Kinetis SDK v.2.0 API Reference Manual

Freescale Semiconductor, Inc.

Document Number: KSDK20APIRM
Rev. 0
Jan 2016

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Chapter 1
Introduction

The Kinetis Software Development Kit (KSDK) 2.0 is a collection of software enablement, for NXP Kinetis Microcontrollers, that includes peripheral drivers, high level stacks including USB and LWIP, integration with WolfSSL and mbed TLS cryptography libraries, other middleware packages (multicore support, 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
Boards	<install_dir>/boards
Demo applications	<install_dir>/boards/<board_name>/demo_apps
USB demo applications	<install_dir>/boards/<board_name>/usb_examples
Driver examples	<install_dir>/boards/<board_name>/driver_examples
RTOS examples	<install_dir>/boards/<board_name>/rtos_examples
Multicore examples	<install_dir>/boards/<board_name>/multicore_examples
Documentation	<install_dir>/docs
USB Documentation	<install_dir>/doc/usb
lwIP Documentation	<install_dir>/doc/lwip
Middleware	<install_dir>/middleware
lwIP stack	<install_dir>/middleware/lwip_<version>
DMA manager	<install_dir>/middleware/dma_manager_<version>
EMV stack	<install_dir>/middleware/emv_<version>
FatFS stack	<install_dir>/middleware/fatfs_<version>
mmCAU	<install_dir>/middleware/mmcau_<version>
Multicore stack	<install_dir>/middleware/multicore_<version>
SDMMC card driver	<install_dir>/middleware/sdmmc_<version>
USB stack	<install_dir>/middleware/usb_<version>
WolfSSL stack	<install_dir>/middleware/wolfssl_<version>
Driver, SoC header files, extension header files and feature header files, utilities	<install_dir>/devices/<device_name>
Cortex Microcontroller Software Interface Standard (CMSIS) ARM Cortex®-M header files, DSP library source	<install_dir>/CMSIS
Peripheral Drivers	<install_dir>/devices/<device_name>/drivers
Utilities such as debug console	<install_dir>/devices/<device_name>/utilities
RTOS Kernel Code	<install_dir>/rtos
Tools	<install_dir>/tools

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_DMA_Busy = 5000
- kStatus_DSPI_Error = 601
- kStatus_EDMA_QueueFull = 5100
- kStatus_EDMA_Busy = 5101
- kStatus_ENET_RxFrameError = 4000
- kStatus_ENET_RxFrameFail = 4001
- kStatus_ENET_RxFrameEmpty = 4002
- kStatus_ENET_TxFrameBusy = 4003
- kStatus_ENET_TxFrameFail = 4004
- kStatus_ENET_PtpTsRingFull = 4005
- kStatus_ENET_PtpTsRingEmpty = 4006
- kStatus_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_SPI_Busy = 1400

- kStatus_SPI_Idle = 1401
- kStatus_SPI_Error = 1402
- kStatus_TRGMUX_Locked = 4200
- 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

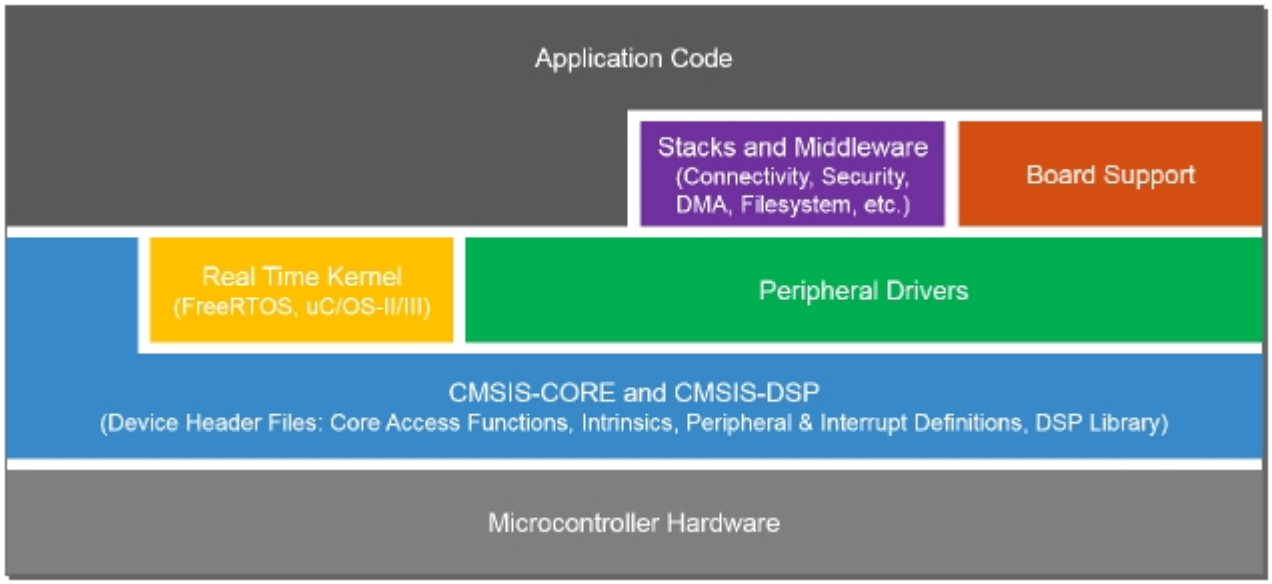

Figure 1: KSDK Block Diagram

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 will ensure 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 deinit 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 will 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 will not meet all customer needs, but provide a baseline for development of custom user APIs.

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

Interrupt handling for transactional APIs

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

```
        PUBWEAK SPI0_IRQHandler
        PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
        LDR     R0, =SPI0_DriverIRQHandler
        BX      R0
```

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

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

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

Feature Header Files

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

Application

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

Chapter 4
Trademarks

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

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

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

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

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

5.1 Overview

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

5.2 Typical use case

5.2.1 Polling Configuration

```
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

5.2.2 Interrupt Configuration

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

Typical use case

```
    // ...

    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 any key in 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 conversion result to clear the conversion completed flag.
        g_Adc16ConversionValue = ADC16_ChannelConversionValue(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP
      );
        g_Adc16InterruptCount++;
    }
```

Files

- file fsl_adc16.h

Data Structures

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

Enumerations

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

Driver version

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

Initialization

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

Advanced Feature

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

Conversion Channel

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

5.3 Data Structure Documentation

5.3.1 struct adc16_config_t

Data Fields

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

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

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource

5.3.1.0.0.1.2 adc16_clock_source_t adc16_config_t::clockSource

5.3.1.0.0.1.3 bool adc16_config_t::enableAsynchronousClock

5.3.1.0.0.1.4 adc16_clock_divider_t adc16_config_t::clockDivider

5.3.1.0.0.1.5 adc16_resolution_t adc16_config_t::resolution

5.3.1.0.0.1.6 adc16_long_sample_mode_t adc16_config_t::longSampleMode

5.3.1.0.0.1.7 bool adc16_config_t::enableHighSpeed

5.3.1.0.0.1.8 bool adc16_config_t::enableLowPower

5.3.1.0.0.1.9 bool adc16_config_t::enableContinuousConversion

5.3.2 struct adc16_hardware_compare_config_t

Data Fields

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

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode

See "adc16_hardware_compare_mode_t".

5.3.2.0.0.2.2 int16_t adc16_hardware_compare_config_t::value1

5.3.2.0.0.2.3 int16_t adc16_hardware_compare_config_t::value2

5.3.3 struct adc16_channel_config_t

Data Fields

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

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 uint32_t adc16_channel_config_t::channelNumber

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

5.3.3.0.0.3.2 bool adc16_channel_config_t::enableInterruptOnConversionCompleted

5.4 Macro Definition Documentation

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

5.5 Enumeration Type Documentation

5.5.1 enum _adc16_channel_status_flags

Enumerator

kADC16_ChannelConversionDoneFlag Conversion done.

5.5.2 enum _adc16_status_flags

Enumerator

kADC16_ActiveFlag Converter is active.

5.5.3 enum adc16_clock_divider_t

Enumerator

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

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

5.5.4 enum adc16_resolution_t

Enumerator

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

5.5.5 enum adc16_clock_source_t

Enumerator

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

5.5.6 enum adc16_long_sample_mode_t

Enumerator

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

5.5.7 enum adc16_reference_voltage_source_t

Enumerator

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

5.5.8 enum adc16_hardware_compare_mode_t

Enumerator

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

5.6 Function Documentation

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

Parameters

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

5.6.2 void ADC16_Deinit (ADC_Type * *base*)

Parameters

base	ADC16 peripheral base address.

5.6.3 void ADC16_GetDefaultConfig (adc16_config_t * *config*)

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

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

Parameters

config	Pointer to configuration structure.

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

Parameters

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

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

The hardware compare mode provides a way to process the conversion result automatically by hardware. Only the result in compare range is available. To compare the range, see "adc16_hardware_compare_-mode_t", or the reference manual document for more detailed information.

Parameters

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

5.6.6 uint32_t ADC16_GetStatusFlags (ADC_Type * *base*)

Parameters

base	ADC16 peripheral base address.

Returns

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

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

Parameters

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

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

This operation triggers the conversion if 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 can have more than one group of status and control register, one for each conversion. The channel group parameter indicates which group of registers are used 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. Channel group 0 is used for both software and hardware trigger modes of operation. Channel groups 1 and greater indicate potentially multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the MCU reference manual about the number of SC1n registers (channel groups) specific to this device. None of the channel groups 1 or greater are used for software trigger operation and therefore writes to these channel groups do not initiate a new conversion. Updating 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 "adc16_channel_config_t" structure for conversion channel.

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

Parameters

base	ADC16 peripheral base address.
channelGroup	Channel group index.

Returns

Conversion value.

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

Parameters

base	ADC16 peripheral base address.
channelGroup	Channel group index.

Returns

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

Chapter 6
Clock Driver

6.1 Overview

The KSDK provides APIs for Kinetis devices clock operation.

6.2 Get frequency

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

6.3 External clock frequency

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

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

The next section shows the MCG_Lite based APIs, the MCG based APIs are similar, will not explain in this document.

Modules

- Multipurpose Clock Generator Lite (MCGLITE)

6.4 Multipurpose Clock Generator Lite (MCGLITE)

6.4.1 Overview

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

6.4.2 Function description

The MCG_Lite driver provides three kinds of APIs:

1. APIs to get the MCG_Lite frequency.
2. APIs for MCG_Lite mode.
3. APIs for OSC setup.

6.4.2.1 MCG_Lite clock frequency

The CLOCK_GetOutClkFreq(), CLOCK_GetInternalRefClkFreq() and CLOCK_GetPeriphClkFreq() functions are used to get the frequency of MCGOUTCLK, MCGIRCLK, and MCGPCLK based on the current hardware setting.

6.4.2.2 MCG_Lite mode

The function CLOCK_GetMode() gets the current MCG_Lite mode.

The function CLOCK_SetMcgliteConfig() sets the MCG_Lite to a desired configuration. The MCG_Lite can't switch between the LIRC2M and LIRC8M. Instead, the function switches to the HIRC mode first and then switches to the target mode.

6.4.2.3 OSC configuration

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

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 mcglite_config_t
  MCG_Lite configure structure for mode change. *More...*

Macros

- #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  Clock driver version.
- #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 SPI_CLOCKS
  Clock ip name array for SPI.
- #define SLCD_CLOCKS
  Clock ip name array for SLCD.
- #define PIT_CLOCKS
  Clock ip name array for PIT.
- #define PORT_CLOCKS
  Clock ip name array for PORT.
- #define LPUART_CLOCKS
  Clock ip name array for LPUART.
- #define DAC_CLOCKS
  Clock ip name array for DAC.
- #define LPTMR_CLOCKS
  Clock ip name array for LPTMR.
- #define ADC16_CLOCKS
  Clock ip name array for ADC16.
- #define FLEXIO_CLOCKS
  Clock ip name array for FLEXIO.
- #define VREF_CLOCKS
  Clock ip name array for VREF.
- #define DMA_CLOCKS
  Clock ip name array for DMA.
- #define UART_CLOCKS
  Clock ip name array for UART.
- #define TPM_CLOCKS
  Clock ip name array for TPM.
- #define I2C_CLOCKS
  Clock ip name array for I2C.
- #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_UsbSrcIrc48M = SIM_SOPT2_USBSRC(1U),
  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_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 osc_mode_t {
  kOSC_ModeExt = 0U,
  kOSC_ModeOscLowPower = MCG_C2_EREFS0_MASK,
  kOSC_ModeOscHighGain = MCG_C2_EREFS0_MASK | MCG_C2_HGO0_MASK }
  OSC work mode.
- enum mcglite_clkout_src_t {

kMCGLITE_ClkSrcHirc,
kMCGLITE_ClkSrcLirc,
kMCGLITE_ClkSrcExt }
MCG_Lite clock source selection.

- enum mcglite_lirc_mode_t {
kMCGLITE_Lirc2M,
kMCGLITE_Lirc8M }
MCG_Lite LIRC select.
- enum mcglite_lirc_div_t {
kMCGLITE_LircDivBy1 = 0U,
kMCGLITE_LircDivBy2,
kMCGLITE_LircDivBy4,
kMCGLITE_LircDivBy8,
kMCGLITE_LircDivBy16,
kMCGLITE_LircDivBy32,
kMCGLITE_LircDivBy64,
kMCGLITE_LircDivBy128 }
MCG_Lite divider factor selection for clock source.
- enum mcglite_mode_t {
kMCGLITE_ModeHirc48M,
kMCGLITE_ModeLirc8M,
kMCGLITE_ModeLirc2M,
kMCGLITE_ModeExt,
kMCGLITE_ModeError }
MCG_Lite clock mode definitions.
- enum _mcglite_irclk_enable_mode {
kMCGLITE_IrclkEnable = MCG_C1_IRCLKEN_MASK,
kMCGLITE_IrclkEnableInStop = MCG_C1_IREFSTEN_MASK }
MCG internal reference clock (MCGIRCLK) enable mode definition.

Functions

- static void CLOCK_SetXtal0Freq (uint32_t freq)
Set the XTAL0 frequency based on board setting.
- static void CLOCK_SetXtal32Freq (uint32_t freq)
Set the XTAL32/RTC_CLKIN frequency based on board setting.
- 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_SetLpuart0Clock (uint32_t src)
Set LPUART0 clock source.
- static void CLOCK_SetLpuart1Clock (uint32_t src)
Set LPUART1 clock source.
- static void CLOCK_SetTpmClock (uint32_t src)

Set TPM clock source.
- static void CLOCK_SetFlexio0Clock (uint32_t src)
  Set FLEXIO clock 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_SetClkOutClock (uint32_t src)
  Set CLKOUT source.
- static void CLOCK_SetRtcClkOutClock (uint32_t src)
  Set RTC_CLKOUT source.
- static void CLOCK_SetOutDiv (uint32_t outdiv1, 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_GetFlashClkFreq (void)
  Get the flash clock frequency.
- uint32_t CLOCK_GetEr32kClkFreq (void)
  Get the external reference 32K clock frequency (ERCLK32K).
- 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.

MCG_Lite clock frequency

- uint32_t CLOCK_GetOutClkFreq (void)
  Gets the MCG_Lite output clock (MCGOUTCLK) frequency.
- uint32_t CLOCK_GetInternalRefClkFreq (void)
  Gets the MCG internal reference clock (MCGIRCLK) frequency.
- uint32_t CLOCK_GetPeriphClkFreq (void)
  Gets the current MCGPCLK frequency.

MCG_Lite mode.

- mcglite_mode_t CLOCK_GetMode (void)
  Gets the current MCG_Lite mode.
- status_t CLOCK_SetMcgliteConfig (mcglite_config_t const *targetConfig)
  Sets the MCG_Lite configuration.

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)
  Initialize OSC0.
- void CLOCK_DeinitOsc0 (void)
  Deinitializes the OSC0.

6.4.3 Data Structure Documentation

6.4.3.1 struct sim_clock_config_t

Data Fields

- uint8_t er32kSrc
  ERCLK32K source selection.
- uint32_t clkdiv1
  SIM_CLKDIV1.

6.4.3.1.0.4 Field Documentation

6.4.3.1.0.4.1 uint8_t sim_clock_config_t::er32kSrc

6.4.3.1.0.4.2 uint32_t sim_clock_config_t::clkdiv1

6.4.3.2 struct oscer_config_t

Data Fields

- uint8_t enableMode
  OSCERCLK enable mode.

6.4.3.2.0.5 Field Documentation

6.4.3.2.0.5.1 uint8_t oscer_config_t::enableMode

OR'ed value of _oscer_enable_mode.

6.4.3.3 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 board settings:

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.

6.4.3.3.0.6 Field Documentation

6.4.3.3.0.6.1 uint32_t osc_config_t::freq

6.4.3.3.0.6.2 uint8_t osc_config_t::capLoad

6.4.3.3.0.6.3 osc_mode_t osc_config_t::workMode

6.4.3.3.0.6.4 oscer_config_t osc_config_t::oscerConfig

6.4.3.4 struct mcglite_config_t

Data Fields

- mcglite_clkout_src_t outSrc
  MCGOUT clock select.
- uint8_t irclkEnableMode
  MCGIRCLK enable mode, OR'ed value of _mcglite_irclk_enable_mode.
- mcglite_lirc_mode_t ircs
  MCG_C2[IRCS].
- mcglite_lirc_div_t fcrdiv
  MCG_SC[FCRDIV].
- mcglite_lirc_div_t lircDiv2
  MCG_MC[LIRC_DIV2].
- bool hircEnableInNotHircMode
  HIRC enable when not in HIRC mode.

6.4.3.4.0.7 Field Documentation

6.4.3.4.0.7.1 mcglite_clkout_src_t mcglite_config_t::outSrc

6.4.3.4.0.7.2 uint8_t mcglite_config_t::irclkEnableMode

6.4.3.4.0.7.3 mcglite_lirc_mode_t mcglite_config_t::ircs

6.4.3.4.0.7.4 mcglite_lirc_div_t mcglite_config_t::fcrdiv

6.4.3.4.0.7.5 mcglite_lirc_div_t mcglite_config_t::lircDiv2

6.4.3.4.0.7.6 bool mcglite_config_t::hircEnableInNotHircMode

6.4.4 Macro Definition Documentation

6.4.4.1 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

Version 2.1.0.

6.4.4.2 #define DMAMUX_CLOCKS

Value:

```
{                   \
        kCLOCK_Dmamux0 \
    }
```

6.4.4.3 #define RTC_CLOCKS

Value:

```
{               \
        kCLOCK_Rtc0 \
    }
```

6.4.4.4 #define SAI_CLOCKS

Value:

```
{               \
        kCLOCK_Sai0 \
    }
```

6.4.4.5 #define SPI_CLOCKS

Value:

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

6.4.4.6 #define SLCD_CLOCKS

Value:

```
{                \
        kCLOCK_Slcd0 \
    }
```

6.4.4.7 #define PIT_CLOCKS

Value:

```
{                \
        kCLOCK_Pit0 \
    }
```

6.4.4.8 #define PORT_CLOCKS

Value:

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

6.4.4.9 #define LPUART_CLOCKS

Value:

```
{                                   \
        kCLOCK_Lpuart0, kCLOCK_Lpuart1 \
    }
```

6.4.4.10 #define DAC_CLOCKS

Value:

```
{                \
        kCLOCK_Dac0 \
    }
```

6.4.4.11 #define LPTMR_CLOCKS

Value:

```
{                  \
        kCLOCK_Lptmr0 \
    }
```

6.4.4.12 #define ADC16_CLOCKS

Value:

```
{                 \
        kCLOCK_Adc0  \
    }
```

6.4.4.13 #define FLEXIO_CLOCKS

Value:

```
{                   \
        kCLOCK_Flexio0 \
    }
```

6.4.4.14 #define VREF_CLOCKS

Value:

```
{                 \
        kCLOCK_Vref0 \
    }
```

6.4.4.15 #define DMA_CLOCKS

Value:

```
{               \
        kCLOCK_Dma0 \
    }
```

6.4.4.16 #define UART_CLOCKS

Value:

```
{                                                   \
        kCLOCK_IpInvalid, kCLOCK_IpInvalid, kCLOCK_Uart2 \
    }
```

6.4.4.17 #define TPM_CLOCKS

Value:

```
{                                       \
        kCLOCK_Tpm0, kCLOCK_Tpm1, kCLOCK_Tpm2 \
    }
```

6.4.4.18 #define I2C_CLOCKS

Value:

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

6.4.4.19 #define FTF_CLOCKS

Value:

```
{               \
        kCLOCK_Ftf0 \
    }
```

6.4.4.20 #define CMP_CLOCKS

Value:

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

6.4.4.21 #define SYS_CLK kCLOCK_CoreSysClk

6.4.5 Enumeration Type Documentation

6.4.5.1 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.

6.4.5.2 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcIrc48M Use IRC48M.
kCLOCK_UsbSrcExt Use USB_CLKIN.

6.4.5.3 enum clock_ip_name_t

6.4.5.4 enum _osc_cap_load

Enumerator

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

6.4.5.5 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.
kOSC_ErClkEnableInStop Enable in stop mode.

6.4.5.6 enum osc_mode_t

Enumerator

kOSC_ModeExt Use external clock.
kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

6.4.5.7 enum mcglite_clkout_src_t

Enumerator

kMCGLITE_ClkSrcHirc MCGOUTCLK source is HIRC.
kMCGLITE_ClkSrcLirc MCGOUTCLK source is LIRC.
kMCGLITE_ClkSrcExt MCGOUTCLK source is external clock source.

6.4.5.8 enum mcglite_lirc_mode_t

Enumerator

kMCGLITE_Lirc2M Slow internal reference(LIRC) 2MHz clock selected.
kMCGLITE_Lirc8M Slow internal reference(LIRC) 8MHz clock selected.

6.4.5.9 enum mcglite_lirc_div_t

Enumerator

kMCGLITE_LircDivBy1 Divider is 1.
kMCGLITE_LircDivBy2 Divider is 2.
kMCGLITE_LircDivBy4 Divider is 4.
kMCGLITE_LircDivBy8 Divider is 8.
kMCGLITE_LircDivBy16 Divider is 16.
kMCGLITE_LircDivBy32 Divider is 32.
kMCGLITE_LircDivBy64 Divider is 64.
kMCGLITE_LircDivBy128 Divider is 128.

6.4.5.10 enum mcglite_mode_t

Enumerator

kMCGLITE_ModeHirc48M Clock mode is HIRC 48 M.
kMCGLITE_ModeLirc8M Clock mode is LIRC 8 M.
kMCGLITE_ModeLirc2M Clock mode is LIRC 2 M.
kMCGLITE_ModeExt Clock mode is EXT.
kMCGLITE_ModeError Unknown mode.

6.4.5.11 enum _mcglite_irclk_enable_mode

Enumerator

kMCGLITE_IrclkEnable MCGIRCLK enable.
kMCGLITE_IrclkEnableInStop MCGIRCLK enable in stop mode.

6.4.6 Function Documentation

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

Parameters

freq	The XTAL0/EXTAL0 input clock frequency in Hz.

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

Parameters

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

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

Parameters

name	Which clock to enable, see clock_ip_name_t.

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

Parameters

name	Which clock to disable, see clock_ip_name_t.

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

Parameters

src	The value to set ERCLK32K clock source.

6.4.6.6 static void CLOCK_SetLpuart0Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set LPUART0 clock source.

6.4.6.7 static void CLOCK_SetLpuart1Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set LPUART1 clock source.

6.4.6.8 static void CLOCK_SetTpmClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set TPM clock source.

6.4.6.9 static void CLOCK_SetFlexio0Clock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set FLEXIO clock source.

6.4.6.10 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.

6.4.6.11 static void CLOCK_DisableUsbfs0Clock (void) [inline], [static]

Disable USB FS clock.

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

Parameters

src	The value to set CLKOUT source.

6.4.6.13 static void CLOCK_SetRtcClkOutClock (uint32_t *src*) [inline], [static]

Parameters

src	The value to set RTC_CLKOUT source.

6.4.6.14 static void CLOCK_SetOutDiv (uint32_t *outdiv1,* uint32_t *outdiv4*) [inline], [static]

Set the SIM_CLKDIV1[OUTDIV1], SIM_CLKDIV1[OUTDIV4].

Parameters

outdiv1	Clock 1 output divider value.
outdiv4	Clock 4 output divider value.

6.4.6.15 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

6.4.6.16 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.17 uint32_t CLOCK_GetPlatClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.18 uint32_t CLOCK_GetBusClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.19 uint32_t CLOCK_GetFlashClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.20 uint32_t CLOCK_GetEr32kClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.21 uint32_t CLOCK_GetOsc0ErClkFreq (void)

Returns

Clock frequency in Hz.

6.4.6.22 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.

6.4.6.23 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.

6.4.6.24 uint32_t CLOCK_GetOutClkFreq (void)

This function gets the MCG_Lite output clock frequency (Hz) based on the current MCG_Lite register value.

Returns

The frequency of MCGOUTCLK.

6.4.6.25 uint32_t CLOCK_GetInternalRefClkFreq (void)

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

Returns

The frequency of MCGIRCLK.

6.4.6.26 uint32_t CLOCK_GetPeriphClkFreq (void)

This function gets the MCGPCLK frequency (Hertz) based on the current MCG_Lite register settings.

Returns

The frequency of MCGPCLK.

6.4.6.27 mcglite_mode_t CLOCK_GetMode (void)

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

Returns

Current MCG_Lite mode or error code.

6.4.6.28 status_t CLOCK_SetMcgliteConfig (mcglite_config_t const * *targetConfig*)

This function configures the MCG_Lite, include output clock source, MCGIRCLK setting, HIRC setting and so on, see mcglite_config_t for details.

Parameters

targetConfig	Pointer to the target MCG_Lite mode configuration structure.

Returns

Error code.

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

This function configures the OSC external reference clock (OSCERCLK). For example, to enable the OSCERCLK in normal mode and stop mode, and also set the output divider to 1, as follows:

```
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.

6.4.6.30 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);
```

6.4.6.31 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.

6.4.6.32 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

6.4.7 Variable Documentation

6.4.7.1 uint32_t g_xtal0Freq

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

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

This is important for the multicore platforms, only one core needs to setup OSC0 using CLOCK_InitOsc0, all other cores need to call CLOCK_SetXtal0Freq to get valid clock frequency.

6.4.7.2 uint32_t g_xtal32Freq

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

This is important for the multicore platforms, only one core needs to setup the clock, all other cores need to call CLOCK_SetXtal32Freq to get valid clock frequency.

Chapter 7
CMP: Analog Comparator Driver

7.1 Overview

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

Overview

The CMP driver is a basic comparator with advanced features. The APIs for the basic comparator enable the CMP as a general comparator, which compares the two voltage of the two input channels and creates the output of the comparator result. The APIs for advanced features can be used as the plug-in function based on the basic comparator. They can process the comparator's output with hardware support.

7.2 Typical use case

7.2.1 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.
        }
    }
}
```

7.2.2 Interrupt Configuration

```
volatile uint32_t g_CmpFlags = 0U;

// ...

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

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

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

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

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

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

    while (1)
    {
    }
}
```

Files

- file fsl_cmp.h

Data Structures

- struct cmp_config_t
  Configure the comparator. *More...*
- struct cmp_filter_config_t
  Configure the filter. *More...*

- struct cmp_dac_config_t
  Configure the internal DAC. More...

Enumerations

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

Driver version

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

Initialization

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

Advanced Features

- void CMP_SetFilterConfig (CMP_Type *base, const cmp_filter_config_t *config)
  Configures the filter.
- void CMP_SetDACConfig (CMP_Type *base, const cmp_dac_config_t *config)
  Configures the internal DAC.

- void CMP_EnableInterrupts (CMP_Type *base, uint32_t mask)
  Enables the interrupts.
- void CMP_DisableInterrupts (CMP_Type *base, uint32_t mask)
  Disables the interrupts.

Results

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

7.3 Data Structure Documentation

7.3.1 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 inverted comparator output.
- bool useUnfilteredOutput
  Set compare output(COUT) to equal COUTA(true) or COUT(false).
- bool enablePinOut
  The comparator output is available on the associated pin.

7.3.1.0.0.8 Field Documentation

7.3.1.0.0.8.1 bool cmp_config_t::enableCmp

7.3.1.0.0.8.2 cmp_hysteresis_mode_t cmp_config_t::hysteresisMode

7.3.1.0.0.8.3 bool cmp_config_t::enableHighSpeed

7.3.1.0.0.8.4 bool cmp_config_t::enableInvertOutput

7.3.1.0.0.8.5 bool cmp_config_t::useUnfilteredOutput

7.3.1.0.0.8.6 bool cmp_config_t::enablePinOut

7.3.2 struct cmp_filter_config_t

Data Fields

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

7.3.2.0.0.9 Field Documentation

7.3.2.0.0.9.1 uint8_t cmp_filter_config_t::filterCount

Available range is 1-7, 0 would cause the filter disabled.

7.3.2.0.0.9.2 uint8_t cmp_filter_config_t::filterPeriod

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

7.3.3 struct cmp_dac_config_t

Data Fields

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

7.3.3.0.0.10 Field Documentation

7.3.3.0.0.10.1 cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource

7.3.3.0.0.10.2 uint8_t cmp_dac_config_t::DACValue

Available range is 0-63.

7.4 Macro Definition Documentation

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

7.5 Enumeration Type Documentation

7.5.1 enum _cmp_interrupt_enable

Enumerator

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

7.5.2 enum _cmp_status_flags

Enumerator

kCMP_OutputRisingEventFlag Rising-edge on compare output has occurred.
kCMP_OutputFallingEventFlag Falling-edge on compare output has occurred.
kCMP_OutputAssertEventFlag Return the current value of the analog comparator output.

7.5.3 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.

7.5.4 enum cmp_reference_voltage_source_t

Enumerator

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

7.6 Function Documentation

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

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

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

Parameters

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

7.6.2 void CMP_Deinit (CMP_Type * *base*)

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

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

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

Parameters

base	CMP peripheral base address.

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

Parameters

base	CMP peripheral base address.

enable	Enable the module or not.

7.6.4 void CMP_GetDefaultConfig (cmp_config_t * *config*)

This function initializes the user configure 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.

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

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

Parameters

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

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

Parameters

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

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

Parameters

base	CMP peripheral base address.
config	Pointer to configuration structure. "NULL" is for disabling the feature.

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

Parameters

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

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

Parameters

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

7.6.10 uint32_t CMP_GetStatusFlags (CMP_Type * *base*)

Parameters

base	CMP peripheral base address.

Returns

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

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

Parameters

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

Chapter 8
CMT: Carrier Modulator Transmitter Driver

8.1 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.

8.2 Clock formulas

The CMT module has internal clock dividers. It was originally designed to be based on an 8 MHZ bus clock that could be divided by 1, 2, 4, or 8 according to the specification. To be compatible with 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 8MHz 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 approximately 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

8.3 Typical use case

This is an example code to initialize the 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;
```

Typical use case

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

// Prepares the modulate configuration for a user 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 the 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);
            }
        }
    }
}
```

Files

- file fsl_cmt.h

Data Structures

- struct cmt_modulate_config_t
  CMT carrier generator and modulator configure 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, 0))
  CMT driver version 2.0.0.

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 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.

8.4 Data Structure Documentation

8.4.1 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.

8.4.1.0.0.11 Field Documentation

8.4.1.0.0.11.1 uint8_t cmt_modulate_config_t::highCount1

8.4.1.0.0.11.2 uint8_t cmt_modulate_config_t::lowCount1

8.4.1.0.0.11.3 uint8_t cmt_modulate_config_t::highCount2

8.4.1.0.0.11.4 uint8_t cmt_modulate_config_t::lowCount2

8.4.1.0.0.11.5 uint16_t cmt_modulate_config_t::markCount

8.4.1.0.0.11.6 uint16_t cmt_modulate_config_t::spaceCount

8.4.2 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.

8.4.2.0.0.12 Field Documentation

8.4.2.0.0.12.1 bool cmt_config_t::isInterruptEnabled

8.4.2.0.0.12.2 bool cmt_config_t::isIroEnabled

8.4.2.0.0.12.3 cmt_infrared_output_polarity_t cmt_config_t::iroPolarity

8.4.2.0.0.12.4 cmt_second_clkdiv_t cmt_config_t::divider

8.5 Macro Definition Documentation

8.5.1 #define FSL_CMT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

8.6 Enumeration Type Documentation

8.6.1 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.

8.6.2 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.

8.6.3 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.

8.6.4 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.

8.6.5 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.

8.6.6 enum _cmt_interrupt_enable

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

Enumerator

kCMT_EndOfCycleInterruptEnable CMT end of cycle interrupt.

8.7 Function Documentation

8.7.1 void CMT_GetDefaultConfig (cmt_config_t * *config*)

The purpose of this API is to get the default configuration structure for the CMT_Init(). Use the initialized structure unchanged in CMT_Init(), or modify some fields of the structure before calling the CMT_Init().

Parameters

config	The CMT configuration structure pointer.

8.7.2 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.

8.7.3 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.

8.7.4 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.

8.7.5 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.

8.7.6 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.

8.7.7 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 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.

8.7.8 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.

8.7.9 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 ~ 0xFFFF.
spaceCount	The number of clock period for CMT modulator signal space period, in the range of the 0 ~ 0xFFFF.

8.7.10 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.

8.7.11 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"

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

This function enables the CMT interrupts according to the provided maskIf 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.

8.7.13 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.

8.7.14 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 9
COP: Watchdog Driver

9.1 Overview

The KSDK provides a peripheral driver for the COP module (COP) of Kinetis devices.

Typical use case

```
cop_config_t config;
COP_GetDefaultConfig(&config);
config.timeoutCycles = kCOP_2Power8CyclesOr2Power16Cycles;
COP_Init(sim_base,&config);
```

Files

- file fsl_cop.h

Data Structures

- struct cop_config_t
  Describes COP configuration structure. *More...*

Enumerations

- enum cop_clock_source_t {
  kCOP_LpoClock = 0U,
  kCOP_BusClock = 3U }
  COP clock source selection.
- enum cop_timeout_cycles_t {
  kCOP_2Power5CyclesOr2Power13Cycles = 1U,
  kCOP_2Power8CyclesOr2Power16Cycles = 2U,
  kCOP_2Power10CyclesOr2Power18Cycles = 3U }
  Define the COP timeout cycles.

Driver version

- #define FSL_COP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  COP driver version 2.0.0.

COP refresh sequence.

- #define COP_FIRST_BYTE_OF_REFRESH (0x55U)
  First byte of refresh sequence.
- #define COP_SECOND_BYTE_OF_REFRESH (0xAAU)
  Second byte of refresh sequence.

COP Functional Operation

- void COP_GetDefaultConfig (cop_config_t *config)
  Initializes the COP configuration structure.
- void COP_Init (SIM_Type *base, const cop_config_t *config)
  Initializes the COP module.
- static void COP_Disable (SIM_Type *base)
  De-initializes the COP module.
- void COP_Refresh (SIM_Type *base)
  Refreshes the COP timer.

9.2 Data Structure Documentation

9.2.1 struct cop_config_t

Data Fields

- bool enableWindowMode
  COP run mode: window mode or normal mode.
- cop_clock_source_t clockSource
  Set COP clock source.
- cop_timeout_cycles_t timeoutCycles
  Set COP timeout value.

9.3 Macro Definition Documentation

9.3.1 #define FSL_COP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

9.4 Enumeration Type Documentation

9.4.1 enum cop_clock_source_t

Enumerator

kCOP_LpoClock COP clock sourced from LPO.
kCOP_BusClock COP clock sourced from Bus clock.

9.4.2 enum cop_timeout_cycles_t

Enumerator

kCOP_2Power5CyclesOr2Power13Cycles 2^5 or 2^13 clock cycles
kCOP_2Power8CyclesOr2Power16Cycles 2^8 or 2^16 clock cycles
kCOP_2Power10CyclesOr2Power18Cycles 2^10 or 2^18 clock cycles

9.5 Function Documentation

9.5.1 void COP_GetDefaultConfig (cop_config_t * *config*)

This function initializes the COP configuration structure to default values. The default values are:

```
copConfig->enableWindowMode = false;
copConfig->timeoutMode = kCOP_LongTimeoutMode;
copConfig->enableStop = false;
copConfig->enableDebug = false;
copConfig->clockSource = kCOP_LpoClock;
copConfig->timeoutCycles = kCOP_2Power10CyclesOr2Power18Cycles;
```

Parameters

config	Pointer to the COP configuration structure.

See Also

cop_config_t

9.5.2 void COP_Init (SIM_Type * *base,* const cop_config_t * *config*)

This function configures the COP. After it is called, the COP starts running according to the configuration. Because all COP control registers are write-once only, the COP_Init function and the COP_Disable function can be called only once. A second call has no effect.

Example:

```
cop_config_t config;
COP_GetDefaultConfig(&config);
config.timeoutCycles = kCOP_2Power8CyclesOr2Power16Cycles;
COP_Init(sim_base,&config);
```

Parameters

base	SIM peripheral base address.
config	The configuration of COP.

9.5.3 static void COP_Disable (SIM_Type * *base*) [inline], [static]

This dedicated function is not provided. Instead, the COP_Disable function can be used to disable the COP.

Disables the COP module.

This function disables the COP Watchdog. Note: The COP configuration register is a write-once after reset. To disable the COP Watchdog, call this function first.

Parameters

base	SIM peripheral base address.

9.5.4 void COP_Refresh (SIM_Type * *base*)

This function feeds the COP.

Parameters

base	SIM peripheral base address.

Chapter 10
CRC: Cyclic Redundancy Check Driver

10.1 Overview

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

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

10.2 CRC Driver Initialization and Configuration

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

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

10.3 CRC Write Data

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

10.4 CRC Get Checksum

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

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

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

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

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

10.5 Comments about API usage in RTOS

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

The triplets

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

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

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

10.6 Comments about API usage in interrupt handler

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

10.7 CRC Driver Examples

Simple examples

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

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

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

Simple example with CRC-32 protocol configuration

```
crc_config_t config;
uint32_t checksum;

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

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

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

Advanced examples

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

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

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

Corresponding init functions:

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

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

    CRC_Init(base, &config);
}

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

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

    CRC_Init(base, &config);
}

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

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

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

The following context switches show possible API usage:

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

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

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

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

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

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

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

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

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

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

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

Files

- file fsl_crc.h

Data Structures

- struct crc_config_t
  CRC protocol configuration. More...

Macros

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

Enumerations

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

Functions

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

Driver version

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

10.8 Data Structure Documentation

10.8.1 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()

10.8.1.0.0.13 Field Documentation

10.8.1.0.0.13.1 uint32_t crc_config_t::polynomial

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

10.8.1.0.0.13.2 bool crc_config_t::reflectIn

10.8.1.0.0.13.3 bool crc_config_t::reflectOut

10.8.1.0.0.13.4 bool crc_config_t::complementChecksum

10.8.1.0.0.13.5 crc_bits_t crc_config_t::crcBits

10.9 Macro Definition Documentation

10.9.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

10.9.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as defeault.

10.10 Enumeration Type Documentation

10.10.1 enum crc_bits_t

Enumerator

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

10.10.2 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.

10.11 Function Documentation

10.11.1 void CRC_Init (CRC_Type * *base,* const crc_config_t * *config*)

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

Parameters

base	CRC peripheral address.
config	CRC module configuration structure

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

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

Parameters

base	CRC peripheral address.

10.11.3 void CRC_GetDefaultConfig (crc_config_t * *config*)

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

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

Parameters

config	CRC protocol configuration structure

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

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

Parameters

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

10.11.5 static uint32_t CRC_Get32bitResult (CRC_Type * *base*) [inline], [static]

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

Parameters

base	CRC peripheral address.

Returns

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

10.11.6 uint16_t CRC_Get16bitResult (CRC_Type * *base*)

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

Parameters

base	CRC peripheral address.

Returns

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

Chapter 11
DAC: Digital-to-Analog Converter Driver

11.1 Overview

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

#Overview

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

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

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

The DAC buffer plays a major part when using the DAC module, as the most functional features are designed for the DAC hardware buffer.

Typical use case

Working as a basic DAC without the hardware buffer feature.

```
// ...

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

// ...

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

Working with the hardware buffer.

```
    // ...

    EnableIRQ(DEMO_DAC_IRQ_ID);

    // ...

    // Configures the DAC.
```

```
    DAC_GetDefaultConfig(&dacConfigStruct);
    DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);

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

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

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

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

Files

- file fsl_dac.h

Data Structures

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

Enumerations

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

Driver version

- #define FSL_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DAC driver version 2.0.0.

Initialization

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

Buffer

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

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

11.2 Data Structure Documentation

11.2.1 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.

11.2.1.0.0.14 Field Documentation

11.2.1.0.0.14.1 dac_reference_voltage_source_t dac_config_t::referenceVoltageSource

11.2.1.0.0.14.2 bool dac_config_t::enableLowPowerMode

11.2.2 struct dac_buffer_config_t

Data Fields

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

11.2.2.0.0.15 Field Documentation

11.2.2.0.0.15.1 dac_buffer_trigger_mode_t dac_buffer_config_t::triggerMode

11.2.2.0.0.15.2 dac_buffer_work_mode_t dac_buffer_config_t::workMode

11.2.2.0.0.15.3 uint8_t dac_buffer_config_t::upperLimit

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

11.3 Macro Definition Documentation

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

11.4 Enumeration Type Documentation

11.4.1 enum _dac_buffer_status_flags

Enumerator

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

11.4.2 enum _dac_buffer_interrupt_enable

Enumerator

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

11.4.3 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.

11.4.4 enum dac_buffer_trigger_mode_t

Enumerator

kDAC_BufferTriggerByHardwareMode The DAC hardware trigger is selected.

kDAC_BufferTriggerBySoftwareMode The DAC software trigger is selected.

11.4.5 enum dac_buffer_work_mode_t

Enumerator

kDAC_BufferWorkAsNormalMode Normal mode.
kDAC_BufferWorkAsOneTimeScanMode One-Time Scan mode.

11.5 Function Documentation

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

This function initializes the DAC module, including:

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

Parameters

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

11.5.2 void DAC_Deinit (DAC_Type * *base*)

This function de-initializes the DAC module, including:

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

Parameters

base	DAC peripheral base address.

11.5.3 void DAC_GetDefaultConfig (dac_config_t * *config*)

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

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

Parameters

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

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

Parameters

base	DAC peripheral base address.
enable	Enables the feature or not.

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

Parameters

base	DAC peripheral base address.
enable	Enables the feature or not.

11.5.6 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".

11.5.7 void DAC_GetDefaultBufferConfig (dac_buffer_config_t * *config*)

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

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

Parameters

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

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

Parameters

base	DAC peripheral base address.
enable	Enables the feature or not.

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

Parameters

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

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

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

Parameters

base	DAC peripheral base address.

11.5.11 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 by software trigger or hardware trigger.

Parameters

base	DAC peripheral base address.

Returns

Current read pointer of DAC buffer.

11.5.12 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 by software trigger or hardware trigger. After the read pointer changes, the DAC output value also changes.

Parameters

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

11.5.13 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".

11.5.14 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".

11.5.15 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".

11.5.16 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 12
Debug Console

Initializes the the peripheral used to debug messages.

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.

12.1 Function groups

12.1.1 Initialization

To initialize the debug console, call the DbgConsole_Init() function with these parameters. This function automatically enables the module and the clock.

