
- #define SMARTCARD_NCN8025_STATUS_CARD_DEACTIVATED (0x10u)
  Smart card phy NCN8025 Smart card deactivated status.

Functions

- void SMARTCARD_PHY_NCN8025_GetDefaultConfig (smartcard_interface_config_t ∗config)
  Fills in the configuration structure with default values.
- status_t SMARTCARD_PHY_NCN8025_Init (void ∗base, smartcard_interface_config_t const ∗config, uint32_t srcClock_Hz)
  Initializes a Smart card interface instance for operation.
- void SMARTCARD_PHY_NCN8025_Deinit (void ∗base, smartcard_interface_config_t ∗config)
  De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.
- status_t SMARTCARD_PHY_NCN8025_Activate (void ∗base, smartcard_context_t ∗context, smartcard_reset_type_t resetType)
  Activates the Smart card IC.
- status_t SMARTCARD_PHY_NCN8025_Deactivate (void ∗base, smartcard_context_t ∗context)
  De-activates the Smart card IC.
- status_t SMARTCARD_PHY_NCN8025_Control (void ∗base, smartcard_context_t ∗context, smartcard_interface_control_t control, uint32_t param)
  Controls the Smart card interface IC.
- void SMARTCARD_PHY_NCN8025_IRQHandler (void ∗base, smartcard_context_t ∗context)
  Smart card interface IC IRQ ISR.

46.2 Macro Definition Documentation

46.2.1 #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMEN-T (4200u)

46.2.2 #define SMARTCARD_NCN8025_STATUS_PRES (0x01u)

Smart card phy NCN8025 Smart card present status

46.3 Function Documentation

46.3.1 void SMARTCARD_PHY_NCN8025_GetDefaultConfig (smartcard_interface_config_t * *config*)

Parameters

config	The Smart card user configuration structure which contains configuration structure of type smartcard_interface_config_t. Function fill in members: clockToResetDelay = 42000, vcc = kSmartcardVoltageClassB3_3V, with default values.

46.3.2 status_t SMARTCARD_PHY_NCN8025_Init (void * *base*, smartcard_interface_config_t const * *config*, uint32_t *srcClock_Hz*)

Parameters

base	The Smart card peripheral base address.
config	The user configuration structure of type smartcard_interface_config_t. Call the function SMARTCARD_PHY_NCN8025_GetDefaultConfig() to fill out the configuration structure.
srcClock_Hz	Smart card clock generation module source clock.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

46.3.3 void SMARTCARD_PHY_NCN8025_Deinit (void * *base*, smartcard_interface_config_t * *config*)

Parameters

base	The Smart card peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t.

46.3.4 status_t SMARTCARD_PHY_NCN8025_Activate (void * *base*, smartcard_context_t * *context*, smartcard_reset_type_t *resetType*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

46.3.5 status_t SMARTCARD_PHY_NCN8025_Deactivate (void * *base*, smartcard_context_t * *context*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

46.3.6 status_t SMARTCARD_PHY_NCN8025_Control (void * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Parameters

base	The Smart card peripheral module base address.
context	A pointer to a Smart card driver context structure.
control	A interface command type.
param	Integer value specific to control type

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

46.3.7 void SMARTCARD_PHY_NCN8025_IRQHandler (void * *base*, smartcard_context_t * *context*)

Parameters

base	The Smart card peripheral module base address.
context	The Smart card context pointer.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

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.

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.

Document Number: KSDK20K80FAPIRM
Rev. 0
Aug 2016