```
/*
 * * @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
```

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;
```

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);
```

12.1.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width] [.precision][length]specifier", which is explained below

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
Do not support	

specifier	Description
d or i	Signed decimal integer
f	Decimal floating point
F	Decimal floating point capital letters
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer capital letters
o	Signed octal
b	Binary value
p	Pointer address
u	Unsigned decimal integer
c	Character
s	String of characters
n	Nothing printed

- Support a format specifier for SCANF following this prototype " %[∗][width][length]specifier", which is explained below

∗	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e., it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X), and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char ∗

specifier	Qualifying Input	Type of argument
i	Integer: : Number optionally preceded with a + or - sign	int *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char *
u	Unsigned decimal integer.	unsigned int *

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file:

```
int DbgConsole_Printf(const char *fmt_s, ...);
```

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
```

12.2 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 (;;)
    {}
}
```

Chapter 13
DMA: Direct Memory Access Controller Driver

13.1 Overview

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

13.2 Typical use case

13.2.1 DMA Operation

```
dma_transfer_config_t transferConfig;
uint32_t transferDone = false;

DMA_Init(DMA0);
DMA_CreateHandle(&g_DMA_Handle, DMA0, channel);
DMA_InstallCallback(&g_DMA_Handle, DMA_Callback, &transferDone);
DMA_PrepareTransfer(&transferConfig, srcAddr, srcWidth, destAddr, destWidth,
      transferBytes,
                    kDMA_MemoryToMemory);
DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig, true);
DMA_StartTransfer(&g_DMA_Handle);
/* Wait for DMA transfer finish
while (transferDone != true);
```

Files

- file fsl_dma.h

Data Structures

- struct dma_transfer_config_t
  DMA transfer configuration structure. *More...*
- struct dma_channel_link_config_t
  DMA transfer configuration structure. *More...*
- struct dma_handle_t
  DMA DMA handle structure. *More...*

Typedefs

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

Enumerations

- enum _dma_channel_status_flags {
  kDMA_TransactionsBCRFlag = DMA_DSR_BCR_BCR_MASK,
  kDMA_TransactionsDoneFlag = DMA_DSR_BCR_DONE_MASK,
  kDMA_TransactionsBusyFlag = DMA_DSR_BCR_BSY_MASK,
  kDMA_TransactionsRequestFlag = DMA_DSR_BCR_REQ_MASK,
  kDMA_BusErrorOnDestinationFlag = DMA_DSR_BCR_BED_MASK,
  kDMA_BusErrorOnSourceFlag = DMA_DSR_BCR_BES_MASK,
  kDMA_ConfigurationErrorFlag = DMA_DSR_BCR_CE_MASK }
  status flag for the DMA driver.
- enum dma_transfer_size_t {
  kDMA_Transfersize32bits = 0x0U,
  kDMA_Transfersize8bits,
  kDMA_Transfersize16bits }
  DMA transfer size type.
- enum dma_modulo_t {
  kDMA_ModuloDisable = 0x0U,
  kDMA_Modulo16Bytes,
  kDMA_Modulo32Bytes,
  kDMA_Modulo64Bytes,
  kDMA_Modulo128Bytes,
  kDMA_Modulo256Bytes,
  kDMA_Modulo512Bytes,
  kDMA_Modulo1KBytes,
  kDMA_Modulo2KBytes,
  kDMA_Modulo4KBytes,
  kDMA_Modulo8KBytes,
  kDMA_Modulo16KBytes,
  kDMA_Modulo32KBytes,
  kDMA_Modulo64KBytes,
  kDMA_Modulo128KBytes,
  kDMA_Modulo256KBytes }
  Configuration type for the DMA modulo.
- enum dma_channel_link_type_t {
  kDMA_ChannelLinkDisable = 0x0U,
  kDMA_ChannelLinkChannel1AndChannel2,
  kDMA_ChannelLinkChannel1,
  kDMA_ChannelLinkChannel1AfterBCR0 }
  DMA channel link type.
- enum dma_transfer_type_t {
  kDMA_MemoryToMemory = 0x0U,
  kDMA_PeripheralToMemory,
  kDMA_MemoryToPeripheral }
  DMA transfer type.
- enum dma_transfer_options_t {
  kDMA_NoOptions = 0x0U,

kDMA_EnableInterrupt }
DMA transfer options.
- enum _dma_transfer_status
DMA transfer status.

Driver version

- #define FSL_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
DMA driver version 2.0.0.

DMA Initialization and De-initialization

- void DMA_Init (DMA_Type *base)
Initializes the DMA peripheral.
- void DMA_Deinit (DMA_Type *base)
Deinitializes the DMA peripheral.

DMA Channel Operation

- void DMA_ResetChannel (DMA_Type *base, uint32_t channel)
Resets the DMA channel.
- void DMA_SetTransferConfig (DMA_Type *base, uint32_t channel, const dma_transfer_config_t *config)
Configures the DMA transfer attribute.
- void DMA_SetChannelLinkConfig (DMA_Type *base, uint32_t channel, const dma_channel_link_config_t *config)
Configures the DMA channel link feature.
- static void DMA_SetSourceAddress (DMA_Type *base, uint32_t channel, uint32_t srcAddr)
Sets the DMA source address for the DMA transfer.
- static void DMA_SetDestinationAddress (DMA_Type *base, uint32_t channel, uint32_t destAddr)
Sets the DMA destination address for the DMA transfer.
- static void DMA_SetTransferSize (DMA_Type *base, uint32_t channel, uint32_t size)
Sets the DMA transfer size for the DMA transfer.
- void DMA_SetModulo (DMA_Type *base, uint32_t channel, dma_modulo_t srcModulo, dma_modulo_t destModulo)
Sets the DMA modulo for the DMA transfer.
- static void DMA_EnableCycleSteal (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA cycle steal for the DMA transfer.
- static void DMA_EnableAutoAlign (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA auto align for the DMA transfer.
- static void DMA_EnableAsyncRequest (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA async request for the DMA transfer.
- static void DMA_EnableInterrupts (DMA_Type *base, uint32_t channel)
Enables an interrupt for the DMA transfer.
- static void DMA_DisableInterrupts (DMA_Type *base, uint32_t channel)
Disables an interrupt for the DMA transfer.

DMA Channel Transfer Operation

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

Enables the DMA hardware channel request.
- static void DMA_DisableChannelRequest (DMA_Type *base, uint32_t channel)
  Disables the DMA hardware channel request.
- static void DMA_TriggerChannelStart (DMA_Type *base, uint32_t channel)
  Starts the DMA transfer with a software trigger.

DMA Channel Status Operation

- static uint32_t DMA_GetRemainingBytes (DMA_Type *base, uint32_t channel)
  Gets the remaining bytes of the current DMA transfer.
- static uint32_t DMA_GetChannelStatusFlags (DMA_Type *base, uint32_t channel)
  Gets the DMA channel status flags.
- static void DMA_ClearChannelStatusFlags (DMA_Type *base, uint32_t channel, uint32_t mask)
  Clears the DMA channel status flags.

DMA Channel Transactional Operation

- void DMA_CreateHandle (dma_handle_t *handle, DMA_Type *base, uint32_t channel)
  Creates the DMA handle.
- void DMA_SetCallback (dma_handle_t *handle, dma_callback callback, void *userData)
  Sets the DMA callback function.
- void DMA_PrepareTransfer (dma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, dma_transfer_type_t type)
  Prepares the DMA transfer configuration structure.
- status_t DMA_SubmitTransfer (dma_handle_t *handle, const dma_transfer_config_t *config, uint32_t options)
  Submits the DMA transfer request.
- static void DMA_StartTransfer (dma_handle_t *handle)
  DMA starts a transfer.
- static void DMA_StopTransfer (dma_handle_t *handle)
  DMA stops a transfer.
- void DMA_AbortTransfer (dma_handle_t *handle)
  DMA aborts a transfer.
- void DMA_HandleIRQ (dma_handle_t *handle)
  DMA IRQ handler for current transfer complete.

13.3 Data Structure Documentation

13.3.1 struct dma_transfer_config_t

Data Fields

- uint32_t srcAddr
  DMA transfer source address.
- uint32_t destAddr
  DMA destination address.
- bool enableSrcIncrement
  Source address increase after each transfer.
- dma_transfer_size_t srcSize
  Source transfer size unit.

- bool enableDestIncrement
  Destination address increase after each transfer.
- dma_transfer_size_t destSize
  Destination transfer unit.
- uint32_t transferSize
  The number of bytes to be transferred.

13.3.1.0.0.16 Field Documentation

13.3.1.0.0.16.1 uint32_t dma_transfer_config_t::srcAddr

13.3.1.0.0.16.2 uint32_t dma_transfer_config_t::destAddr

13.3.1.0.0.16.3 bool dma_transfer_config_t::enableSrcIncrement

13.3.1.0.0.16.4 dma_transfer_size_t dma_transfer_config_t::srcSize

13.3.1.0.0.16.5 bool dma_transfer_config_t::enableDestIncrement

13.3.1.0.0.16.6 dma_transfer_size_t dma_transfer_config_t::destSize

13.3.1.0.0.16.7 uint32_t dma_transfer_config_t::transferSize

13.3.2 struct dma_channel_link_config_t

Data Fields

- dma_channel_link_type_t linkType
  Channel link type.
- uint32_t channel1
  The index of channel 1.
- uint32_t channel2
  The index of channel 2.

13.3.2.0.0.17 Field Documentation

13.3.2.0.0.17.1 dma_channel_link_type_t dma_channel_link_config_t::linkType

13.3.2.0.0.17.2 uint32_t dma_channel_link_config_t::channel1

13.3.2.0.0.17.3 uint32_t dma_channel_link_config_t::channel2

13.3.3 struct dma_handle_t

Data Fields

- DMA_Type * base
  DMA peripheral address.
- uint8_t channel

DMA channel used.

- dma_callback callback
  DMA callback function.
- void * userData
  Callback parameter.

13.3.3.0.0.18 Field Documentation

13.3.3.0.0.18.1 DMA_Type* dma_handle_t::base

13.3.3.0.0.18.2 uint8_t dma_handle_t::channel

13.3.3.0.0.18.3 dma_callback dma_handle_t::callback

13.3.3.0.0.18.4 void* dma_handle_t::userData

13.4 Macro Definition Documentation

13.4.1 #define FSL_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

13.5 Typedef Documentation

13.5.1 typedef void(* dma_callback)(struct _dma_handle *handle, void *userData)

13.6 Enumeration Type Documentation

13.6.1 enum _dma_channel_status_flags

Enumerator

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

kDMA_TransactionsDoneFlag Transactions Done.
kDMA_TransactionsBusyFlag Transactions Busy.
kDMA_TransactionsRequestFlag Transactions Request.
kDMA_BusErrorOnDestinationFlag Bus Error on Destination.
kDMA_BusErrorOnSourceFlag Bus Error on Source.
kDMA_ConfigurationErrorFlag Configuration Error.

13.6.2 enum dma_transfer_size_t

Enumerator

kDMA_Transfersize32bits 32 bits are transferred for every read/write
kDMA_Transfersize8bits 8 bits are transferred for every read/write
kDMA_Transfersize16bits 16b its are transferred for every read/write

13.6.3 enum dma_modulo_t

Enumerator

kDMA_ModuloDisable Buffer disabled.
kDMA_Modulo16Bytes Circular buffer size is 16 bytes.
kDMA_Modulo32Bytes Circular buffer size is 32 bytes.
kDMA_Modulo64Bytes Circular buffer size is 64 bytes.
kDMA_Modulo128Bytes Circular buffer size is 128 bytes.
kDMA_Modulo256Bytes Circular buffer size is 256 bytes.
kDMA_Modulo512Bytes Circular buffer size is 512 bytes.
kDMA_Modulo1KBytes Circular buffer size is 1 KB.
kDMA_Modulo2KBytes Circular buffer size is 2 KB.
kDMA_Modulo4KBytes Circular buffer size is 4 KB.
kDMA_Modulo8KBytes Circular buffer size is 8 KB.
kDMA_Modulo16KBytes Circular buffer size is 16 KB.
kDMA_Modulo32KBytes Circular buffer size is 32 KB.
kDMA_Modulo64KBytes Circular buffer size is 64 KB.
kDMA_Modulo128KBytes Circular buffer size is 128 KB.
kDMA_Modulo256KBytes Circular buffer size is 256 KB.

13.6.4 enum dma_channel_link_type_t

Enumerator

kDMA_ChannelLinkDisable No channel link.
kDMA_ChannelLinkChannel1AndChannel2 Perform a link to channel LCH1 after each cycle-steal transfer. followed by a link to LCH2 after the BCR decrements to 0.
kDMA_ChannelLinkChannel1 Perform a link to LCH1 after each cycle-steal transfer.
kDMA_ChannelLinkChannel1AfterBCR0 Perform a link to LCH1 after the BCR decrements.

13.6.5 enum dma_transfer_type_t

Enumerator

kDMA_MemoryToMemory Memory to Memory transfer.
kDMA_PeripheralToMemory Peripheral to Memory transfer.
kDMA_MemoryToPeripheral Memory to Peripheral transfer.

13.6.6 enum dma_transfer_options_t

Enumerator

kDMA_NoOptions Transfer without options.
kDMA_EnableInterrupt Enable interrupt while transfer complete.

13.7 Function Documentation

13.7.1 void DMA_Init (DMA_Type * *base*)

This function ungates the DMA clock.

Parameters

base	DMA peripheral base address.

13.7.2 void DMA_Deinit (DMA_Type * *base*)

This function gates the DMA clock.

Parameters

base	DMA peripheral base address.

13.7.3 void DMA_ResetChannel (DMA_Type * *base,* uint32_t *channel*)

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

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

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

Parameters

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

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

This function allows DMA channels to have their transfers linked. The current DMA channel triggers a DMA request to the linked channels (LCH1 or LCH2) depending on the channel link type. Perform a link to channel LCH1 after each cycle-steal transfer followed by a link to LCH2 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AndChannel2. Perform a link to LCH1 after each cycle-steal transfer if the type is kDMA_ChannelLinkChannel1. Perform a link to LCH1 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AfterBCR0.

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
config	Pointer to the channel link configuration structure.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
srcAddr	DMA source address.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
destAddr	DMA destination address.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
size	The number of bytes to be transferred.

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

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

Parameters

base	DMA peripheral base address.

channel	DMA channel number.
srcModulo	source address modulo.
destModulo	destination address modulo.

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

Parameters

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

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

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

This function starts only one read/write iteration.

Parameters

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

13.7.18 static uint32_t DMA_GetRemainingBytes (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

The number of bytes which have not been transferred yet.

13.7.19 static uint32_t DMA_GetChannelStatusFlags (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.

channel	DMA channel number.

Returns

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

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

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
mask	The mask of the channel status to be cleared. Use the defined _dma_channel_status-_flags type.

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

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

Parameters

handle	DMA handle pointer. The DMA handle stores callback function and parameters.
base	DMA peripheral base address.
channel	DMA channel number.

13.7.22 void DMA_SetCallback (dma_handle_t * *handle,* dma_callback *callback,* void * *userData*)

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

Parameters

handle	DMA handle pointer.
callback	DMA callback function pointer.
userData	Parameter for callback function. If it is not needed, just set to NULL.

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

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

Parameters

config	Pointer to the user configuration structure of type dma_transfer_config_t.
srcAddr	DMA transfer source address.
srcWidth	DMA transfer source address width (byte).
destAddr	DMA transfer destination address.
destWidth	DMA transfer destination address width (byte).
transferBytes	DMA transfer bytes to be transferred.
type	DMA transfer type.

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

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

Parameters

handle	DMA handle pointer.
config	Pointer to DMA transfer configuration structure.
options	Additional configurations for transfer. Use the defined dma_transfer_options_t type.

Return values

kStatus_DMA_Success	It indicates that the DMA submit transfer request succeeded.
kStatus_DMA_Busy	It indicates that the DMA is busy. Submit transfer request is not allowed.

Note

This function can't process multi transfer request.

13.7.25 static void DMA_StartTransfer (dma_handle_t * *handle*) [inline], [static]

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

Parameters

handle	DMA handle pointer.

Return values

kStatus_DMA_Success	It indicates that the DMA start transfer succeed.
kStatus_DMA_Busy	It indicates that the DMA has started a transfer.

13.7.26 static void DMA_StopTransfer (dma_handle_t * *handle*) [inline], [static]

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

Parameters

handle	DMA handle pointer.

13.7.27 void DMA_AbortTransfer (dma_handle_t * *handle*)

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

Parameters

handle	DMA handle pointer.

13.7.28 void DMA_HandleIRQ (dma_handle_t * *handle*)

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

Parameters

handle	DMA handle pointer.

Chapter 14
DMAMUX: Direct Memory Access Multiplexer Driver

14.1 Overview

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

14.2 Typical use case

14.2.1 DMAMUX Operation

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

Files

- file fsl_dmamux.h

Driver version

- #define FSL_DMAMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DMAMUX driver version 2.0.0.

DMAMUX Initialize and De-initialize

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

DMAMUX Channel Operation

- static void DMAMUX_EnableChannel (DMAMUX_Type *base, uint32_t channel)
  Enable DMAMUX channel.
- static void DMAMUX_DisableChannel (DMAMUX_Type *base, uint32_t channel)
  Disable DMAMUX channel.
- static void DMAMUX_SetSource (DMAMUX_Type *base, uint32_t channel, uint8_t source)
  Configure DMAMUX channel source.

14.3 Macro Definition Documentation

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

14.4 Function Documentation

14.4.1 void DMAMUX_Init (DMAMUX_Type * *base*)

This function ungate the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

14.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gate the DMAMUX clock.

Parameters

base	DMAMUX peripheral base address.

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

This function enable DMAMUX channel to work.

Parameters

base	DMAMUX peripheral base address.
channel	DMAMUX channel number.

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

This function disable DMAMUX channel.

Note

User must disable DMAMUX channel before configure it.

Parameters

base	DMAMUX peripheral base address.

channel	DMAMUX channel number.

14.4.5 static void DMAMUX_SetSource (DMAMUX_Type * *base*, uint32_t *channel*, uint8_t *source*) [inline], [static]

Parameters

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

Chapter 15
DSPI: Serial Peripheral Interface Driver

15.1 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

15.2 DSPI Driver

15.2.1 Overview

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

15.2.2 Typical use case

15.2.2.1 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);
```

15.2.2.2 Slave Operation

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

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

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

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

DSPI_SlaveTransferNonBlocking(&g_s_handle, &slaveXfer);


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

Files

- file fsl_dspi.h

Data Structures

- struct dspi_command_data_config_t
  DSPI master command date configuration used for 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 transactional API. More...

Macros

- #define DSPI_MASTER_CTAR_SHIFT (0U)
  DSPI master CTAR shift macro , internal used.
- #define DSPI_MASTER_CTAR_MASK (0x0FU)
  DSPI master CTAR mask macro , internal used.
- #define DSPI_MASTER_PCS_SHIFT (4U)
  DSPI master PCS shift macro , internal used.
- #define DSPI_MASTER_PCS_MASK (0xF0U)
  DSPI master PCS mask macro , internal used.
- #define DSPI_SLAVE_CTAR_SHIFT (0U)
  DSPI slave CTAR shift macro , internal used.
- #define DSPI_SLAVE_CTAR_MASK (0x07U)
  DSPI slave CTAR mask macro , internal used.

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 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 }
Can use this enumeration for DSPI master transfer configFlags.
- enum _dspi_transfer_config_flag_for_slave { kDSPI_SlaveCtar0 = 0U << DSPI_SLAVE_CTAR-_SHIFT }
Can use this enum for DSPI slave transfer configFlags.
- enum _dspi_transfer_state {
kDSPI_Idle = 0x0U,
kDSPI_Busy,

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

Driver version

- #define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  DSPI driver version 2.1.0.

Dummy data

- #define DSPI_MASTER_DUMMY_DATA (0x00U)
  Master dummy data used for tx if there is not txData.
- #define DSPI_SLAVE_DUMMY_DATA (0x00U)
  Slave dummy data used for tx if there is not txData.

Initialization and deinitialization

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

Status

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

Interrupts

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

DMA Control

- static void DSPI_EnableDMA (SPI_Type *base, uint32_t mask)
  Enables the DSPI DMA request.
- static void DSPI_DisableDMA (SPI_Type *base, uint32_t mask)
  Disables the DSPI DMA request.
- static uint32_t DSPI_MasterGetTxRegisterAddress (SPI_Type *base)
  Gets the DSPI master PUSHR data register address for the DMA operation.
- static uint32_t DSPI_SlaveGetTxRegisterAddress (SPI_Type *base)
  Gets the DSPI slave PUSHR data register address for the DMA operation.
- static uint32_t DSPI_GetRxRegisterAddress (SPI_Type *base)
  Gets the DSPI POPR data register address for the DMA operation.

Bus Operations

- static void DSPI_SetMasterSlaveMode (SPI_Type *base, dspi_master_slave_mode_t mode)
  Configures the DSPI for master or slave.
- static bool DSPI_IsMaster (SPI_Type *base)
  Returns whether the DSPI module is in master mode.
- static void DSPI_StartTransfer (SPI_Type *base)
  Starts the DSPI transfers and clears HALT bit in MCR.
- static void DSPI_StopTransfer (SPI_Type *base)
  Stops (halts) DSPI transfers and sets 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 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.

15.2.3 Data Structure Documentation

15.2.3.1 struct dspi_command_data_config_t

Data Fields

- bool isPcsContinuous

Option to enable the continuous assertion of 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 SPI Transfer Counter (SPI_TCNT) before transmission starts.

15.2.3.1.0.19 Field Documentation

15.2.3.1.0.19.1 bool dspi_command_data_config_t::isPcsContinuous

15.2.3.1.0.19.2 dspi_ctar_selection_t dspi_command_data_config_t::whichCtar

15.2.3.1.0.19.3 dspi_which_pcs_t dspi_command_data_config_t::whichPcs

15.2.3.1.0.19.4 bool dspi_command_data_config_t::isEndOfQueue

15.2.3.1.0.19.5 bool dspi_command_data_config_t::clearTransferCount

15.2.3.2 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 with nanosecond , set to 0 sets the minimum delay.
- uint32_t lastSckToPcsDelayInNanoSec
  Last SCK to PCS delay time with nanosecond , set to 0 sets the minimum delay.It sets the boundary value if out of range that can be set.
- uint32_t betweenTransferDelayInNanoSec
  After SCK delay time with nanosecond , set to 0 sets the minimum delay.It sets the boundary value if out of range that can be set.

15.2.3.2.0.20 Field Documentation

15.2.3.2.0.20.1 uint32_t dspi_master_ctar_config_t::baudRate

15.2.3.2.0.20.2 uint32_t dspi_master_ctar_config_t::bitsPerFrame

15.2.3.2.0.20.3 dspi_clock_polarity_t dspi_master_ctar_config_t::cpol

15.2.3.2.0.20.4 dspi_clock_phase_t dspi_master_ctar_config_t::cpha

15.2.3.2.0.20.5 dspi_shift_direction_t dspi_master_ctar_config_t::direction

15.2.3.2.0.20.6 uint32_t dspi_master_ctar_config_t::pcsToSckDelayInNanoSec

It sets the boundary value if out of range that can be set.

15.2.3.2.0.20.7 uint32_t dspi_master_ctar_config_t::lastSckToPcsDelayInNanoSec

15.2.3.2.0.20.8 uint32_t dspi_master_ctar_config_t::betweenTransferDelayInNanoSec

15.2.3.3 struct dspi_master_config_t

Data Fields

- dspi_ctar_selection_t whichCtar
  Desired CTAR to use.
- dspi_master_ctar_config_t ctarConfig
  Set the ctarConfig to the desired CTAR.
- dspi_which_pcs_t whichPcs
  Desired Peripheral Chip Select (pcs).
- dspi_pcs_polarity_config_t pcsActiveHighOrLow
  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 it's true.
- dspi_master_sample_point_t samplePoint
  Controls when the module master samples SIN in Modified Transfer Format.

15.2.3.3.0.21 Field Documentation

15.2.3.3.0.21.1 dspi_ctar_selection_t dspi_master_config_t::whichCtar

15.2.3.3.0.21.2 dspi_master_ctar_config_t dspi_master_config_t::ctarConfig

15.2.3.3.0.21.3 dspi_which_pcs_t dspi_master_config_t::whichPcs

15.2.3.3.0.21.4 dspi_pcs_polarity_config_t dspi_master_config_t::pcsActiveHighOrLow

15.2.3.3.0.21.5 bool dspi_master_config_t::enableContinuousSCK

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

15.2.3.3.0.21.6 bool dspi_master_config_t::enableRxFifoOverWrite

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

15.2.3.3.0.21.7 bool dspi_master_config_t::enableModifiedTimingFormat

15.2.3.3.0.21.8 dspi_master_sample_point_t dspi_master_config_t::samplePoint

It's valid only when CPHA=0.

15.2.3.4 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.

15.2.3.4.0.22 Field Documentation

15.2.3.4.0.22.1 uint32_t dspi_slave_ctar_config_t::bitsPerFrame

15.2.3.4.0.22.2 dspi_clock_polarity_t dspi_slave_ctar_config_t::cpol

15.2.3.4.0.22.3 dspi_clock_phase_t dspi_slave_ctar_config_t::cpha

Slave only supports MSB , does not support LSB.

15.2.3.5 struct dspi_slave_config_t

Data Fields

- dspi_ctar_selection_t whichCtar
  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 it's true.
- dspi_master_sample_point_t samplePoint
  Controls when the module master samples SIN in Modified Transfer Format.

15.2.3.5.0.23 Field Documentation

15.2.3.5.0.23.1 dspi_ctar_selection_t dspi_slave_config_t::whichCtar

15.2.3.5.0.23.2 dspi_slave_ctar_config_t dspi_slave_config_t::ctarConfig

15.2.3.5.0.23.3 bool dspi_slave_config_t::enableContinuousSCK

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

15.2.3.5.0.23.4 bool dspi_slave_config_t::enableRxFifoOverWrite

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

15.2.3.5.0.23.5 bool dspi_slave_config_t::enableModifiedTimingFormat

15.2.3.5.0.23.6 dspi_master_sample_point_t dspi_slave_config_t::samplePoint

It's valid only when CPHA=0.

15.2.3.6 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.

15.2.3.6.0.24 Field Documentation

15.2.3.6.0.24.1 uint8_t* dspi_transfer_t::txData

15.2.3.6.0.24.2 uint8_t* dspi_transfer_t::rxData

15.2.3.6.0.24.3 volatile size_t dspi_transfer_t::dataSize

15.2.3.6.0.24.4 uint32_t dspi_transfer_t::configFlags

15.2.3.7 struct _dspi_master_handle

Forward declaration of the _dspi_master_handle typedefs.

Data Fields

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

15.2.3.7.0.25 Field Documentation

15.2.3.7.0.25.1 uint32_t dspi_master_handle_t::bitsPerFrame

15.2.3.7.0.25.2 volatile uint32_t dspi_master_handle_t::command

15.2.3.7.0.25.3 volatile uint32_t dspi_master_handle_t::lastCommand

15.2.3.7.0.25.4 uint8_t dspi_master_handle_t::fifoSize

15.2.3.7.0.25.5 volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer

15.2.3.7.0.25.6 volatile bool dspi_master_handle_t::isThereExtraByte

15.2.3.7.0.25.7 uint8_t* volatile dspi_master_handle_t::txData

15.2.3.7.0.25.8 uint8_t* volatile dspi_master_handle_t::rxData

15.2.3.7.0.25.9 volatile size_t dspi_master_handle_t::remainingSendByteCount

15.2.3.7.0.25.10 volatile size_t dspi_master_handle_t::remainingReceiveByteCount

15.2.3.7.0.25.11 volatile uint8_t dspi_master_handle_t::state

15.2.3.7.0.25.12 dspi_master_transfer_callback_t dspi_master_handle_t::callback

15.2.3.7.0.25.13 void* dspi_master_handle_t::userData

15.2.3.8 struct _dspi_slave_handle

Forward declaration of the _dspi_slave_handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  Desired number of bits per frame.
- volatile bool isThereExtraByte
  Is there extra byte.
- uint8_t *volatile txData
  Send buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  Number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  Number of bytes remaining to receive.
- size_t totalByteCount
  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.

15.2.3.8.0.26 Field Documentation

15.2.3.8.0.26.1 uint32_t dspi_slave_handle_t::bitsPerFrame

15.2.3.8.0.26.2 volatile bool dspi_slave_handle_t::isThereExtraByte

15.2.3.8.0.26.3 uint8_t* volatile dspi_slave_handle_t::txData

15.2.3.8.0.26.4 uint8_t* volatile dspi_slave_handle_t::rxData

15.2.3.8.0.26.5 volatile size_t dspi_slave_handle_t::remainingSendByteCount

15.2.3.8.0.26.6 volatile size_t dspi_slave_handle_t::remainingReceiveByteCount

15.2.3.8.0.26.7 volatile uint8_t dspi_slave_handle_t::state

15.2.3.8.0.26.8 volatile uint32_t dspi_slave_handle_t::errorCount

15.2.3.8.0.26.9 dspi_slave_transfer_callback_t dspi_slave_handle_t::callback

15.2.3.8.0.26.10 void* dspi_slave_handle_t::userData

15.2.4 Macro Definition Documentation

15.2.4.1 #define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

15.2.4.2 #define DSPI_MASTER_DUMMY_DATA (0x00U)

15.2.4.3 #define DSPI_SLAVE_DUMMY_DATA (0x00U)

15.2.4.4 #define DSPI_MASTER_CTAR_SHIFT (0U)

15.2.4.5 #define DSPI_MASTER_CTAR_MASK (0x0FU)

15.2.4.6 #define DSPI_MASTER_PCS_SHIFT (4U)

15.2.4.7 #define DSPI_MASTER_PCS_MASK (0xF0U)

15.2.4.8 #define DSPI_SLAVE_CTAR_SHIFT (0U)

15.2.4.9 #define DSPI_SLAVE_CTAR_MASK (0x07U)

15.2.5 Typedef Documentation

15.2.5.1 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.

15.2.5.2 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.

15.2.6 Enumeration Type Documentation

15.2.6.1 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.

15.2.6.2 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 status above.

15.2.6.3 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.

15.2.6.4 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.

15.2.6.5 enum dspi_master_slave_mode_t

Enumerator

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

15.2.6.6 enum dspi_master_sample_point_t

This field is valid only when CPHA bit in 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.

15.2.6.7 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].

15.2.6.8 enum dspi_pcs_polarity_config_t

Enumerator

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

15.2.6.9 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).

15.2.6.10 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).

15.2.6.11 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.

15.2.6.12 enum dspi_shift_direction_t

Enumerator

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

15.2.6.13 enum dspi_delay_type_t

Enumerator

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

15.2.6.14 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 device do not support CTAR2.
kDSPI_Ctar3 CTAR3 selection option for master mode only , note that some device do not support CTAR3.
kDSPI_Ctar4 CTAR4 selection option for master mode only , note that some device do not support CTAR4.
kDSPI_Ctar5 CTAR5 selection option for master mode only , note that some device do not support CTAR5.
kDSPI_Ctar6 CTAR6 selection option for master mode only , note that some device do not support CTAR6.
kDSPI_Ctar7 CTAR7 selection option for master mode only , note that some device do not support CTAR7.

15.2.6.15 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 Is PCS signal continuous.
kDSPI_MasterActiveAfterTransfer Is PCS signal active after last frame transfer.

15.2.6.16 enum _dspi_transfer_config_flag_for_slave

Enumerator

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

15.2.6.17 enum _dspi_transfer_state

Enumerator

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

15.2.7 Function Documentation

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

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

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

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

Parameters

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

15.2.7.2 void DSPI_MasterGetDefaultConfig (dspi_master_config_t * *masterConfig*)

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

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

Parameters

masterConfig	pointer to dspi_master_config_t structure

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

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

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

Parameters

base	DSPI peripheral address.
slaveConfig	Pointer to structure dspi_master_config_t.

15.2.7.4 void DSPI_SlaveGetDefaultConfig (dspi_slave_config_t * *slaveConfig*)

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

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

Parameters

slaveConfig	pointer to dspi_slave_config_t structure.

15.2.7.5 void DSPI_Deinit (SPI_Type * *base*)

Call this API to disable the DSPI clock.

Parameters

base	DSPI peripheral address.

15.2.7.6 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.

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

Parameters

base	DSPI peripheral address.

Returns

The DSPI status(in SR register).

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

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

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

Parameters

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

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

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

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

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

Parameters

base	DSPI peripheral address.

mask	The interrupt mask, can use the enum _dspi_interrupt_enable.

15.2.7.10 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, can use the enum _dspi_interrupt_enable.

15.2.7.11 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 base and a DMA mask.

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

Parameters

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

15.2.7.12 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 base and a DMA mask.

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

Parameters

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

15.2.7.13 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.

15.2.7.14 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.

15.2.7.15 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.

15.2.7.16 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.

15.2.7.17 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.

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

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

Parameters

base	DSPI peripheral address.

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

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

Parameters

base	DSPI peripheral address.

15.2.7.20 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, the caller must pass in a logic 0 (false) for the particular FIFO configuration. To enable, the caller must pass in a logic 1 (true).

Parameters

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

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

Parameters

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

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

For example, PCS0 and PCS1 set to active low and other PCS 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 , can use the enum _dspi_pcs_polarity.

15.2.7.23 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

15.2.7.24 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 type dspi_delay_type_t.

The user passes the delay to configure along with the prescaler and scaler value. This allows the user to directly set the prescaler/scaler values if they have pre-calculated them or if they simply wish 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

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

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

These delay names are available in type dspi_delay_type_t.

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

Parameters

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

Returns

The actual calculated delay value.

15.2.7.26 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 command structure.
data	The data word to be sent.

15.2.7.27 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(). User may use the initialized structure unchanged in DSPI_MasterWrite_xx() or modify the structure before calling DSPI_MasterWrite_xx(). Example:

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

Parameters

command	pointer to dspi_command_data_config_t structure.

15.2.7.28 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, receive data is available when transmit completes.

Parameters

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

15.2.7.29 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 command structure.

Returns

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

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

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

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

15.2.7.31 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.

15.2.7.32 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.

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

Parameters

base	DSPI peripheral address.

Returns

The data from the read data buffer.

15.2.7.34 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.

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

This function transfers data with 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 dspi_transfer_t structure.

Returns

status of status_t.

15.2.7.36 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 have been transferred, the callback function is called.

Parameters

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

Returns

status of status_t.

15.2.7.37 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 dspi_master_handle_t structure which stores the transfer state.
count	Number of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

15.2.7.38 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 dspi_master_handle_t structure which stores the transfer state.

15.2.7.39 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 dspi_master_handle_t structure which stores the transfer state.

15.2.7.40 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 dspi_slave_handle_t.
base	DSPI peripheral base address.
callback	DSPI callback.
userData	callback function parameter.

15.2.7.41 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 have been transferred, the callback function is called.

Parameters

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

Returns

status of status_t.

15.2.7.42 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 dspi_master_handle_t structure which stores the transfer state.
count	Number of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

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

This function aborts transfer using an interrupt.

Parameters

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

15.2.7.44 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 dspi_slave_handle_t structure which stores the transfer state.

15.3 DSPI DMA Driver

15.3.1 Overview

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

Files

- file fsl_dspi_dma.h

Data Structures

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

Typedefs

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

Functions

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

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

15.3.2 Data Structure Documentation

15.3.2.1 struct _dspi_master_dma_handle

Forward declaration of the DSPI DMA master handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  Desired number of bits per frame.
- volatile uint32_t command
  Desired data command.
- volatile uint32_t lastCommand
  Desired last data command.
- uint8_t fifoSize
  FIFO dataSize.
- volatile bool isPcsActiveAfterTransfer
  Is PCS signal keep active after the last frame transfer.
- volatile bool isThereExtraByte
  Is there extra byte.
- uint8_t *volatile txData
  Send buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  Number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  Number of bytes remaining to receive.
- size_t totalByteCount
  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 , _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

15.3.2.1.0.27 Field Documentation

15.3.2.1.0.27.1 uint32_t dspi_master_dma_handle_t::bitsPerFrame

15.3.2.1.0.27.2 volatile uint32_t dspi_master_dma_handle_t::command

15.3.2.1.0.27.3 volatile uint32_t dspi_master_dma_handle_t::lastCommand

15.3.2.1.0.27.4 uint8_t dspi_master_dma_handle_t::fifoSize

15.3.2.1.0.27.5 volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer

15.3.2.1.0.27.6 volatile bool dspi_master_dma_handle_t::isThereExtraByte

15.3.2.1.0.27.7 uint8_t* volatile dspi_master_dma_handle_t::txData

15.3.2.1.0.27.8 uint8_t* volatile dspi_master_dma_handle_t::rxData

15.3.2.1.0.27.9 volatile size_t dspi_master_dma_handle_t::remainingSendByteCount

15.3.2.1.0.27.10 volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount

15.3.2.1.0.27.11 uint32_t dspi_master_dma_handle_t::rxBuffIfNull

15.3.2.1.0.27.12 uint32_t dspi_master_dma_handle_t::txBuffIfNull

15.3.2.1.0.27.13 volatile uint8_t dspi_master_dma_handle_t::state

15.3.2.1.0.27.14 dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback

15.3.2.1.0.27.15 void* dspi_master_dma_handle_t::userData

15.3.2.2 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
  Is there extra byte.
- uint8_t ∗volatile txData
  Send buffer.
- uint8_t ∗volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  Number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  Number of bytes remaining to receive.
- size_t totalByteCount
  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 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

15.3.2.2.0.28 Field Documentation

15.3.2.2.0.28.1 uint32_t dspi_slave_dma_handle_t::bitsPerFrame

15.3.2.2.0.28.2 volatile bool dspi_slave_dma_handle_t::isThereExtraByte

15.3.2.2.0.28.3 uint8_t* volatile dspi_slave_dma_handle_t::txData

15.3.2.2.0.28.4 uint8_t* volatile dspi_slave_dma_handle_t::rxData

15.3.2.2.0.28.5 volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount

15.3.2.2.0.28.6 volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount

15.3.2.2.0.28.7 uint32_t dspi_slave_dma_handle_t::rxBuffIfNull

15.3.2.2.0.28.8 uint32_t dspi_slave_dma_handle_t::txBuffIfNull

15.3.2.2.0.28.9 uint32_t dspi_slave_dma_handle_t::txLastData

15.3.2.2.0.28.10 volatile uint8_t dspi_slave_dma_handle_t::state

15.3.2.2.0.28.11 uint32_t dspi_slave_dma_handle_t::errorCount

15.3.2.2.0.28.12 dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback

15.3.2.2.0.28.13 void* dspi_slave_dma_handle_t::userData

15.3.3 Typedef Documentation

15.3.3.1 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.

15.3.3.2 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.

15.3.4 Function Documentation

15.3.4.1 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	callback function parameter.
dmaRxRegToRxDataHandle	dmaRxRegToRxDataHandle pointer to dma_handle_t.
dmaTxDataToIntermediaryHandle	dmaTxDataToIntermediaryHandle pointer to dma_handle_t.
dmaIntermediaryToTxRegHandle	dmaIntermediaryToTxRegHandle pointer to dma_handle_t.

15.3.4.2 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 master DMA transfer cannot support the transfer_size of 1 when the bitsPerFrame is greater than 8.

Parameters

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

Returns

status of status_t.

15.3.4.3 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	pointer to dspi_master_dma_handle_t structure which stores the transfer state.

15.3.4.4 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	pointer to dspi_master_dma_handle_t structure which stores the transfer state.
count	number point of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

15.3.4.5 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 one time 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	callback function parameter.
dmaRxRegToRxDataHandle	dmaRxRegToRxDataHandle pointer to dma_handle_t.
dmaTxDataToTxRegHandle	dmaTxDataToTxRegHandle pointer to dma_handle_t.

15.3.4.6 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 cannot support the transfer_size of 1 when the bitsPerFrame is greater than 8.

Parameters

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

Returns

status of status_t.

15.3.4.7 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	pointer to dspi_slave_dma_handle_t structure which stores the transfer state.

15.3.4.8 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	pointer to dspi_slave_dma_handle_t structure which stores the transfer state.
count	number point of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

15.4 DSPI eDMA Driver

15.4.1 Overview

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

Files

- file fsl_dspi_edma.h

Data Structures

- struct dspi_master_edma_handle_t
  DSPI master eDMA transfer handle structure used for transactional API. More...
- struct dspi_slave_edma_handle_t
  DSPI slave eDMA transfer handle structure used for 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 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 using eDMA.
- status_t DSPI_SlaveTransferGetCountEDMA (SPI_Type *base, dspi_slave_edma_handle_t *handle, size_t *count)
  Gets the slave eDMA transfer count.

15.4.2 Data Structure Documentation

15.4.2.1 struct _dspi_master_edma_handle

Forward declaration of the DSPI eDMA master handle typedefs.

Data Fields

- uint32_t bitsPerFrame
  Desired number of bits per frame.
- volatile uint32_t command
  Desired data command.
- volatile uint32_t lastCommand
  Desired last data command.
- uint8_t fifoSize
  FIFO dataSize.
- volatile bool isPcsActiveAfterTransfer
  Is PCS signal keep active after the last frame transfer.
- volatile bool isThereExtraByte
  Is there extra byte.
- uint8_t *volatile txData
  Send buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t remainingSendByteCount
  Number of bytes remaining to send.
- volatile size_t remainingReceiveByteCount
  Number of bytes remaining to receive.
- size_t totalByteCount
  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 , _dspi_transfer_state.

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

15.4.2.1.0.29 Field Documentation

15.4.2.1.0.29.1 uint32_t dspi_master_edma_handle_t::bitsPerFrame

15.4.2.1.0.29.2 volatile uint32_t dspi_master_edma_handle_t::command

15.4.2.1.0.29.3 volatile uint32_t dspi_master_edma_handle_t::lastCommand

15.4.2.1.0.29.4 uint8_t dspi_master_edma_handle_t::fifoSize

15.4.2.1.0.29.5 volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer

15.4.2.1.0.29.6 volatile bool dspi_master_edma_handle_t::isThereExtraByte

15.4.2.1.0.29.7 uint8_t* volatile dspi_master_edma_handle_t::txData

15.4.2.1.0.29.8 uint8_t* volatile dspi_master_edma_handle_t::rxData

15.4.2.1.0.29.9 volatile size_t dspi_master_edma_handle_t::remainingSendByteCount

15.4.2.1.0.29.10 volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount

15.4.2.1.0.29.11 uint32_t dspi_master_edma_handle_t::rxBuffIfNull

15.4.2.1.0.29.12 uint32_t dspi_master_edma_handle_t::txBuffIfNull

15.4.2.1.0.29.13 volatile uint8_t dspi_master_edma_handle_t::state

15.4.2.1.0.29.14 dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback

15.4.2.1.0.29.15 void* dspi_master_edma_handle_t::userData

15.4.2.2 struct _dspi_slave_edma_handle

Forward declaration of the DSPI eDMA slave handle typedefs.

Data Fields

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

15.4.2.2.0.30 Field Documentation

15.4.2.2.0.30.1 uint32_t dspi_slave_edma_handle_t::bitsPerFrame

15.4.2.2.0.30.2 volatile bool dspi_slave_edma_handle_t::isThereExtraByte

15.4.2.2.0.30.3 uint8_t* volatile dspi_slave_edma_handle_t::txData

15.4.2.2.0.30.4 uint8_t* volatile dspi_slave_edma_handle_t::rxData

15.4.2.2.0.30.5 volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount

15.4.2.2.0.30.6 volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount

15.4.2.2.0.30.7 uint32_t dspi_slave_edma_handle_t::rxBuffIfNull

15.4.2.2.0.30.8 uint32_t dspi_slave_edma_handle_t::txBuffIfNull

15.4.2.2.0.30.9 uint32_t dspi_slave_edma_handle_t::txLastData

15.4.2.2.0.30.10 volatile uint8_t dspi_slave_edma_handle_t::state

15.4.2.2.0.30.11 uint32_t dspi_slave_edma_handle_t::errorCount

15.4.2.2.0.30.12 dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback

15.4.2.2.0.30.13 void* dspi_slave_edma_handle_t::userData

15.4.3 Typedef Documentation

15.4.3.1 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	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.

15.4.3.2 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	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.

15.4.4 Function Documentation

15.4.4.1 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, user need only 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	callback function parameter.
edmaRxRegToRxDataHandle	edmaRxRegToRxDataHandle pointer to edma_handle_t.
edmaTxDataToIntermediaryHandle	edmaTxDataToIntermediaryHandle pointer to edma_handle_t.
edmaIntermediaryToTxRegHandle	edmaIntermediaryToTxRegHandle pointer to edma_handle_t.

15.4.4.2 status_t DSPI_MasterTransferEDMA (SPI_Type * *base*, dspi_master_edma_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function transfer data using eDMA. This is non-blocking function, which returns right away. When all data have been transfer, the callback function is called.

Parameters

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

Returns

status of status_t.

15.4.4.3 void DSPI_MasterTransferAbortEDMA (SPI_Type * *base*, dspi_master_edma_handle_t * *handle*)

This function aborts a transfer which using eDMA.

Parameters

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

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

This function get the master eDMA transfer count.

Parameters

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

Returns

status of status_t.

15.4.4.5 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	callback function parameter.
edmaRxRegToRxDataHandle	edmaRxRegToRxDataHandle pointer to edma_handle_t.
edmaTxDataToTxRegHandle	edmaTxDataToTxRegHandle pointer to edma_handle_t.

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

This function transfer data using eDMA. This is non-blocking function, which returns right away. When all data have been transfer, the callback function is called. Note that slave EDMA transfer cannot support the situation that transfer_size is 1 when the bitsPerFrame is greater than 8 .

Parameters

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

Returns

status of status_t.

15.4.4.7 void DSPI_SlaveTransferAbortEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*)

This function aborts a transfer which using eDMA.

Parameters

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

15.4.4.8 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	pointer to dspi_slave_edma_handle_t structure which stores the transfer state.
count	Number of bytes transferred so far by the non-blocking transaction.

Returns

status of status_t.

15.5 DSPI FreeRTOS Driver

15.5.1 Overview

Files

- file fsl_dspi_freertos.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI FreeRTOS handle. More...

Driver version

- #define FSL_DSPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DSPI FreeRTOS driver version 2.0.0.

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 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 SPI transfer.

15.5.2 Data Structure Documentation

15.5.2.1 struct dspi_rtos_handle_t

DSPI FreeRTOS handle.

Data Fields

- SPI_Type * base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.

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

15.5.3 Macro Definition Documentation

15.5.3.1 #define FSL_DSPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

15.5.4 Function Documentation

15.5.4.1 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 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	Configuration structure to set-up DSPI in master mode.
srcClock_Hz	Frequency of input clock of the DSPI module.

Returns

status of the operation.

15.5.4.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and related RTOS context.

Parameters

handle	The RTOS DSPI handle.

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

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

Parameters

handle	The RTOS DSPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

15.6 DSPI µCOS/II Driver

15.6.1 Overview

Files

- file fsl_dspi_ucosii.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI µC/OS-II handle. More...

Driver version

- #define FSL_DSPI_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DSPI µC/OS-II driver version 2.0.0.

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 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 SPI transfer.

15.6.2 Data Structure Documentation

15.6.2.1 struct dspi_rtos_handle_t

DSPI µC/OS-II handle.

Data Fields

- SPI_Type * base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.

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

15.6.3 Macro Definition Documentation

15.6.3.1 #define FSL_DSPI_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

15.6.4 Function Documentation

15.6.4.1 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 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	Configuration structure to set-up DSPI in master mode.
srcClock_Hz	Frequency of input clock of the DSPI module.

Returns

status of the operation.

15.6.4.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and related RTOS context.

Parameters

handle	The RTOS DSPI handle.

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

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

Parameters

handle	The RTOS DSPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

15.7 DSPI µCOS/III Driver

15.7.1 Overview

Files

- file fsl_dspi_ucosiii.h

Data Structures

- struct dspi_rtos_handle_t
  DSPI µC/OS-III handle. More...

Driver version

- #define FSL_DSPI_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DSPI µC/OS-III driver version 2.0.0.

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 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 SPI transfer.

15.7.2 Data Structure Documentation

15.7.2.1 struct dspi_rtos_handle_t

DSPI µC/OS-III handle.

Data Fields

- SPI_Type * base
  DSPI base address.
- dspi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a transfer.

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

15.7.3 Macro Definition Documentation

15.7.3.1 #define FSL_DSPI_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

15.7.4 Function Documentation

15.7.4.1 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 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	Configuration structure to set-up DSPI in master mode.
srcClock_Hz	Frequency of input clock of the DSPI module.

Returns

status of the operation.

15.7.4.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * *handle*)

This function deinitializes the DSPI module and related RTOS context.

Parameters

handle	The RTOS DSPI handle.

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

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

Parameters

handle	The RTOS DSPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 16
eDMA: Enhanced Direct Memory Access Controller Driver

16.1 Overview

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

16.2 Typical use case

16.2.1 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);
/* Wait for EDMA transfer finish
while (transferDone != true);
```

Files

- file fsl_edma.h

Data Structures

- struct edma_config_t
  eDMA global configuration structure. *More...*
- struct edma_transfer_config_t
  eDMA transfer configuration *More...*
- struct edma_channel_Preemption_config_t
  eDMA channel priority configuration *More...*
- struct edma_minor_offset_config_t
  eDMA minor offset configuration *More...*
- struct edma_tcd_t
  eDMA TCD. *More...*
- struct edma_handle_t
  eDMA transfer handle structure *More...*

Macros

- #define DMA_DCHPRI_INDEX(channel) (((channel) & ~0x03U) | (3 - ((channel)&0x03U)))
  Compute the offset unit from DCHPRI3.

Typical use case

- #define DMA_DCHPRIn(base, channel) ((volatile uint8_t *)&(base->DCHPRI3))[DMA_DCHPRI_INDEX(channel)]
  Get the pointer of DCHPRIn.

Typedefs

- typedef void(* edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)
  Define Callback function for eDMA.

Enumerations

- enum edma_transfer_size_t {
  kEDMA_TransferSize1Bytes = 0x0U,
  kEDMA_TransferSize2Bytes = 0x1U,
  kEDMA_TransferSize4Bytes = 0x2U,
  kEDMA_TransferSize16Bytes = 0x4U,
  kEDMA_TransferSize32Bytes = 0x5U }
  eDMA transfer configuration
- enum edma_modulo_t {

kEDMA_ModuloDisable = 0x0U,
kEDMA_Modulo2bytes,
kEDMA_Modulo4bytes,
kEDMA_Modulo8bytes,
kEDMA_Modulo16bytes,
kEDMA_Modulo32bytes,
kEDMA_Modulo64bytes,
kEDMA_Modulo128bytes,
kEDMA_Modulo256bytes,
kEDMA_Modulo512bytes,
kEDMA_Modulo1Kbytes,
kEDMA_Modulo2Kbytes,
kEDMA_Modulo4Kbytes,
kEDMA_Modulo8Kbytes,
kEDMA_Modulo16Kbytes,
kEDMA_Modulo32Kbytes,
kEDMA_Modulo64Kbytes,
kEDMA_Modulo128Kbytes,
kEDMA_Modulo256Kbytes,
kEDMA_Modulo512Kbytes,
kEDMA_Modulo1Mbytes,
kEDMA_Modulo2Mbytes,
kEDMA_Modulo4Mbytes,
kEDMA_Modulo8Mbytes,
kEDMA_Modulo16Mbytes,
kEDMA_Modulo32Mbytes,
kEDMA_Modulo64Mbytes,
kEDMA_Modulo128Mbytes,
kEDMA_Modulo256Mbytes,
kEDMA_Modulo512Mbytes,
kEDMA_Modulo1Gbytes,
kEDMA_Modulo2Gbytes }
eDMA modulo configuration

- enum edma_bandwidth_t {
  kEDMA_BandwidthStallNone = 0x0U,
  kEDMA_BandwidthStall4Cycle = 0x2U,
  kEDMA_BandwidthStall8Cycle = 0x3U }
  Bandwidth control.
- enum edma_channel_link_type_t {
  kEDMA_LinkNone = 0x0U,
  kEDMA_MinorLink,
  kEDMA_MajorLink }
  Channel link type.
- enum _edma_channel_status_flags {

kEDMA_DoneFlag = 0x1U,
kEDMA_ErrorFlag = 0x2U,
kEDMA_InterruptFlag = 0x4U }
eDMA channel status flags.

- enum _edma_error_status_flags {
kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,
kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,
kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,
kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,
kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,
kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,
kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,
kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,
kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,
kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,
kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,
kEDMA_ValidFlag = DMA_ES_VLD_MASK }
eDMA channel error status flags.
- enum edma_interrupt_enable_t {
kEDMA_ErrorInterruptEnable = 0x1U,
kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,
kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }
eDMA interrupt source
- enum edma_transfer_type_t {
kEDMA_MemoryToMemory = 0x0U,
kEDMA_PeripheralToMemory,
kEDMA_MemoryToPeripheral }
eDMA transfer type
- enum _edma_transfer_status {
kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),
kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }
eDMA transfer status

Driver version

- #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
eDMA driver version

eDMA initialization and De-initialization

- void EDMA_Init (DMA_Type *base, const edma_config_t *config)
Initializes eDMA peripheral.
- void EDMA_Deinit (DMA_Type *base)
Deinitializes 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 a default value.
- 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 destination modulo for 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 eDMA TCD.
- static void EDMA_TcdSetBandWidth (edma_tcd_t *tcd, edma_bandwidth_t bandWidth)
  Sets the bandwidth for the eDMA TCD.
- void EDMA_TcdSetModulo (edma_tcd_t *tcd, edma_modulo_t srcModulo, edma_modulo_t destModulo)
  Sets the source modulo and destination modulo for 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.

Typical use case

- 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 software trigger.

eDMA Channel Status Operation

- uint32_t EDMA_GetRemainingBytes (DMA_Type *base, uint32_t channel)
  Gets the Remaining bytes from the eDMA current channel TCD.
- static uint32_t EDMA_GetErrorStatusFlags (DMA_Type *base)
  Gets the eDMA channel error status flags.
- uint32_t EDMA_GetChannelStatusFlags (DMA_Type *base, uint32_t channel)
  Gets the eDMA channel status flags.
- void EDMA_ClearChannelStatusFlags (DMA_Type *base, uint32_t channel, uint32_t mask)
  Clears the eDMA channel status flags.

eDMA Transactional Operation

- void EDMA_CreateHandle (edma_handle_t *handle, DMA_Type *base, uint32_t channel)
  Creates the eDMA handle.
- void EDMA_InstallTCDMemory (edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
  Installs the TCDs memory pool into 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 start transfer.
- void EDMA_StopTransfer (edma_handle_t *handle)
  eDMA stop transfer.
- void EDMA_AbortTransfer (edma_handle_t *handle)
  eDMA abort transfer.
- void EDMA_HandleIRQ (edma_handle_t *handle)
  eDMA IRQ handler for current major loop transfer complete.

16.3 Data Structure Documentation

16.3.1 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.

16.3.1.0.0.31 Field Documentation

16.3.1.0.0.31.1 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.

16.3.1.0.0.31.2 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.

16.3.1.0.0.31.3 bool edma_config_t::enableDebugMode

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

16.3.2 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.
- uint16_t minorLoopBytes
  Bytes to transfer in a minor loop.
- uint32_t majorLoopCounts
  Major loop iteration count.

16.3.2.0.0.32 Field Documentation

16.3.2.0.0.32.1 uint32_t edma_transfer_config_t::srcAddr

16.3.2.0.0.32.2 uint32_t edma_transfer_config_t::destAddr

16.3.2.0.0.32.3 edma_transfer_size_t edma_transfer_config_t::srcTransferSize

16.3.2.0.0.32.4 edma_transfer_size_t edma_transfer_config_t::destTransferSize

16.3.2.0.0.32.5 int16_t edma_transfer_config_t::srcOffset

16.3.2.0.0.32.6 int16_t edma_transfer_config_t::destOffset

16.3.2.0.0.32.7 uint32_t edma_transfer_config_t::majorLoopCounts

16.3.3 struct edma_channel_Preemption_config_t

Data Fields

- bool enableChannelPreemption
  If true: channel can be suspended by other channel with higher priority.
- bool enablePreemptAbility
  If true: channel can suspend other channel with low priority.
- uint8_t channelPriority
  Channel priority.

16.3.4 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 minor loop mapping.

16.3.4.0.0.33 Field Documentation

16.3.4.0.0.33.1 bool edma_minor_offset_config_t::enableSrcMinorOffset

16.3.4.0.0.33.2 bool edma_minor_offset_config_t::enableDestMinorOffset

16.3.4.0.0.33.3 uint32_t edma_minor_offset_config_t::minorOffset

16.3.5 struct edma_tcd_t

This structure is same as TCD register which is described in reference manual, and is used to configure scatter/gather feature as a next hardware TCD.

Data Fields

- __IO uint32_t SADDR
  SADDR register, used to save source address.
- __IO uint16_t SOFF
  SOFF register, save offset bytes every transfer.
- __IO uint16_t ATTR
  ATTR register, source/destination transfer size and modulo.
- __IO uint32_t NBYTES
  Nbytes register, minor loop length in bytes.
- __IO uint32_t SLAST
  SLAST register.
- __IO uint32_t DADDR
  DADDR register, used for destination address.
- __IO uint16_t DOFF
  DOFF register, used for destination offset.
- __IO uint16_t CITER
  CITER register, current minor loop numbers, for unfinished minor loop.
- __IO uint32_t DLAST_SGA
  DLASTSGA register, next stcd address used in scatter-gather mode.
- __IO uint16_t CSR
  CSR register, for TCD control status.
- __IO uint16_t BITER
  BITER register, begin minor loop count.

16.3.5.0.0.34 Field Documentation

16.3.5.0.0.34.1 __IO uint16_t edma_tcd_t::CITER

16.3.5.0.0.34.2 __IO uint16_t edma_tcd_t::BITER

16.3.6 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.

16.3.6.0.0.35 Field Documentation

16.3.6.0.0.35.1 edma_callback edma_handle_t::callback

16.3.6.0.0.35.2 void* edma_handle_t::userData

16.3.6.0.0.35.3 DMA_Type* edma_handle_t::base

16.3.6.0.0.35.4 edma_tcd_t* edma_handle_t::tcdPool

16.3.6.0.0.35.5 uint8_t edma_handle_t::channel

16.3.6.0.0.35.6 volatile int8_t edma_handle_t::header

16.3.6.0.0.35.7 volatile int8_t edma_handle_t::tail

16.3.6.0.0.35.8 volatile int8_t edma_handle_t::tcdUsed

16.3.6.0.0.35.9 volatile int8_t edma_handle_t::tcdSize

16.3.6.0.0.35.10 uint8_t edma_handle_t::flags

16.4 Macro Definition Documentation

16.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

16.5 Typedef Documentation

16.5.1 typedef void(* edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)

16.6 Enumeration Type Documentation

16.6.1 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.

16.6.2 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 1K bytes.
kEDMA_Modulo2Kbytes Circular buffer size is 2K bytes.
kEDMA_Modulo4Kbytes Circular buffer size is 4K bytes.
kEDMA_Modulo8Kbytes Circular buffer size is 8K bytes.
kEDMA_Modulo16Kbytes Circular buffer size is 16K bytes.
kEDMA_Modulo32Kbytes Circular buffer size is 32K bytes.
kEDMA_Modulo64Kbytes Circular buffer size is 64K bytes.
kEDMA_Modulo128Kbytes Circular buffer size is 128K bytes.
kEDMA_Modulo256Kbytes Circular buffer size is 256K bytes.
kEDMA_Modulo512Kbytes Circular buffer size is 512K bytes.
kEDMA_Modulo1Mbytes Circular buffer size is 1M bytes.
kEDMA_Modulo2Mbytes Circular buffer size is 2M bytes.
kEDMA_Modulo4Mbytes Circular buffer size is 4M bytes.
kEDMA_Modulo8Mbytes Circular buffer size is 8M bytes.
kEDMA_Modulo16Mbytes Circular buffer size is 16M bytes.
kEDMA_Modulo32Mbytes Circular buffer size is 32M bytes.
kEDMA_Modulo64Mbytes Circular buffer size is 64M bytes.
kEDMA_Modulo128Mbytes Circular buffer size is 128M bytes.
kEDMA_Modulo256Mbytes Circular buffer size is 256M bytes.
kEDMA_Modulo512Mbytes Circular buffer size is 512M bytes.
kEDMA_Modulo1Gbytes Circular buffer size is 1G bytes.
kEDMA_Modulo2Gbytes Circular buffer size is 2G bytes.

16.6.3 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.

16.6.4 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.

16.6.5 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

16.6.6 enum _edma_error_status_flags

Enumerator

kEDMA_DestinationBusErrorFlag Bus error on destination address.
kEDMA_SourceBusErrorFlag Bus error on the source address.
kEDMA_ScatterGatherErrorFlag Error on the Scatter/Gather address, not 32byte aligned.
kEDMA_NbytesErrorFlag NBYTES/CITER configuration error.
kEDMA_DestinationOffsetErrorFlag Destination offset not aligned with destination size.
kEDMA_DestinationAddressErrorFlag Destination address not aligned with destination size.
kEDMA_SourceOffsetErrorFlag Source offset not aligned with source size.
kEDMA_SourceAddressErrorFlag Source address not aligned with source size.
kEDMA_ErrorChannelFlag Error channel number of the cancelled channel number.
kEDMA_ChannelPriorityErrorFlag Channel priority is not unique.
kEDMA_TransferCanceledFlag Transfer cancelled.
kEDMA_ValidFlag No error occurred, this bit will be 0, otherwise be 1.

16.6.7 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.

16.6.8 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.

16.6.9 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.

16.7 Function Documentation

16.7.1 void EDMA_Init (DMA_Type * *base*, const edma_config_t * *config*)

This function ungates the eDMA clock and configure eDMA peripheral according to the configuration structure.

Parameters

base	eDMA peripheral base address.
config	Pointer to configuration structure, see "edma_config_t".

Note

This function enable the minor loop map feature.

16.7.2 void EDMA_Deinit (DMA_Type * *base*)

This function gates the eDMA clock.

Parameters

base	eDMA peripheral base address.

16.7.3 void EDMA_GetDefaultConfig (edma_config_t * *config*)

This function sets the configuration structure to a default value. The default configuration is set to the following value:

```
config.enableContinuousLinkMode = false;
config.enableHaltOnError = true;
config.enableRoundRobinArbitration = false;
config.enableDebugMode = false;
```

Parameters

config	Pointer to eDMA configuration structure.

16.7.4 void EDMA_ResetChannel (DMA_Type * *base*, uint32_t *channel*)

This function sets TCD registers for this channel to default value.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Note

This function must not be called while the channel transfer is on-going, or it will case unpredicated results.
This function will enable auto stop request feature.

16.7.5 void EDMA_SetTransferConfig (DMA_Type * *base*, uint32_t *channel*, const edma_transfer_config_t * *config*, edma_tcd_t * *nextTcd*)

This function configure 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 user do not want to enable scatter/gather feature.

Note

If nextTcd is not NULL, it means scatter gather feature will be enabled. And DREQ bit will be cleared in the previous transfer configuration which will be set in eDMA_ResetChannel.

16.7.6 void EDMA_SetMinorOffsetConfig (DMA_Type * *base*, uint32_t *channel*, const edma_minor_offset_config_t * *config*)

Minor offset means signed-extended value added to source address or destination address after each minor loop.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
config	Pointer to Minor offset configuration structure.

16.7.7 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	Pointer to channel preemption configuration structure.

16.7.8 void EDMA_SetChannelLink (DMA_Type * *base*, uint32_t *channel*, edma_channel_link_type_t *type*, uint32_t *linkedChannel*)

This function configures minor link or 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	Channel link type, it can be one of: • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink
linkedChannel	The linked channel number.

Note

User should ensure that DONE flag is cleared before call this interface, or the configuration will be invalid.

16.7.9 void EDMA_SetBandWidth (DMA_Type * *base*, uint32_t *channel*, edma_bandwidth_t *bandWidth*)

In general, 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	Bandwidth setting, it can be one of: • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

16.7.10 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	Source modulo value.
destModulo	Destination modulo value.

16.7.11 static void EDMA_EnableAutoStopRequest (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

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

16.7.12 void EDMA_EnableChannelInterrupts (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of interrupt source to be set. User need to use the defined edma_interrupt-_enable_t type.

16.7.13 void EDMA_DisableChannelInterrupts (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of interrupt source to be set. Use the defined edma_interrupt_enable_t type.

16.7.14 void EDMA_TcdReset (edma_tcd_t * *tcd*)

This function sets all fields for this TCD structure to default value.

Parameters

tcd	Pointer to the TCD structure.

Note

This function will enable auto stop request feature.

16.7.15 void EDMA_TcdSetTransferConfig (edma_tcd_t * *tcd*, const edma_transfer_config_t * *config*, edma_tcd_t * *nextTcd*)

TCD is a transfer control descriptor. The content of the TCD is the same as hardware TCD registers. STCD is used in 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);
```

Function Documentation

```
edma_tcd_t nextTcd __aligned(32);
EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
```

Parameters

tcd	Pointer to the TCD structure.
config	Pointer to eDMA transfer configuration structure.
nextTcd	Pointer to the next TCD structure. It can be NULL if user do not want to enable scatter/gather feature.

Note

TCD address should be 32 bytes aligned, or it will cause eDMA error.
If nextTcd is not NULL, it means scatter gather feature will be enabled. And DREQ bit will be cleared in the previous transfer configuration which will be set in EDMA_TcdReset.

16.7.16 void EDMA_TcdSetMinorOffsetConfig (edma_tcd_t * *tcd,* const edma_minor_offset_config_t * *config*)

Minor offset is a signed-extended value added to the source address or destination address after each minor loop.

Parameters

tcd	Point to the TCD structure.
config	Pointer to Minor offset configuration structure.

16.7.17 void EDMA_TcdSetChannelLink (edma_tcd_t * *tcd,* edma_channel_link_type_t *type,* uint32_t *linkedChannel*)

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note

User should ensure that DONE flag is cleared before call this interface, or the configuration will be 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.

16.7.18 static void EDMA_TcdSetBandWidth (edma_tcd_t * *tcd,* edma_bandwidth_t *bandWidth*) [inline], [static]

In general, because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. 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	Point to the TCD structure.
bandWidth	Bandwidth setting, it can be one of: • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

16.7.19 void EDMA_TcdSetModulo (edma_tcd_t * *tcd,* edma_modulo_t *srcModulo,* edma_modulo_t *destModulo*)

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

Parameters

tcd	Point to the TCD structure.

srcModulo	Source modulo value.
destModulo	Destination modulo value.

16.7.20 static void EDMA_TcdEnableAutoStopRequest (edma_tcd_t * *tcd,* bool *enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

tcd	Point to the TCD structure.
enable	The command for enable(ture) or disable(false).

16.7.21 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. User need to use the defined edma_interrupt-_enable_t type.

16.7.22 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. User need to use the defined edma_interrupt-_enable_t type.

16.7.23 static void EDMA_EnableChannelRequest (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

This function enables the hardware channel request.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

16.7.24 static void EDMA_DisableChannelRequest (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

This function disables the hardware channel request.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

16.7.25 static void EDMA_TriggerChannelStart (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

This function starts a minor loop transfer.

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

16.7.26 uint32_t EDMA_GetRemainingBytes (DMA_Type * *base,* uint32_t *channel*)

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

Parameters

base	eDMA peripheral base address.

channel	eDMA channel number.

Returns

Bytes have not been transferred yet for the current TCD.

Note

This function can only be used to get unfinished bytes of transfer without the next TCD, or it might be inaccuracy.

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

Parameters

base	eDMA peripheral base address.

Returns

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

16.7.28 uint32_t EDMA_GetChannelStatusFlags (DMA_Type * *base,* uint32_t *channel*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.

Returns

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

16.7.29 void EDMA_ClearChannelStatusFlags (DMA_Type * *base,* uint32_t *channel,* uint32_t *mask*)

Parameters

base	eDMA peripheral base address.
channel	eDMA channel number.
mask	The mask of channel status to be cleared. User need to use the defined _edma_channel_status_flags type.

16.7.30 void EDMA_CreateHandle (edma_handle_t * *handle*, DMA_Type * *base*, uint32_t *channel*)

This function is called if using transaction API for eDMA. This function initializes the internal state of eDMA handle.

Parameters

handle	eDMA handle pointer. The eDMA handle stores callback function and parameters.
base	eDMA peripheral base address.
channel	eDMA channel number.

16.7.31 void EDMA_InstallTCDMemory (edma_handle_t * *handle*, edma_tcd_t * *tcdPool*, uint32_t *tcdSize*)

This function is called after the EDMA_CreateHandle to use scatter/gather feature.

Parameters

handle	eDMA handle pointer.
tcdPool	Memory pool to store TCDs. It must be 32 bytes aligned.
tcdSize	The number of TCD slots.

16.7.32 void EDMA_SetCallback (edma_handle_t * *handle*, edma_callback *callback*, void * *userData*)

This callback is called in 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	Parameter for callback function.

16.7.33 void EDMA_PrepareTransfer (edma_transfer_config_t * *config*, void * *srcAddr*, uint32_t *srcWidth*, void * *destAddr*, uint32_t *destWidth*, uint32_t *bytesEachRequest*, uint32_t *transferBytes*, edma_transfer_type_t *type*)

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

Parameters

config	The user configuration structure of type edma_transfer_t.
srcAddr	eDMA transfer source address.
srcWidth	eDMA transfer source address width(bytes).
destAddr	eDMA transfer destination address.
destWidth	eDMA transfer destination address width(bytes).
bytesEach-Request	eDMA transfer bytes per channel request.
transferBytes	eDMA transfer bytes to be transferred.
type	eDMA transfer type.

Note

The data address and the data width must be consistent. For example, if the SRC is 4 bytes, so the source address must be 4 bytes aligned, or it shall result in source address error(SAE).

16.7.34 status_t EDMA_SubmitTransfer (edma_handle_t * *handle*, const edma_transfer_config_t * *config*)

This function submits the eDMA transfer request according to the transfer configuration structure. If the user submits 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.

16.7.35 void EDMA_StartTransfer (edma_handle_t * *handle*)

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

Parameters

handle	eDMA handle pointer.

16.7.36 void EDMA_StopTransfer (edma_handle_t * *handle*)

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

Parameters

handle	eDMA handle pointer.

16.7.37 void EDMA_AbortTransfer (edma_handle_t * *handle*)

This function disables the channel request and clear transfer status bits. User can submit another transfer after calling this API.

Parameters

handle	DMA handle pointer.

16.7.38 void EDMA_HandleIRQ (edma_handle_t * *handle*)

This function clears the channel major interrupt flag and call the callback function if it is not NULL.

Parameters

handle	eDMA handle pointer.

Chapter 17
ENET: Ethernet MAC Driver

17.1 Overview

The KSDK provides a peripheral driver for the 10/100 Mbps Ethernet MAC (ENET) module of Kinetis devices.

The MII interface is the interface connected with MAC and PHY. the Serial management interface - MII management interface should be set firstly before any access to external PHY chip register. So call ENET_SetSMI() to initialize MII management interface. Use ENET_StartSMIRead(), ENET_StartSMIWrite() and ENET_ReadSMIData() to read/write phy registers. This function group sets up the MII and serial management SMI interface, gets data from the SMI interface, and starts the SMI read and write command. Use ENET_SetMII() to configure the MII before successfully get the data from the external PHY.

This group sets/gets the ENET mac address, setting the multicast group address filter. ENET_AddMulticastGroup() should be called to add the ENET MAC to multicast group. It is important for 1588 feature to receive the PTP message.

For ENET receive side, ENET_GetRxFrameSize() must be called firstly used to get the received data size, then call ENET_ReadFrame() to get the received data. If the received error happen, call ENET_GetRxErrBeforeReadFrame() after ENET_GetRxFrameSize() and before ENET_ReadFrame() to get the detail error informations.

For ENET transmit side, simply call ENET_SendFrame() to send the data out. The transmit data error information only accessible for 1588 enhanced buffer descriptor mode. So when ENET_ENHANCEDBUFFERDESCRIPTOR_MODE is defined the ENET_GetTxErrAfterSendFrame() can be used to get the detail transmit error information. The transmit error information only be updated by uDMA after the data is transmit. So ENET_GetTxErrAfterSendFrame() is recommended to be called on transmit interrupt handler.

This function group configures the PTP 1588 feature, starts/stops/gets/sets/adjusts the PTP IEEE 1588 timer, gets the receive/transmit frame timestamp, and PTP IEEE 1588 timer channel feature setting.

ENET_Ptp1588Configure() must be called when ENET_ENHANCEDBUFFERDESCRIPTOR_MODE is defined and the 1588 feature is required. The ENET_GetRxFrameTime() and ENET_GetTxFrameTime() are called by PTP stack to get the timestamp captured by ENET driver.

17.2 Typical use case

17.2.1 ENET Initialization, receive, and transmit operation

For ENET_ENHANCEDBUFFERDESCRIPTOR_MODE not defined use case, use the legacy type buffer descriptor transmit/receive the frame:

```
enet_config_t config;
```

```
  uint32_t length = 0;
  uint32_t sysClock;
  uint32_t phyAddr = 0;
  bool link = false;
  phy_speed_t speed;
  phy_duplex_t duplex;
  enet_status_t result;
  enet_data_error_stats_t eErrorStatic;
  // Prepares the buffer configuration.
  enet_buffer_config_t buffCfg =
  {
      ENET_RXBD_NUM,
      ENET_TXBD_NUM,
      ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
      ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
      &RxBuffDescrip[0],  // Prepare buffers
      &TxBuffDescrip[0],  // Prepare buffers
      &RxDataBuff[0][0],  // Prepare buffers
      &TxDataBuff[0][0],  // Prepare buffers
  };

  sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

  // Gets the default configuration.
  ENET_GetDefaultConfig(&config);
  PHY_Init(EXAMPLE_ENET, 0, sysClock);
  // Changes the link status to PHY auto-negotiated link status.
  PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
  if (link)
  {
      PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
      config.miiSpeed = (enet_mii_speed_t)speed;
      config.miiDuplex = (enet_mii_duplex_t)duplex;
  }
  ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);
  ENET_ActiveRead(EXAMPLE_ENET);

  while (1)
  {
      // Gets the frame size.
      result = ENET_GetRxFrameSize(&handle, &length);
      // Calls the ENET_ReadFrame when there is a received frame.
      if (length != 0)
      {
          // Receives a valid frame and delivers the receive buffer with the size equal to length.
          uint8_t *data = (uint8_t *)malloc(length);
          ENET_ReadFrame(EXAMPLE_ENET, &handle, data, length);
          // Delivers the data to the upper layer.
          .........
          free(data);
      }
      else if (result == kStatus_ENET_RxFrameErr)
      {
         // Updates the received buffer when an error occurs.
         ENET_GetRxErrBeforeReadFrame(&handle, &eErrStatic);
         // Updates the receive buffer.
         ENET_ReadFrame(EXAMPLE_ENET, &handle, NULL, 0);
      }

     // Sends a multicast frame when the PHY is linked up.
     if(kStatus_Success == PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link))
     {
        if(link)
        {
           ENET_SendFrame(EXAMPLE_ENET, &handle, &frame[0], ENET_DATA_LENGTH);
        }
     }
}
```

For ENET_ENHANCEDBUFFERDESCRIPTOR_MODE defined case, add the PTP IEEE 1588 configuration to enable the PTP IEEE 1588 feature. The initialization occurs as follows:

```
enet_config_t config;
uint32_t length = 0;
uint32_t sysClock;
uint32_t phyAddr = 0;
bool link = false;
phy_speed_t speed;
phy_duplex_t duplex;
enet_status_t result;
enet_data_err_stats_t eErrStatic;
enet_buffer_config_t buffCfg =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
    &RxBuffDescrip[0],
    &TxBuffDescrip[0],
    &RxDataBuff[0][0],
    &TxDataBuff[0][0],
};

sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

// Sets the PTP 1588 source.
CLOCK_SetEnetTime0Clock(2);
ptpClock = CLOCK_GetFreq(kCLOCK_Osc0ErClk);
// Prepares the PTP configuration.
enet_ptp_config_t ptpConfig =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    &g_rxPtpTsBuff[0],
    &g_txPtpTsBuff[0],
    kENET_PtpTimerChannel1,
    ptpClock,
};

// Gets the default configuration.
ENET_GetDefaultConfig(&config);

PHY_Init(EXAMPLE_ENET, 0, sysClock);
// Changes the link status to PHY auto-negotiated link status.
PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
if (link)
{
    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
    config.miiSpeed = (enet_mii_speed_t)speed;
    config.miiDuplex = (enet_mii_duplex_t)duplex;
}

ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);

// Configures the PTP 1588 feature.
ENET_Ptp1588Configure(EXAMPLE_ENET, &handle, &ptpConfig);
// Adds the device to the PTP multicast group.
ENET_AddMulticastGroup(EXAMPLE_ENET, &mGAddr[0]);

ENET_ActiveRead(EXAMPLE_ENET);
```

Files

- file fsl_enet.h

Typical use case

Data Structures

- struct enet_rx_bd_struct_t
  Defines the receive buffer descriptor structure for the little endian system. More...
- struct enet_tx_bd_struct_t
  Defines the enhanced transmit buffer descriptor structure for the little endian system. More...
- struct enet_data_error_stats_t
  Defines the ENET data error statistic structure. More...
- struct enet_buffer_config_t
  Defines the receive buffer descriptor configure structure. More...
- struct enet_config_t
  Defines the basic configuration structure for the ENET device. More...
- struct enet_handle_t
  Defines the ENET handler structure. More...

Macros

- #define ENET_BUFFDESCRIPTOR_RX_ERR_MASK
  Defines the receive error status flag mask.
- #define ENET_FIFO_MIN_RX_FULL 5U
  ENET minimum receive FIFO full.
- #define ENET_RX_MIN_BUFFERSIZE 256U
  ENET minimum buffer size.
- #define ENET_BUFF_ALIGNMENT 16U
  Ethernet buffer alignment.
- #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)
  Defines the PHY address scope for the ENET.

Typedefs

- typedef void(* enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, void *userData)
  ENET callback function.

Enumerations

- enum _enet_status {
  kStatus_ENET_RxFrameError = MAKE_STATUS(kStatusGroup_ENET, 0U),
  kStatus_ENET_RxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 1U),
  kStatus_ENET_RxFrameEmpty = MAKE_STATUS(kStatusGroup_ENET, 2U),
  kStatus_ENET_TxFrameBusy,
  kStatus_ENET_TxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 4U) }
  Defines the status return codes for transaction.
- enum enet_mii_mode_t {
  kENET_MiiMode = 0U,
  kENET_RmiiMode }
  Defines the RMII or MII mode for data interface between the MAC and the PHY.

- enum enet_mii_speed_t {
  kENET_MiiSpeed10M = 0U,
  kENET_MiiSpeed100M }
  Defines the 10 Mbps or 100 Mbps speed for the MII data interface.
- enum enet_mii_duplex_t {
  kENET_MiiHalfDuplex = 0U,
  kENET_MiiFullDuplex }
  Defines the half or full duplex for the MII data interface.
- enum enet_mii_write_t {
  kENET_MiiWriteNoCompliant = 0U,
  kENET_MiiWriteValidFrame }
  Defines the write operation for the MII management frame.
- enum enet_mii_read_t {
  kENET_MiiReadValidFrame = 2U,
  kENET_MiiReadNoCompliant = 3U }
  Defines the read operation for the MII management frame.
- enum enet_special_control_flag_t {
  kENET_ControlFlowControlEnable = 0x0001U,
  kENET_ControlRxPayloadCheckEnable = 0x0002U,
  kENET_ControlRxPadRemoveEnable = 0x0004U,
  kENET_ControlRxBroadCastRejectEnable = 0x0008U,
  kENET_ControlMacAddrInsert = 0x0010U,
  kENET_ControlStoreAndFwdDisable = 0x0020U,
  kENET_ControlSMIPreambleDisable = 0x0040U,
  kENET_ControlPromiscuousEnable = 0x0080U,
  kENET_ControlMIILoopEnable = 0x0100U,
  kENET_ControlVLANTagEnable = 0x0200U }
  Defines a special configuration for ENET MAC controller.
- enum enet_interrupt_enable_t {
  kENET_BabrInterrupt = ENET_EIR_BABR_MASK,
  kENET_BabtInterrupt = ENET_EIR_BABT_MASK,
  kENET_GraceStopInterrupt = ENET_EIR_GRA_MASK,
  kENET_TxFrameInterrupt = ENET_EIR_TXF_MASK,
  kENET_TxByteInterrupt = ENET_EIR_TXB_MASK,
  kENET_RxFrameInterrupt = ENET_EIR_RXF_MASK,
  kENET_RxByteInterrupt = ENET_EIR_RXB_MASK,
  kENET_MiiInterrupt = ENET_EIR_MII_MASK,
  kENET_EBusERInterrupt = ENET_EIR_EBERR_MASK,
  kENET_LateCollisionInterrupt = ENET_EIR_LC_MASK,
  kENET_RetryLimitInterrupt = ENET_EIR_RL_MASK,
  kENET_UnderrunInterrupt = ENET_EIR_UN_MASK,
  kENET_PayloadRxInterrupt = ENET_EIR_PLR_MASK,
  kENET_WakeupInterrupt = ENET_EIR_WAKEUP_MASK }
  List of interrupts supported by the peripheral.
- enum enet_event_t {

kENET_RxEvent,
kENET_TxEvent,
kENET_ErrEvent,
kENET_WakeUpEvent }
Defines the common interrupt event for callback use.

- enum enet_tx_accelerator_t {
  kENET_TxAccelIsShift16Enabled = ENET_TACC_SHIFT16_MASK,
  kENET_TxAccelIpCheckEnabled = ENET_TACC_IPCHK_MASK,
  kENET_TxAccelProtoCheckEnabled = ENET_TACC_PROCHK_MASK }
  Defines the transmit accelerator configuration.
- enum enet_rx_accelerator_t {
  kENET_RxAccelPadRemoveEnabled = ENET_RACC_PADREM_MASK,
  kENET_RxAccelIpCheckEnabled = ENET_RACC_IPDIS_MASK,
  kENET_RxAccelProtoCheckEnabled = ENET_RACC_PRODIS_MASK,
  kENET_RxAccelMacCheckEnabled = ENET_RACC_LINEDIS_MASK,
  kENET_RxAccelisShift16Enabled = ENET_RACC_SHIFT16_MASK }
  Defines the receive accelerator configuration.

Driver version

- #define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Defines the driver version.

Control and status region bit masks of the receive buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U
  Empty bit mask.
- #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U
  Software owner one mask.
- #define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U
  Next buffer descriptor is the start address.
- #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U
  Software owner two mask.
- #define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U
  Last BD of the frame mask.
- #define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U
  Received because of the promiscuous mode.
- #define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U
  Broadcast packet mask.
- #define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U
  Multicast packet mask.
- #define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK 0x0020U
  Length violation mask.
- #define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U
  Non-octet aligned frame mask.
- #define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U
  CRC error mask.
- #define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U
  FIFO overrun mask.

- #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U
  Frame is truncated mask.

Control and status bit masks of the transmit buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U
  Ready bit mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U
  Software owner one mask.
- #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U
  Wrap buffer descriptor mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U
  Software owner two mask.
- #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U
  Last BD of the frame mask.
- #define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U
  Transmit CRC mask.

Defines the maximum Ethernet frame size.

- #define ENET_FRAME_MAX_FRAMELEN 1518U
  Maximum Ethernet frame size.
- #define ENET_FRAME_MAX_VALNFRAMELEN 1522U
  Maximum VLAN frame size.

Initialization and De-initialization

- void ENET_GetDefaultConfig (enet_config_t *config)
  Gets the ENET default configuration structure.
- void ENET_Init (ENET_Type *base, enet_handle_t *handle, const enet_config_t *config, const enet_buffer_config_t *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)
  Initializes the ENET module.
- void ENET_Deinit (ENET_Type *base)
  Deinitializes the ENET module.
- static void ENET_Reset (ENET_Type *base)
  Resets the ENET module.

MII interface operation

- void ENET_SetMII (ENET_Type *base, enet_mii_speed_t speed, enet_mii_duplex_t duplex)
  Sets the ENET MII speed and duplex.
- void ENET_SetSMI (ENET_Type *base, uint32_t srcClock_Hz, bool isPreambleDisabled)
  Sets the ENET SMI(serial management interface)- MII management interface.
- static bool ENET_GetSMI (ENET_Type *base)
  Gets the ENET SMI- MII management interface configuration.
- static uint32_t ENET_ReadSMIData (ENET_Type *base)
  Reads data from the PHY register through SMI interface.
- void ENET_StartSMIRead (ENET_Type *base, uint32_t phyAddr, uint32_t phyReg, enet_mii_-read_t operation)
  Starts an SMI (Serial Management Interface) read command.

Typical use case

- void ENET_StartSMIWrite (ENET_Type *base, uint32_t phyAddr, uint32_t phyReg, enet_mii_write_t operation, uint32_t data)
  Starts a SMI write command.

MAC Address Filter

- void ENET_SetMacAddr (ENET_Type *base, uint8_t *macAddr)
  Sets the ENET module Mac address.
- void ENET_GetMacAddr (ENET_Type *base, uint8_t *macAddr)
  Gets the ENET module Mac address.
- void ENET_AddMulticastGroup (ENET_Type *base, uint8_t *address)
  Adds the ENET device to a multicast group.
- void ENET_LeaveMulticastGroup (ENET_Type *base, uint8_t *address)
  Moves the ENET device from a multicast group.

Other basic operation

- static void ENET_ActiveRead (ENET_Type *base)
  Activates ENET read or receive.
- static void ENET_EnableSleepMode (ENET_Type *base, bool enable)
  Enables/disables the MAC to enter sleep mode.
- static void ENET_GetAccelFunction (ENET_Type *base, uint32_t *txAccelOption, uint32_t *rxAccelOption)
  Gets ENET transmit and receive accelerator functions from MAC controller.

Interrupts.

- static void ENET_EnableInterrupts (ENET_Type *base, uint32_t mask)
  Enables the ENET interrupt.
- static void ENET_DisableInterrupts (ENET_Type *base, uint32_t mask)
  Disables the ENET interrupt.
- static uint32_t ENET_GetInterruptStatus (ENET_Type *base)
  Gets the ENET interrupt status flag.
- static void ENET_ClearInterruptStatus (ENET_Type *base, uint32_t mask)
  Clears the ENET interrupt events status flag.

Transactional operation

- void ENET_SetCallback (enet_handle_t *handle, enet_callback_t callback, void *userData)
  Set the callback function.
- void ENET_GetRxErrBeforeReadFrame (enet_handle_t *handle, enet_data_error_stats_t *eErrorStatic)
  Gets the ENET the error statistics of a received frame.
- status_t ENET_GetRxFrameSize (enet_handle_t *handle, uint32_t *length)
  Gets the size of the read frame.
- status_t ENET_ReadFrame (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length)
  Reads a frame from the ENET device.
- status_t ENET_SendFrame (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length)

Transmits an ENET frame.
- void ENET_TransmitIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The transmit IRQ handler.
- void ENET_ReceiveIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The receive IRQ handler.
- void ENET_ErrorIRQHandler (ENET_Type ∗base, enet_handle_t ∗handle)
  The error IRQ handler.

17.3 Data Structure Documentation

17.3.1 struct enet_rx_bd_struct_t

Data Fields

- uint16_t length
  Buffer descriptor data length.
- uint16_t control
  Buffer descriptor control and status.
- uint8_t ∗ buffer
  Data buffer pointer.

17.3.1.0.0.36 Field Documentation

17.3.1.0.0.36.1 uint16_t enet_rx_bd_struct_t::length

17.3.1.0.0.36.2 uint16_t enet_rx_bd_struct_t::control

17.3.1.0.0.36.3 uint8_t∗ enet_rx_bd_struct_t::buffer

17.3.2 struct enet_tx_bd_struct_t

Data Fields

- uint16_t length
  Buffer descriptor data length.
- uint16_t control
  Buffer descriptor control and status.
- uint8_t ∗ buffer
  Data buffer pointer.

17.3.2.0.0.37 Field Documentation

17.3.2.0.0.37.1 uint16_t enet_tx_bd_struct_t::length

17.3.2.0.0.37.2 uint16_t enet_tx_bd_struct_t::control

17.3.2.0.0.37.3 uint8_t* enet_tx_bd_struct_t::buffer

17.3.3 struct enet_data_error_stats_t

Data Fields

- uint32_t statsRxLenGreaterErr
  Receive length greater than RCR[MAX_FL].
- uint32_t statsRxAlignErr
  Receive non-octet alignment/.
- uint32_t statsRxFcsErr
  Receive CRC error.
- uint32_t statsRxOverRunErr
  Receive over run.
- uint32_t statsRxTruncateErr
  Receive truncate.

17.3.3.0.0.38 Field Documentation

17.3.3.0.0.38.1 uint32_t enet_data_error_stats_t::statsRxLenGreaterErr

17.3.3.0.0.38.2 uint32_t enet_data_error_stats_t::statsRxFcsErr

17.3.3.0.0.38.3 uint32_t enet_data_error_stats_t::statsRxOverRunErr

17.3.3.0.0.38.4 uint32_t enet_data_error_stats_t::statsRxTruncateErr

17.3.4 struct enet_buffer_config_t

Note: For the internal DMA requirements, the buffers have a corresponding alignment requirement:

1. The aligned receive and transmit buffer size must be evenly divisible by 16.
2. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by 16.
3. The aligned transmit and receive buffer start address must be evenly divisible by 16. Receive buffers should be continuous with the total size equal to "rxBdNumber * rxBuffSizeAlign". Transmit buffers should be continuous with the total size equal to "txBdNumber * txBuffSizeAlign".

Data Fields

- uint16_t rxBdNumber
  Receive buffer descriptor number.

- uint16_t txBdNumber
  Transmit buffer descriptor number.
- uint32_t rxBuffSizeAlign
  Aligned receive data buffer size.
- uint32_t txBuffSizeAlign
  Aligned transmit data buffer size.
- volatile enet_rx_bd_struct_t * rxBdStartAddrAlign
  Aligned receive buffer descriptor start address.
- volatile enet_tx_bd_struct_t * txBdStartAddrAlign
  Aligned transmit buffer descriptor start address.
- uint8_t * rxBufferAlign
  Receive data buffer start address.
- uint8_t * txBufferAlign
  Transmit data buffer start address.

17.3.4.0.0.39 Field Documentation

17.3.4.0.0.39.1 uint16_t enet_buffer_config_t::rxBdNumber

17.3.4.0.0.39.2 uint16_t enet_buffer_config_t::txBdNumber

17.3.4.0.0.39.3 uint32_t enet_buffer_config_t::rxBuffSizeAlign

17.3.4.0.0.39.4 uint32_t enet_buffer_config_t::txBuffSizeAlign

17.3.4.0.0.39.5 volatile enet_rx_bd_struct_t* enet_buffer_config_t::rxBdStartAddrAlign

17.3.4.0.0.39.6 volatile enet_tx_bd_struct_t* enet_buffer_config_t::txBdStartAddrAlign

17.3.4.0.0.39.7 uint8_t* enet_buffer_config_t::rxBufferAlign

17.3.4.0.0.39.8 uint8_t* enet_buffer_config_t::txBufferAlign

17.3.5 struct enet_config_t

Note:

1. macSpecialConfig is used for a special control configuration, A logical OR of "enet_special_control-_flag_t". For a special configuration for MAC, set this parameter to 0.
2. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO .... 3 - 192 bytes written to TX FIFO .... The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO .... txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
3. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULL and the maximum is 0xFF. If the end of the frame is stored in FIFO and the frame size if smaller than the txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.
4. When "kENET_ControlFlowControlEnable" is set in the macSpecialConfig, ensure that the pause-

Duration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.

5. When "kENET_ControlStoreAndFwdDisabled" is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
6. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The "enet_tx_accelerator_t" and "enet_rx_accelerator_t" are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.

Data Fields

- uint32_t macSpecialConfig
  Mac special configuration.
- uint32_t interrupt
  Mac interrupt source.
- uint16_t rxMaxFrameLen
  Receive maximum frame length.
- enet_mii_mode_t miiMode
  MII mode.
- enet_mii_speed_t miiSpeed
  MII Speed.
- enet_mii_duplex_t miiDuplex
  MII duplex.
- uint8_t rxAccelerConfig
  Receive accelerator, A logical OR of "enet_rx_accelerator_t".
- uint8_t txAccelerConfig
  Transmit accelerator, A logical OR of "enet_rx_accelerator_t".
- uint16_t pauseDuration
  For flow control enabled case: Pause duration.
- uint8_t rxFifoEmptyThreshold
  For flow control enabled case: when RX FIFO level reaches this value, it makes MAC generate XOFF pause frame.
- uint8_t rxFifoFullThreshold
  For store and forward disable case, the data required in RX FIFO to notify the MAC receive ready status.
- uint8_t txFifoWatermark
  For store and forward disable case, the data required in TX FIFO before a frame transmit start.

17.3.5.0.0.40 Field Documentation

17.3.5.0.0.40.1 uint32_t enet_config_t::macSpecialConfig

A logical OR of "enet_special_control_flag_t".

17.3.5.0.0.40.2 uint32_t enet_config_t::interrupt

A logical OR of "enet_interrupt_enable_t".

17.3.5.0.0.40.3 uint16_t enet_config_t::rxMaxFrameLen

17.3.5.0.0.40.4 enet_mii_mode_t enet_config_t::miiMode

17.3.5.0.0.40.5 enet_mii_speed_t enet_config_t::miiSpeed

17.3.5.0.0.40.6 enet_mii_duplex_t enet_config_t::miiDuplex

17.3.5.0.0.40.7 uint8_t enet_config_t::rxAccelerConfig

17.3.5.0.0.40.8 uint8_t enet_config_t::txAccelerConfig

17.3.5.0.0.40.9 uint16_t enet_config_t::pauseDuration

17.3.5.0.0.40.10 uint8_t enet_config_t::rxFifoEmptyThreshold

17.3.5.0.0.40.11 uint8_t enet_config_t::rxFifoFullThreshold

17.3.5.0.0.40.12 uint8_t enet_config_t::txFifoWatermark

17.3.6 struct _enet_handle

Data Fields

- volatile enet_rx_bd_struct_t * rxBdBase
  Receive buffer descriptor base address pointer.
- volatile enet_rx_bd_struct_t * rxBdCurrent
  The current available receive buffer descriptor pointer.
- volatile enet_rx_bd_struct_t * rxBdDirty
  The dirty receive buffer descriptor needed to be updated from.
- volatile enet_tx_bd_struct_t * txBdBase
  Transmit buffer descriptor base address pointer.
- volatile enet_tx_bd_struct_t * txBdCurrent
  The current available transmit buffer descriptor pointer.
- uint32_t rxBuffSizeAlign
  Receive buffer size alignment.
- uint32_t txBuffSizeAlign
  Transmit buffer size alignment.
- enet_callback_t callback
  Callback function.
- void * userData
  Callback function parameter.

17.3.6.0.0.41 Field Documentation

17.3.6.0.0.41.1 volatile enet_rx_bd_struct_t* enet_handle_t::rxBdBase

17.3.6.0.0.41.2 volatile enet_rx_bd_struct_t* enet_handle_t::rxBdCurrent

17.3.6.0.0.41.3 volatile enet_rx_bd_struct_t* enet_handle_t::rxBdDirty

17.3.6.0.0.41.4 volatile enet_tx_bd_struct_t* enet_handle_t::txBdBase

17.3.6.0.0.41.5 volatile enet_tx_bd_struct_t* enet_handle_t::txBdCurrent

17.3.6.0.0.41.6 uint32_t enet_handle_t::rxBuffSizeAlign

17.3.6.0.0.41.7 uint32_t enet_handle_t::txBuffSizeAlign

17.3.6.0.0.41.8 enet_callback_t enet_handle_t::callback

17.3.6.0.0.41.9 void* enet_handle_t::userData

17.4 Macro Definition Documentation

17.4.1 #define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

17.4.2 #define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U

17.4.3 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U

17.4.4 #define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U

17.4.5 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U

17.4.6 #define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U

17.4.7 #define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U

17.4.8 #define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U

17.4.9 #define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U

17.4.10 #define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK 0x0020U

17.4.11 #define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U

17.4.12 #define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U

17.4.13 #define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U

17.4.14 #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U

17.4.15 #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U

17.4.16 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U

17.4.17 #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U

17.4.18 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U

17.4.19 #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U

17.4.20 #define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U

17.4.21 #define ENET_BUFFDESCRIPTOR_RX_ERR_MASK

Value:

```
(ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK |
      ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK | \
     ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK |
      ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK |
      ENET_BUFFDESCRIPTOR_RX_CRC_MASK)
```

17.4.22 #define ENET_FRAME_MAX_FRAMELEN 1518U

17.4.23 #define ENET_FRAME_MAX_VALNFRAMELEN 1522U

17.4.24 #define ENET_FIFO_MIN_RX_FULL 5U

17.4.25 #define ENET_RX_MIN_BUFFERSIZE 256U

17.4.26 #define ENET_BUFF_ALIGNMENT 16U

17.4.27 #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)

17.5 Typedef Documentation

17.5.1 typedef void(* enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, void *userData)

17.6 Enumeration Type Documentation

17.6.1 enum _enet_status

Enumerator

kStatus_ENET_RxFrameError A frame received but data error happen.
kStatus_ENET_RxFrameFail Failed to receive a frame.
kStatus_ENET_RxFrameEmpty No frame arrive.
kStatus_ENET_TxFrameBusy Transmit buffer descriptors are under process.
kStatus_ENET_TxFrameFail Transmit frame fail.

17.6.2 enum enet_mii_mode_t

Enumerator

kENET_MiiMode MII mode for data interface.
kENET_RmiiMode RMII mode for data interface.

17.6.3 enum enet_mii_speed_t

Enumerator

kENET_MiiSpeed10M Speed 10 Mbps.
kENET_MiiSpeed100M Speed 100 Mbps.

17.6.4 enum enet_mii_duplex_t

Enumerator

kENET_MiiHalfDuplex Half duplex mode.
kENET_MiiFullDuplex Full duplex mode.

17.6.5 enum enet_mii_write_t

Enumerator

kENET_MiiWriteNoCompliant Write frame operation, but not MII-compliant.
kENET_MiiWriteValidFrame Write frame operation for a valid MII management frame.

17.6.6 enum enet_mii_read_t

Enumerator

kENET_MiiReadValidFrame Read frame operation for a valid MII management frame.
kENET_MiiReadNoCompliant Read frame operation, but not MII-compliant.

17.6.7 enum enet_special_control_flag_t

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to macSpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifoFullThreshold and txFifoWatermark in the enet_config_t.

Enumerator

kENET_ControlFlowControlEnable Enable ENET flow control: pause frame.
kENET_ControlRxPayloadCheckEnable Enable ENET receive payload length check.
kENET_ControlRxPadRemoveEnable Padding is removed from received frames.

kENET_ControlRxBroadCastRejectEnable Enable broadcast frame reject.
kENET_ControlMacAddrInsert Enable MAC address insert.
kENET_ControlStoreAndFwdDisable Enable FIFO store and forward.
kENET_ControlSMIPreambleDisable Enable SMI preamble.
kENET_ControlPromiscuousEnable Enable promiscuous mode.
kENET_ControlMIILoopEnable Enable ENET MII loop back.
kENET_ControlVLANTagEnable Enable VLAN tag frame.

17.6.8 enum enet_interrupt_enable_t

This enumeration uses one-bot encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Enumerator

kENET_BabrInterrupt Babbling receive error interrupt source.
kENET_BabtInterrupt Babbling transmit error interrupt source.
kENET_GraceStopInterrupt Graceful stop complete interrupt source.
kENET_TxFrameInterrupt TX FRAME interrupt source.
kENET_TxByteInterrupt TX BYTE interrupt source.
kENET_RxFrameInterrupt RX FRAME interrupt source.
kENET_RxByteInterrupt RX BYTE interrupt source.
kENET_MiiInterrupt MII interrupt source.
kENET_EBusERInterrupt Ethernet bus error interrupt source.
kENET_LateCollisionInterrupt Late collision interrupt source.
kENET_RetryLimitInterrupt Collision Retry Limit interrupt source.
kENET_UnderrunInterrupt Transmit FIFO underrun interrupt source.
kENET_PayloadRxInterrupt Payload Receive interrupt source.
kENET_WakeupInterrupt WAKEUP interrupt source.

17.6.9 enum enet_event_t

Enumerator

kENET_RxEvent Receive event.
kENET_TxEvent Transmit event.
kENET_ErrEvent Error event: BABR/BABT/EBERR/LC/RL/UN/PLR .
kENET_WakeUpEvent Wake up from sleep mode event.

17.6.10 enum enet_tx_accelerator_t

Enumerator

kENET_TxAccelIsShift16Enabled Transmit FIFO shift-16.
kENET_TxAccelIpCheckEnabled Insert IP header checksum.
kENET_TxAccelProtoCheckEnabled Insert protocol checksum.

17.6.11 enum enet_rx_accelerator_t

Enumerator

kENET_RxAccelPadRemoveEnabled Padding removal for short IP frames.
kENET_RxAccelIpCheckEnabled Discard with wrong IP header checksum.
kENET_RxAccelProtoCheckEnabled Discard with wrong protocol checksum.
kENET_RxAccelMacCheckEnabled Discard with Mac layer errors.
kENET_RxAccelisShift16Enabled Receive FIFO shift-16.

17.7 Function Documentation

17.7.1 void ENET_GetDefaultConfig (enet_config_t * *config*)

The purpose of this API is to get the default ENET MAC controller configure structure for ENET_Init(). User may use the initialized structure unchanged in ENET_Init(), or modify some fields of the structure before calling ENET_Init(). Example:

```
enet_config_t config;
ENET_GetDefaultConfig(&config);
```

Parameters

config	The ENET mac controller configuration structure pointer.

17.7.2 void ENET_Init (ENET_Type * *base*, enet_handle_t * *handle*, const enet_config_t * *config*, const enet_buffer_config_t * *bufferConfig*, uint8_t * *macAddr*, uint32_t *srcClock_Hz*)

This function ungates the module clock and initializes it with the ENET configuration.

Parameters

base	ENET peripheral base address.
handle	ENET handler pointer.
config	ENET mac configuration structure pointer. The "enet_config_t" type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
bufferConfig	ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization.
macAddr	ENET mac address of Ethernet device. This MAC address should be provided.
srcClock_Hz	The internal module clock source for MII clock.

Note

ENET has two buffer descriptors: legacy buffer descriptors and enhanced 1588 buffer descriptors. The legacy descriptor is used by default. To use 1588 feature, use the enhanced 1588 buffer descriptor by defining "ENET_ENHANCEDBUFFERDESCRIPTOR_MODE" and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Init().

17.7.3 void ENET_Deinit (ENET_Type * *base*)

This function gates the module clock, clears ENET interrupts, and disables the ENET module.

Parameters

base	ENET peripheral base address.

17.7.4 static void ENET_Reset (ENET_Type * *base*) [inline], [static]

This function restores the ENET module to reset state. Note that this function sets all registers to reset state. As a result, the ENET module can't work after calling this function.

Parameters

base	ENET peripheral base address.

17.7.5 void ENET_SetMII (ENET_Type * *base*, enet_mii_speed_t *speed*, enet_mii_duplex_t *duplex*)

Parameters

base	ENET peripheral base address.
speed	The speed of the RMII mode.
duplex	The duplex of the RMII mode.

17.7.6 void ENET_SetSMI (ENET_Type * *base,* uint32_t *srcClock_Hz,* bool *isPreambleDisabled*)

Parameters

base	ENET peripheral base address.
srcClock_Hz	This is the ENET module clock frequency. Normally it's the system clock. See clock distribution.
isPreamble-Disabled	The preamble disable flag. • true Enables the preamble. • false Disables the preamble.

17.7.7 static bool ENET_GetSMI (ENET_Type * *base*) [inline], [static]

This API is used to get the SMI configuration to check if the MII management interface has been set.

Parameters

base	ENET peripheral base address.

Returns

The SMI setup status true or false.

17.7.8 static uint32_t ENET_ReadSMIData (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The data read from PHY

17.7.9 void ENET_StartSMIRead (ENET_Type * *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_read_t *operation*)

Parameters

base	ENET peripheral base address.
phyAddr	The PHY address.
phyReg	The PHY register.
operation	The read operation.

17.7.10 void ENET_StartSMIWrite (ENET_Type * *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_write_t *operation*, uint32_t *data*)

Parameters

base	ENET peripheral base address.
phyAddr	The PHY address.
phyReg	The PHY register.
operation	The write operation.
data	The data written to PHY.

17.7.11 void ENET_SetMacAddr (ENET_Type * *base*, uint8_t * *macAddr*)

Parameters

base	ENET peripheral base address.
macAddr	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

17.7.12 void ENET_GetMacAddr (ENET_Type * *base,* uint8_t * *macAddr*)

Parameters

base	ENET peripheral base address.
macAddr	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

17.7.13 void ENET_AddMulticastGroup (ENET_Type * *base,* uint8_t * *address*)

Parameters

base	ENET peripheral base address.
address	The six-byte multicast group address which is provided by application.

17.7.14 void ENET_LeaveMulticastGroup (ENET_Type * *base,* uint8_t * *address*)

Parameters

base	ENET peripheral base address.
address	The six-byte multicast group address which is provided by application.

17.7.15 static void ENET_ActiveRead (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Note

This must be called after the MAC configuration and state are ready. It must be called after the ENET_Init() and ENET_Ptp1588Configure(). This should be called when the ENET receive required.

17.7.16 static void ENET_EnableSleepMode (ENET_Type * *base*, bool *enable*) [inline], [static]

This function is used to set the MAC enter sleep mode. When entering sleep mode, the magic frame wakeup interrupt should be enabled to wake up MAC from the sleep mode and reset it to normal mode.

Parameters

base	ENET peripheral base address.
enable	True enable sleep mode, false disable sleep mode.

17.7.17 static void ENET_GetAccelFunction (ENET_Type * *base*, uint32_t * *txAccelOption*, uint32_t * *rxAccelOption*) [inline], [static]

Parameters

base	ENET peripheral base address.
txAccelOption	The transmit accelerator option. The "enet_tx_accelerator_t" is recommended to be used to as the mask to get the exact the accelerator option.
rxAccelOption	The receive accelerator option. The "enet_rx_accelerator_t" is recommended to be used to as the mask to get the exact the accelerator option.

17.7.18 static void ENET_EnableInterrupts (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to enable the TX frame interrupt and RX frame interrupt, do this:

```
ENET_EnableInterrupts(ENET, kENET_TxFrameInterrupt |
      kENET_RxFrameInterrupt);
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to enable. This is a logical OR of the enumeration :: enet_interrupt-_enable_t.

17.7.19 static void ENET_DisableInterrupts (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to disable the TX frame interrupt and RX frame interrupt, do this:

```
ENET_DisableInterrupts(ENET, kENET_TxFrameInterrupt |
      kENET_RxFrameInterrupt);
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to disable. This is a logical OR of the enumeration :: enet_interrupt-_enable_t.

17.7.20 static uint32_t ENET_GetInterruptStatus (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: enet_interrupt_enable_t.

17.7.21 static void ENET_ClearInterruptStatus (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the enet_interrupt_enable_t. For example, to clear the TX frame interrupt and RX frame interrupt, do this:

```
ENET_ClearInterruptStatus(ENET, kENET_TxFrameInterrupt |
      kENET_RxFrameInterrupt);
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupt source to be cleared. This is the logical OR of members of the enumeration :: enet_interrupt_enable_t.

17.7.22 void ENET_SetCallback (enet_handle_t * *handle*, enet_callback_t *callback*, void * *userData*)

This API is provided for application callback required case when ENET interrupt is enabled. This API should be called after calling ENET_Init.

Parameters

handle	ENET handler pointer. Should be provided by application.
callback	The ENET callback function.
userData	The callback function parameter.

17.7.23 void ENET_GetRxErrBeforeReadFrame (enet_handle_t * *handle*, enet_data_error_stats_t * *eErrorStatic*)

This API must be called after the ENET_GetRxFrameSize and before the ENET_ReadFrame(). If the ENET_GetRxFrameSize returns kStatus_ENET_RxFrameError, the ENET_GetRxErrBeforeReadFrame can be used to get the exact error statistics. For example:

```
status = ENET_GetRxFrameSize(&g_handle, &length);
if (status == kStatus_ENET_RxFrameError)
{
    // Get the error information of the received frame.
    ENET_GetRxErrBeforeReadFrame(&g_handle, &eErrStatic);
    // update the receive buffer.
    ENET_ReadFrame(EXAMPLE_ENET, &g_handle, NULL, 0);
}
```

Parameters

handle	The ENET handler structure pointer. This is the same handler pointer used in the ENET_Init.
eErrorStatic	The error statistics structure pointer.

17.7.24 status_t ENET_GetRxFrameSize (enet_handle_t * *handle*, uint32_t * *length*)

This function reads a received frame size from the ENET buffer descriptors.

Note

The FCS of the frame is removed by MAC controller and the size is the length without the FCS. After calling ENET_GetRxFrameSize, ENET_ReadFrame() should be called to update the receive buffers If the result is not "kStatus_ENET_RxFrameEmpty".

Parameters

handle	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
length	The length of the valid frame received.

Return values

kStatus_ENET_RxFrame-Empty	No frame received. Should not call ENET_ReadFrame to read frame.
kStatus_ENET_RxFrame-Error	Data error happens. ENET_ReadFrame should be called with NULL data and NULL length to update the receive buffers.
kStatus_Success	Receive a frame Successfully then the ENET_ReadFrame should be called with the right data buffer and the captured data length input.

17.7.25 status_t ENET_ReadFrame (ENET_Type * *base*, enet_handle_t * *handle*, uint8_t * *data*, uint32_t *length*)

This function reads a frame (both the data and the length) from the ENET buffer descriptors. The ENET-_GetRxFrameSize should be used to get the size of the prepared data buffer.

Note

The FCS of the frame is removed by MAC controller and is not delivered to the application.

Parameters

base	ENET peripheral base address.
handle	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to store the frame which memory size should be at least "length".
length	The size of the data buffer which is still the length of the received frame.

Returns

The execute status, successful or failure.

17.7.26 status_t ENET_SendFrame (ENET_Type * *base,* enet_handle_t * *handle,* uint8_t * *data,* uint32_t *length*)

Note

The CRC is automatically appended to the data. Input the data to send without the CRC.

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to be send.
length	The length of the data to be send.

Return values

kStatus_Success	Send frame succeed.
kStatus_ENET_TxFrame-Busy	Transmit buffer descriptor is busy under transmit.
kStatus_ENET_TxFrame-Fail	Transmit frame fail.

17.7.27 void ENET_TransmitIRQHandler (ENET_Type * *base,* enet_handle_t * *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

17.7.28 void ENET_ReceiveIRQHandler (ENET_Type * *base,* enet_handle_t * *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

17.7.29 void ENET_ErrorIRQHandler (ENET_Type * *base,* enet_handle_t * *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

Chapter 18
EWM: External Watchdog Monitor Driver

18.1 Overview

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

Typical use case

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.enableInterrupt = true;
config.compareLowValue = 0U;
config.compareHighValue = 0xAAU;
NVIC_EnableIRQ(WDOG_EWM_IRQn);
EWM_Init(base, &config);
```

Files

- file fsl_ewm.h

Data Structures

- struct ewm_config_t
  Describes ewm clock source. More...

Enumerations

- enum _ewm_interrupt_enable_t { kEWM_InterruptEnable = EWM_CTRL_INTEN_MASK }
  EWM interrupt configuration structure, default settings all disabled.
- enum _ewm_status_flags_t { kEWM_RunningFlag = EWM_CTRL_EWMEN_MASK }
  EWM status flags.

Driver version

- #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  EWM driver version 2.0.1.

EWM Initialization and De-initialization

- void EWM_Init (EWM_Type *base, const ewm_config_t *config)
  Initializes the EWM peripheral.
- void EWM_Deinit (EWM_Type *base)
  Deinitializes the EWM peripheral.
- void EWM_GetDefaultConfig (ewm_config_t *config)
  Initializes the EWM configuration structure.

EWM functional Operation

- static void EWM_EnableInterrupts (EWM_Type *base, uint32_t mask)
  Enables the EWM interrupt.
- static void EWM_DisableInterrupts (EWM_Type *base, uint32_t mask)
  Disables the EWM interrupt.
- static uint32_t EWM_GetStatusFlags (EWM_Type *base)
  Gets EWM all status flags.
- void EWM_Refresh (EWM_Type *base)
  Service EWM.

18.2 Data Structure Documentation

18.2.1 struct ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

Data Fields

- bool enableEwm
  Enable EWM module.
- bool enableEwmInput
  Enable EWM_in input.
- bool setInputAssertLogic
  EWM_in signal assertion state.
- bool enableInterrupt
  Enable EWM interrupt.
- uint8_t compareLowValue
  Compare low register value.
- uint8_t compareHighValue
  Compare high register value.

18.3 Macro Definition Documentation

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

18.4 Enumeration Type Documentation

18.4.1 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM_InterruptEnable Enable EWM to generate an interrupt.

18.4.2 enum _ewm_status_flags_t

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

Enumerator

kEWM_RunningFlag Running flag, set when ewm is enabled.

18.5 Function Documentation

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

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

Example:

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

Parameters

base	EWM peripheral base address
config	The configuration of EWM

18.5.2 void EWM_Deinit (EWM_Type * *base*)

This function is used to shut down the EWM.

Parameters

base	EWM peripheral base address

18.5.3 void EWM_GetDefaultConfig (ewm_config_t * *config*)

This function initializes the EWM configure structure to default values. The default values are:

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

Parameters

config	Pointer to EWM configuration structure.

See Also

ewm_config_t

18.5.4 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

18.5.5 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

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

This function gets all status flags.

Example for getting Running Flag:

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

Parameters

base	EWM peripheral base address

Returns

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

See Also

_ewm_status_flags_t
- true: related status flag has been set.
- false: related status flag is not set.

18.5.7 void EWM_Refresh (EWM_Type * *base*)

This function reset EWM counter to zero.

Parameters

base	EWM peripheral base address

Chapter 19
C90TFS Flash Driver

19.1 Overview

The flash provides the C90TFS Flash driver of Kinetis devices with the C90TFS Flash module inside. The flash 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 fileds. More...
- struct flash_operation_config_t
  Active flash information for current operation. More...
- struct flash_config_t
  Flash driver state information. More...

Typedefs

- typedef void(* flash_callback_t)(void)
  callback type used for pflash block

Enumerations

- enum flash_margin_value_t {
  kFLASH_marginValueNormal,
  kFLASH_marginValueUser,
  kFLASH_marginValueFactory,
  kFLASH_marginValueInvalid }
  Enumeration for supported flash margin levels.
- enum flash_security_state_t {
  kFLASH_securityStateNotSecure,
  kFLASH_securityStateBackdoorEnabled,
  kFLASH_securityStateBackdoorDisabled }
  Enumeration for the three possible flash security states.
- enum flash_protection_state_t {
  kFLASH_protectionStateUnprotected,
  kFLASH_protectionStateProtected,

kFLASH_protectionStateMixed }
Enumeration for the three possible flash protection levels.

- enum flash_execute_only_access_state_t {
  kFLASH_accessStateUnLimited,
  kFLASH_accessStateExecuteOnly,
  kFLASH_accessStateMixed }
  Enumeration for the three possible flash execute access levels.
- enum flash_property_tag_t {
  kFLASH_propertyPflashSectorSize = 0x00U,
  kFLASH_propertyPflashTotalSize = 0x01U,
  kFLASH_propertyPflashBlockSize = 0x02U,
  kFLASH_propertyPflashBlockCount = 0x03U,
  kFLASH_propertyPflashBlockBaseAddr = 0x04U,
  kFLASH_propertyPflashFacSupport = 0x05U,
  kFLASH_propertyPflashAccessSegmentSize = 0x06U,
  kFLASH_propertyPflashAccessSegmentCount = 0x07U,
  kFLASH_propertyFlexRamBlockBaseAddr = 0x08U,
  kFLASH_propertyFlexRamTotalSize = 0x09U,
  kFLASH_propertyDflashSectorSize = 0x10U,
  kFLASH_propertyDflashTotalSize = 0x11U,
  kFLASH_propertyDflashBlockSize = 0x12U,
  kFLASH_propertyDflashBlockCount = 0x13U,
  kFLASH_propertyDflashBlockBaseAddr = 0x14U }
  Enumeration for various flash properties.
- enum _flash_execute_in_ram_function_constants {
  kFLASH_executeInRamFunctionMaxSize = 64U,
  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 = 0x40000U,
  kFLASH_resourceRangePflashSwapIfrEnd = 0x403FFU,
  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_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 FlexRAM load during reset option.

Flash version

- enum _flash_driver_version_constants {
  kFLASH_driverVersionName = 'F',
  kFLASH_driverVersionMajor = 2,
  kFLASH_driverVersionMinor = 1,
  kFLASH_driverVersionBugfix = 0 }
  FLASH driver version for ROM.
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
  Construct the version number for drivers.
- #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  FLASH driver version for SDK.

Flash configuration

- #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1
  Whether to support FlexNVM in flash driver.
- #define FLASH_SSD_IS_FLEXNVM_ENABLED (FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT && FSL_FEATURE_FLASH_HAS_FLEX_NVM)
  Whether the FlexNVM is enabled in flash driver.

- #define FLASH_DRIVER_IS_FLASH_RESIDENT 1
  Flash driver location.
- #define FLASH_DRIVER_IS_EXPORTED 0
  Flash Driver Export option.

Flash status

- enum _flash_status {
  kStatus_FLASH_Success = MAKE_STATUS(kStatusGroupGeneric, 0),
  kStatus_FLASH_InvalidArgument = MAKE_STATUS(kStatusGroupGeneric, 4),
  kStatus_FLASH_SizeError = MAKE_STATUS(kStatusGroupFlashDriver, 0),
  kStatus_FLASH_AlignmentError,
  kStatus_FLASH_AddressError = MAKE_STATUS(kStatusGroupFlashDriver, 2),
  kStatus_FLASH_AccessError,
  kStatus_FLASH_ProtectionViolation,
  kStatus_FLASH_CommandFailure,
  kStatus_FLASH_UnknownProperty = MAKE_STATUS(kStatusGroupFlashDriver, 6),
  kStatus_FLASH_EraseKeyError = MAKE_STATUS(kStatusGroupFlashDriver, 7),
  kStatus_FLASH_RegionExecuteOnly = MAKE_STATUS(kStatusGroupFlashDriver, 8),
  kStatus_FLASH_ExecuteInRamFunctionNotReady,
  kStatus_FLASH_PartitionStatusUpdateFailure,
  kStatus_FLASH_SetFlexramAsEepromError,
  kStatus_FLASH_RecoverFlexramAsRamError,
  kStatus_FLASH_SetFlexramAsRamError = MAKE_STATUS(kStatusGroupFlashDriver, 13),
  kStatus_FLASH_RecoverFlexramAsEepromError,
  kStatus_FLASH_CommandNotSupported = MAKE_STATUS(kStatusGroupFlashDriver, 15),
  kStatus_FLASH_SwapSystemNotInUninitialized,
  kStatus_FLASH_SwapIndicatorAddressError }
  Flash driver status codes.
- #define kStatusGroupGeneric 0
  Flash driver status group.
- #define **kStatusGroupFlashDriver** 1
- #define MAKE_STATUS(group, code) ((((group)*100) + (code)))
  Construct a status code value from a group and 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)))
  Construct the four char code for flash driver API key.

Initialization

- status_t FLASH_Init (flash_config_t *config)
  Initializes global flash properties structure members.
- status_t FLASH_SetCallback (flash_config_t *config, flash_callback_t callback)
  Set the desired flash callback function.

- status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t *config)
  Prepare 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 flash sectors encompassed by parameters passed into function.
- status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t *config, uint32_t key)
  Erases entire flash, including protected sectors.

Programming

- status_t FLASH_Program (flash_config_t *config, uint32_t start, uint32_t *src, uint32_t lengthInBytes)
  Programs flash with data at locations passed in through parameters.
- status_t FLASH_ProgramOnce (flash_config_t *config, uint32_t index, uint32_t *src, uint32_t lengthInBytes)
  Programs Program Once Field through parameters.

Reading

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

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

Parameters

config	Pointer to storage for the driver runtime state.
start	The start address of the desired flash memory to be programmed. Must be word-aligned.
src	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	Invalid argument is provided.

kStatus_FLASH_AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_AddressError	Address is out of range.
kStatus_FLASH_SetFlexramAsRamError	Failed to set flexram as ram
kStatus_FLASH_ExecuteInRamFunctionNotReady	Execute-in-ram function is not available.
kStatus_FLASH_AccessError	Invalid instruction codes and out-of bounds addresses.
kStatus_FLASH_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_CommandFailure	Run-time error during command execution.
kStatus_FLASH_RecoverFlexramAsEepromError	Failed to recover flexram as eeprom

Programs EEPROM with data at locations passed in through parameters

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

Parameters

config	Pointer to storage for the driver runtime state.
start	The start address of the desired flash memory to be programmed. Must be word-aligned.
src	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	Invalid argument is provided.

kStatus_FLASH_AddressError	Address is out of range.
kStatus_FLASH_SetFlexramAsEepromError	Failed to set flexram as eeprom.
kStatus_FLASH_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_RecoverFlexramAsRamError	Failed to recover flexram as ram

- status_t FLASH_ReadOnce (flash_config_t *config, uint32_t index, uint32_t *dst, uint32_t lengthInBytes)
  Read resource with data at locations passed in through parameters.

Security

- status_t FLASH_GetSecurityState (flash_config_t *config, flash_security_state_t *state)
  Returns the security state via the pointer passed into the function.
- status_t FLASH_SecurityBypass (flash_config_t *config, const uint8_t *backdoorKey)
  Allows user to bypass security with a backdoor key.

Verification

- status_t FLASH_VerifyEraseAll (flash_config_t *config, flash_margin_value_t margin)
  Verifies erasure of entire flash at specified margin level.
- status_t FLASH_VerifyErase (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_margin_value_t margin)
  Verifies erasure of desired flash area at 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 desired flash area at specified margin level.
- status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t *config, flash_margin_value_t margin)
  Verifies if the program flash executeonly 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 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 desired flash area via the pointer passed into the function.

Properties

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

Returns the desired flash property.

Flash Protection Utilities

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

Parameters

config	Pointer to storage for the driver runtime state.
option	The option used to set FlexRAM load behavior during reset.
eepromDataSizeCode	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
flexnvmPartitionCode	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

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

- status_t FLASH_PflashSetProtection (flash_config_t *config, uint32_t protectStatus)
  Set PFLASH Protection to the intended protection status.
- status_t FLASH_PflashGetProtection (flash_config_t *config, uint32_t *protectStatus)
  Get PFLASH Protection Status.

19.2 Data Structure Documentation

19.2.1 struct flash_execute_in_ram_function_config_t

Data Fields

- uint32_t activeFunctionCount
  Number of available execute-in-ram functions.
- uint8_t * flashRunCommand

execute-in-ram function: flash_run_command.
- uint8_t * flashCacheClearCommand
  execute-in-ram function: flash_cache_clear_command.

19.2.1.0.0.42 Field Documentation

19.2.1.0.0.42.1 uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

19.2.1.0.0.42.2 uint8_t* flash_execute_in_ram_function_config_t::flashRunCommand

19.2.1.0.0.42.3 uint8_t* flash_execute_in_ram_function_config_t::flashCacheClearCommand

19.2.2 struct flash_swap_state_config_t

Data Fields

- flash_swap_state_t flashSwapState
  Current swap system status.
- flash_swap_block_status_t currentSwapBlockStatus
  Current swap block status.
- flash_swap_block_status_t nextSwapBlockStatus
  Next swap block status.

19.2.2.0.0.43 Field Documentation

19.2.2.0.0.43.1 flash_swap_state_t flash_swap_state_config_t::flashSwapState

19.2.2.0.0.43.2 flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus

19.2.2.0.0.43.3 flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus

19.2.3 struct flash_swap_ifr_field_config_t

Data Fields

- uint16_t swapIndicatorAddress
  Swap indicator address field.
- uint16_t swapEnableWord
  Swap enable word field.
- uint8_t reserved0 [6]
  Reserved field.
- uint16_t swapDisableWord
  Swap disable word field.
- uint8_t reserved1 [4]
  Reserved field.

19.2.3.0.0.44 Field Documentation

19.2.3.0.0.44.1 uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress

19.2.3.0.0.44.2 uint16_t flash_swap_ifr_field_config_t::swapEnableWord

19.2.3.0.0.44.3 uint8_t flash_swap_ifr_field_config_t::reserved0[6]

19.2.3.0.0.44.4 uint16_t flash_swap_ifr_field_config_t::swapDisableWord

19.2.3.0.0.44.5 uint8_t flash_swap_ifr_field_config_t::reserved1[4]

19.2.4 struct flash_operation_config_t

Data Fields

- uint32_t convertedAddress
  Converted address for current flash type.
- uint32_t activeSectorSize
  Sector size of current flash type.
- uint32_t activeBlockSize
  Block size of current flash type.
- uint32_t blockWriteUnitSize
  write unit size.
- uint32_t sectorCmdAddressAligment
  Erase sector command address alignment.
- uint32_t partCmdAddressAligment
  Program/Verify part command address alignment.
- 32_t resourceCmdAddressAligment
  Read resource command address alignment.
- uint32_t checkCmdAddressAligment
  Program check command address alignment.

19.2.4.0.0.45 Field Documentation

19.2.4.0.0.45.1 uint32_t flash_operation_config_t::convertedAddress

19.2.4.0.0.45.2 uint32_t flash_operation_config_t::activeSectorSize

19.2.4.0.0.45.3 uint32_t flash_operation_config_t::activeBlockSize

19.2.4.0.0.45.4 uint32_t flash_operation_config_t::blockWriteUnitSize

19.2.4.0.0.45.5 uint32_t flash_operation_config_t::sectorCmdAddressAligment

19.2.4.0.0.45.6 uint32_t flash_operation_config_t::partCmdAddressAligment

19.2.4.0.0.45.7 uint32_t flash_operation_config_t::resourceCmdAddressAligment

19.2.4.0.0.45.8 uint32_t flash_operation_config_t::checkCmdAddressAligment

19.2.5 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
  Base address of the first PFlash block.
- uint32_t PFlashTotalSize
  Size of all combined PFlash block.
- uint32_t PFlashBlockCount
  Number of PFlash blocks.
- uint32_t PFlashSectorSize
  Size in bytes of a sector of PFlash.
- flash_callback_t PFlashCallback
  Callback function for flash API.
- uint32_t PFlashAccessSegmentSize
  Size in bytes of a access segment of PFlash.
- uint32_t PFlashAccessSegmentCount
  Number of PFlash access segments.
- uint32_t * flashExecuteInRamFunctionInfo
  Info struct of flash execute-in-ram function.
- uint32_t FlexRAMBlockBase
  For FlexNVM device, this is the base address of FlexRAM For non-FlexNVM device, this is the base address of acceleration RAM memory.
- uint32_t FlexRAMTotalSize
  For FlexNVM device, this is the size of FlexRAM For non-FlexNVM device, this is the size of acceleration RAM memory.
- uint32_t DFlashBlockBase
  For FlexNVM device, this is the base address of D-Flash memory (FlexNVM memory); For non-FlexNVM

device, this field is unused.

- uint32_t DFlashTotalSize
  For FlexNVM device, this is total size of the FlexNVM memory; For non-FlexNVM device, this field is unused.
- uint32_t EEpromTotalSize
  For FlexNVM device, this is the size in byte of EEPROM area which was partitioned from FlexRAM; For non-FlexNVM device, this field is unused.

19.2.5.0.0.46 Field Documentation

19.2.5.0.0.46.1 uint32_t flash_config_t::PFlashTotalSize

19.2.5.0.0.46.2 uint32_t flash_config_t::PFlashBlockCount

19.2.5.0.0.46.3 uint32_t flash_config_t::PFlashSectorSize

19.2.5.0.0.46.4 flash_callback_t flash_config_t::PFlashCallback

19.2.5.0.0.46.5 uint32_t flash_config_t::PFlashAccessSegmentSize

19.2.5.0.0.46.6 uint32_t flash_config_t::PFlashAccessSegmentCount

19.2.5.0.0.46.7 uint32_t* flash_config_t::flashExecuteInRamFunctionInfo

19.3 Macro Definition Documentation

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

19.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

Version 2.1.0.

19.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enable FlexNVM support by default.

19.3.4 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for flash resident application.

19.3.5 #define FLASH_DRIVER_IS_EXPORTED 0

Used for SDK application.

19.3.6 #define kStatusGroupGeneric 0

19.3.7 #define MAKE_STATUS(*group, code*) ((((group)∗100) + (code)))

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

19.4 Enumeration Type Documentation

19.4.1 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.

19.4.2 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 specified baseline.
kStatus_FLASH_AddressError Address is out of range.
kStatus_FLASH_AccessError Invalid instruction codes and out-of bounds addresses.
kStatus_FLASH_ProtectionViolation The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_CommandFailure Run-time error during command execution.
kStatus_FLASH_UnknownProperty Unknown property.
kStatus_FLASH_EraseKeyError Api erase key is invalid.
kStatus_FLASH_RegionExecuteOnly 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 uninitialzed state.
kStatus_FLASH_SwapIndicatorAddressError Swap indicator address is invalid.

19.4.3 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.

19.4.4 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.

19.4.5 enum flash_security_state_t

Enumerator

kFLASH_securityStateNotSecure Flash is not secure.
kFLASH_securityStateBackdoorEnabled Flash backdoor is enabled.
kFLASH_securityStateBackdoorDisabled Flash backdoor is disabled.

19.4.6 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.

19.4.7 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.

19.4.8 enum flash_property_tag_t

Enumerator

kFLASH_propertyPflashSectorSize Pflash sector size property.
kFLASH_propertyPflashTotalSize Pflash total size property.
kFLASH_propertyPflashBlockSize Pflash block size property.
kFLASH_propertyPflashBlockCount Pflash block count property.
kFLASH_propertyPflashBlockBaseAddr Pflash block base address property.
kFLASH_propertyPflashFacSupport Pflash fac support property.
kFLASH_propertyPflashAccessSegmentSize Pflash access segment size property.
kFLASH_propertyPflashAccessSegmentCount Pflash access segment count property.
kFLASH_propertyFlexRamBlockBaseAddr FlexRam block base address property.
kFLASH_propertyFlexRamTotalSize FlexRam total size property.
kFLASH_propertyDflashSectorSize Dflash sector size property.
kFLASH_propertyDflashTotalSize Dflash total size property.
kFLASH_propertyDflashBlockSize Dflash block count property.
kFLASH_propertyDflashBlockCount Dflash block base address property.
kFLASH_propertyDflashBlockBaseAddr Eeprom total size property.

19.4.9 enum _flash_execute_in_ram_function_constants

Enumerator

kFLASH_executeInRamFunctionMaxSize Max size of execute-in-ram function.
kFLASH_executeInRamFunctionTotalNum Total number of execute-in-ram functions.

19.4.10 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 Version ID.

19.4.11 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.

19.4.12 enum flash_flexram_function_option_t

Enumerator

kFLASH_flexramFunctionOptionAvailableAsRam Option used to make FlexRAM available as RAM.
kFLASH_flexramFunctionOptionAvailableForEeprom Option used to make FlexRAM available for EEPROM.

19.4.13 enum flash_swap_function_option_t

Enumerator

kFLASH_swapFunctionOptionEnable Option used to enable Swap function.
kFLASH_swapFunctionOptionDisable Option used to Disable Swap function.

19.4.14 enum flash_swap_control_option_t

Enumerator

kFLASH_swapControlOptionIntializeSystem Option used to Intialize Swap System.
kFLASH_swapControlOptionSetInUpdateState Option used to Set Swap in Update State.
kFLASH_swapControlOptionSetInCompleteState Option used to Set Swap in Complete State.
kFLASH_swapControlOptionReportStatus Option used to Report Swap Status.
kFLASH_swapControlOptionDisableSystem Option used to Disable Swap Status.

19.4.15 enum flash_swap_state_t

Enumerator

kFLASH_swapStateUninitialized Flash swap system is in uninitialized state.
kFLASH_swapStateReady Flash swap system is in ready state.
kFLASH_swapStateUpdate Flash swap system is in update state.
kFLASH_swapStateUpdateErased Flash swap system is in updateErased state.
kFLASH_swapStateComplete Flash swap system is in complete state.
kFLASH_swapStateDisabled Flash swap system is in disabled state.

19.4.16 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.

19.4.17 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.

19.5 Function Documentation

19.5.1 status_t FLASH_Init (flash_config_t * *config*)

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

Parameters

config	Pointer to storage for the driver runtime state.

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_-PartitionStatusUpdate-Failure	Failed to update partition status.

19.5.2 status_t FLASH_SetCallback (flash_config_t * *config*, flash_callback_t *callback*)

Parameters

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

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	Invalid argument is provided.

19.5.3 status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t * *config*)

Parameters

config	Pointer to storage for the driver runtime state.

Return values

kStatus_FLASH_Success	Api was executed successfully.

kStatus_FLASH_Invalid-Argument	Invalid argument is provided.

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

Parameters

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

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	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 partition status

19.5.5 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	Pointer to 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	value used to validate all flash erase APIs.

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	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_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.

19.5.6 status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t * *config*, uint32_t *key*)

Parameters

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

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	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 partition status

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

Parameters

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

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	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.

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

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

Parameters

config	Pointer to storage for the driver runtime state.
start	The start address of the desired flash memory to be programmed. Must be word-aligned.
src	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	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 command execution.

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

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

Parameters

config	Pointer to storage for the driver runtime state.
index	The index indicating which area of Program Once Field to be programmed.
src	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_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.

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

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

Parameters

config	Pointer to storage for the driver runtime state.
start	The start address of the desired flash memory to be programmed. Must be word-aligned.
dst	Pointer to the destination buffer of data that is used to store data to be read.
lengthInBytes	The length, given in bytes (not words or long-words) to be read. Must be word-aligned.
option	The resource option which indicates which area should be read back.

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_InvalidArgument	Invalid argument is provided.
kStatus_FLASH_AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_ExecuteInRamFunctionNotReady	Execute-in-ram function is not available.
kStatus_FLASH_AccessError	Invalid instruction codes and out-of bounds addresses.
kStatus_FLASH_ProtectionViolation	The program/erase operation is requested to execute on protected areas.

kStatus_FLASH_CommandFailure	Run-time error during command execution.

Read Program Once Field through parameters

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

Parameters

config	Pointer to storage for the driver runtime state.
index	The index indicating the area of program once field to be read.
dst	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	Invalid argument is provided.
kStatus_FLASH_ExecuteInRamFunctionNotReady	Execute-in-ram function is not available.
kStatus_FLASH_AccessError	Invalid instruction codes and out-of bounds addresses.
kStatus_FLASH_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_CommandFailure	Run-time error during command execution.

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

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

Parameters

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

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	Invalid argument is provided.

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

If the MCU is in secured state, this function will unsecure the MCU by comparing the provided backdoor key with ones in the Flash Configuration Field.

Parameters

config	Pointer to storage for the driver runtime state.
backdoorKey	Pointer to the user buffer containing the backdoor key.

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.

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

This function will check to see if the flash have been erased to the specified read margin level.

Parameters

config	Pointer to storage for the driver runtime state.
margin	Read margin choice

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.

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

This function will check the appropriate number of flash sectors based on the desired start address and length to see if the flash have been erased to the specified read margin level.

Parameters

config	Pointer to 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	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 command execution.

19.5.14 status_t FLASH_VerifyProgram (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, const uint32_t * *expectedData*, flash_margin_value_t *margin*, uint32_t * *failedAddress*, uint32_t * *failedData*)

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it with expected data for a given flash area as determined by the start address and length.

Parameters

config	Pointer to 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	Pointer to the expected data that is to be verified against.
margin	Read margin choice
failedAddress	Pointer to returned failing address.

failedData	Pointer to returned failing data. Some derivitives do not included failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	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 command execution.

19.5.15 status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t * *config*, flash_margin_value_t *margin*)

Parameters

config	Pointer to storage for the driver runtime state.
margin	Read margin choice

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.

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

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

Parameters

config	Pointer to 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	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	Invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_Address-Error	Address is out of range.

19.5.17 status_t FLASH_IsExecuteOnly (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, flash_execute_only_access_state_t * *access_state*)

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

Parameters

config	Pointer to 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	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_Invalid-Argument	Invalid argument is provided.
kStatus_FLASH_-AlignmentError	Parameter is not aligned with specified baseline.
kStatus_FLASH_Address-Error	Address is out of range.

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

Parameters

config	Pointer to storage for the driver runtime state.
whichProperty	The desired property from the list of properties in enum flash_property_tag_t

value	Pointer to the value returned for the desired flash property

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	Invalid argument is provided.
kStatus_FLASH_-UnknownProperty	unknown property tag

19.5.19 status_t FLASH_PflashSetProtection (flash_config_t * *config*, uint32_t *protectStatus*)

Parameters

config	Pointer to storage for the driver runtime state.
protectStatus	The expected protect status user wants to set to PFlash protection register. Each bit is corresponding to protection of 1/32 of the total PFlash. The least significant bit is corresponding to the lowest address area of P-Flash. 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	Invalid argument is provided.
kStatus_FLASH_-CommandFailure	Run-time error during command execution.

19.5.20 status_t FLASH_PflashGetProtection (flash_config_t * *config,* uint32_t * *protectStatus*)

Parameters

config	Pointer to storage for the driver runtime state.
protectStatus	Protect status returned by PFlash IP. Each bit is corresponding to protection of 1/32 of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. Thee are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

kStatus_FLASH_Success	Api was executed successfully.
kStatus_FLASH_Invalid-Argument	Invalid argument is provided.

Chapter 20
FlexBus: External Bus Interface Driver

20.1 Overview

The KSDK provides a peripheral driver for the Crossbar External Bus Interface (FlexBus) block of Kinetis devices.

20.2 Overview

A multifunction external bus interface is provided on the device with a basic functionality to interface to slave-only devices. It can be directly connected to the following asynchronous or synchronous devices with little or no additional circuitry:

- External ROMs
- Flash memories
- Programmable logic devices
- Other simple target (slave) devices

For asynchronous devices, a simple chip-select based interface can be used. The FlexBus interface has up to six general purpose chip-selects, FB_CS[5:0]. The actual number of chip selects available depends upon the device and its pin configuration.

20.3 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.

20.4 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);
```

Typical use case and example

Files

- file fsl_flexbus.h

Data Structures

- struct flexbus_config_t
  Configuration structure that the user needs to set. More...

Enumerations

- enum flexbus_port_size_t {
  kFLEXBUS_4Bytes = 0x00U,
  kFLEXBUS_1Byte = 0x01U,
  kFLEXBUS_2Bytes = 0x02U }
  Defines port size for FlexBus peripheral.
- enum flexbus_write_address_hold_t {
  kFLEXBUS_Hold1Cycle = 0x00U,
  kFLEXBUS_Hold2Cycles = 0x01U,
  kFLEXBUS_Hold3Cycles = 0x02U,
  kFLEXBUS_Hold4Cycles = 0x03U }
  Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum flexbus_read_address_hold_t {
  kFLEXBUS_Hold1Or0Cycles = 0x00U,
  kFLEXBUS_Hold2Or1Cycles = 0x01U,
  kFLEXBUS_Hold3Or2Cycle = 0x02U,
  kFLEXBUS_Hold4Or3Cycle = 0x03U }
  Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum flexbus_address_setup_t {
  kFLEXBUS_FirstRisingEdge = 0x00U,
  kFLEXBUS_SecondRisingEdge = 0x01U,
  kFLEXBUS_ThirdRisingEdge = 0x02U,
  kFLEXBUS_FourthRisingEdge = 0x03U }
  Address setup for FlexBus peripheral.
- enum flexbus_bytelane_shift_t {
  kFLEXBUS_NotShifted = 0x00U,
  kFLEXBUS_Shifted = 0x01U }
  Defines byte-lane shift for FlexBus peripheral.
- enum flexbus_multiplex_group1_t {
  kFLEXBUS_MultiplexGroup1_FB_ALE = 0x00U,
  kFLEXBUS_MultiplexGroup1_FB_CS1 = 0x01U,
  kFLEXBUS_MultiplexGroup1_FB_TS = 0x02U }
  Defines multiplex group1 valid signals.
- enum flexbus_multiplex_group2_t {
  kFLEXBUS_MultiplexGroup2_FB_CS4 = 0x00U,
  kFLEXBUS_MultiplexGroup2_FB_TSIZ0 = 0x01U,
  kFLEXBUS_MultiplexGroup2_FB_BE_31_24 = 0x02U }
  Defines multiplex group2 valid signals.

- enum flexbus_multiplex_group3_t {
  kFLEXBUS_MultiplexGroup3_FB_CS5 = 0x00U,
  kFLEXBUS_MultiplexGroup3_FB_TSIZ1 = 0x01U,
  kFLEXBUS_MultiplexGroup3_FB_BE_23_16 = 0x02U }
  Defines multiplex group3 valid signals.
- enum flexbus_multiplex_group4_t {
  kFLEXBUS_MultiplexGroup4_FB_TBST = 0x00U,
  kFLEXBUS_MultiplexGroup4_FB_CS2 = 0x01U,
  kFLEXBUS_MultiplexGroup4_FB_BE_15_8 = 0x02U }
  Defines multiplex group4 valid signals.
- enum flexbus_multiplex_group5_t {
  kFLEXBUS_MultiplexGroup5_FB_TA = 0x00U,
  kFLEXBUS_MultiplexGroup5_FB_CS3 = 0x01U,
  kFLEXBUS_MultiplexGroup5_FB_BE_7_0 = 0x02U }
  Defines multiplex group5 valid signals.

Driver version

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

FlexBus functional operation

- void FLEXBUS_Init (FB_Type *base, const flexbus_config_t *config)
  Initializes and configures the FlexBus module.
- void FLEXBUS_Deinit (FB_Type *base)
  De-initializes a FlexBus instance.
- void FLEXBUS_GetDefaultConfig (flexbus_config_t *config)
  Initializes the FlexBus configuration structure.

20.5 Data Structure Documentation

20.5.1 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.

20.6 Macro Definition Documentation

20.6.1 #define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

20.7 Enumeration Type Documentation

20.7.1 enum flexbus_port_size_t

Enumerator

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

20.7.2 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.

20.7.3 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.

20.7.4 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.

20.7.5 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

20.7.6 enum flexbus_multiplex_group1_t

Enumerator

kFLEXBUS_MultiplexGroup1_FB_ALE FB_ALE.

kFLEXBUS_MultiplexGroup1_FB_CS1 FB_CS1.
kFLEXBUS_MultiplexGroup1_FB_TS FB_TS.

20.7.7 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.

20.7.8 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.

20.7.9 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.

20.7.10 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.

20.8 Function Documentation

20.8.1 void FLEXBUS_Init (FB_Type * *base,* const flexbus_config_t * *config*)

This function enables the clock gate for FlexBus module. Only chip 0 is validated and set to known values. Other chips are disabled. NOTE: In this function, certain parameters, depending on external memories,

must be set before using FLEXBUS_Init() function. This example shows how to set up the uart_state-_t and the flexbus_config_t parameters and how to call the FLEXBUS_Init function by passing in these parameters:

```
flexbus_config_t flexbusConfig;
FLEXBUS_GetDefaultConfig(&flexbusConfig);
flexbusConfig.waitStates            = 2U;
flexbusConfig.chipBaseAddress       = 0x60000000U;
flexbusConfig.chipBaseAddressMask   = 7U;
FLEXBUS_Init(FB, &flexbusConfig);
```

Parameters

base	FlexBus peripheral address.
config	Pointer to the configure structure

20.8.2 void FLEXBUS_Deinit (FB_Type * *base*)

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

Parameters

base	FlexBus peripheral address.

20.8.3 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;
```

Function Documentation

Parameters

config	Pointer to the initialization structure.

See Also

FLEXBUS_Init

Chapter 21
FlexCAN: Flex Controller Area Network Driver

21.1 Overview

The KSDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of Kinetis devices.

Modules

- FlexCAN Driver
- FlexCAN eDMA Driver

21.2 FlexCAN Driver

21.2.1 Overview

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

21.2.2 Typical use case

21.2.2.1 Message Buffer Send Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t txFrame;

/* Init FlexCAN module.
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enable FlexCAN module.
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the transmit message buffer.
FLEXCAN_SetTxMbConfig(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, true);

/* Prepares the transmit frame for sending.
txFrame.format = KFLEXCAN_FrameFormatStandard;
txFrame.type   = KFLEXCAN_FrameTypeData;
txFrame.id     = FLEXCAN_ID_STD(0x123);
txFrame.length = 8;
txFrame.dataWord0 = CAN_WORD0_DATA_BYTE_0(0x11) |
                    CAN_WORD0_DATA_BYTE_1(0x22) |
                    CAN_WORD0_DATA_BYTE_2(0x33) |
                    CAN_WORD0_DATA_BYTE_3(0x44);
txFrame.dataWord1 = CAN_WORD1_DATA_BYTE_4(0x55) |
                    CAN_WORD1_DATA_BYTE_5(0x66) |
                    CAN_WORD1_DATA_BYTE_6(0x77) |
                    CAN_WORD1_DATA_BYTE_7(0x88);
/* Writes a transmit message buffer to send a CAN Message.
FLEXCAN_WriteTxMb(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, &txFrame);

/* Waits until the transmit message buffer is empty.
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX));

/* Cleans the transmit message buffer empty status.
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX);
```

21.2.2.2 Message Buffer Receive Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module.
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module.
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive message buffer.
```

```
mbConfig.format    = KFLEXCAN_FrameFormatStandard;
mbConfig.type      = KFLEXCAN_FrameTypeData;
mbConfig.id        = FLEXCAN_ID_STD(0x123);
FLEXCAN_SetRxMbConfig(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &mbConfig, true);

/* Waits until the receive message buffer is full.
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX));

/* Reads the received message from the receive message buffer.
FLEXCAN_ReadRxMb(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &rxFrame);

/* Cleans the receive message buffer full status.
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX);
```

21.2.2.3 Receive FIFO Operation

```
uint32_t rxFifoFilter[] = {FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 1, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 1, 0)}
      ;

flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module.
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module.
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive FIFO.
rxFifoConfig.idFilterTable = rxFifoFilter;
rxFifoConfig.idFilterType  = KFLEXCAN_RxFifoFilterTypeA;
rxFifoConfig.idFilterNum   = sizeof(rxFifoFilter) / sizeof(rxFifoFilter[0]);
rxFifoConfig.priority      = KFLEXCAN_RxFifoPrioHigh;
FlEXCAN_SetRxFifoConfig(EXAMPLE_CAN, &rxFifoConfig, true);

/* Waits until the receive FIFO becomes available.
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag));

/* Reads the message from the receive FIFO.
FlEXCAN_ReadRxFifo(EXAMPLE_CAN, &rxFrame);

/* Cleans the receive FIFO available status.
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag);
```

Files

- file fsl_flexcan.h

Data Structures

- struct flexcan_frame_t
  FlexCAN message frame structure. *More...*
- struct flexcan_config_t
  FlexCAN module configuration structure. *More...*

- struct flexcan_timing_config_t
  FlexCAN protocol timing characteristic configuration structure. More...
- struct flexcan_rx_mb_config_t
  FlexCAN Receive Message Buffer configuration structure. More...
- struct flexcan_rx_fifo_config_t
  FlexCAN Rx FIFO configure structure. More...
- struct flexcan_mb_transfer_t
  FlexCAN Message Buffer transfer. More...
- struct flexcan_fifo_transfer_t
  FlexCAN Rx FIFO transfer. More...
- struct flexcan_handle_t
  FlexCAN handle structure. More...

Macros

- #define FLEXCAN_ID_STD(id) (((uint32_t)((uint32_t)(id)) << CAN_ID_STD_SHIFT)) & CAN_ID_STD_MASK)
  FlexCAN Frame ID helper macro.
- #define FLEXCAN_ID_EXT(id)
  Extend Frame ID helper macro.
- #define FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)
  FlexCAN Rx Message Buffer Mask helper macro.
- #define FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)
  Extend Rx Message Buffer Mask helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)
  FlexCAN Rx FIFO Mask helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)
  Standard Rx FIFO Mask helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)
  Standard Rx FIFO Mask helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 21)
  Standard Rx FIFO Mask helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 13)
  Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 5)
  Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id) ((FLEXCAN_ID_STD(id) & 0x7F8) >> 3)
  Standard Rx FIFO Mask helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type A helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide)
  Extend Rx FIFO Mask helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id) ((FLEXCAN_ID_EXT(id) &

0x1FE00000) << 3)
Extend Rx FIFO Mask helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id)
Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id)
Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 21)
Extend Rx FIFO Mask helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(id, rtr, ide) FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)
FlexCAN Rx FIFO Filter helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(id, rtr, ide)
Standard Rx FIFO Filter helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(id, rtr, ide)
Standard Rx FIFO Filter helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(id)
Standard Rx FIFO Filter helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(id)
Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(id)
Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id) FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)
Standard Rx FIFO Filter helper macro Type C lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type A helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type B upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)
Extend Rx FIFO Filter helper macro Type B lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)
Extend Rx FIFO Filter helper macro Type C upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)
Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)
Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(id) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)
Extend Rx FIFO Filter helper macro Type C lower part helper macro.

Typedefs

- typedef void(* flexcan_transfer_callback_t)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint32_t result, void *userData)
FlexCAN transfer callback function.

Enumerations

- enum _flexcan_status {
  kStatus_FLEXCAN_TxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
  kStatus_FLEXCAN_TxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 1),
  kStatus_FLEXCAN_TxSwitchToRx,
  kStatus_FLEXCAN_RxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 3),
  kStatus_FLEXCAN_RxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 4),
  kStatus_FLEXCAN_RxOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 5),
  kStatus_FLEXCAN_RxFifoBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 6),
  kStatus_FLEXCAN_RxFifoIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 7),
  kStatus_FLEXCAN_RxFifoOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 8),
  kStatus_FLEXCAN_RxFifoWarning = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
  kStatus_FLEXCAN_ErrorStatus = MAKE_STATUS(kStatusGroup_FLEXCAN, 10),
  kStatus_FLEXCAN_UnHandled = MAKE_STATUS(kStatusGroup_FLEXCAN, 11) }
  FlexCAN transfer status.
- enum flexcan_frame_format_t {
  kFLEXCAN_FrameFormatStandard = 0x0U,
  kFLEXCAN_FrameFormatExtend = 0x1U }
  FlexCAN frame format.
- enum flexcan_frame_type_t {
  kFLEXCAN_FrameTypeData = 0x0U,
  kFLEXCAN_FrameTypeRemote = 0x1U }
  FlexCAN frame type.
- enum flexcan_clock_source_t {
  kFLEXCAN_ClkSrcOsc = 0x0U,
  kFLEXCAN_ClkSrcPeri = 0x1U }
  FlexCAN clock source.
- enum flexcan_rx_fifo_filter_type_t {
  kFLEXCAN_RxFifoFilterTypeA = 0x0U,
  kFLEXCAN_RxFifoFilterTypeB,
  kFLEXCAN_RxFifoFilterTypeC,
  kFLEXCAN_RxFifoFilterTypeD = 0x3U }
  FlexCAN Rx Fifo Filter type.
- enum flexcan_rx_fifo_priority_t {
  kFLEXCAN_RxFifoPrioLow = 0x0U,
  kFLEXCAN_RxFifoPrioHigh = 0x1U }
  FlexCAN Rx FIFO priority.
- enum _flexcan_interrupt_enable {
  kFLEXCAN_BusOffInterruptEnable = CAN_CTRL1_BOFFMSK_MASK,
  kFLEXCAN_ErrorInterruptEnable = CAN_CTRL1_ERRMSK_MASK,
  kFLEXCAN_RxWarningInterruptEnable = CAN_CTRL1_RWRNMSK_MASK,
  kFLEXCAN_TxWarningInterruptEnable = CAN_CTRL1_TWRNMSK_MASK,
  kFLEXCAN_WakeUpInterruptEnable = CAN_MCR_WAKMSK_MASK }
  FlexCAN interrupt configuration structure, default settings all disabled.
- enum _flexcan_flags {

kFLEXCAN_SynchFlag = CAN_ESR1_SYNCH_MASK,
kFLEXCAN_TxWarningIntFlag = CAN_ESR1_TWRNINT_MASK,
kFLEXCAN_RxWarningIntFlag = CAN_ESR1_RWRNINT_MASK,
kFLEXCAN_TxErrorWarningFlag = CAN_ESR1_TXWRN_MASK,
kFLEXCAN_RxErrorWarningFlag = CAN_ESR1_RXWRN_MASK,
kFLEXCAN_IdleFlag = CAN_ESR1_IDLE_MASK,
kFLEXCAN_FaultConfinementFlag = CAN_ESR1_FLTCONF_MASK,
kFLEXCAN_TransmittingFlag = CAN_ESR1_TX_MASK,
kFLEXCAN_ReceivingFlag = CAN_ESR1_RX_MASK,
kFLEXCAN_BusOffIntFlag = CAN_ESR1_BOFFINT_MASK,
kFLEXCAN_ErrorIntFlag = CAN_ESR1_ERRINT_MASK,
kFLEXCAN_WakeUpIntFlag = CAN_ESR1_WAKINT_MASK,
kFLEXCAN_ErrorFlag }
FlexCAN status flags.

- enum _flexcan_error_flags {
  kFLEXCAN_StuffingError = CAN_ESR1_STFERR_MASK,
  kFLEXCAN_FormError = CAN_ESR1_FRMERR_MASK,
  kFLEXCAN_CrcError = CAN_ESR1_CRCERR_MASK,
  kFLEXCAN_AckError = CAN_ESR1_ACKERR_MASK,
  kFLEXCAN_Bit0Error = CAN_ESR1_BIT0ERR_MASK,
  kFLEXCAN_Bit1Error = CAN_ESR1_BIT1ERR_MASK }
  FlexCAN error status flags.
- enum _flexcan_rx_fifo_flags {
  kFLEXCAN_RxFifoOverflowFlag = CAN_IFLAG1_BUF7I_MASK,
  kFLEXCAN_RxFifoWarningFlag = CAN_IFLAG1_BUF6I_MASK,
  kFLEXCAN_RxFifoFrameAvlFlag = CAN_IFLAG1_BUF5I_MASK }
  FlexCAN Rx FIFO status flags.

Driver version

- #define FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  FlexCAN driver version 2.1.0.

Initialization and deinitialization

- void FLEXCAN_Init (CAN_Type *base, const flexcan_config_t *config, uint32_t sourceClock_Hz)
  Initializes a FlexCAN instance.
- void FLEXCAN_Deinit (CAN_Type *base)
  De-initializes a FlexCAN instance.
- void FLEXCAN_GetDefaultConfig (flexcan_config_t *config)
  Get the default configuration structure.

Configuration.

- void FLEXCAN_SetTimingConfig (CAN_Type *base, const flexcan_timing_config_t *config)
  Sets the FlexCAN protocol timing characteristic.
- void FlEXCAN_SetRxMbGlobalMask (CAN_Type *base, uint32_t mask)
  Sets the FlexCAN receive message buffer global mask.
- void FlEXCAN_SetRxFifoGlobalMask (CAN_Type *base, uint32_t mask)
  Sets the FlexCAN receive FIFO global mask.
- void FlEXCAN_SetRxIndividualMask (CAN_Type *base, uint8_t maskIdx, uint32_t mask)
  Sets the FlexCAN receive individual mask.
- void FLEXCAN_SetTxMbConfig (CAN_Type *base, uint8_t mbIdx, bool enable)
  Configures a FlexCAN transmit message buffer.
- void FLEXCAN_SetRxMbConfig (CAN_Type *base, uint8_t mbIdx, const flexcan_rx_mb_config_t *config, bool enable)
  Configures a FlexCAN Receive Message Buffer.
- void FlEXCAN_SetRxFifoConfig (CAN_Type *base, const flexcan_rx_fifo_config_t *config, bool enable)
  Configures the FlexCAN Rx FIFO.

Status

- static uint32_t FLEXCAN_GetStatusFlags (CAN_Type *base)
  Gets the FlexCAN module interrupt flags.
- static void FLEXCAN_ClearStatusFlags (CAN_Type *base, uint32_t mask)
  Clears status flags with the provided mask.
- static void FlEXCAN_GetBusErrCount (CAN_Type *base, uint8_t *txErrBuf, uint8_t *rxErrBuf)
  Gets the FlexCAN Bus Error Counter value.
- static uint32_t FLEXCAN_GetMbStatusFlags (CAN_Type *base, uint32_t mask)
  Gets the FlexCAN Message Buffer interrupt flags.
- static void FLEXCAN_ClearMbStatusFlags (CAN_Type *base, uint32_t mask)
  Clears the FlexCAN Message Buffer interrupt flags.

Interrupts

- static void FLEXCAN_EnableInterrupts (CAN_Type *base, uint32_t mask)
  Enables FlexCAN interrupts according to provided mask.
- static void FLEXCAN_DisableInterrupts (CAN_Type *base, uint32_t mask)
  Disables FlexCAN interrupts according to provided mask.
- static void FLEXCAN_EnableMbInterrupts (CAN_Type *base, uint32_t mask)
  Enables FlexCAN Message Buffer interrupts.
- static void FLEXCAN_DisableMbInterrupts (CAN_Type *base, uint32_t mask)
  Disables FlexCAN Message Buffer interrupts.

Bus Operations

- static void FLEXCAN_Enable (CAN_Type *base, bool enable)
  Enables or disables the FlexCAN module operation.

- status_t FLEXCAN_WriteTxMb (CAN_Type *base, uint8_t mbIdx, const flexcan_frame_t *txFrame)
  Writes a FlexCAN Message to Transmit Message Buffer.
- status_t FLEXCAN_ReadRxMb (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *rxFrame)
  Reads a FlexCAN Message from Receive Message Buffer.
- status_t FlEXCAN_ReadRxFifo (CAN_Type *base, flexcan_frame_t *rxFrame)
  Reads a FlexCAN Message from Rx FIFO.

Transactional

- status_t FlEXCAN_TransferSendBlocking (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *txFrame)
  Performs a polling send transaction on the CAN bus.
- status_t FlEXCAN_TransferReceiveBlocking (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *rxFrame)
  Performs a polling receive transaction on the CAN bus.
- status_t FlEXCAN_TransferReceiveFifoBlocking (CAN_Type *base, flexcan_frame_t *rxFrame)
  Performs a polling receive transaction from Rx FIFO on the CAN bus.
- void FLEXCAN_TransferCreateHandle (CAN_Type *base, flexcan_handle_t *handle, flexcan_transfer_callback_t callback, void *userData)
  Initializes the FlexCAN handle.
- status_t FLEXCAN_TransferSendNonBlocking (CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *xfer)
  Sends a message using IRQ.
- status_t FLEXCAN_TransferReceiveNonBlocking (CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *xfer)
  Receives a message using IRQ.
- status_t FLEXCAN_TransferReceiveFifoNonBlocking (CAN_Type *base, flexcan_handle_t *handle, flexcan_fifo_transfer_t *xfer)
  Receives a message from Rx FIFO using IRQ.
- void FLEXCAN_TransferAbortSend (CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)
  Aborts the interrupt driven message send process.
- void FLEXCAN_TransferAbortReceive (CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)
  Aborts the interrupt driven message receive process.
- void FLEXCAN_TransferAbortReceiveFifo (CAN_Type *base, flexcan_handle_t *handle)
  Aborts the interrupt driven message receive from Rx FIFO process.
- void FLEXCAN_TransferHandleIRQ (CAN_Type *base, flexcan_handle_t *handle)
  FlexCAN IRQ handle function.

21.2.3 Data Structure Documentation

21.2.3.1 struct flexcan_frame_t

21.2.3.1.0.47 Field Documentation

21.2.3.1.0.47.1 uint32_t flexcan_frame_t::timestamp

21.2.3.1.0.47.2 uint32_t flexcan_frame_t::length

21.2.3.1.0.47.3 uint32_t flexcan_frame_t::type

21.2.3.1.0.47.4 uint32_t flexcan_frame_t::format

21.2.3.1.0.47.5 uint32_t flexcan_frame_t::reserve1

21.2.3.1.0.47.6 uint32_t flexcan_frame_t::idhit

21.2.3.1.0.47.7 uint32_t flexcan_frame_t::id

21.2.3.1.0.47.8 uint32_t flexcan_frame_t::reserve2

21.2.3.1.0.47.9 uint32_t flexcan_frame_t::dataWord0

21.2.3.1.0.47.10 uint32_t flexcan_frame_t::dataWord1

21.2.3.1.0.47.11 uint8_t flexcan_frame_t::dataByte3

21.2.3.1.0.47.12 uint8_t flexcan_frame_t::dataByte2

21.2.3.1.0.47.13 uint8_t flexcan_frame_t::dataByte1

21.2.3.1.0.47.14 uint8_t flexcan_frame_t::dataByte0

21.2.3.1.0.47.15 uint8_t flexcan_frame_t::dataByte7

21.2.3.1.0.47.16 uint8_t flexcan_frame_t::dataByte6

21.2.3.1.0.47.17 uint8_t flexcan_frame_t::dataByte5

21.2.3.1.0.47.18 uint8_t flexcan_frame_t::dataByte4

21.2.3.2 struct flexcan_config_t

Data Fields

- uint32_t baudRate
  FlexCAN baud rate in bps.
- flexcan_clock_source_t clkSrc
  Clock source for FlexCAN Protocol Engine.

- uint8_t maxMbNum
  The maximum number of Message Buffers used by user.
- bool enableLoopBack
  Enable or Disable Loop Back Self Test Mode.
- bool enableSelfWakeup
  Enable or Disable Self Wakeup Mode.
- bool enableIndividMask
  Enable or Disable Rx Individual Mask.

21.2.3.2.0.48 Field Documentation

21.2.3.2.0.48.1 uint32_t flexcan_config_t::baudRate

21.2.3.2.0.48.2 flexcan_clock_source_t flexcan_config_t::clkSrc

21.2.3.2.0.48.3 uint8_t flexcan_config_t::maxMbNum

21.2.3.2.0.48.4 bool flexcan_config_t::enableLoopBack

21.2.3.2.0.48.5 bool flexcan_config_t::enableSelfWakeup

21.2.3.2.0.48.6 bool flexcan_config_t::enableIndividMask

21.2.3.3 struct flexcan_timing_config_t

Data Fields

- uint8_t preDivider
  Clock Pre-scaler Division Factor.
- uint8_t rJumpwidth
  Re-sync Jump Width.
- uint8_t phaseSeg1
  Phase Segment 1.
- uint8_t phaseSeg2
  Phase Segment 2.
- uint8_t propSeg
  Propagation Segment.

21.2.3.3.0.49 Field Documentation

21.2.3.3.0.49.1 uint8_t flexcan_timing_config_t::preDivider

21.2.3.3.0.49.2 uint8_t flexcan_timing_config_t::rJumpwidth

21.2.3.3.0.49.3 uint8_t flexcan_timing_config_t::phaseSeg1

21.2.3.3.0.49.4 uint8_t flexcan_timing_config_t::phaseSeg2

21.2.3.3.0.49.5 uint8_t flexcan_timing_config_t::propSeg

21.2.3.4 struct flexcan_rx_mb_config_t

This structure is used as the parameter of FLEXCAN_SetRxMbConfig() function. The FLEXCAN_SetRxMbConfig() function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Data Fields

- uint32_t id
  CAN Message Buffer Frame Identifier, should be set using FLEXCAN_ID_EXT() or FLEXCAN_ID_STD() macro.
- flexcan_frame_format_t format
  CAN Frame Identifier format(Standard of Extend).
- flexcan_frame_type_t type
  CAN Frame Type(Data or Remote).

21.2.3.4.0.50 Field Documentation

21.2.3.4.0.50.1 uint32_t flexcan_rx_mb_config_t::id

21.2.3.4.0.50.2 flexcan_frame_format_t flexcan_rx_mb_config_t::format

21.2.3.4.0.50.3 flexcan_frame_type_t flexcan_rx_mb_config_t::type

21.2.3.5 struct flexcan_rx_fifo_config_t

Data Fields

- uint32_t * idFilterTable
  Pointer to FlexCAN Rx FIFO identifier filter table.
- uint8_t idFilterNum
  The quantity of filter elements.
- flexcan_rx_fifo_filter_type_t idFilterType
  The FlexCAN Rx FIFO Filter type.
- flexcan_rx_fifo_priority_t priority
  The FlexCAN Rx FIFO receive priority.

21.2.3.5.0.51 Field Documentation

21.2.3.5.0.51.1 uint32_t* flexcan_rx_fifo_config_t::idFilterTable

21.2.3.5.0.51.2 uint8_t flexcan_rx_fifo_config_t::idFilterNum

21.2.3.5.0.51.3 flexcan_rx_fifo_filter_type_t flexcan_rx_fifo_config_t::idFilterType

21.2.3.5.0.51.4 flexcan_rx_fifo_priority_t flexcan_rx_fifo_config_t::priority

21.2.3.6 struct flexcan_mb_transfer_t

Data Fields

- flexcan_frame_t * frame
  The buffer of CAN Message to be transfer.
- uint8_t mbIdx
  The index of Message buffer used to transfer Message.

21.2.3.6.0.52 Field Documentation

21.2.3.6.0.52.1 flexcan_frame_t* flexcan_mb_transfer_t::frame

21.2.3.6.0.52.2 uint8_t flexcan_mb_transfer_t::mbIdx

21.2.3.7 struct flexcan_fifo_transfer_t

Data Fields

- flexcan_frame_t * frame
  The buffer of CAN Message to be received from Rx FIFO.

21.2.3.7.0.53 Field Documentation

21.2.3.7.0.53.1 flexcan_frame_t* flexcan_fifo_transfer_t::frame

21.2.3.8 struct _flexcan_handle

FlexCAN handle structure definition.

Data Fields

- flexcan_transfer_callback_t callback
  Callback function.
- void * userData
  FlexCAN callback function parameter.
- flexcan_frame_t *volatile mbFrameBuf [CAN_WORD1_COUNT]
  The buffer for received data from Message Buffers.
- flexcan_frame_t *volatile rxFifoFrameBuf
  The buffer for received data from Rx FIFO.

- volatile uint8_t mbState [CAN_WORD1_COUNT]
  Message Buffer transfer state.
- volatile uint8_t rxFifoState
  Rx FIFO transfer state.

21.2.3.8.0.54 Field Documentation

21.2.3.8.0.54.1 flexcan_transfer_callback_t flexcan_handle_t::callback

21.2.3.8.0.54.2 void* flexcan_handle_t::userData

21.2.3.8.0.54.3 flexcan_frame_t* volatile flexcan_handle_t::mbFrameBuf[CAN_WORD1_COUNT]

21.2.3.8.0.54.4 flexcan_frame_t* volatile flexcan_handle_t::rxFifoFrameBuf

21.2.3.8.0.54.5 volatile uint8_t flexcan_handle_t::mbState[CAN_WORD1_COUNT]

21.2.3.8.0.54.6 volatile uint8_t flexcan_handle_t::rxFifoState

21.2.4 Macro Definition Documentation

21.2.4.1 #define FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

21.2.4.2 #define FLEXCAN_ID_STD(*id*) (((uint32_t)(((uint32_t)(id)) << CAN_ID_STD_SHIFT)) & CAN_ID_STD_MASK)

Standard Frame ID helper macro.

21.2.4.3 #define FLEXCAN_ID_EXT(*id*)

Value:

```
(((uint32_t)(((uint32_t)(id)) << CAN_ID_EXT_SHIFT)) & \
     (CAN_ID_EXT_MASK | CAN_ID_STD_MASK))
```

21.2.4.4 #define FLEXCAN_RX_MB_STD_MASK(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     FLEXCAN_ID_STD(id))
```

Standard Rx Message Buffer Mask helper macro.

21.2.4.5 #define FLEXCAN_RX_MB_EXT_MASK(id, rtr, ide)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     FLEXCAN_ID_EXT(id))
```

21.2.4.6 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_STD(id) << 1))
```

Standard Rx FIFO Mask helper macro Type A helper macro.

21.2.4.7 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(id, rtr, ide)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_STD(id) << 16))
```

21.2.4.8 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(id, rtr, ide)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
     FLEXCAN_ID_STD(id))
```

21.2.4.9 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 21)

21.2.4.10 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 13)

\

21.2.4.11 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) << 5)

21.2.4.12 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(*id*) ((FLEXCAN_ID_STD(id) & 0x7F8) >> 3)

21.2.4.13 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
     (FLEXCAN_ID_EXT(id) << 1))
```

21.2.4.14 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
(                                                                    \
        ((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
        ((FLEXCAN_ID_EXT(id) & 0x1FFF8000)                             \
         << 1))
```

21.2.4.15 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
(((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
     ((FLEXCAN_ID_EXT(id) & 0x1FFF8000) >>                             \
      15))
```

21.2.4.16 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(*id*) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) << 3)

21.2.4.17 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >>           \
     5)
```

21.2.4.18 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >>           \
     13)
```

21.2.4.19 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(*id*) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 21)

21.2.4.20 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(*id, rtr, ide*) FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(id, rtr, ide)

Standard Rx FIFO Filter helper macro Type A helper macro.

21.2.4.21 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(                   \
        id, rtr, ide)
```

21.2.4.22 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(                    \
        id, rtr, ide)
```

21.2.4.23 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(           \
        id)
```

21.2.4.24 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(           \
        id)
```

21.2.4.25 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(          \
        id)
```

21.2.4.26 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(*id*) FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)

\

21.2.4.27 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(*id, rtr, ide*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)

21.2.4.28 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(                  \
        id, rtr, ide)
```

21.2.4.29 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(*id, rtr, ide*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(                   \
        id, rtr, ide)
```

21.2.4.30 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(*id*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)

\

21.2.4.31 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(          \
        id)
```

21.2.4.32 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(*id*)

Value:

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(          \
        id)
```

21.2.4.33 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW(*id*) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)

21.2.5 Typedef Documentation

21.2.5.1 typedef void(∗ flexcan_transfer_callback_t)(CAN_Type ∗base, flexcan_handle_t ∗handle, status_t status, uint32_t result, void ∗userData)

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_FLEXCAN_ErrorStatus, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

21.2.6 Enumeration Type Documentation

21.2.6.1 enum _flexcan_status

Enumerator

kStatus_FLEXCAN_TxBusy Tx Message Buffer is Busy.
kStatus_FLEXCAN_TxIdle Tx Message Buffer is Idle.
kStatus_FLEXCAN_TxSwitchToRx Remote Message is send out and Message buffer changed to Receive one.
kStatus_FLEXCAN_RxBusy Rx Message Buffer is Busy.
kStatus_FLEXCAN_RxIdle Rx Message Buffer is Idle.
kStatus_FLEXCAN_RxOverflow Rx Message Buffer is Overflowed.
kStatus_FLEXCAN_RxFifoBusy Rx Message FIFO is Busy.
kStatus_FLEXCAN_RxFifoIdle Rx Message FIFO is Idle.
kStatus_FLEXCAN_RxFifoOverflow Rx Message FIFO is overflowed.
kStatus_FLEXCAN_RxFifoWarning Rx Message FIFO is almost overflowed.
kStatus_FLEXCAN_ErrorStatus FlexCAN Module Error and Status.
kStatus_FLEXCAN_UnHandled UnHadled Interrupt asserted.

21.2.6.2 enum flexcan_frame_format_t

Enumerator

kFLEXCAN_FrameFormatStandard Standard frame format attribute.
kFLEXCAN_FrameFormatExtend Extend frame format attribute.

21.2.6.3 enum flexcan_frame_type_t

Enumerator

kFLEXCAN_FrameTypeData Data frame type attribute.
kFLEXCAN_FrameTypeRemote Remote frame type attribute.

21.2.6.4 enum flexcan_clock_source_t

Enumerator

kFLEXCAN_ClkSrcOsc FlexCAN Protocol Engine clock from Oscillator.
kFLEXCAN_ClkSrcPeri FlexCAN Protocol Engine clock from Peripheral Clock.

21.2.6.5 enum flexcan_rx_fifo_filter_type_t

Enumerator

kFLEXCAN_RxFifoFilterTypeA One full ID (standard and extended) per ID Filter element.
kFLEXCAN_RxFifoFilterTypeB Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.
kFLEXCAN_RxFifoFilterTypeC Four partial 8-bit Standard or extended ID slices per ID Filter Table element.
kFLEXCAN_RxFifoFilterTypeD All frames rejected.

21.2.6.6 enum flexcan_rx_fifo_priority_t

The matching process starts from the Rx MB(or Rx FIFO) with higher priority. If no MB(or Rx FIFO filter) is satisfied, the matching process goes on with the Rx FIFO(or Rx MB) with lower priority.

Enumerator

kFLEXCAN_RxFifoPrioLow Matching process start from Rx Message Buffer first.
kFLEXCAN_RxFifoPrioHigh Matching process start from Rx FIFO first.

21.2.6.7 enum _flexcan_interrupt_enable

This structure contains the settings for all of the FlexCAN Module interrupt configurations. Note: FlexCAN Message Buffers and Rx FIFO have their own interrupts.

Enumerator

kFLEXCAN_BusOffInterruptEnable Bus Off interrupt.
kFLEXCAN_ErrorInterruptEnable Error interrupt.
kFLEXCAN_RxWarningInterruptEnable Rx Warning interrupt.
kFLEXCAN_TxWarningInterruptEnable Tx Warning interrupt.
kFLEXCAN_WakeUpInterruptEnable Wake Up interrupt.

21.2.6.8 enum _flexcan_flags

This provides constants for the FlexCAN status flags for use in the FlexCAN functions. Note: The CPU read action clears FlEXCAN_ErrorFlag, therefore user need to read FlEXCAN_ErrorFlag and distinguish which error is occur using _flexcan_error_flags enumerations.

Enumerator

kFLEXCAN_SynchFlag CAN Synchronization Status.
kFLEXCAN_TxWarningIntFlag Tx Warning Interrupt Flag.
kFLEXCAN_RxWarningIntFlag Rx Warning Interrupt Flag.
kFLEXCAN_TxErrorWarningFlag Tx Error Warning Status.
kFLEXCAN_RxErrorWarningFlag Rx Error Warning Status.
kFLEXCAN_IdleFlag CAN IDLE Status Flag.
kFLEXCAN_FaultConfinementFlag Fault Confinement State Flag.
kFLEXCAN_TransmittingFlag FlexCAN In Transmission Status.
kFLEXCAN_ReceivingFlag FlexCAN In Reception Status.
kFLEXCAN_BusOffIntFlag Bus Off Interrupt Flag.
kFLEXCAN_ErrorIntFlag Error Interrupt Flag.
kFLEXCAN_WakeUpIntFlag Wake-Up Interrupt Flag.
kFLEXCAN_ErrorFlag All FlexCAN Error Status.

21.2.6.9 enum _flexcan_error_flags

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN_ErrorFlag in _flexcan_flags enumerations to ditermine which error is generated.

Enumerator

kFLEXCAN_StuffingError Stuffing Error.

kFLEXCAN_FormError Form Error.
kFLEXCAN_CrcError Cyclic Redundancy Check Error.
kFLEXCAN_AckError Received no ACK on transmission.
kFLEXCAN_Bit0Error Unable to send dominant bit.
kFLEXCAN_Bit1Error Unable to send recessive bit.

21.2.6.10 enum _flexcan_rx_fifo_flags

The FlexCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Enumerator

kFLEXCAN_RxFifoOverflowFlag Rx FIFO overflow flag.
kFLEXCAN_RxFifoWarningFlag Rx FIFO almost full flag.
kFLEXCAN_RxFifoFrameAvlFlag Frames available in Rx FIFO flag.

21.2.7 Function Documentation

21.2.7.1 void FLEXCAN_Init (CAN_Type * *base*, const flexcan_config_t * *config*, uint32_t *sourceClock_Hz*)

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the flexcan_config_t parameters and how to call the FLEXCAN_Init function by passing in these parameters:

```
flexcan_config_t flexcanConfig;
flexcanConfig.clkSrc            = KFLEXCAN_ClkSrcOsc;
flexcanConfig.baudRate          = 125000U;
flexcanConfig.maxMbNum          = 16;
flexcanConfig.enableLoopBack    = false;
flexcanConfig.enableSelfWakeup  = false;
flexcanConfig.enableIndividMask = false;
flexcanConfig.enableDoze        = false;
FLEXCAN_Init(CAN0, &flexcanConfig, 8000000UL);
```

Parameters

base	FlexCAN peripheral base address.

config	Pointer to user-defined configuration structure.
sourceClock_-Hz	FlexCAN Protocol Engine clock source frequency in Hz.

21.2.7.2 void FLEXCAN_Deinit (CAN_Type * *base*)

This function disable the FlexCAN module clock and set all register value to reset value.

Parameters

base	FlexCAN peripheral base address.

21.2.7.3 void FLEXCAN_GetDefaultConfig (flexcan_config_t * *config*)

This function initializes the FlexCAN configure structure to default value. The default value are: flexcanConfig->clkSrc = KFLEXCAN_ClkSrcOsc; flexcanConfig->baudRate = 125000U; flexcanConfig->maxMbNum = 16; flexcanConfig->enableLoopBack = false; flexcanConfig->enableSelfWakeup = false; flexcanConfig->enableIndividMask = false; flexcanConfig->enableDoze = false;

Parameters

config	Pointer to FlexCAN configuration structure.

21.2.7.4 void FLEXCAN_SetTimingConfig (CAN_Type * *base*, const flexcan_timing_config_t * *config*)

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the FLEXCAN_Init() and fill the baud rate field with a desired value. This provides the default timing characteristics to the module.

Note that calling FLEXCAN_SetTimingConfig() overrides the baud rate set in FLEXCAN_Init().

Parameters

base	FlexCAN peripheral base address.
config	Pointer to the timing configuration structure.

21.2.7.5 void FlEXCAN_SetRxMbGlobalMask (CAN_Type * *base*, uint32_t *mask*)

This function sets the global mask for FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the FLEXCAN_Init().

Parameters

base	FlexCAN peripheral base address.
mask	Rx Message Buffer Global Mask value.

21.2.7.6 void FlEXCAN_SetRxFifoGlobalMask (CAN_Type * *base*, uint32_t *mask*)

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

base	FlexCAN peripheral base address.
mask	Rx Fifo Global Mask value.

21.2.7.7 void FlEXCAN_SetRxIndividualMask (CAN_Type * *base*, uint8_t *maskIdx*, uint32_t *mask*)

This function sets the individual mask for FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in FLEXCAN_Init(). If Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with same index. What calls for special attention is that only the first 32 individual masks can be used as Rx FIFO filter mask.

Parameters

base	FlexCAN peripheral base address.
maskIdx	The Index of individual Mask.
mask	Rx Individual Mask value.

21.2.7.8 void FLEXCAN_SetTxMbConfig (CAN_Type * *base*, uint8_t *mbIdx*, bool *enable*)

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

base	FlexCAN peripheral base address.
mbIdx	The Message Buffer index.
enable	Enable/Disable Tx Message Buffer. • true: Enable Tx Message Buffer. • false: Disable Tx Message Buffer.

21.2.7.9 void FLEXCAN_SetRxMbConfig (CAN_Type * *base*, uint8_t *mbIdx*, const flexcan_rx_mb_config_t * *config*, bool *enable*)

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

base	FlexCAN peripheral base address.
mbIdx	The Message Buffer index.
config	Pointer to FlexCAN Message Buffer configuration structure.
enable	Enable/Disable Rx Message Buffer. • true: Enable Rx Message Buffer. • false: Disable Rx Message Buffer.

21.2.7.10 void FlEXCAN_SetRxFifoConfig (CAN_Type * *base*, const flexcan_rx_fifo_config_t * *config*, bool *enable*)

This function configures the Rx FIFO with given Rx FIFO configuration.

Parameters

base	FlexCAN peripheral base address.
config	Pointer to FlexCAN Rx FIFO configuration structure.
enable	Enable/Disable Rx FIFO. • true: Enable Rx FIFO. • false: Disable Rx FIFO.

21.2.7.11 static uint32_t FLEXCAN_GetStatusFlags (CAN_Type * *base*) [inline], [static]

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators _flexcan_flags. To check the specific status, compare the return value with enumerators in _flexcan_flags.

Parameters

base	FlexCAN peripheral base address.

Returns

FlexCAN status flags which are ORed by the enumerators in the _flexcan_flags.

21.2.7.12 static void FLEXCAN_ClearStatusFlags (CAN_Type * *base,* uint32_t *mask*) [inline], [static]

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

base	FlexCAN peripheral base address.
mask	The status flags to be cleared, it is logical OR value of _flexcan_flags.

21.2.7.13 static void FlEXCAN_GetBusErrCount (CAN_Type * *base,* uint8_t * *txErrBuf,* uint8_t * *rxErrBuf*) [inline], [static]

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

base	FlexCAN peripheral base address.
txErrBuf	Buffer to store Tx Error Counter value.
rxErrBuf	Buffer to store Rx Error Counter value.

21.2.7.14 static uint32_t FLEXCAN_GetMbStatusFlags (CAN_Type * *base,* uint32_t *mask*) [inline], [static]

This function gets the interrupt flags of a given Message Buffers.

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

21.2.7.15 static void FLEXCAN_ClearMbStatusFlags (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears the interrupt flags of a given Message Buffers.

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

21.2.7.16 static void FLEXCAN_EnableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the FlexCAN interrupts according to provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

base	FlexCAN peripheral base address.
mask	The interrupts to enable. Logical OR of _flexcan_interrupt_enable.

21.2.7.17 static void FLEXCAN_DisableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the FlexCAN interrupts according to provided mask. The mask is a logical OR of enumeration members, see _flexcan_interrupt_enable.

Parameters

base	FlexCAN peripheral base address.
mask	The interrupts to disable. Logical OR of _flexcan_interrupt_enable.

21.2.7.18 static void FLEXCAN_EnableMbInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the interrupts of given Message Buffers

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

21.2.7.19 static void FLEXCAN_DisableMbInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the interrupts of given Message Buffers

Parameters

base	FlexCAN peripheral base address.
mask	The ORed FlexCAN Message Buffer mask.

21.2.7.20 static void FLEXCAN_Enable (CAN_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the FlexCAN module.

Parameters

base	FlexCAN base pointer.
enable	true to enable, false to disable.

21.2.7.21 status_t FLEXCAN_WriteTxMb (CAN_Type * *base*, uint8_t *mbIdx*, const flexcan_frame_t * *txFrame*)

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

Parameters

base	FlexCAN peripheral base address.
mbIdx	The FlexCAN Message Buffer index.
txFrame	Pointer to CAN message frame to be sent.

Return values

kStatus_Success	- Write Tx Message Buffer Successfully.
kStatus_Fail	- Tx Message Buffer is currently in use.

21.2.7.22 status_t FLEXCAN_ReadRxMb (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

base	FlexCAN peripheral base address.
mbIdx	The FlexCAN Message Buffer index.
rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Rx Message Buffer is full and has been read successfully.
kStatus_FLEXCAN_RxOverflow	- Rx Message Buffer is already overflowed and has been read successfully.
kStatus_Fail	- Rx Message Buffer is empty.

21.2.7.23 status_t FlEXCAN_ReadRxFifo (CAN_Type * *base*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message from the FlexCAN build-in Rx FIFO.

Parameters

base	FlexCAN peripheral base address.
rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Read Message from Rx FIFO successfully.
kStatus_Fail	- Rx FIFO is not enabled.

21.2.7.24 status_t FlEXCAN_TransferSendBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *txFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
mbIdx	The FlexCAN Message Buffer index.
txFrame	Pointer to CAN message frame to be sent.

Return values

kStatus_Success	- Write Tx Message Buffer Successfully.
kStatus_Fail	- Tx Message Buffer is currently in use.

21.2.7.25 status_t FlEXCAN_TransferReceiveBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
mbIdx	The FlexCAN Message Buffer index.

rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Rx Message Buffer is full and has been read successfully.
kStatus_FLEXCAN_RxOverflow	- Rx Message Buffer is already overflowed and has been read successfully.
kStatus_Fail	- Rx Message Buffer is empty.

21.2.7.26 status_t FlEXCAN_TransferReceiveFifoBlocking (CAN_Type * *base,* flexcan_frame_t * *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	FlexCAN peripheral base pointer.
rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Read Message from Rx FIFO successfully.
kStatus_Fail	- Rx FIFO is not enabled.

21.2.7.27 void FLEXCAN_TransferCreateHandle (CAN_Type * *base,* flexcan_handle_t * *handle,* flexcan_transfer_callback_t *callback,* void * *userData*)

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

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

21.2.7.28 status_t FLEXCAN_TransferSendNonBlocking (CAN_Type * *base,* flexcan_handle_t * *handle,* flexcan_mb_transfer_t * *xfer*)

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

kStatus_Success	Start Tx Message Buffer sending process successfully.
kStatus_Fail	Write Tx Message Buffer failed.
kStatus_FLEXCAN_Tx-Busy	Tx Message Buffer is in use.

21.2.7.29 status_t FLEXCAN_TransferReceiveNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_mb_transfer_t * *xfer*)

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t.

Return values

kStatus_Success	- Start Rx Message Buffer receiving process successfully.
kStatus_FLEXCAN_Rx-Busy	- Rx Message Buffer is in use.

21.2.7.30 status_t FLEXCAN_TransferReceiveFifoNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
xfer	FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t.

Return values

kStatus_Success	- Start Rx FIFO receiving process successfully.
kStatus_FLEXCAN_RxFifoBusy	- Rx FIFO is currently in use.

21.2.7.31 void FLEXCAN_TransferAbortSend (CAN_Type * *base*, flexcan_handle_t * *handle*, uint8_t *mbIdx*)

This function aborts the interrupt driven message send process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
mbIdx	The FlexCAN Message Buffer index.

21.2.7.32 void FLEXCAN_TransferAbortReceive (CAN_Type * *base*, flexcan_handle_t * *handle*, uint8_t *mbIdx*)

This function aborts the interrupt driven message receive process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.
mbIdx	The FlexCAN Message Buffer index.

21.2.7.33 void FLEXCAN_TransferAbortReceiveFifo (CAN_Type * *base*, flexcan_handle_t * *handle*)

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.

21.2.7.34 void FLEXCAN_TransferHandleIRQ (CAN_Type * *base*, flexcan_handle_t * *handle*)

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

base	FlexCAN peripheral base address.
handle	FlexCAN handle pointer.

21.3 FlexCAN eDMA Driver

21.3.1 Overview

Files

- file fsl_flexcan_edma.h

Data Structures

- struct flexcan_edma_handle_t
  FlexCAN eDMA handle. More...

Typedefs

- typedef void(∗ flexcan_edma_transfer_callback_t)(CAN_Type ∗base, flexcan_edma_handle_t ∗handle, status_t status, void ∗userData)
  FlexCAN transfer callback function.

eDMA transactional

- void FLEXCAN_TransferCreateHandleEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle, flexcan_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗rxFifoEdmaHandle)
  Initializes the FlexCAN handle, which is used in transactional functions.
- status_t FLEXCAN_TransferReceiveFifoEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle, flexcan_fifo_transfer_t ∗xfer)
  Receives the CAN Message from the Rx FIFO using eDMA.
- void FLEXCAN_TransferAbortReceiveFifoEDMA (CAN_Type ∗base, flexcan_edma_handle_t ∗handle)
  Aborts the receive process which used eDMA.

21.3.2 Data Structure Documentation

21.3.2.1 struct _flexcan_edma_handle

Data Fields

- flexcan_edma_transfer_callback_t callback
  Callback function.
- void ∗ userData
  FlexCAN callback function parameter.
- edma_handle_t ∗ rxFifoEdmaHandle
  The EDMA Rx FIFO channel used.

- volatile uint8_t rxFifoState
  Rx FIFO transfer state.

21.3.2.1.0.55 Field Documentation

21.3.2.1.0.55.1 flexcan_edma_transfer_callback_t flexcan_edma_handle_t::callback

21.3.2.1.0.55.2 void* flexcan_edma_handle_t::userData

21.3.2.1.0.55.3 edma_handle_t* flexcan_edma_handle_t::rxFifoEdmaHandle

21.3.2.1.0.55.4 volatile uint8_t flexcan_edma_handle_t::rxFifoState

21.3.3 Typedef Documentation

21.3.3.1 typedef void(* flexcan_edma_transfer_callback_t)(CAN_Type *base, flexcan_edma_handle_t *handle, status_t status, void *userData)

21.3.4 Function Documentation

21.3.4.1 void FLEXCAN_TransferCreateHandleEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*, flexcan_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *rxFifoEdmaHandle*)

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.
callback	The callback function.
userData	The parameter of the callback function.
rxFifoEdma-Handle	User-requested DMA handle for Rx FIFO DMA transfer.

21.3.4.2 status_t FLEXCAN_TransferReceiveFifoEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.
xfer	FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXCAN_RxFifoBusy	Previous transfer ongoing.

21.3.4.3 void FLEXCAN_TransferAbortReceiveFifoEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*)

This function aborts the receive process which used eDMA.

Parameters

base	FlexCAN peripheral base address.
handle	Pointer to flexcan_edma_handle_t structure.

Chapter 22
FlexIO: FlexIO Driver

22.1 Overview

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

Modules

22.2 FlexIO Driver

22.2.1 Overview

Files

- file fsl_flexio.h

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 }
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 }
Define FlexIO shifter buffer type.

Driver version

- #define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  FlexIO driver version 2.0.0.

FlexIO Initialization and De-initialization

- void FLEXIO_GetDefaultConfig (flexio_config_t *userConfig)
  Gets the default configuration to configure FlexIO module.
- void FLEXIO_Init (FLEXIO_Type *base, const flexio_config_t *userConfig)
  Configures the FlexIO with 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.
- void FLEXIO_SetShifterConfig (FLEXIO_Type *base, uint8_t index, const flexio_shifter_config_t *shifterConfig)
  Configures the shifter with 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.

22.2.2 Data Structure Documentation

22.2.2.1 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.

22.2.2.1.0.56 Field Documentation

22.2.2.1.0.56.1 bool flexio_config_t::enableFastAccess

22.2.2.2 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.

22.2.2.2.0.57 Field Documentation

22.2.2.2.0.57.1 uint32_t flexio_timer_config_t::triggerSelect

22.2.2.2.0.57.2 flexio_timer_trigger_polarity_t flexio_timer_config_t::triggerPolarity

22.2.2.2.0.57.3 flexio_timer_trigger_source_t flexio_timer_config_t::triggerSource

22.2.2.2.0.57.4 flexio_pin_config_t flexio_timer_config_t::pinConfig

22.2.2.2.0.57.5 uint32_t flexio_timer_config_t::pinSelect

22.2.2.2.0.57.6 flexio_pin_polarity_t flexio_timer_config_t::pinPolarity

22.2.2.2.0.57.7 flexio_timer_mode_t flexio_timer_config_t::timerMode

22.2.2.2.0.57.8 flexio_timer_output_t flexio_timer_config_t::timerOutput

22.2.2.2.0.57.9 flexio_timer_decrement_source_t flexio_timer_config_t::timerDecrement

22.2.2.2.0.57.10 flexio_timer_reset_condition_t flexio_timer_config_t::timerReset

22.2.2.2.0.57.11 flexio_timer_disable_condition_t flexio_timer_config_t::timerDisable

22.2.2.2.0.57.12 flexio_timer_enable_condition_t flexio_timer_config_t::timerEnable

22.2.2.2.0.57.13 flexio_timer_stop_bit_condition_t flexio_timer_config_t::timerStop

22.2.2.2.0.57.14 flexio_timer_start_bit_condition_t flexio_timer_config_t::timerStart

22.2.2.2.0.57.15 uint32_t flexio_timer_config_t::timerCompare

22.2.2.3 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.
- 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.

22.2.2.3.0.58 Field Documentation

22.2.2.3.0.58.1 uint32_t flexio_shifter_config_t::timerSelect

22.2.2.3.0.58.2 flexio_shifter_timer_polarity_t flexio_shifter_config_t::timerPolarity

22.2.2.3.0.58.3 flexio_pin_config_t flexio_shifter_config_t::pinConfig

22.2.2.3.0.58.4 uint32_t flexio_shifter_config_t::pinSelect

22.2.2.3.0.58.5 flexio_pin_polarity_t flexio_shifter_config_t::pinPolarity

22.2.2.3.0.58.6 flexio_shifter_mode_t flexio_shifter_config_t::shifterMode

22.2.2.3.0.58.7 flexio_shifter_input_source_t flexio_shifter_config_t::inputSource

22.2.2.3.0.58.8 flexio_shifter_stop_bit_t flexio_shifter_config_t::shifterStop

22.2.2.3.0.58.9 flexio_shifter_start_bit_t flexio_shifter_config_t::shifterStart

22.2.3 Macro Definition Documentation

22.2.3.1 #define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

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

22.2.4 Typedef Documentation

22.2.4.1 typedef void(* flexio_isr_t)(void *base, void *handle)

22.2.5 Enumeration Type Documentation

22.2.5.1 enum flexio_timer_trigger_polarity_t

Enumerator

kFLEXIO_TimerTriggerPolarityActiveHigh Active high.
kFLEXIO_TimerTriggerPolarityActiveLow Active low.

22.2.5.2 enum flexio_timer_trigger_source_t

Enumerator

kFLEXIO_TimerTriggerSourceExternal External trigger selected.
kFLEXIO_TimerTriggerSourceInternal Internal trigger selected.

22.2.5.3 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.

22.2.5.4 enum flexio_pin_polarity_t

Enumerator

kFLEXIO_PinActiveHigh Active high.
kFLEXIO_PinActiveLow Active low.

22.2.5.5 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.

22.2.5.6 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.

22.2.5.7 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.

22.2.5.8 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.

22.2.5.9 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.

22.2.5.10 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.

22.2.5.11 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.

22.2.5.12 enum flexio_timer_start_bit_condition_t

Enumerator

kFLEXIO_TimerStartBitDisabled Start bit disabled.
kFLEXIO_TimerStartBitEnabled Start bit enabled.

22.2.5.13 enum flexio_shifter_timer_polarity_t

22.2.5.14 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.

22.2.5.15 enum flexio_shifter_input_source_t

Enumerator

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

22.2.5.16 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.

22.2.5.17 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.

22.2.5.18 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.

22.2.6 Function Documentation

22.2.6.1 void FLEXIO_GetDefaultConfig (flexio_config_t * *userConfig*)

The configuration can used directly for calling FLEXIO_Configure().

Example:

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

Parameters

userConfig	pointer to flexio_config_t structure

22.2.6.2 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_GetDefaultConfig().

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

22.2.6.3 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

22.2.6.4 void FLEXIO_Reset (FLEXIO_Type * *base*)

Parameters

base	FlexIO peripheral base address

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

Parameters

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

22.2.6.6 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

22.2.6.7 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 flexio_timer_config_t structure

22.2.6.8 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 could be calculated by (1 << shifter index)

Note

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

22.2.6.9 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 could be calculated by (1 << shifter index)

Note

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

22.2.6.10 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 could be calculated by (1 << shifter index)

Note

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

22.2.6.11 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 could be calculated by (1 << shifter index)

Note

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

22.2.6.12 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 could be calculated by (1 << timer index)

Note

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

22.2.6.13 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 could be calculated by (1 << timer index)

Note

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

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

Parameters

base	FlexIO peripheral base address

Returns

shifter status flags

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

Parameters

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

Note

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

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

Parameters

base	FlexIO peripheral base address

Returns

shifter error flags

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

Parameters

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

Note

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

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

Parameters

base	FlexIO peripheral base address

Returns

timer status flags

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

Parameters

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

Note

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

22.2.6.20 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 could be calculated by (1 << shifter index)

enable	True to enable, false to disable.

22.2.6.21 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

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

Parameters

base	pointer to FlexIO simulated peripheral type.
handle	pointer to 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.

22.2.6.23 status_t FLEXIO_UnregisterHandleIRQ (void * *base*)

Parameters

base	pointer to FlexIO simulated peripheral type.

Return values

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

22.3 FlexIO Camera Driver

22.3.1 Overview

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

22.3.2 Overview

FlexIO CAMERA driver includes 2 parts: functional APIs and EDMA transactional APIs. Functional APIs are feature/property target low level APIs. User can use functional APIs for FLEXIO CAMERA initialization/configuration/operation purpose. Using the functional API require user get knowledge of the FLEXIO CAMERA peripheral and know how to organize functional APIs to meet the requirement of 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 are transaction target high level APIs. User 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 can satisfy requirement. If the code size and performance are critical requirement, user can refer to the transactional API implementation and write their own code. All transactional APIs use the flexio_camera_edma_handle_t as the second parameter and user need to initialize the handle by calling FLEXIO_CAMERA_TransferCreateHandleEDMA() API.

EDMA transactional APIs support asynchronized receive. It means, the functions FLEXIO_CAMERA_TransferReceiveEDMA() setup interrupt for data receive, when the receive complete, upper layer is notified through callback function with status kStatus_FLEXIO_CAMERA_RxIdle.

22.3.3 Typical use case

22.3.3.1 FLEXIO CAMERA Receive in EDMA way

```
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

Files

- file fsl_flexio_camera.h

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.

Initialize and configuration

- void FLEXIO_CAMERA_Init (FLEXIO_CAMERA_Type *base, const flexio_camera_config_t *config)
  Ungates the FlexIO clock, reset the FlexIO module and do FlexIO CAMERA hardware configuration.
- void FLEXIO_CAMERA_Deinit (FLEXIO_CAMERA_Type *base)
  Disables the FlexIO CAMERA and gate the FlexIO clock.
- void FLEXIO_CAMERA_GetDefaultConfig (flexio_camera_config_t *config)
  Get the default configuration to configure 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.

22.3.4 Data Structure Documentation

22.3.4.1 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.

22.3.4.1.0.59 Field Documentation

22.3.4.1.0.59.1 FLEXIO_Type* FLEXIO_CAMERA_Type::flexioBase

22.3.4.1.0.59.2 uint32_t FLEXIO_CAMERA_Type::datPinStartIdx

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

22.3.4.1.0.59.3 uint32_t FLEXIO_CAMERA_Type::pclkPinIdx

22.3.4.1.0.59.4 uint32_t FLEXIO_CAMERA_Type::hrefPinIdx

22.3.4.1.0.59.5 uint32_t FLEXIO_CAMERA_Type::shifterStartIdx

22.3.4.1.0.59.6 uint32_t FLEXIO_CAMERA_Type::shifterCount

22.3.4.1.0.59.7 uint32_t FLEXIO_CAMERA_Type::timerIdx

22.3.4.2 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.

22.3.4.2.0.60 Field Documentation

22.3.4.2.0.60.1 bool flexio_camera_config_t::enablecamera

22.3.4.2.0.60.2 bool flexio_camera_config_t::enableFastAccess

22.3.4.3 struct flexio_camera_transfer_t

Data Fields

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

22.3.5 Macro Definition Documentation

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

22.3.5.2 #define FLEXIO_CAMERA_PARALLEL_DATA_WIDTH (8U)

22.3.6 Enumeration Type Documentation

22.3.6.1 enum _flexio_camera_status

Enumerator

kStatus_FLEXIO_CAMERA_RxBusy Receiver is busy.
kStatus_FLEXIO_CAMERA_RxIdle CAMERA receiver is idle.

22.3.6.2 enum _flexio_camera_status_flags

Enumerator

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

22.3.7 Function Documentation

22.3.7.1 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

22.3.7.2 void FLEXIO_CAMERA_Deinit (FLEXIO_CAMERA_Type * *base*)

Note

After calling this API, user need to call FLEXO_CAMERA_Init to use the FlexIO CAMERA module.

Parameters

base	pointer to FLEXIO_CAMERA_Type structure

22.3.7.3 void FLEXIO_CAMERA_GetDefaultConfig (flexio_camera_config_t * *config*)

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

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

Parameters

config	pointer to flexio_camera_config_t structure

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

Parameters

base	pointer to FLEXIO_CAMERA_Type
enable	True to enable, false to disable.

22.3.7.5 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

22.3.7.6 void FLEXIO_CAMERA_ClearStatusFlags (FLEXIO_CAMERA_Type * *base*, uint32_t *mask*)

Parameters

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

22.3.7.7 void FLEXIO_CAMERA_EnableInterrupt (FLEXIO_CAMERA_Type * *base*)

Parameters

base	pointer to the device.

22.3.7.8 void FLEXIO_CAMERA_DisableInterrupt (FLEXIO_CAMERA_Type * *base*)

Parameters

base	pointer to the device.

22.3.7.9 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 channel, one for transferring data from camera, such as 0V7670 to FlexIO buffer, another is for transferring data from FlexIO buffer to LCD.

22.3.7.10 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 would keep the data with count of base->shifterCount .

22.3.8 FlexIO DMA I2S Driver

22.3.8.1 Overview

Files

- file fsl_flexio_i2s_dma.h

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.

22.3.8.2 Data Structure Documentation

22.3.8.2.1 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.

22.3.8.2.1.1 Field Documentation

22.3.8.2.1.1.1 flexio_i2s_transfer_t flexio_i2s_dma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]

22.3.8.2.1.1.2 volatile uint8_t flexio_i2s_dma_handle_t::queueUser

22.3.8.3 Function Documentation

22.3.8.3.1 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, user need only 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 shall be a static value allocated by users.

22.3.8.3.2 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, user need only 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 shall be a static value allocated by users.

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

22.3.8.3.4 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.

22.3.8.3.5 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_GetReceiveRemainingBytes 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.

22.3.8.3.6 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.

22.3.8.3.7 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.

22.3.8.3.8 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.

22.3.8.3.9 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_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

22.3.9 FlexIO DMA SPI Driver

22.3.9.1 Overview

Files

- file fsl_flexio_spi_dma.h

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.

22.3.9.2 Data Structure Documentation

22.3.9.2.1 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.

22.3.9.2.1.1 Field Documentation

22.3.9.2.1.1.1 size_t flexio_spi_master_dma_handle_t::transferSize

22.3.9.3 Typedef Documentation

22.3.9.3.1 typedef flexio_spi_master_dma_handle_t flexio_spi_slave_dma_handle_t

22.3.9.4 Function Documentation

22.3.9.4.1 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, user need only 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.

22.3.9.4.2 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, users could call FLEXIO_SPI_MasterGetTransferCountDMA to poll the transfer status to check whether FlexIO SPI transfer 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.

22.3.9.4.3 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.

22.3.9.4.4 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.

22.3.9.4.5 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.

22.3.9.4.6 status_t FLEXIO_SPI_SlaveTransferDMA (FLEXIO_SPI_Type * *base*, flexio_spi_slave_dma_handle_t * *handle*, flexio_spi_transfer_t * *xfer*)

Note

This interface returned immediately after transfer initiates, users could call FLEXIO_SPI_SlaveGetTransferCountDMA to poll the transfer status to check whether FlexIO SPI transfer 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.

22.3.9.4.7 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.

22.3.9.4.8 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.

22.3.10 FlexIO DMA UART Driver

22.3.10.1 Overview

Files

- file fsl_flexio_uart_dma.h

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 still not 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 still not received.

22.3.10.2 Data Structure Documentation

22.3.10.2.1 struct _flexio_uart_dma_handle

Data Fields

- flexio_uart_dma_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- size_t txSize
  Total bytes to be sent.
- size_t rxSize
  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.

22.3.10.2.1.1 Field Documentation

22.3.10.2.1.1.1 flexio_uart_dma_transfer_callback_t flexio_uart_dma_handle_t::callback

22.3.10.2.1.1.2 void* flexio_uart_dma_handle_t::userData

22.3.10.2.1.1.3 size_t flexio_uart_dma_handle_t::txSize

22.3.10.2.1.1.4 size_t flexio_uart_dma_handle_t::rxSize

22.3.10.2.1.1.5 dma_handle_t* flexio_uart_dma_handle_t::txDmaHandle

22.3.10.2.1.1.6 dma_handle_t* flexio_uart_dma_handle_t::rxDmaHandle

22.3.10.2.1.1.7 volatile uint8_t flexio_uart_dma_handle_t::txState

22.3.10.3 Typedef Documentation

22.3.10.3.1 typedef void(* flexio_uart_dma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_dma_handle_t *handle, status_t status, void *userData)

22.3.10.4 Function Documentation

22.3.10.4.1 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.

22.3.10.4.2 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 return right away. When all data have been 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, refer to flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_TxBusy	Previous transfer on going.

22.3.10.4.3 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 have been 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 sturcture, refer to flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_RxBusy	Previous transfer on going.

22.3.10.4.4 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

22.3.10.4.5 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

22.3.10.4.6 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 still not 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.

22.3.10.4.7 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 still not 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.

22.3.11 FlexIO eDMA Camera Driver

22.3.11.1 Overview

Files

- file fsl_flexio_camera_edma.h

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.

22.3.11.2 Data Structure Documentation

22.3.11.2.1 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.
- volatile uint8_t rxState
  RX transfer state.

22.3.11.2.1.1 Field Documentation

22.3.11.2.1.1.1 flexio_camera_edma_transfer_callback_t flexio_camera_edma_handle_t::callback

22.3.11.2.1.1.2 void∗ flexio_camera_edma_handle_t::userData

22.3.11.2.1.1.3 size_t flexio_camera_edma_handle_t::rxSize

22.3.11.2.1.1.4 edma_handle_t∗ flexio_camera_edma_handle_t::rxEdmaHandle

22.3.11.3 Typedef Documentation

22.3.11.3.1 typedef void(∗ flexio_camera_edma_transfer_callback_t)(FLEXIO_CAMERA_Type ∗base, flexio_camera_edma_handle_t ∗handle, status_t status, void ∗userData)

22.3.11.4 Function Documentation

22.3.11.4.1 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 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.

22.3.11.4.2 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_RxBusy	Previous transfer on going.

22.3.11.4.3 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.

22.3.11.4.4 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.

22.3.12 FlexIO eDMA I2S Driver

22.3.12.1 Overview

Files

- file fsl_flexio_i2s_edma.h

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.

22.3.12.2 Data Structure Documentation

22.3.12.2.1 struct _flexio_i2s_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  DMA handler for FlexIO I2S send.
- uint8_t bytesPerFrame
  Bytes in a frame.
- 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.

22.3.12.2.1.1 Field Documentation

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

22.3.12.2.1.1.2 flexio_i2s_transfer_t flexio_i2s_edma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]

22.3.12.2.1.1.3 volatile uint8_t flexio_i2s_edma_handle_t::queueUser

22.3.12.3 Function Documentation

22.3.12.3.1 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, user need only 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 shall be a static value allocated by users.

22.3.12.3.2 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, user need only 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 shall be a static value allocated by users.

22.3.12.3.3 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 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.

22.3.12.3.4 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 to 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 eDMA send successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_TxBusy	FlexIO I2S is busy sending data.

22.3.12.3.5 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 to 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 eDMA receive successfully.
kStatus_InvalidArgument	The input arguments is invalid.
kStatus_RxBusy	FlexIO I2S is busy receiving data.

22.3.12.3.6 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.

22.3.12.3.7 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.

22.3.12.3.8 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.

22.3.12.3.9 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_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

22.3.13 FlexIO eDMA SPI Driver

22.3.13.1 Overview

Files

- file fsl_flexio_spi_edma.h

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 FLEXO 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.

22.3.13.2 Data Structure Documentation

22.3.13.2.1 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.
- 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.

22.3.13.2.1.1 Field Documentation

22.3.13.2.1.1.1 size_t flexio_spi_master_edma_handle_t::transferSize

22.3.13.3 Typedef Documentation

22.3.13.3.1 typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t

22.3.13.4 Function Documentation

22.3.13.4.1 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 FLEXO SPI master eDMA handle which can be used for other FLEXO SPI master transactional APIs. 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_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.

22.3.13.4.2 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_MasterGetTransferCountEDMA to poll the transfer status to check whether FlexIO SPI transfer 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.

22.3.13.4.3 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.

22.3.13.4.4 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.

22.3.13.4.5 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.

22.3.13.4.6 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_SlaveGetTransferCountEDMA to poll the transfer status to check whether FlexIO SPI transfer 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.

22.3.13.4.7 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.

22.3.13.4.8 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.

22.3.14 FlexIO eDMA UART Driver

22.3.14.1 Overview

Files

- file fsl_flexio_uart_edma.h

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 still not 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 still not received.

22.3.14.2 Data Structure Documentation

22.3.14.2.1 struct _flexio_uart_edma_handle

Data Fields

- flexio_uart_edma_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- size_t txSize
  Total bytes to be sent.
- size_t rxSize
  Total bytes to be received.
- edma_handle_t * txEdmaHandle
  The eDMA TX channel used.
- edma_handle_t * rxEdmaHandle
  The eDMA RX channel used.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

22.3.14.2.1.1 Field Documentation

22.3.14.2.1.1.1 flexio_uart_edma_transfer_callback_t flexio_uart_edma_handle_t::callback

22.3.14.2.1.1.2 void* flexio_uart_edma_handle_t::userData

22.3.14.2.1.1.3 size_t flexio_uart_edma_handle_t::txSize

22.3.14.2.1.1.4 size_t flexio_uart_edma_handle_t::rxSize

22.3.14.2.1.1.5 edma_handle_t* flexio_uart_edma_handle_t::txEdmaHandle

22.3.14.2.1.1.6 edma_handle_t* flexio_uart_edma_handle_t::rxEdmaHandle

22.3.14.2.1.1.7 volatile uint8_t flexio_uart_edma_handle_t::txState

22.3.14.3 Typedef Documentation

22.3.14.3.1 typedef void(* flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_edma_handle_t *handle, status_t status, void *userData)

22.3.14.4 Function Documentation

22.3.14.4.1 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.

22.3.14.4.2 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 have been sent out, the send callback function is called.

Parameters

base	pointer to FLEXIO_UART_Type
handle	UART handle pointer.
xfer	UART EDMA transfer structure, refer to flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_FLEXIO_UART-_TxBusy	Previous transfer on going.

22.3.14.4.3 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 have been 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, refer to flexio_uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.

22.3.14.4.4 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

22.3.14.4.5 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

22.3.14.4.6 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 still not 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.

22.3.14.4.7 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 still not received.

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.

22.4 FlexIO I2C Master Driver

22.4.1 Overview

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

22.4.2 Overview

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

Functional APIs are feature/property target low level APIs. Functional APIs can be used for the FlexIO I2C master initialization/configuration/operation for 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 are transaction 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 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.

22.4.3 Typical use case

22.4.3.1 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)
    {
    }

    // ...
}
```

Files

- file fsl_flexio_i2c_master.h

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, 0))
FlexIO I2C master driver version 2.1.0.

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_handle_t *handle, 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.

22.4.4 Data Structure Documentation

22.4.4.1 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.

22.4.4.1.0.1 Field Documentation

22.4.4.1.0.1.1 FLEXIO_Type* FLEXIO_I2C_Type::flexioBase

22.4.4.1.0.1.2 uint8_t FLEXIO_I2C_Type::SDAPinIndex

22.4.4.1.0.1.3 uint8_t FLEXIO_I2C_Type::SCLPinIndex

22.4.4.1.0.1.4 uint8_t FLEXIO_I2C_Type::shifterIndex[2]

22.4.4.1.0.1.5 uint8_t FLEXIO_I2C_Type::timerIndex[2]

22.4.4.2 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.

22.4.4.2.0.2 Field Documentation

22.4.4.2.0.2.1 bool flexio_i2c_master_config_t::enableMaster

22.4.4.2.0.2.2 bool flexio_i2c_master_config_t::enableInDoze

22.4.4.2.0.2.3 bool flexio_i2c_master_config_t::enableInDebug

22.4.4.2.0.2.4 bool flexio_i2c_master_config_t::enableFastAccess

22.4.4.2.0.2.5 uint32_t flexio_i2c_master_config_t::baudRate_Bps

22.4.4.3 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.

22.4.4.3.0.3 Field Documentation

22.4.4.3.0.3.1 uint32_t flexio_i2c_master_transfer_t::flags

22.4.4.3.0.3.2 uint8_t flexio_i2c_master_transfer_t::slaveAddress

22.4.4.3.0.3.3 flexio_i2c_direction_t flexio_i2c_master_transfer_t::direction

22.4.4.3.0.3.4 uint32_t flexio_i2c_master_transfer_t::subaddress

Transferred MSB first.

22.4.4.3.0.3.5 uint8_t flexio_i2c_master_transfer_t::subaddressSize

22.4.4.3.0.3.6 uint8_t volatile* flexio_i2c_master_transfer_t::data

22.4.4.3.0.3.7 volatile size_t flexio_i2c_master_transfer_t::dataSize

22.4.4.4 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.

22.4.4.4.0.4 Field Documentation

22.4.4.4.0.4.1 flexio_i2c_master_transfer_t flexio_i2c_master_handle_t::transfer

22.4.4.4.0.4.2 size_t flexio_i2c_master_handle_t::transferSize

22.4.4.4.0.4.3 uint8_t flexio_i2c_master_handle_t::state

22.4.4.4.0.4.4 flexio_i2c_master_transfer_callback_t flexio_i2c_master_handle_t::completionCallback

Callback function called at transfer event.

22.4.4.4.0.4.5 void* flexio_i2c_master_handle_t::userData

22.4.5 Macro Definition Documentation

22.4.5.1 #define FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

22.4.6 Typedef Documentation

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

22.4.7 Enumeration Type Documentation

22.4.7.1 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.

22.4.7.2 enum _flexio_i2c_master_interrupt

Enumerator

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

22.4.7.3 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.

22.4.7.4 enum flexio_i2c_direction_t

Enumerator

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

22.4.8 Function Documentation

22.4.8.1 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.

22.4.8.2 void FLEXIO_I2C_MasterDeinit (FLEXIO_I2C_Type * *base*)

Calling this API gates the FlexIO clock, so the FlexIO I2C master module can't work unless call FLEXIO_I2C_MasterInit.

Parameters

base	pointer to FLEXIO_I2C_Type structure.

22.4.8.3 void FLEXIO_I2C_MasterGetDefaultConfig (flexio_i2c_master_config_t * *masterConfig*)

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

Example:

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

Parameters

masterConfig	pointer to flexio_i2c_master_config_t structure.

22.4.8.4 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.

22.4.8.5 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 could get the related status.

22.4.8.6 void FLEXIO_I2C_MasterClearStatusFlags (FLEXIO_I2C_Type * *base*, uint32_t *mask*)

Parameters

base	pointer to FLEXIO_I2C_Type structure.
mask	status flag. The parameter could be any combination of the following values: • kFLEXIO_I2C_RxFullFlag • kFLEXIO_I2C_ReceiveNakFlag

22.4.8.7 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

22.4.8.8 void FLEXIO_I2C_MasterDisableInterrupts (FLEXIO_I2C_Type * *base*, uint32_t *mask*)

Parameters

base	pointer to FLEXIO_I2C_Type structure.
mask	interrupt source.

22.4.8.9 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

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

Note

This is API should be called when 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 not address transfer 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

22.4.8.11 void FLEXIO_I2C_MasterStop (FLEXIO_I2C_Type * *base*)

Parameters

base	pointer to FLEXIO_I2C_Type structure.

22.4.8.12 void FLEXIO_I2C_MasterRepeatedStart (FLEXIO_I2C_Type * *base*)

Parameters

base	pointer to FLEXIO_I2C_Type structure.

22.4.8.13 void FLEXIO_I2C_MasterAbortStop (FLEXIO_I2C_Type * *base*)

Parameters

base	pointer to FLEXIO_I2C_Type structure.

22.4.8.14 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.

22.4.8.15 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.

22.4.8.16 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 not data transfer 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.

22.4.8.17 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.

22.4.8.18 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.

22.4.8.19 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.

22.4.8.20 status_t FLEXIO_I2C_MasterTransferBlocking (FLEXIO_I2C_Type * *base*, flexio_i2c_master_handle_t * *handle*, 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.
handle	pointer to flexio_i2c_master_handle_t structure which stores the transfer state.
xfer	pointer to flexio_i2c_master_transfer_t structure.

Returns

status of status_t.

22.4.8.21 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.

22.4.8.22 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_MasterGetTransferCount 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.

22.4.8.23 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.

22.4.8.24 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

22.4.8.25 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

22.5 FlexIO I2S Driver

22.5.1 Overview

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

22.5.2 Overview

The FlexIO I2S driver includes functional APIs and transactional APIs.

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

Transactional APIs are transaction target high level APIs. The 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 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.

22.5.3 Typical use case

22.5.3.1 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)
    {
    }

    // ...
}
```

22.5.3.2 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

Files

- file fsl_flexio_i2s.h

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, 0))

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 a piece of data using a blocking method.
- static void FLEXIO_I2S_WriteData (FLEXIO_I2S_Type *base, uint8_t bitWidth, uint32_t data)
  Writes a data into 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.

22.5.4 Data Structure Documentation

22.5.4.1 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.

22.5.4.2 struct flexio_i2s_config_t

Data Fields

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

22.5.4.3 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.

22.5.4.4 struct flexio_i2s_transfer_t

Data Fields

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

22.5.4.4.0.5 Field Documentation

22.5.4.4.0.5.1 size_t flexio_i2s_transfer_t::dataSize

22.5.4.5 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.

22.5.5 Macro Definition Documentation

22.5.5.1 #define FSL_FLEXIO_I2S_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

22.5.5.2 #define FLEXIO_I2S_XFER_QUEUE_SIZE (4)

22.5.6 Enumeration Type Documentation

22.5.6.1 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.

22.5.6.2 enum flexio_i2s_master_slave_t

Enumerator

kFLEXIO_I2S_Master Master mode.
kFLEXIO_I2S_Slave Slave mode.

22.5.6.3 enum _flexio_i2s_interrupt_enable

Enumerator

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

22.5.6.4 enum _flexio_i2s_status_flags

Enumerator

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

22.5.6.5 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.

22.5.6.6 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.

22.5.7 Function Documentation

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

This API configures FlexIO pins and shifter to I2S and configure FlexIO I2S with 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, or any access to the FlexIO I2S module could cause hard fault because clock is not enabled.

Parameters

base	FlexIO I2S base pointer
config	FlexIO I2S configure structure.

22.5.7.2 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(). User 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

22.5.7.3 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

22.5.7.4 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.

22.5.7.5 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.

22.5.7.6 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

22.5.7.7 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

22.5.7.8 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.

22.5.7.9 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.

22.5.7.10 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.

22.5.7.11 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.

22.5.7.12 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.

22.5.7.13 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.

22.5.7.14 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.

22.5.7.15 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.

22.5.7.16 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.

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

Parameters

base	FlexIO I2S base pointer

Returns

Data read from data register.

22.5.7.18 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.

22.5.7.19 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 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.
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.

22.5.7.20 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. Usually, user only need to 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.

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

Note

Calling 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_TxBusy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	The input parameter is invalid.

22.5.7.22 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.

22.5.7.23 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

22.5.7.24 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

22.5.7.25 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.

22.5.7.26 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_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

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

Parameters

i2sBase	pointer to FLEXIO_I2S_Type structure.
i2sHandle	pointer to flexio_i2s_handle_t structure

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

Parameters

i2sBase	pointer to FLEXIO_I2S_Type structure.
i2sHandle	pointer to flexio_i2s_handle_t structure

22.6 FlexIO SPI Driver

22.6.1 Overview

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

22.6.2 Overview

FlexIO SPI driver includes functional APIs and transactional APIs.

Functional APIs are feature/property 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 are transaction 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. Please notice that FLEXIO SPI slave driver only support discontinuous PCS access, this is a limitation. The FLEXIO SPI slave driver could support continuous PCS, but the slave can't adapt discontinuous and continuous PCS automatically. User could change the timer disable mode in FLEXIO_SPI_SlaveInit manually, from kFLEXIO_TimerDisableOnTimerCompare to kFLEXIO_TimerDisableNever to enable discontinuous PCS access, only support CPHA = 0.

22.6.3 Typical use case

22.6.3.1 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)
    {
    }

        // ...
}
```

22.6.3.2 FlexIO_SPI Send/Receive in DMA way

```
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

Files

- file fsl_flexio_spi.h

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 and 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.

22.6.4 Data Structure Documentation

22.6.4.1 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.

22.6.4.1.0.6 Field Documentation

22.6.4.1.0.6.1 FLEXIO_Type* FLEXIO_SPI_Type::flexioBase

22.6.4.1.0.6.2 uint8_t FLEXIO_SPI_Type::SDOPinIndex

22.6.4.1.0.6.3 uint8_t FLEXIO_SPI_Type::SDIPinIndex

22.6.4.1.0.6.4 uint8_t FLEXIO_SPI_Type::SCKPinIndex

22.6.4.1.0.6.5 uint8_t FLEXIO_SPI_Type::CSnPinIndex

22.6.4.1.0.6.6 uint8_t FLEXIO_SPI_Type::shifterIndex[2]

22.6.4.1.0.6.7 uint8_t FLEXIO_SPI_Type::timerIndex[2]

22.6.4.2 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.

22.6.4.2.0.7 Field Documentation

22.6.4.2.0.7.1 bool flexio_spi_master_config_t::enableMaster

22.6.4.2.0.7.2 bool flexio_spi_master_config_t::enableInDoze

22.6.4.2.0.7.3 bool flexio_spi_master_config_t::enableInDebug

22.6.4.2.0.7.4 bool flexio_spi_master_config_t::enableFastAccess

22.6.4.2.0.7.5 uint32_t flexio_spi_master_config_t::baudRate_Bps

22.6.4.2.0.7.6 flexio_spi_clock_phase_t flexio_spi_master_config_t::phase

22.6.4.2.0.7.7 flexio_spi_data_bitcount_mode_t flexio_spi_master_config_t::dataMode

22.6.4.3 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.

22.6.4.3.0.8 Field Documentation

22.6.4.3.0.8.1 bool flexio_spi_slave_config_t::enableSlave

22.6.4.3.0.8.2 bool flexio_spi_slave_config_t::enableInDoze

22.6.4.3.0.8.3 bool flexio_spi_slave_config_t::enableInDebug

22.6.4.3.0.8.4 bool flexio_spi_slave_config_t::enableFastAccess

22.6.4.3.0.8.5 flexio_spi_clock_phase_t flexio_spi_slave_config_t::phase

22.6.4.3.0.8.6 flexio_spi_data_bitcount_mode_t flexio_spi_slave_config_t::dataMode

22.6.4.4 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.

22.6.4.4.0.9 Field Documentation

22.6.4.4.0.9.1 uint8_t* flexio_spi_transfer_t::txData

22.6.4.4.0.9.2 uint8_t* flexio_spi_transfer_t::rxData

22.6.4.4.0.9.3 size_t flexio_spi_transfer_t::dataSize

22.6.4.4.0.9.4 uint8_t flexio_spi_transfer_t::flags

22.6.4.5 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.

22.6.4.5.0.10 Field Documentation

22.6.4.5.0.10.1 uint8_t* flexio_spi_master_handle_t::txData

22.6.4.5.0.10.2 uint8_t* flexio_spi_master_handle_t::rxData

22.6.4.5.0.10.3 size_t flexio_spi_master_handle_t::transferSize

22.6.4.5.0.10.4 volatile size_t flexio_spi_master_handle_t::txRemainingBytes

22.6.4.5.0.10.5 volatile size_t flexio_spi_master_handle_t::rxRemainingBytes

22.6.4.5.0.10.6 volatile uint32_t flexio_spi_master_handle_t::state

22.6.4.5.0.10.7 flexio_spi_shift_direction_t flexio_spi_master_handle_t::direction

22.6.4.5.0.10.8 flexio_spi_master_transfer_callback_t flexio_spi_master_handle_t::callback

22.6.4.5.0.10.9 void* flexio_spi_master_handle_t::userData

22.6.5 Macro Definition Documentation

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

22.6.5.2 #define FLEXIO_SPI_DUMMYDATA (0xFFFFU)

22.6.6 Typedef Documentation

22.6.6.1 typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t

22.6.7 Enumeration Type Documentation

22.6.7.1 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.

22.6.7.2 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.

22.6.7.3 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.

22.6.7.4 enum flexio_spi_data_bitcount_mode_t

Enumerator

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

22.6.7.5 enum _flexio_spi_interrupt_enable

Enumerator

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

22.6.7.6 enum _flexio_spi_status_flags

Enumerator

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

22.6.7.7 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.

22.6.7.8 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.

22.6.8 Function Documentation

22.6.8.1 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.

22.6.8.2 void FLEXIO_SPI_MasterDeinit (FLEXIO_SPI_Type * *base*)

Parameters

base	Pointer to the FLEXIO_SPI_Type.

22.6.8.3 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.

22.6.8.4 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.

22.6.8.5 void FLEXIO_SPI_SlaveDeinit (FLEXIO_SPI_Type * *base*)

Parameters

base	Pointer to the FLEXIO_SPI_Type.

22.6.8.6 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.

22.6.8.7 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

22.6.8.8 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

22.6.8.9 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

22.6.8.10 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

22.6.8.11 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.

22.6.8.12 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.

22.6.8.13 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.

22.6.8.14 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.

22.6.8.15 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.

22.6.8.16 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.

22.6.8.17 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.

22.6.8.18 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.

22.6.8.19 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.

22.6.8.20 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.

22.6.8.21 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.

22.6.8.22 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.

22.6.8.23 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.

22.6.8.24 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.

22.6.8.25 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.

22.6.8.26 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.

22.6.8.27 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.

22.6.8.28 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.

22.6.8.29 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.

22.6.8.30 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.

22.7 FlexIO UART Driver

22.7.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) function using the Flexible I/O.

22.7.2 Overview

FlexIO UART driver includes 2 parts: functional APIs and transactional APIs. Functional APIs are feature/property 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 are transaction 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_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.

22.7.3 Typical use case

22.7.3.1 FlexIO UART send/receive using a polling method

```
uint8_t ch;
FLEXIO_UART_Type uartDev;
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;

FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);

FLEXIO_UART_WriteBlocking(&uartDev, txbuff, sizeof(txbuff));

while(1)
{
    FLEXIO_UART_ReadBlocking(&uartDev, &ch, 1);
    FLEXIO_UART_WriteBlocking(&uartDev, &ch, 1);
}
```

22.7.3.2 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;

    FLEXIO_UART_Init(&uartDev, &user_config, 120000000U);
    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)
    {
    }

    // ...
}
```

22.7.3.3 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;

    FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
    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)
        {
        }
    }

    // ...
}
```

22.7.3.4 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;
    FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);

    /*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

Files

- file fsl_flexio_uart.h

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_UART, 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, 0))
  FlexIO UART driver version 2.1.0.

Initialization and deinitialization

- void 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_StopRingBuffer (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 remaining bytes not 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 remaining bytes not received.
- void FLEXIO_UART_TransferHandleIRQ (void *uartType, void *uartHandle)
  FlexIO UART IRQ handler function.

22.7.4 Data Structure Documentation

22.7.4.1 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.

22.7.4.1.0.11 Field Documentation

22.7.4.1.0.11.1 FLEXIO_Type* FLEXIO_UART_Type::flexioBase

22.7.4.1.0.11.2 uint8_t FLEXIO_UART_Type::TxPinIndex

22.7.4.1.0.11.3 uint8_t FLEXIO_UART_Type::RxPinIndex

22.7.4.1.0.11.4 uint8_t FLEXIO_UART_Type::shifterIndex[2]

22.7.4.1.0.11.5 uint8_t FLEXIO_UART_Type::timerIndex[2]

22.7.4.2 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

22.7.4.2.0.12 Field Documentation

22.7.4.2.0.12.1 bool flexio_uart_config_t::enableUart

22.7.4.2.0.12.2 bool flexio_uart_config_t::enableFastAccess

22.7.4.2.0.12.3 uint32_t flexio_uart_config_t::baudRate_Bps

22.7.4.3 struct flexio_uart_transfer_t

Data Fields

- uint8_t * data
  Transfer buffer.
- size_t dataSize
  Transfer size.

22.7.4.4 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 txSize
  Total bytes to be sent.
- size_t rxSize
  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.

22.7.4.4.0.13 Field Documentation

22.7.4.4.0.13.1 uint8_t* volatile flexio_uart_handle_t::txData

22.7.4.4.0.13.2 volatile size_t flexio_uart_handle_t::txDataSize

22.7.4.4.0.13.3 uint8_t* volatile flexio_uart_handle_t::rxData

22.7.4.4.0.13.4 volatile size_t flexio_uart_handle_t::rxDataSize

22.7.4.4.0.13.5 size_t flexio_uart_handle_t::txSize

22.7.4.4.0.13.6 size_t flexio_uart_handle_t::rxSize

22.7.4.4.0.13.7 uint8_t* flexio_uart_handle_t::rxRingBuffer

22.7.4.4.0.13.8 size_t flexio_uart_handle_t::rxRingBufferSize

22.7.4.4.0.13.9 volatile uint16_t flexio_uart_handle_t::rxRingBufferHead

22.7.4.4.0.13.10 volatile uint16_t flexio_uart_handle_t::rxRingBufferTail

22.7.4.4.0.13.11 flexio_uart_transfer_callback_t flexio_uart_handle_t::callback

22.7.4.4.0.13.12 void* flexio_uart_handle_t::userData

22.7.4.4.0.13.13 volatile uint8_t flexio_uart_handle_t::txState

22.7.5 Macro Definition Documentation

22.7.5.1 #define FSL_FLEXIO_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

22.7.6 Typedef Documentation

22.7.6.1 typedef void(* flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)

22.7.7 Enumeration Type Documentation

22.7.7.1 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.

22.7.7.2 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

22.7.7.3 enum _flexio_uart_interrupt_enable

Enumerator

kFLEXIO_UART_TxDataRegEmptyInterruptEnable Transmit buffer empty interrupt enable.
kFLEXIO_UART_RxDataRegFullInterruptEnable Receive buffer full interrupt enable.

22.7.7.4 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.

22.7.8 Function Documentation

22.7.8.1 void 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.

22.7.8.2 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

22.7.8.3 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.

22.7.8.4 uint32_t FLEXIO_UART_GetStatusFlags (FLEXIO_UART_Type * *base*)

Parameters

base	Pointer to the FLEXIO_UART_Type structure.

Returns

FlexIO UART status flags.

22.7.8.5 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

22.7.8.6 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.

22.7.8.7 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.

22.7.8.8 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.

22.7.8.9 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.

22.7.8.10 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.

22.7.8.11 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.

22.7.8.12 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.

22.7.8.13 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.

22.7.8.14 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.

22.7.8.15 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.

22.7.8.16 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.

22.7.8.17 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 user 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, user 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.

22.7.8.18 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 are already data received in the ring buffer, user can get the received data from the ring buffer directly.

Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

base	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.

22.7.8.19 void FLEXIO_UART_StopRingBuffer (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.

22.7.8.20 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 are written to 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.

22.7.8.21 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 know 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.

22.7.8.22 status_t FLEXIO_UART_TransferGetSendCount (FLEXIO_UART_Type * *base,* flexio_uart_handle_t * *handle,* size_t * *count*)

This function gets the number of remaining bytes not 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_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

22.7.8.23 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.

22.7.8.24 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.

22.7.8.25 status_t FLEXIO_UART_TransferGetReceiveCount (FLEXIO_UART_Type * *base*, flexio_uart_handle_t * *handle*, size_t * *count*)

This function gets the number of remaining bytes not 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_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

22.7.8.26 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.

Chapter 23
FTM: FlexTimer Driver

23.1 Overview

The KSDK provides a driver for the FlexTimer Module (FTM) of Kinetis devices.

23.2 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.

23.2.1 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 BDM mode.

23.2.2 PWM Operations

The function FTM_SetupPwm() sets up FTM channels for PWM output. The function can set 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.

23.2.3 Input capture operations

The function FTM_SetupInputCapture() sets up an FTM channel for 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.

23.2.4 Output compare operations

The function FTM_SetupOutputCompare() sets up an FTM channel for output compare. The user can specify the channel output on a successful comparison and a comparison value.

23.2.5 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.

23.2.6 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.

23.3 Register Update

Some of the FTM registers have buffers. The driver support 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)
```

23.4 Typical use case

23.4.1 PWM output

Output a PWM signal on 2 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];

    /* Configure ftm params with frequency 24kHZ
    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 see the change of LEDs brightness.
        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);
    }
}
```

Typical use case

Files

- file fsl_ftm.h

Data Structures

- struct ftm_chnl_pwm_signal_param_t
  Options to configure a FTM channel's PWM signal. More...
- struct ftm_dual_edge_capture_param_t
  FlexTimer dual edge capture parameters. More...
- struct ftm_phase_params_t
  FlexTimer quadrature decode phase parameters. More...
- struct ftm_fault_param_t
  Structure is used to hold the parameters to configure a FTM fault. More...
- struct ftm_config_t
  FTM configuration structure. More...

Enumerations

- enum ftm_chnl_t {
  kFTM_Chnl_0 = 0U,
  kFTM_Chnl_1,
  kFTM_Chnl_2,
  kFTM_Chnl_3,
  kFTM_Chnl_4,
  kFTM_Chnl_5,
  kFTM_Chnl_6,
  kFTM_Chnl_7 }
  List of FTM channels.
- enum ftm_fault_input_t {
  kFTM_Fault_0 = 0U,
  kFTM_Fault_1,
  kFTM_Fault_2,
  kFTM_Fault_3 }
  List of FTM faults.
- enum ftm_pwm_mode_t {
  kFTM_EdgeAlignedPwm = 0U,
  kFTM_CenterAlignedPwm,
  kFTM_CombinedPwm }
  FTM PWM operation modes.
- enum ftm_pwm_level_select_t {
  kFTM_NoPwmSignal = 0U,
  kFTM_LowTrue,
  kFTM_HighTrue }
  FTM PWM output pulse mode: high-true, low-true or no output.
- enum ftm_output_compare_mode_t {
  kFTM_NoOutputSignal = (1U << FTM_CnSC_MSA_SHIFT),
  kFTM_ToggleOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (1U << FTM_CnSC_ELSA_S-

HIFT)),
kFTM_ClearOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (2U << FTM_CnSC_ELSA_SHIFT)),
kFTM_SetOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (3U << FTM_CnSC_ELSA_SHIFT)) }
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.

Functions

- 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.
- 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, 0))
Version 2.0.0.

Typical use case

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.

23.5 Data Structure Documentation

23.5.1 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.

23.5.1.0.0.14 Field Documentation

23.5.1.0.0.14.1 ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber

In combined mode, this represents the channel pair number.

23.5.1.0.0.14.2 ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level

23.5.1.0.0.14.3 uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

23.5.1.0.0.14.4 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

23.5.2 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.

23.5.3 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.

23.5.4 struct ftm_fault_param_t

Data Fields

- bool enableFaultInput
  True: Fault input is enabled; false: Fault input is disabled.
- bool faultLevel
  True: Fault polarity is active low i.e '0' indicates a fault; False: Fault polarity is active high.
- bool useFaultFilter
  True: Use the filtered fault signal; False: Use the direct path from fault input.

23.5.5 struct ftm_config_t

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

The configuration structure can be made constant so as to reside in flash.

Data Fields

- ftm_clock_prescale_t prescale
  FTM clock prescale value.
- ftm_bdm_mode_t bdmMode
  FTM behavior in BDM mode.
- uint32_t pwmSyncMode
  Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration ftm_pwm_sync_method_t.
- uint32_t reloadPoints
  FTM reload points; When using this, the PWM synchronization is not required.
- ftm_fault_mode_t faultMode
  FTM fault control mode.
- uint8_t faultFilterValue
  Fault input filter value.
- ftm_deadtime_prescale_t deadTimePrescale
  The dead time prescalar value.
- uint8_t deadTimeValue
  The dead time value.
- uint32_t extTriggers
  External triggers to enable.
- uint8_t chnlInitState
  Defines the initialization value of the channels in OUTINT register.
- uint8_t chnlPolarity
  Defines the output polarity of the channels in POL register.
- bool useGlobalTimeBase
  True: Use of an external global time base is enabled; False: disabled.

23.5.5.0.0.15 Field Documentation

23.5.5.0.0.15.1 uint32_t ftm_config_t::pwmSyncMode

23.5.5.0.0.15.2 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.

23.5.5.0.0.15.3 uint32_t ftm_config_t::extTriggers

Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration ftm_external_trigger_t.

23.6 Enumeration Type Documentation

23.6.1 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.

23.6.2 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.

23.6.3 enum ftm_pwm_mode_t

Enumerator

kFTM_EdgeAlignedPwm Edge-aligned PWM.
kFTM_CenterAlignedPwm Center-aligned PWM.
kFTM_CombinedPwm Combined PWM.

23.6.4 enum ftm_pwm_level_select_t

Enumerator

kFTM_NoPwmSignal No PWM output on pin.
kFTM_LowTrue Low true pulses.
kFTM_HighTrue High true pulses.

23.6.5 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.

23.6.6 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.

23.6.7 enum ftm_dual_edge_capture_mode_t

Enumerator

kFTM_OneShot One-shot capture mode.
kFTM_Continuous Continuous capture mode.

23.6.8 enum ftm_quad_decode_mode_t

Enumerator

kFTM_QuadPhaseEncode Phase A and Phase B encoding mode.
kFTM_QuadCountAndDir Count and direction encoding mode.

23.6.9 enum ftm_phase_polarity_t

Enumerator

kFTM_QuadPhaseNormal Phase input signal is not inverted.
kFTM_QuadPhaseInvert Phase input signal is inverted.

23.6.10 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.

23.6.11 enum ftm_clock_source_t

Enumerator

kFTM_SystemClock System clock selected.
kFTM_FixedClock Fixed frequency clock.
kFTM_ExternalClock External clock.

23.6.12 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.

23.6.13 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.

23.6.14 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.

23.6.15 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.

23.6.16 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.

23.6.17 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.

23.6.18 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.

23.6.19 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.

23.7 Function Documentation

23.7.1 status_t FTM_Init (FTM_Type * *base*, const ftm_config_t * *config*)

Note

This API should be called at the beginning of the application using the FTM driver.

Parameters

base	FTM peripheral base address
config	Pointer to the user configuration structure.

Returns

kStatus_Success indicates success; Else indicates failure.

23.7.2 void FTM_Deinit (FTM_Type * *base*)

Parameters

base	FTM peripheral base address

23.7.3 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.

23.7.4 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

23.7.5 void FTM_UpdatePwmDutycycle (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)

Parameters

base	FTM peripheral base address
chnlNumber	The channel/channel pair number. In combined mode, this represents the channel pair number
currentPwmMode	The current PWM mode set during PWM setup
dutyCyclePercent	New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

23.7.6 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.

23.7.7 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.

23.7.8 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.

23.7.9 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.

23.7.10 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

23.7.11 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

23.7.12 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

23.7.13 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

23.7.14 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

23.7.15 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

23.7.16 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

23.7.17 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.

23.7.18 static void FTM_StopTimer (FTM_Type * *base*) [inline], [static]

Parameters

base	FTM peripheral base address

23.7.19 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

23.7.20 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

23.7.21 static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	FTM peripheral base address
enable	true to enable, false to disable

23.7.22 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

23.7.23 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

23.7.24 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

23.7.25 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

23.7.26 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

23.7.27 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

23.7.28 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 24
GPIO: General-Purpose Input/Output Driver

24.1 Overview

Modules

- FGPIO Driver
- GPIO Driver

Files

- file fsl_gpio.h

Data Structures

- struct gpio_pin_config_t
  The GPIO pin configuration structure. More...

Enumerations

- enum gpio_pin_direction_t {
  kGPIO_DigitalInput = 0U,
  kGPIO_DigitalOutput = 1U }
  GPIO direction definition.

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  GPIO driver version 2.1.0.

24.2 Data Structure Documentation

24.2.1 struct gpio_pin_config_t

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused Note : In some 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 default output logic, no use in input.

24.3 Macro Definition Documentation

24.3.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

24.4 Enumeration Type Documentation

24.4.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.
kGPIO_DigitalOutput Set current pin as digital output.

24.5 GPIO Driver

24.5.1 Overview

The KSDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of Kinetis devices.

24.5.2 Typical use case

24.5.2.1 Output Operation

```
/* Output pin configuration
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

24.5.2.2 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 one GPIO pin 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 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 specified GPIO pin.

GPIO Interrupt

- uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type *base)
  Reads whole GPIO port interrupt status flag.
- void GPIO_ClearPinsInterruptFlags (GPIO_Type *base, uint32_t mask)
  Clears multiple GPIO pins' interrupt status flag.

24.5.3 Function Documentation

24.5.3.1 void GPIO_PinInit (GPIO_Type * *base*, uint32_t *pin*, const gpio_pin_config_t * *config*)

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

This is an example to define an input pin or 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

24.5.3.2 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's number
output	GPIO pin output logic level. • 0: corresponding pin output low logic level. • 1: corresponding pin output high logic level.

24.5.3.3 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 pins' numbers macro

24.5.3.4 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 pins' numbers macro

24.5.3.5 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 pins' numbers macro

24.5.3.6 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's number

Return values

GPIO	port input value • 0: corresponding pin input low logic level. • 1: corresponding pin input high logic level.

24.5.3.7 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

Current	GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

24.5.3.8 void GPIO_ClearPinsInterruptFlags (GPIO_Type * *base,* uint32_t *mask*)

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pins' numbers macro

24.6 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.

24.6.1 Typical use case

24.6.1.1 Output Operation

```
/* Output pin configuration
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration
FGPIO_PinInit(FGPIO_LED, LED_PINNUM, &led_config);
```

24.6.1.2 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 25
I2C: Inter-Integrated Circuit Driver

25.1 Overview

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver
- I2C µCOS/II Driver
- I2C µCOS/III Driver

25.2 I2C Driver

25.2.1 Overview

The KSDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of Kinetis devices.

25.2.2 Overview

The I2C driver includes functional APIs and transactional APIs.

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

Transactional APIs are transaction 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 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.

25.2.3 Typical use case

25.2.3.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Get default configuration for master.
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init I2C master.
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Send start and slave address.
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address, kI2C_Write/kI2C_Read);

/* Wait address sent out.
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 error occours, send STOP.
    I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
    return result;
}

while(!(I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR) & kI2C_IntPendingFlag))
{

}

/* Wait all data sent out, send STOP.
I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
```

25.2.3.2 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;
    }
}

/* Get default configuration for master.
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init 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);

/*  Wait for transfer completed.
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

25.2.3.3 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;
    }
}

/* Get default configuration for master.
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init 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;
```

25.2.3.4 Slave Operation in functional method

```
i2c_slave_config_t slaveConfig;
uint8_t status;
status_t result = kStatus_Success;

I2C_SlaveGetDefaultConfig(&slaveConfig); /*default configuration 7-bit addressing
       mode
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/kI2C_RangeMatch;
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

/* Wait address match.
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag))
{

}

/* Slave transmit, master reading from slave.
if (status & kI2C_TransferDirectionFlag)
```

```
{
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}
else
{
    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}

return result;
```

25.2.3.5 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;

        /*  Receive request
        case kI2C_SlaveReceiveEvent:
            /*  Update information for received process
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /*  Transfer done
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*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);

/*  Wait for transfer completed.
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;
```

Files

- file fsl_i2c.h

Data Structures

- struct i2c_master_config_t
  I2C master user configuration. More...
- struct i2c_slave_config_t
  I2C slave user configuration. More...
- struct i2c_master_transfer_t
  I2C master transfer structure. More...
- struct i2c_master_handle_t
  I2C master handle structure. More...
- struct i2c_slave_transfer_t
  I2C slave transfer structure. More...
- struct i2c_slave_handle_t
  I2C slave handle structure. More...

Typedefs

- typedef void(∗ i2c_master_transfer_callback_t)(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t status, void ∗userData)
  I2C master transfer callback typedef.
- typedef void(∗ i2c_slave_transfer_callback_t)(I2C_Type ∗base, i2c_slave_transfer_t ∗xfer, void ∗userData)
  I2C slave transfer callback typedef.

Enumerations

- enum _i2c_status {
  kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_I2C, 0),
  kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_I2C, 1),
  kStatus_I2C_Nak = MAKE_STATUS(kStatusGroup_I2C, 2),
  kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_I2C, 3),
  kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_I2C, 4) }
  I2C status return codes.
- enum _i2c_flags {
  kI2C_ReceiveNakFlag = I2C_S_RXAK_MASK,
  kI2C_IntPendingFlag = I2C_S_IICIF_MASK,
  kI2C_TransferDirectionFlag = I2C_S_SRW_MASK,
  kI2C_RangeAddressMatchFlag = I2C_S_RAM_MASK,
  kI2C_ArbitrationLostFlag = I2C_S_ARBL_MASK,
  kI2C_BusBusyFlag = I2C_S_BUSY_MASK,
  kI2C_AddressMatchFlag = I2C_S_IAAS_MASK,

kI2C_TransferCompleteFlag = I2C_S_TCF_MASK }
I2C peripheral flags.
- enum _i2c_interrupt_enable { kI2C_GlobalInterruptEnable = I2C_C1_IICIE_MASK }
I2C feature interrupt source.
- enum i2c_direction_t {
kI2C_Write = 0x0U,
kI2C_Read = 0x1U }
Direction of master and slave transfers.
- enum i2c_slave_address_mode_t {
kI2C_Address7bit = 0x0U,
kI2C_RangeMatch = 0X2U }
Addressing mode.
- enum _i2c_master_transfer_flags {
kI2C_TransferDefaultFlag = 0x0U,
kI2C_TransferNoStartFlag = 0x1U,
kI2C_TransferRepeatedStartFlag = 0x2U,
kI2C_TransferNoStopFlag = 0x4U }
I2C transfer control flag.
- enum i2c_slave_transfer_event_t {
kI2C_SlaveAddressMatchEvent = 0x01U,
kI2C_SlaveTransmitEvent = 0x02U,
kI2C_SlaveReceiveEvent = 0x04U,
kI2C_SlaveTransmitAckEvent = 0x08U,
kI2C_SlaveCompletionEvent = 0x20U,
kI2C_SlaveAllEvents }
Set of events sent to the callback for nonblocking slave transfers.

Driver version

- #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
I2C driver version 2.0.0.

Initialization and deinitialization

- void I2C_MasterInit (I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
Initializes the I2C peripheral.
- void I2C_SlaveInit (I2C_Type *base, const i2c_slave_config_t *slaveConfig)
Initializes the I2C peripheral.
- void I2C_MasterDeinit (I2C_Type *base)
De-initializes the I2C master peripheral.
- void I2C_SlaveDeinit (I2C_Type *base)
De-initializes the I2C slave peripheral.
- void I2C_MasterGetDefaultConfig (i2c_master_config_t *masterConfig)
Sets the I2C master configuration structure to default values.
- void I2C_SlaveGetDefaultConfig (i2c_slave_config_t *slaveConfig)

Sets the I2C slave configuration structure to default values.

- static void I2C_Enable (I2C_Type *base, bool enable)
  Enables or disabless the I2C peripheral operation.

Status

- uint32_t I2C_MasterGetStatusFlags (I2C_Type *base)
  Gets the I2C status flags.
- static uint32_t I2C_SlaveGetStatusFlags (I2C_Type *base)
  Gets the I2C status flags.
- static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C status flag state.
- static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C status flag state.

Interrupts

- void I2C_EnableInterrupts (I2C_Type *base, uint32_t mask)
  Enables I2C interrupt requests.
- void I2C_DisableInterrupts (I2C_Type *base, uint32_t mask)
  Disables I2C interrupt requests.

DMA Control

- static uint32_t I2C_GetDataRegAddr (I2C_Type *base)
  Gets the I2C tx/rx data register address.

Bus Operations

- void I2C_MasterSetBaudRate (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the I2C master transfer baud rate.
- status_t I2C_MasterStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)
  Sends a START on the I2C bus.
- status_t I2C_MasterStop (I2C_Type *base)
  Sends a STOP signal on the I2C bus.
- status_t I2C_MasterRepeatedStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)
  Sends a REPEATED START on the I2C bus.
- status_t I2C_MasterWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
  Performs a polling send transaction on the I2C bus without a STOP signal.
- status_t I2C_MasterReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
  Performs a polling receive transaction on the I2C bus with a STOP signal.
- status_t I2C_SlaveWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
  Performs a polling send transaction on the I2C bus.
- void I2C_SlaveReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
  Performs a polling receive transaction on the I2C bus.
- status_t I2C_MasterTransferBlocking (I2C_Type *base, i2c_master_transfer_t *xfer)

Performs a master polling transfer on the I2C bus.

Transactional

- void I2C_MasterTransferCreateHandle (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_callback_t callback, void *userData)
  Initializes the I2C handle which is used in transactional functions.
- status_t I2C_MasterTransferNonBlocking (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_t *xfer)
  Performs a master interrupt non-blocking transfer on the I2C bus.
- status_t I2C_MasterTransferGetCount (I2C_Type *base, i2c_master_handle_t *handle, size_t *count)
  Gets the master transfer status during a interrupt non-blocking transfer.
- void I2C_MasterTransferAbort (I2C_Type *base, i2c_master_handle_t *handle)
  Aborts an interrupt non-blocking transfer early.
- void I2C_MasterTransferHandleIRQ (I2C_Type *base, void *i2cHandle)
  Master interrupt handler.
- void I2C_SlaveTransferCreateHandle (I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)
  Initializes the I2C handle which is used in transactional functions.
- status_t I2C_SlaveTransferNonBlocking (I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)
  Starts accepting slave transfers.
- void I2C_SlaveTransferAbort (I2C_Type *base, i2c_slave_handle_t *handle)
  Aborts the slave transfer.
- status_t I2C_SlaveTransferGetCount (I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
  Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.
- void I2C_SlaveTransferHandleIRQ (I2C_Type *base, void *i2cHandle)
  Slave interrupt handler.

25.2.4 Data Structure Documentation

25.2.4.1 struct i2c_master_config_t

Data Fields

- bool enableMaster
  Enables the I2C peripheral at initialization time.
- uint32_t baudRate_Bps
  Baud rate configuration of I2C peripheral.
- uint8_t glitchFilterWidth
  Controls the width of the glitch.

25.2.4.1.0.16 Field Documentation

25.2.4.1.0.16.1 bool i2c_master_config_t::enableMaster

25.2.4.1.0.16.2 uint32_t i2c_master_config_t::baudRate_Bps

25.2.4.1.0.16.3 uint8_t i2c_master_config_t::glitchFilterWidth

25.2.4.2 struct i2c_slave_config_t

Data Fields

- bool enableSlave
  Enables the I2C peripheral at initialization time.
- bool enableGeneralCall
  Enable general call addressing mode.
- bool enableWakeUp
  Enables/disables waking up MCU from low power mode.
- bool enableBaudRateCtl
  Enables/disables independent slave baud rate on SCL in very fast I2C modes.
- uint16_t slaveAddress
  Slave address configuration.
- uint16_t upperAddress
  Maximum boundary slave address used in range matching mode.
- i2c_slave_address_mode_t addressingMode
  Addressing mode configuration of i2c_slave_address_mode_config_t.

25.2.4.2.0.17 Field Documentation

25.2.4.2.0.17.1 bool i2c_slave_config_t::enableSlave

25.2.4.2.0.17.2 bool i2c_slave_config_t::enableGeneralCall

25.2.4.2.0.17.3 bool i2c_slave_config_t::enableWakeUp

25.2.4.2.0.17.4 bool i2c_slave_config_t::enableBaudRateCtl

25.2.4.2.0.17.5 uint16_t i2c_slave_config_t::slaveAddress

25.2.4.2.0.17.6 uint16_t i2c_slave_config_t::upperAddress

25.2.4.2.0.17.7 i2c_slave_address_mode_t i2c_slave_config_t::addressingMode

25.2.4.3 struct i2c_master_transfer_t

Data Fields

- uint32_t flags
  Transfer flag which controls the transfer.
- uint8_t slaveAddress
  7-bit slave address.

- 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.

25.2.4.3.0.18 Field Documentation

25.2.4.3.0.18.1 uint32_t i2c_master_transfer_t::flags

25.2.4.3.0.18.2 uint8_t i2c_master_transfer_t::slaveAddress

25.2.4.3.0.18.3 i2c_direction_t i2c_master_transfer_t::direction

25.2.4.3.0.18.4 uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

25.2.4.3.0.18.5 uint8_t i2c_master_transfer_t::subaddressSize

25.2.4.3.0.18.6 uint8_t* volatile i2c_master_transfer_t::data

25.2.4.3.0.18.7 volatile size_t i2c_master_transfer_t::dataSize

25.2.4.4 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
  Transfer state maintained during transfer.
- i2c_master_transfer_callback_t completionCallback
  Callback function called when transfer finished.
- void * userData
  Callback parameter passed to callback function.

25.2.4.4.0.19 Field Documentation

25.2.4.4.0.19.1 i2c_master_transfer_t i2c_master_handle_t::transfer

25.2.4.4.0.19.2 size_t i2c_master_handle_t::transferSize

25.2.4.4.0.19.3 uint8_t i2c_master_handle_t::state

25.2.4.4.0.19.4 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

25.2.4.4.0.19.5 void* i2c_master_handle_t::userData

25.2.4.5 struct i2c_slave_transfer_t

Data Fields

- i2c_slave_transfer_event_t event
  Reason the callback is being invoked.
- uint8_t *volatile data
  Transfer buffer.
- volatile size_t dataSize
  Transfer size.
- status_t completionStatus
  Success or error code describing how the transfer completed.
- size_t transferredCount
  Number of bytes actually transferred since start or last repeated start.

25.2.4.5.0.20 Field Documentation

25.2.4.5.0.20.1 i2c_slave_transfer_event_t i2c_slave_transfer_t::event

25.2.4.5.0.20.2 uint8_t* volatile i2c_slave_transfer_t::data

25.2.4.5.0.20.3 volatile size_t i2c_slave_transfer_t::dataSize

25.2.4.5.0.20.4 status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

25.2.4.5.0.20.5 size_t i2c_slave_transfer_t::transferredCount

25.2.4.6 struct _i2c_slave_handle

I2C slave handle typedef.

Data Fields

- bool isBusy
  Whether transfer is busy.
- i2c_slave_transfer_t transfer

I2C slave transfer copy.
- uint32_t eventMask
  Mask of enabled events.
- i2c_slave_transfer_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback.

25.2.4.6.0.21 Field Documentation

25.2.4.6.0.21.1 bool i2c_slave_handle_t::isBusy

25.2.4.6.0.21.2 i2c_slave_transfer_t i2c_slave_handle_t::transfer

25.2.4.6.0.21.3 uint32_t i2c_slave_handle_t::eventMask

25.2.4.6.0.21.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

25.2.4.6.0.21.5 void* i2c_slave_handle_t::userData

25.2.5 Macro Definition Documentation

25.2.5.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

25.2.6 Typedef Documentation

25.2.6.1 typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)

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

25.2.7 Enumeration Type Documentation

25.2.7.1 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.

25.2.7.2 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.

25.2.7.3 enum _i2c_interrupt_enable

Enumerator

kI2C_GlobalInterruptEnable I2C global interrupt.

25.2.7.4 enum i2c_direction_t

Enumerator

kI2C_Write Master transmit to slave.
kI2C_Read Master receive from slave.

25.2.7.5 enum i2c_slave_address_mode_t

Enumerator

kI2C_Address7bit 7-bit addressing mode.
kI2C_RangeMatch Range address match addressing mode.

25.2.7.6 enum _i2c_master_transfer_flags

Enumerator

kI2C_TransferDefaultFlag Transfer starts with a start signal, stops with a stop signal.
kI2C_TransferNoStartFlag Transfer starts without a start signal.
kI2C_TransferRepeatedStartFlag Transfer starts with a repeated start signal.
kI2C_TransferNoStopFlag Transfer ends without a stop signal.

25.2.7.7 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() in order 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 Callback is requested to provide data to transmit (slave-transmitter role).

kI2C_SlaveReceiveEvent Callback is requested to provide a buffer in which to place received data (slave-receiver role).
kI2C_SlaveTransmitAckEvent Callback needs to either transmit an ACK or NACK.
kI2C_SlaveCompletionEvent A stop was detected or finished transfer, completing the transfer.
kI2C_SlaveAllEvents Bit mask of all available events.

25.2.8 Function Documentation

25.2.8.1 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 to use the I2C driver, or any operation to the I2C module could cause hard fault because clock is not enabled. The configuration structure can be filled by user from scratch, or be set with default values by I2C_MasterGetDefaultConfig(). After calling this API, the master is ready to transfer. Example:

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

Parameters

base	I2C base pointer
masterConfig	pointer to master configuration structure
srcClock_Hz	I2C peripheral clock frequency in Hz

25.2.8.2 void I2C_SlaveInit (I2C_Type * *base*, const i2c_slave_config_t * *slaveConfig*)

Call this API to ungate the I2C clock and initializes the I2C with slave configuration.

Note

This API should be called at the beginning of the application to use the I2C driver, or 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 can be filled by the user. Example

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

Parameters

base	I2C base pointer
slaveConfig	pointer to slave configuration structure

25.2.8.3 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

25.2.8.4 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

25.2.8.5 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_MasterConfigure(). Use the initialized structure unchanged in I2C_MasterConfigure(), or modify some fields of the structure before calling I2C_MasterConfigure(). Example:

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

Parameters

masterConfig	Pointer to the master configuration structure.

25.2.8.6 void I2C_SlaveGetDefaultConfig (i2c_slave_config_t * *slaveConfig*)

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

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

Parameters

slaveConfig	Pointer to the slave configuration structure.

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

Parameters

base	I2C base pointer
enable	pass true to enable module, false to disable module

25.2.8.8 uint32_t I2C_MasterGetStatusFlags (I2C_Type * *base*)

Parameters

base	I2C base pointer

Returns

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

25.2.8.9 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 could get the related status.

25.2.8.10 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 could be any combination of the following values: • kI2C_StartDetectFlag (if available) • kI2C_StopDetectFlag (if available) • kI2C_ArbitrationLostFlag • kI2C_IntPendingFlagFlag

25.2.8.11 static void I2C_SlaveClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared: kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag

Parameters

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

25.2.8.12 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

25.2.8.13 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

25.2.8.14 static uint32_t I2C_GetDataRegAddr (I2C_Type * *base*) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

base	I2C base pointer

Returns

data register address

25.2.8.15 void I2C_MasterSetBaudRate (I2C_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

Parameters

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

25.2.8.16 status_t I2C_MasterStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

base	I2C peripheral base pointer
address	7-bit slave device address.
direction	Master transfer directions(transmit/receive).

Return values

kStatus_Success	Successfully send the start signal.
kStatus_I2C_Busy	Current bus is busy.

25.2.8.17 status_t I2C_MasterStop (I2C_Type * *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

25.2.8.18 status_t I2C_MasterRepeatedStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)

Parameters

base	I2C peripheral base pointer
address	7-bit slave device address.
direction	Master transfer directions(transmit/receive).

Return values

kStatus_Success	Successfully send the start signal.
kStatus_I2C_Busy	Current bus is busy but not occupied by current I2C master.

25.2.8.19 status_t I2C_MasterWriteBlocking (I2C_Type * *base*, const uint8_t * *txBuff*, size_t *txSize*)

Parameters

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

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

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

Note

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

Parameters

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

Return values

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

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

Parameters

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

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

25.2.8.22 void I2C_SlaveReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*)

Parameters

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

25.2.8.23 status_t I2C_MasterTransferBlocking (I2C_Type * *base*, i2c_master_transfer_t * *xfer*)

Note

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

Parameters

base	I2C peripheral base address.
xfer	Pointer to the transfer structure.

Return values

kStatus_Success	Successfully complete the data transmission.
kStatus_I2C_Busy	Previous transmission still not finished.
kStatus_I2C_Timeout	Transfer error, wait signal timeout.
kStatus_I2C_Arbitration-Lost	Transfer error, arbitration lost.
kStataus_I2C_Nak	Transfer error, receive NAK during transfer.

25.2.8.24 void I2C_MasterTransferCreateHandle (I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_callback_t *callback*, void * *userData*)

Parameters

base	I2C base pointer.
handle	pointer to i2c_master_handle_t structure to store the transfer state.
callback	pointer to user callback function.
userData	user paramater passed to the callback function.

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

Note

Calling the API will return immediately after transfer initiates, user needs to call I2C_MasterGet-TransferCount to poll the transfer status to check whether the transfer is finished, if the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

base	I2C base pointer.
handle	pointer to i2c_master_handle_t structure which stores the transfer state.
xfer	pointer to i2c_master_transfer_t structure.

Return values

kStatus_Success	Sucessully start the data transmission.
kStatus_I2C_Busy	Previous transmission still not finished.
kStatus_I2C_Timeout	Transfer error, wait signal timeout.

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

Parameters

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

Return values

kStatus_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

25.2.8.27 void I2C_MasterTransferAbort (I2C_Type * *base*, i2c_master_handle_t * *handle*)

Note

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

Parameters

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

25.2.8.28 void I2C_MasterTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

base	I2C base pointer.
i2cHandle	pointer to i2c_master_handle_t structure.

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

Parameters

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

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

Call this API after calling the I2C_SlaveInit() and I2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of i2c_slave_transfer_event_t enumerators for the events you wish to receive. The kI2C_SlaveTransmitEvent and kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
handle	Pointer to #i2c_slave_handle_t structure which stores the transfer state.
eventMask	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

#kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

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

Note

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

Parameters

base	I2C base pointer.
handle	pointer to i2c_slave_handle_t structure which stores the transfer state.

25.2.8.32 status_t I2C_SlaveTransferGetCount (I2C_Type * *base*, i2c_slave_handle_t * *handle*, size_t * *count*)

Parameters

base	I2C base pointer.
handle	pointer to i2c_slave_handle_t structure.
count	Number of bytes transferred so far by the non-blocking transaction.

Return values

kStatus_InvalidArgument	count is Invalid.
kStatus_Success	Successfully return the count.

25.2.8.33 void I2C_SlaveTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

base	I2C base pointer.
i2cHandle	pointer to i2c_slave_handle_t structure which stores the transfer state

25.3 I2C DMA Driver

25.3.1 Overview

Files

- file fsl_i2c_dma.h

Data Structures

- struct i2c_master_dma_handle_t
  I2C master dma transfer structure. More...

Typedefs

- typedef void(∗ i2c_master_dma_transfer_callback_t)(I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, status_t status, void ∗userData)
  I2C master dma transfer callback typedef.

I2C Block DMA Transfer Operation

- void I2C_MasterTransferCreateHandleDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle, i2c_master_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗dmaHandle)
  Init 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)
  Get master transfer status during a dma non-blocking transfer.
- void I2C_MasterTransferAbortDMA (I2C_Type ∗base, i2c_master_dma_handle_t ∗handle)
  Abort a master dma non-blocking transfer in a early time.

25.3.2 Data Structure Documentation

25.3.2.1 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
  Callback function called after dma transfer finished.
- void * userData
  Callback parameter passed to callback function.

25.3.2.1.0.22 Field Documentation

25.3.2.1.0.22.1 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

25.3.2.1.0.22.2 size_t i2c_master_dma_handle_t::transferSize

25.3.2.1.0.22.3 uint8_t i2c_master_dma_handle_t::state

25.3.2.1.0.22.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

25.3.2.1.0.22.5 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

25.3.2.1.0.22.6 void* i2c_master_dma_handle_t::userData

25.3.3 Typedef Documentation

25.3.3.1 typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)

25.3.4 Function Documentation

25.3.4.1 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 i2c_master_dma_handle_t structure

callback	pointer to user callback function
userData	user param passed to the callback function
dmaHandle	DMA handle pointer

25.3.4.2 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	pointer to i2c_master_dma_handle_t structure
xfer	pointer to transfer structure of i2c_master_transfer_t

Return values

kStatus_Success	Sucessully 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.

25.3.4.3 status_t I2C_MasterTransferGetCountDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	I2C peripheral base address
handle	pointer to i2c_master_dma_handle_t structure
count	Number of bytes transferred so far by the non-blocking transaction.

25.3.4.4 void I2C_MasterTransferAbortDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*)

Parameters

base	I2C peripheral base address
handle	pointer to i2c_master_dma_handle_t structure

25.4 I2C eDMA Driver

25.4.1 Overview

Files

- file fsl_i2c_edma.h

Data Structures

- struct i2c_master_edma_handle_t
  I2C master edma transfer structure. More...

Typedefs

- typedef void(∗ i2c_master_edma_transfer_callback_t)(I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, status_t status, void ∗userData)
  I2C master edma transfer callback typedef.

I2C Block EDMA Transfer Operation

- void I2C_MasterCreateEDMAHandle (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle, i2c_master_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗edmaHandle)
  Init 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)
  Get master transfer status during a edma non-blocking transfer.
- void I2C_MasterTransferAbortEDMA (I2C_Type ∗base, i2c_master_edma_handle_t ∗handle)
  Abort a master edma non-blocking transfer in a early time.

25.4.2 Data Structure Documentation

25.4.2.1 struct _i2c_master_edma_handle

I2C master edma 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.
- edma_handle_t * dmaHandle
  The eDMA handler used.
- i2c_master_edma_transfer_callback_t completionCallback
  Callback function called after edma transfer finished.
- void * userData
  Callback parameter passed to callback function.

25.4.2.1.0.23 Field Documentation

25.4.2.1.0.23.1 i2c_master_transfer_t i2c_master_edma_handle_t::transfer

25.4.2.1.0.23.2 size_t i2c_master_edma_handle_t::transferSize

25.4.2.1.0.23.3 uint8_t i2c_master_edma_handle_t::state

25.4.2.1.0.23.4 edma_handle_t* i2c_master_edma_handle_t::dmaHandle

25.4.2.1.0.23.5 i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completionCallback

25.4.2.1.0.23.6 void* i2c_master_edma_handle_t::userData

25.4.3 Typedef Documentation

25.4.3.1 typedef void(* i2c_master_edma_transfer_callback_t)(I2C_Type *base, i2c_master_edma_handle_t *handle, status_t status, void *userData)

25.4.4 Function Documentation

25.4.4.1 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	pointer to i2c_master_edma_handle_t structure.

callback	pointer to user callback function.
userData	user param passed to the callback function.
edmaHandle	EDMA handle pointer.

25.4.4.2 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	pointer to i2c_master_edma_handle_t structure.
xfer	pointer to transfer structure of i2c_master_transfer_t.

Return values

kStatus_Success	Sucessully 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.

25.4.4.3 status_t I2C_MasterTransferGetCountEDMA (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*, size_t * *count*)

Parameters

base	I2C peripheral base address.
handle	pointer to i2c_master_edma_handle_t structure.
count	Number of bytes transferred so far by the non-blocking transaction.

25.4.4.4 void I2C_MasterTransferAbortEDMA (I2C_Type * *base*, i2c_master_edma_handle_t * *handle*)

Parameters

base	I2C peripheral base address.
handle	pointer to i2c_master_edma_handle_t structure.

25.5 I2C FreeRTOS Driver

25.5.1 Overview

Files

- file fsl_i2c_freertos.h

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

Driver version

- #define FSL_I2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  I2C FreeRTOS driver version 2.0.0.

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t *handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
  Performs I2C transfer.

25.5.2 Data Structure Documentation

25.5.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type * base
  I2C base address.
- i2c_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a trasfer.
- SemaphoreHandle_t sem
  Semaphore to notify and unblock task when transfer ends.
- OS_EVENT * mutex
  Mutex to lock the handle during a trasfer.

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

25.5.3 Macro Definition Documentation

25.5.3.1 #define FSL_I2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

25.5.4 Function Documentation

25.5.4.1 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 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	Configuration structure to set-up I2C in master mode.
srcClock_Hz	Frequency of input clock of the I2C module.

Returns

status of the operation.

25.5.4.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and related RTOS context.

Parameters

handle	The RTOS I2C handle.

25.5.4.3 status_t I2C_RTOS_Transfer (i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)

This function performs an I2C transfer according to data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

25.6 I2C µCOS/II Driver

25.6.1 Overview

Files

- file fsl_i2c_ucosii.h
- file fsl_i2c_ucosiii.h

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

Driver version

- #define FSL_I2C_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  I2C uCOS II driver version 2.0.0.

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t *handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
  Performs I2C transfer.

Driver version

- #define FSL_I2C_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  I2C uCOS III driver version 2.0.0.

25.6.2 Data Structure Documentation

25.6.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type * base
  I2C base address.
- i2c_master_handle_t drv_handle

Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a trasfer.
- SemaphoreHandle_t sem
  Semaphore to notify and unblock task when transfer ends.
- OS_EVENT * mutex
  Mutex to lock the handle during a trasfer.
- OS_FLAG_GRP * event
  Semaphore to notify and unblock task when transfer ends.
- OS_SEM mutex
  Mutex to lock the handle during a trasfer.
- OS_FLAG_GRP event
  Semaphore to notify and unblock task when transfer ends.

25.6.3 Macro Definition Documentation

25.6.3.1 #define FSL_I2C_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

25.6.3.2 #define FSL_I2C_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

25.6.4 Function Documentation

25.6.4.1 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 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	Configuration structure to set-up I2C in master mode.
srcClock_Hz	Frequency of input clock of the I2C module.

Returns

status of the operation.

25.6.4.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and related RTOS context.

Parameters

handle	The RTOS I2C handle.

25.6.4.3 status_t I2C_RTOS_Transfer (i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)

This function performs an I2C transfer according to data given in the transfer structure.

Parameters

handle	The RTOS I2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

25.7 I2C µCOS/III Driver

Chapter 26
LLWU: Low-Leakage Wakeup Unit Driver

26.1 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.

26.2 External wakeup pins configurations

Configures the external wakeup pins' working modes, gets and clears the wake pin flags. External wakeup pins are accessed by `pinIndex` which is started from 1. Numbers of external pins depend on the SoC configuration.

26.3 Internal wakeup modules configurations

Enables/disables the internal wakeup modules, and gets the modules flags. Internal modules are accessed by `moduleIndex` which is started from 1. Numbers of external pins depend the on SoC configuration.

26.4 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 pins filters are accessed by `filterIndex` which is started from 1. Numbers of external pins depends on the SoC configuration.

Files

- file fsl_llwu.h

Enumerations

- enum llwu_external_pin_mode_t {
  kLLWU_ExternalPinDisable = 0U,
  kLLWU_ExternalPinRisingEdge = 1U,
  kLLWU_ExternalPinFallingEdge = 2U,
  kLLWU_ExternalPinAnyEdge = 3U }
  External input pin control modes.
- enum llwu_pin_filter_mode_t {
  kLLWU_PinFilterDisable = 0U,
  kLLWU_PinFilterRisingEdge = 1U,
  kLLWU_PinFilterFallingEdge = 2U,
  kLLWU_PinFilterAnyEdge = 3U }
  Digital filter control modes.

Driver version

- #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  LLWU driver version 2.0.1.

26.5 Macro Definition Documentation

26.5.1 #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

26.6 Enumeration Type Documentation

26.6.1 enum llwu_external_pin_mode_t

Enumerator

kLLWU_ExternalPinDisable Pin disabled as wakeup input.
kLLWU_ExternalPinRisingEdge Pin enabled with rising edge detection.
kLLWU_ExternalPinFallingEdge Pin enabled with falling edge detection.
kLLWU_ExternalPinAnyEdge Pin enabled with any change detection.

26.6.2 enum llwu_pin_filter_mode_t

Enumerator

kLLWU_PinFilterDisable Filter disabled.
kLLWU_PinFilterRisingEdge Filter positive edge detection.
kLLWU_PinFilterFallingEdge Filter negative edge detection.
kLLWU_PinFilterAnyEdge Filter any edge detection.

Chapter 27
LMEM: Local Memory Controller Cache Control Driver

27.1 Overview

The KSDK provides a peripheral driver for the Local Memory Controller Cache Controller module of Kinetis devices.

27.2 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: 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	Unconditionally clear valid and modify bits of a cache entry.
Push	Push a cache entry if it is valid and modified, then clear the modififcation
Clear	Push a cache entry if it is valid

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".

27.3 Function groups

27.3.1 Local Memory Processor Code Bus Cache Control

The invalidate command can be performed on the entire cache, one line and multiple lines by calling LMEM_CodeCacheInvalidateAll(), LMEM_CodeCacheInvalidateLine(), and LMEM_CodeCacheInvalidateMultiLines().

The push command can be performed on the entire cache, one line and multiple lines by calling LMEM_CodeCachePushAll(), LMEM_CodeCachePushLine(), and LMEM_CodeCachePushMultiLines().

The clear command can be performed on the entire cache, one line and 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 you wish to perform the one line cache maintenance operation. In addition, the length 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 WAY1 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. The user should call LMEM_EnableCodeCache() to enable a particular bus cache.

27.3.2 Local Memory Processor System Bus Cache Control

The invalidate command can be performed on the entire cache, one line and multiple lines by calling LMEM_SystemCacheInvalidateAll(), LMEM_SystemCacheInvalidateLine(), and LMEM_SystemCacheInvalidateMultiLines().

The push command can be performed on the entire cache, one line and multiple lines by calling LMEM_SystemCachePushAll(), LMEM_SystemCachePushLine(), and LMEM_SystemCachePushMultiLines().

The clear command can be performed on the entire cache, one line and multiple lines by calling LMEM_SystemCacheClearAll(), LMEM_SystemCacheClearLine(), and LMEM_SystemCacheClearMultiLines().

Note that the parameter "address" must be supplied, which indicates the physical address of the line you wish to perform the one line cache maintenance operation. In addition, the length 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 WAY1 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 detailed in the Chip Configuration part of the Kinetis SoC 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 PS bus cache and the write buffer. However, before enabling these, the function first performs an invalidate all. The user should call LMEM_EnableSystemCache() to enable a particular bus cache.

Files

- file fsl_lmem_cache.h

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, 0, 0))
LMEM controller driver version 2.0.0.

Local Memory Processor Code Bus Cache Control

- void LMEM_EnableCodeCache (LMEM_Type *base, bool enable)
Enables/disables the processor code bus cache.
- 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.

27.4 Macro Definition Documentation

27.4.1 #define FSL_LMEM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

27.4.2 #define LMEM_CACHE_LINE_SIZE (0x10U)

27.4.3 #define LMEM_CACHE_SIZE_ONEWAY (4096U)

27.5 Enumeration Type Documentation

27.5.1 enum lmem_cache_mode_t

Enumerator

kLMEM_NonCacheable CACHE mode: non-cacheable.
kLMEM_CacheWriteThrough CACHE mode: write-through.
kLMEM_CacheWriteBack CACHE mode: write-back.

27.5.2 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.

27.5.3 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.

27.6 Function Documentation

27.6.1 void LMEM_EnableCodeCache (LMEM_Type * *base,* bool *enable*)

This function enables/disables the cache. The function first invalidates the entire cache and then enables/disable 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.

27.6.2 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.

27.6.3 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.

27.6.4 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.

27.6.5 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.

27.6.6 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. Then the function performs an entire cache invalidate function, which is more efficient than invalidating the cache line-by-line. The need to check half the total amount of cache is due to the fact that the cache consists of two ways and that line commands based on the physical address searches both ways. 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.

27.6.7 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.

27.6.8 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. The need to check half the total amount of cache is due to the fact that the cache consists of two ways and that line commands based on the physical address searches both ways. 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.

27.6.9 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.

27.6.10 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. The need to check half the total amount of cache is due to the fact that the cache consists of two ways and that line commands based on the physical address searches both ways. 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.

27.6.11 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.

Chapter 28
LPI2C: Low Power I2C Driver

28.1 Overview

Modules

- LPI2C FreeRTOS Driver
- LPI2C Master DMA Driver
- LPI2C Master Driver
- LPI2C Slave DMA Driver
- LPI2C Slave Driver
- LPI2C µCOS/II Driver
- LPI2C µCOS/III Driver

Files

- file fsl_lpi2c.h

Enumerations

- enum _lpi2c_status {
  kStatus_LPI2C_Busy = MAKE_STATUS(kStatusGroup_LPI2C, 0),
  kStatus_LPI2C_Idle = MAKE_STATUS(kStatusGroup_LPI2C, 1),
  kStatus_LPI2C_Nak = MAKE_STATUS(kStatusGroup_LPI2C, 2),
  kStatus_LPI2C_FifoError = MAKE_STATUS(kStatusGroup_LPI2C, 3),
  kStatus_LPI2C_BitError = MAKE_STATUS(kStatusGroup_LPI2C, 4),
  kStatus_LPI2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_LPI2C, 5),
  kStatus_LPI2C_PinLowTimeout,
  kStatus_LPI2C_NoTransferInProgress,
  kStatus_LPI2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_LPI2C, 7) }
  LPI2C status return codes.

Driver version

- #define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  LPI2C driver version 2.1.0.

28.2 Macro Definition Documentation

28.2.1 #define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

28.3 Enumeration Type Documentation

28.3.1 enum _lpi2c_status

Enumerator

kStatus_LPI2C_Busy The master is already performing a transfer.
kStatus_LPI2C_Idle The slave driver is idle.
kStatus_LPI2C_Nak The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError FIFO under run or overrun.
kStatus_LPI2C_BitError Transferred bit was not seen on the bus.
kStatus_LPI2C_ArbitrationLost Arbitration lost error.
kStatus_LPI2C_PinLowTimeout SCL or SDA were held low longer than the timeout.
kStatus_LPI2C_NoTransferInProgress Attempt to abort a transfer when one is not in progress.
kStatus_LPI2C_DmaRequestFail DMA request failed.

28.4 LPI2C FreeRTOS Driver

28.4.1 Overview

Files

- file fsl_lpi2c_freertos.h

Data Structures

- struct lpi2c_rtos_handle_t
  LPI2C FreeRTOS handle. More...

Driver version

- #define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPI2C FreeRTOS driver version 2.0.0.

LPI2C RTOS Operation

- status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t *handle, LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes LPI2C.
- status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t *handle)
  Deinitializes the LPI2C.
- status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t *handle, lpi2c_master_transfer_t *transfer)
  Performs I2C transfer.

28.4.2 Data Structure Documentation

28.4.2.1 struct lpi2c_rtos_handle_t

LPI2C uCOS III handle.

LPI2C uCOS II handle.

Data Fields

- LPI2C_Type * base
  LPI2C base address.
- lpi2c_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a trasfer.

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

28.4.3 Macro Definition Documentation

28.4.3.1 #define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

28.4.4 Function Documentation

28.4.4.1 status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t * *handle*, LPI2C_Type * *base*, const lpi2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the LPI2C instance to initialize.
masterConfig	Configuration structure to set-up LPI2C in master mode.
srcClock_Hz	Frequency of input clock of the LPI2C module.

Returns

status of the operation.

28.4.4.2 status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t * *handle*)

This function deinitializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle.

28.4.4.3 status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t * *handle*, lpi2c_master_transfer_t * *transfer*)

This function performs an I2C transfer using LPI2C module according to data given in the transfer structure.

Parameters

handle	The RTOS LPI2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

28.5 LPI2C Master Driver

28.5.1 Overview

Data Structures

- struct lpi2c_master_config_t
  Structure with settings to initialize the LPI2C master module. More...
- struct lpi2c_data_match_config_t
  LPI2C master data match configuration structure. More...
- struct lpi2c_master_transfer_t
  Non-blocking transfer descriptor structure. More...
- struct lpi2c_master_handle_t
  Driver handle for master non-blocking APIs. More...

Typedefs

- typedef void(∗ lpi2c_master_transfer_callback_t)(LPI2C_Type ∗base, lpi2c_master_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Master completion callback function pointer type.

Enumerations

- enum _lpi2c_master_flags {
  kLPI2C_MasterTxReadyFlag = LPI2C_MSR_TDF_MASK,
  kLPI2C_MasterRxReadyFlag = LPI2C_MSR_RDF_MASK,
  kLPI2C_MasterEndOfPacketFlag = LPI2C_MSR_EPF_MASK,
  kLPI2C_MasterStopDetectFlag = LPI2C_MSR_SDF_MASK,
  kLPI2C_MasterNackDetectFlag = LPI2C_MSR_NDF_MASK,
  kLPI2C_MasterArbitrationLostFlag = LPI2C_MSR_ALF_MASK,
  kLPI2C_MasterFifoErrFlag = LPI2C_MSR_FEF_MASK,
  kLPI2C_MasterPinLowTimeoutFlag = LPI2C_MSR_PLTF_MASK,
  kLPI2C_MasterDataMatchFlag = LPI2C_MSR_DMF_MASK,
  kLPI2C_MasterBusyFlag = LPI2C_MSR_MBF_MASK,
  kLPI2C_MasterBusBusyFlag = LPI2C_MSR_BBF_MASK }
  LPI2C master peripheral flags.
- enum lpi2c_direction_t {
  kLPI2C_Write = 0U,
  kLPI2C_Read = 1U }
  Direction of master and slave transfers.
- enum lpi2c_master_pin_config_t {

kLPI2C_2PinOpenDrain = 0x0U,
kLPI2C_2PinOutputOnly = 0x1U,
kLPI2C_2PinPushPull = 0x2U,
kLPI2C_4PinPushPull = 0x3U,
kLPI2C_2PinOpenDrainWithSeparateSlave,
kLPI2C_2PinOutputOnlyWithSeparateSlave,
kLPI2C_2PinPushPullWithSeparateSlave,
kLPI2C_4PinPushPullWithInvertedOutput = 0x7U }
LPI2C pin configuration.

- enum lpi2c_host_request_source_t {
kLPI2C_HostRequestExternalPin = 0x0U,
kLPI2C_HostRequestInputTrigger = 0x1U }
LPI2C master host request selection.
- enum lpi2c_host_request_polarity_t {
kLPI2C_HostRequestPinActiveLow = 0x0U,
kLPI2C_HostRequestPinActiveHigh = 0x1U }
LPI2C master host request pin polarity configuration.
- enum lpi2c_data_match_config_mode_t {
kLPI2C_MatchDisabled = 0x0U,
kLPI2C_1stWordEqualsM0OrM1 = 0x2U,
kLPI2C_AnyWordEqualsM0OrM1 = 0x3U,
kLPI2C_1stWordEqualsM0And2ndWordEqualsM1,
kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1,
kLPI2C_1stWordAndM1EqualsM0AndM1,
kLPI2C_AnyWordAndM1EqualsM0AndM1 }
LPI2C master data match configuration modes.
- enum _lpi2c_master_transfer_flags {
kLPI2C_TransferDefaultFlag = 0x00U,
kLPI2C_TransferNoStartFlag = 0x01U,
kLPI2C_TransferRepeatedStartFlag = 0x02U,
kLPI2C_TransferNoStopFlag = 0x04U }
Transfer option flags.

Initialization and deinitialization

- void LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t *masterConfig)
Provides a default configuration for the LPI2C master peripheral.
- void LPI2C_MasterInit (LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t sourceClock_Hz)
Initializes the LPI2C master peripheral.
- void LPI2C_MasterDeinit (LPI2C_Type *base)
Deinitializes the LPI2C master peripheral.
- void LPI2C_MasterConfigureDataMatch (LPI2C_Type *base, const lpi2c_data_match_config_t *config)
Configures LPI2C master data match feature.
- static void LPI2C_MasterReset (LPI2C_Type *base)

Performs a software reset.

- static void LPI2C_MasterEnable (LPI2C_Type *base, bool enable)
  Enables or disables the LPI2C module as master.

Status

- static uint32_t LPI2C_MasterGetStatusFlags (LPI2C_Type *base)
  Gets the LPI2C master status flags.
- static void LPI2C_MasterClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)
  Clears the LPI2C master status flag state.

Interrupts

- static void LPI2C_MasterEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Enables the LPI2C master interrupt requests.
- static void LPI2C_MasterDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Disables the LPI2C master interrupt requests.
- static uint32_t LPI2C_MasterGetEnabledInterrupts (LPI2C_Type *base)
  Returns the set of currently enabled LPI2C master interrupt requests.

DMA control

- static void LPI2C_MasterEnableDMA (LPI2C_Type *base, bool enableTx, bool enableRx)
  Enables or disables LPI2C master DMA requests.
- static uint32_t LPI2C_MasterGetTxFifoAddress (LPI2C_Type *base)
  Gets LPI2C master transmit data register address for DMA transfer.
- static uint32_t LPI2C_MasterGetRxFifoAddress (LPI2C_Type *base)
  Gets LPI2C master receive data register address for DMA transfer.

FIFO control

- static void LPI2C_MasterSetWatermarks (LPI2C_Type *base, size_t txWords, size_t rxWords)
  Sets the watermarks for LPI2C master FIFOs.
- static void LPI2C_MasterGetFifoCounts (LPI2C_Type *base, size_t *rxCount, size_t *txCount)
  Gets the current number of words in the LPI2C master FIFOs.

Bus operations

- void LPI2C_MasterSetBaudRate (LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t baudRate_Hz)
  Sets the I2C bus frequency for master transactions.
- static bool LPI2C_MasterGetBusIdleState (LPI2C_Type *base)
  Returns whether the bus is idle.
- status_t LPI2C_MasterStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)

Sends a START signal and slave address on the I2C bus.

- static status_t LPI2C_MasterRepeatedStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
  Sends a repeated START signal and slave address on the I2C bus.
- status_t LPI2C_MasterSend (LPI2C_Type *base, const void *txBuff, size_t txSize)
  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_MasterReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize)
  Performs a polling receive transfer on the I2C bus.
- status_t LPI2C_MasterStop (LPI2C_Type *base)
  Sends a STOP signal on the I2C bus.

Non-blocking

- void LPI2C_MasterTransferCreateHandle (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)
  Creates a new handle for the LPI2C master non-blocking APIs.
- status_t LPI2C_MasterTransferNonBlocking (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_t *transfer)
  Performs a non-blocking transaction on the I2C bus.
- status_t LPI2C_MasterTransferGetCount (LPI2C_Type *base, lpi2c_master_handle_t *handle, size_t *count)
  Returns number of bytes transferred so far.
- void LPI2C_MasterTransferAbort (LPI2C_Type *base, lpi2c_master_handle_t *handle)
  Terminates a non-blocking LPI2C master transmission early.

IRQ handler

- void LPI2C_MasterTransferHandleIRQ (LPI2C_Type *base, lpi2c_master_handle_t *handle)
  Reusable routine to handle master interrupts.

28.5.2 Data Structure Documentation

28.5.2.1 struct lpi2c_master_config_t

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

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

Data Fields

- bool enableMaster
  Whether to enable master mode.
- bool enableDoze

Whether master is enabled in doze mode.

- bool debugEnable
  Enable transfers to continue when halted in debug mode.
- bool ignoreAck
  Whether to ignore ACK/NACK.
- lpi2c_master_pin_config_t pinConfig
  The pin configuration option.
- uint32_t baudRate_Hz
  Desired baud rate in Hertz.
- uint32_t busIdleTimeout_ns
  Bus idle timeout in nanoseconds.
- uint32_t pinLowTimeout_ns
  Pin low timeout in nanoseconds.
- uint8_t sdaGlitchFilterWidth_ns
  Width in nanoseconds of glitch filter on SDA pin.
- uint8_t sclGlitchFilterWidth_ns
  Width in nanoseconds of glitch filter on SCL pin.
- struct {
  bool enable
  Enable host request.
  lpi2c_host_request_source_t source
  Host request source.
  lpi2c_host_request_polarity_t polarity
  Host request pin polarity.
  } hostRequest

  Host request options.

28.5.2.1.0.24 Field Documentation

28.5.2.1.0.24.1 bool lpi2c_master_config_t::enableMaster

28.5.2.1.0.24.2 bool lpi2c_master_config_t::enableDoze

28.5.2.1.0.24.3 bool lpi2c_master_config_t::debugEnable

28.5.2.1.0.24.4 bool lpi2c_master_config_t::ignoreAck

28.5.2.1.0.24.5 lpi2c_master_pin_config_t lpi2c_master_config_t::pinConfig

28.5.2.1.0.24.6 uint32_t lpi2c_master_config_t::baudRate_Hz

28.5.2.1.0.24.7 uint32_t lpi2c_master_config_t::busIdleTimeout_ns

Set to 0 to disable.

28.5.2.1.0.24.8 uint32_t lpi2c_master_config_t::pinLowTimeout_ns

Set to 0 to disable.

28.5.2.1.0.24.9 uint8_t lpi2c_master_config_t::sdaGlitchFilterWidth_ns

Set to 0 to disable.

28.5.2.1.0.24.10 uint8_t lpi2c_master_config_t::sclGlitchFilterWidth_ns

Set to 0 to disable.

28.5.2.1.0.24.11 bool lpi2c_master_config_t::enable

28.5.2.1.0.24.12 lpi2c_host_request_source_t lpi2c_master_config_t::source

28.5.2.1.0.24.13 lpi2c_host_request_polarity_t lpi2c_master_config_t::polarity

28.5.2.1.0.24.14 struct { ... } lpi2c_master_config_t::hostRequest

28.5.2.2 struct lpi2c_data_match_config_t

Data Fields

- lpi2c_data_match_config_mode_t matchMode
  Data match configuration setting.
- bool rxDataMatchOnly
  When set to true, received data is ignored until a successful match.
- uint32_t match0
  Match value 0.
- uint32_t match1
  Match value 1.

28.5.2.2.0.25 Field Documentation

28.5.2.2.0.25.1 lpi2c_data_match_config_mode_t lpi2c_data_match_config_t::matchMode

28.5.2.2.0.25.2 bool lpi2c_data_match_config_t::rxDataMatchOnly

28.5.2.2.0.25.3 uint32_t lpi2c_data_match_config_t::match0

28.5.2.2.0.25.4 uint32_t lpi2c_data_match_config_t::match1

28.5.2.3 struct _lpi2c_master_transfer

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

Data Fields

- uint32_t flags
  Bit mask of options for the transfer.
- uint16_t slaveAddress
  The 7-bit slave address.
- lpi2c_direction_t direction

Either kLPI2C_Read or kLPI2C_Write.

- uint32_t subaddress
  Sub address.
- size_t subaddressSize
  Length of sub address to send in bytes.
- void * data
  Pointer to data to transfer.
- size_t dataSize
  Number of bytes to transfer.

28.5.2.3.0.26 Field Documentation

28.5.2.3.0.26.1 uint32_t lpi2c_master_transfer_t::flags

See enumeration _lpi2c_master_transfer_flags for available options. Set to 0 or kLPI2C_TransferDefaultFlag for normal transfers.

28.5.2.3.0.26.2 uint16_t lpi2c_master_transfer_t::slaveAddress

28.5.2.3.0.26.3 lpi2c_direction_t lpi2c_master_transfer_t::direction

28.5.2.3.0.26.4 uint32_t lpi2c_master_transfer_t::subaddress

Transferred MSB first.

28.5.2.3.0.26.5 size_t lpi2c_master_transfer_t::subaddressSize

Maximum size is 4 bytes.

28.5.2.3.0.26.6 void* lpi2c_master_transfer_t::data

28.5.2.3.0.26.7 size_t lpi2c_master_transfer_t::dataSize

28.5.2.4 struct _lpi2c_master_handle

Note

The contents of this structure are private and subject to change.

Data Fields

- uint8_t state
  Transfer state machine current state.
- uint16_t remainingBytes
  Remaining byte count in current state.
- uint8_t * buf
  Buffer pointer for current state.
- uint16_t commandBuffer [7]
  LPI2C command sequence.
- lpi2c_master_transfer_t transfer

Copy of the current transfer info.

- lpi2c_master_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.

28.5.2.4.0.27 Field Documentation

28.5.2.4.0.27.1 uint8_t lpi2c_master_handle_t::state

28.5.2.4.0.27.2 uint16_t lpi2c_master_handle_t::remainingBytes

28.5.2.4.0.27.3 uint8_t* lpi2c_master_handle_t::buf

28.5.2.4.0.27.4 uint16_t lpi2c_master_handle_t::commandBuffer[7]

28.5.2.4.0.27.5 lpi2c_master_transfer_t lpi2c_master_handle_t::transfer

28.5.2.4.0.27.6 lpi2c_master_transfer_callback_t lpi2c_master_handle_t::completionCallback

28.5.2.4.0.27.7 void* lpi2c_master_handle_t::userData

28.5.3 Typedef Documentation

28.5.3.1 typedef void(* lpi2c_master_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterTransferCreateHandle().

Parameters

base	The LPI2C peripheral base address.
completion-Status	Either #kStatus_Success or an error code describing how the transfer completed.
userData	Arbitrary pointer-sized value passed from the application.

28.5.4 Enumeration Type Documentation

28.5.4.1 enum _lpi2c_master_flags

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag

- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kLPI2C_MasterTxReadyFlag Transmit data flag.
kLPI2C_MasterRxReadyFlag Receive data flag.
kLPI2C_MasterEndOfPacketFlag End Packet flag.
kLPI2C_MasterStopDetectFlag Stop detect flag.
kLPI2C_MasterNackDetectFlag NACK detect flag.
kLPI2C_MasterArbitrationLostFlag Arbitration lost flag.
kLPI2C_MasterFifoErrFlag FIFO error flag.
kLPI2C_MasterPinLowTimeoutFlag Pin low timeout flag.
kLPI2C_MasterDataMatchFlag Data match flag.
kLPI2C_MasterBusyFlag Master busy flag.
kLPI2C_MasterBusBusyFlag Bus busy flag.

28.5.4.2 enum lpi2c_direction_t

Enumerator

kLPI2C_Write Master transmit.
kLPI2C_Read Master receive.

28.5.4.3 enum lpi2c_master_pin_config_t

Enumerator

kLPI2C_2PinOpenDrain LPI2C Configured for 2-pin open drain mode.
kLPI2C_2PinOutputOnly LPI2C Configured for 2-pin output only mode (ultra-fast mode)
kLPI2C_2PinPushPull LPI2C Configured for 2-pin push-pull mode.
kLPI2C_4PinPushPull LPI2C Configured for 4-pin push-pull mode.
kLPI2C_2PinOpenDrainWithSeparateSlave LPI2C Configured for 2-pin open drain mode with separate LPI2C slave.
kLPI2C_2PinOutputOnlyWithSeparateSlave LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave.

kLPI2C_2PinPushPullWithSeparateSlave LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave.
kLPI2C_4PinPushPullWithInvertedOutput LPI2C Configured for 4-pin push-pull mode(inverted outputs)

28.5.4.4 enum lpi2c_host_request_source_t

Enumerator

kLPI2C_HostRequestExternalPin Select the LPI2C_HREQ pin as the host request input.
kLPI2C_HostRequestInputTrigger Select the input trigger as the host request input.

28.5.4.5 enum lpi2c_host_request_polarity_t

Enumerator

kLPI2C_HostRequestPinActiveLow Configure the LPI2C_HREQ pin active low.
kLPI2C_HostRequestPinActiveHigh Configure the LPI2C_HREQ pin active high.

28.5.4.6 enum lpi2c_data_match_config_mode_t

Enumerator

kLPI2C_MatchDisabled LPI2C Match Disabled.
kLPI2C_1stWordEqualsM0OrM1 LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1.
kLPI2C_AnyWordEqualsM0OrM1 LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1.
kLPI2C_1stWordEqualsM0And2ndWordEqualsM1 LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1.
kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1 LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1.
kLPI2C_1stWordAndM1EqualsM0AndM1 LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1.
kLPI2C_AnyWordAndM1EqualsM0AndM1 LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1.

28.5.4.7 enum _lpi2c_master_transfer_flags

Note

These enumerations are intended to be OR'd together to form a bit mask of options for the _lpi2c_-master_transfer::flags field.

Enumerator

kLPI2C_TransferDefaultFlag Transfer starts with a start signal, stops with a stop signal.
kLPI2C_TransferNoStartFlag Don't send a start condition, address, and sub address.
kLPI2C_TransferRepeatedStartFlag Send a repeated start condition.
kLPI2C_TransferNoStopFlag Don't send a stop condition.

28.5.5 Function Documentation

28.5.5.1 void LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t * *masterConfig*)

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster            = true;
masterConfig->debugEnable             = false;
masterConfig->ignoreAck               = false;
masterConfig->pinConfig               = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz             = 100000U;
masterConfig->busIdleTimeout_ns       = 0;
masterConfig->pinLowTimeout_ns        = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable      = false;
masterConfig->hostRequest.source      = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity    = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with LPI2C_MasterInit().

Parameters

out	*masterConfig*	User provided configuration structure for default values. Refer to lpi2c-_master_config_t.

28.5.5.2 void LPI2C_MasterInit (LPI2C_Type * *base*, const lpi2c_master_config_t * *masterConfig*, uint32_t *sourceClock_Hz*)

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

base	The LPI2C peripheral base address.
masterConfig	User provided peripheral configuration. Use LPI2C_MasterGetDefaultConfig() to get a set of defaults that you can override.
sourceClock_Hz	Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

28.5.5.3 void LPI2C_MasterDeinit (LPI2C_Type * *base*)

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

28.5.5.4 void LPI2C_MasterConfigureDataMatch (LPI2C_Type * *base*, const lpi2c_data_match_config_t * *config*)

Parameters

base	The LPI2C peripheral base address.
config	Settings for the data match feature.

28.5.5.5 static void LPI2C_MasterReset (LPI2C_Type * *base*) [inline], [static]

Restores the LPI2C master peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

28.5.5.6 static void LPI2C_MasterEnable (LPI2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.
enable	Pass true to enable or false to disable the specified LPI2C as master.

28.5.5.7 static uint32_t LPI2C_MasterGetStatusFlags (LPI2C_Type * *base*) [inline], [static]

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

State of the status flags:
- 1: related status flag is set.
- 0: related status flag is not set.

See Also

_lpi2c_master_flags

28.5.5.8 static void LPI2C_MasterClearStatusFlags (LPI2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

Parameters

base	The LPI2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_master_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_MasterGetStatusFlags().

See Also

_lpi2c_master_flags.

28.5.5.9 static void LPI2C_MasterEnableInterrupts (LPI2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

28.5.5.10 static void LPI2C_MasterDisableInterrupts (LPI2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

28.5.5.11 static uint32_t LPI2C_MasterGetEnabledInterrupts (LPI2C_Type * *base*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

28.5.5.12 static void LPI2C_MasterEnableDMA (LPI2C_Type * *base*, bool *enableTx*, bool *enableRx*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.
enableTx	Enable flag for transmit DMA request. Pass true for enable, false for disable.
enableRx	Enable flag for receive DMA request. Pass true for enable, false for disable.

28.5.5.13 static uint32_t LPI2C_MasterGetTxFifoAddress (LPI2C_Type * *base*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

28.5.5.14 static uint32_t LPI2C_MasterGetRxFifoAddress (LPI2C_Type * *base*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

28.5.5.15 static void LPI2C_MasterSetWatermarks (LPI2C_Type * *base*, size_t *txWords*, size_t *rxWords*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.
txWords	Transmit FIFO watermark value in words. The kLPI2C_MasterTxReadyFlag flag is set whenever the number of words in the transmit FIFO is equal or less than *txWords*. Writing a value equal or greater than the FIFO size is truncated.
rxWords	Receive FIFO watermark value in words. The kLPI2C_MasterRxReadyFlag flag is set whenever the number of words in the receive FIFO is greater than *rxWords*. Writing a value equal or greater than the FIFO size is truncated.

28.5.5.16 static void LPI2C_MasterGetFifoCounts (LPI2C_Type * *base*, size_t * *rxCount*, size_t * *txCount*) [inline], [static]

Parameters

	base	The LPI2C peripheral base address.
out	*txCount*	Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
out	*rxCount*	Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.

28.5.5.17 void LPI2C_MasterSetBaudRate (LPI2C_Type * *base*, uint32_t *sourceClock_Hz*, uint32_t *baudRate_Hz*)

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

base	The LPI2C peripheral base address.
sourceClock_-Hz	LPI2C functional clock frequency in Hertz.
baudRate_Hz	Requested bus frequency in Hertz.

28.5.5.18 static bool LPI2C_MasterGetBusIdleState (LPI2C_Type * *base*) [inline], [static]

Requires the master mode to be enabled.

Parameters

base	The LPI2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

28.5.5.19 status_t LPI2C_MasterStart (LPI2C_Type * *base*, uint8_t *address*, lpi2c_direction_t *dir*)

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

base	The LPI2C peripheral base address.
address	7-bit slave device address, in bits [6:0].
dir	Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

#kStatus_Success	START signal and address were successfully enqueued in the transmit FIFO.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.

28.5.5.20 static status_t LPI2C_MasterRepeatedStart (LPI2C_Type * *base*, uint8_t *address*, lpi2c_direction_t *dir*) [inline], [static]

This function is used to send a Repeated START signal when a transfer is already in progress. Like LPI2C_MasterStart(), it also sends the specified 7-bit address.

Note

This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

base	The LPI2C peripheral base address.
address	7-bit slave device address, in bits [6:0].
dir	Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

#kStatus_Success	Repeated START signal and address were successfully enqueued in the transmit FIFO.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.

28.5.5.21 status_t LPI2C_MasterSend (LPI2C_Type * *base*, const void * *txBuff*, size_t *txSize*)

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_LPI2C_Nak.

Parameters

base	The LPI2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.

Return values

#kStatus_Success	Data was sent successfully.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or over run.
kStatus_LPI2C_-ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLow-Timeout	SCL or SDA were held low longer than the timeout.

28.5.5.22 status_t LPI2C_MasterReceive (LPI2C_Type * *base*, void * *rxBuff*, size_t *rxSize*)

Parameters

base	The LPI2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.

Return values

#kStatus_Success	Data was received successfully.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or overrun.
kStatus_LPI2C_-ArbitrationLost	Arbitration lost error.

kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.

28.5.5.23 status_t LPI2C_MasterStop (LPI2C_Type * *base*)

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

base	The LPI2C peripheral base address.

Return values

#kStatus_Success	The STOP signal was successfully sent on the bus and the transaction terminated.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or overrun.
kStatus_LPI2C_ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.

28.5.5.24 void LPI2C_MasterTransferCreateHandle (LPI2C_Type * *base*, lpi2c_master_handle_t * *handle*, lpi2c_master_transfer_callback_t *callback*, void * *userData*)

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_MasterTransferAbort() API shall be called.

Parameters

	base	The LPI2C peripheral base address.
`out`	*handle*	Pointer to the LPI2C master driver handle.

	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

28.5.5.25 status_t LPI2C_MasterTransferNonBlocking (LPI2C_Type * *base*, lpi2c_master_handle_t * *handle*, lpi2c_master_transfer_t * *transfer*)

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.
transfer	The pointer to the transfer descriptor.

Return values

#kStatus_Success	The transaction was started successfully.
kStatus_LPI2C_Busy	Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

28.5.5.26 status_t LPI2C_MasterTransferGetCount (LPI2C_Type * *base*, lpi2c_master_handle_t * *handle*, size_t * *count*)

Parameters

	base	The LPI2C peripheral base address.
	handle	Pointer to the LPI2C master driver handle.
out	*count*	Number of bytes transferred so far by the non-blocking transaction.

Return values

#kStatus_Success	
#kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

28.5.5.27 void LPI2C_MasterTransferAbort (LPI2C_Type * *base*, lpi2c_master_handle_t * *handle*)

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.

Return values

#kStatus_Success	A transaction was successfully aborted.
kStatus_LPI2C_Idle	There is not a non-blocking transaction currently in progress.

28.5.5.28 void LPI2C_MasterTransferHandleIRQ (LPI2C_Type * *base*, lpi2c_master_handle_t * *handle*)

Note

This function does not need to be called unless you are reimplementing the nonblocking API's interrupt handler routines to add special functionality.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.

28.6 LPI2C Master DMA Driver

28.6.1 Overview

Data Structures

- struct lpi2c_master_edma_handle_t
  Driver handle for master DMA APIs. More...

Typedefs

- typedef void(∗ lpi2c_master_edma_transfer_callback_t)(LPI2C_Type ∗base, lpi2c_master_edma_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Master DMA completion callback function pointer type.

Master DMA

- void LPI2C_MasterCreateEDMAHandle (LPI2C_Type ∗base, lpi2c_master_edma_handle_t ∗handle, edma_handle_t ∗rxDmaHandle, edma_handle_t ∗txDmaHandle, lpi2c_master_edma_transfer_callback_t callback, void ∗userData)
  Create a new handle for the LPI2C master DMA APIs.
- status_t LPI2C_MasterTransferEDMA (LPI2C_Type ∗base, lpi2c_master_edma_handle_t ∗handle, lpi2c_master_transfer_t ∗transfer)
  Performs a non-blocking DMA-based transaction on the I2C bus.
- status_t LPI2C_MasterTransferGetCountEDMA (LPI2C_Type ∗base, lpi2c_master_edma_handle_t ∗handle, size_t ∗count)
  Returns number of bytes transferred so far.
- status_t LPI2C_MasterTransferAbortEDMA (LPI2C_Type ∗base, lpi2c_master_edma_handle_t ∗handle)
  Terminates a non-blocking LPI2C master transmission early.

28.6.2 Data Structure Documentation

28.6.2.1 struct _lpi2c_master_edma_handle

Note

The contents of this structure are private and subject to change.

Data Fields

- LPI2C_Type ∗ base
  LPI2C base pointer.
- bool isBusy

Transfer state machine current state.
- uint16_t commandBuffer [7]
  LPI2C command sequence.
- lpi2c_master_transfer_t transfer
  Copy of the current transfer info.
- lpi2c_master_edma_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.
- edma_handle_t * rx
  Handle for receive DMA channel.
- edma_handle_t * tx
  Handle for transmit DMA channel.
- edma_tcd_t tcds [2]
  Software TCD.

28.6.2.1.0.28 Field Documentation

28.6.2.1.0.28.1 LPI2C_Type* lpi2c_master_edma_handle_t::base

28.6.2.1.0.28.2 bool lpi2c_master_edma_handle_t::isBusy

28.6.2.1.0.28.3 uint16_t lpi2c_master_edma_handle_t::commandBuffer[7]

28.6.2.1.0.28.4 lpi2c_master_transfer_t lpi2c_master_edma_handle_t::transfer

28.6.2.1.0.28.5 lpi2c_master_edma_transfer_callback_t lpi2c_master_edma_handle_t::completionCallback

28.6.2.1.0.28.6 void* lpi2c_master_edma_handle_t::userData

28.6.2.1.0.28.7 edma_handle_t* lpi2c_master_edma_handle_t::rx

28.6.2.1.0.28.8 edma_handle_t* lpi2c_master_edma_handle_t::tx

28.6.2.1.0.28.9 edma_tcd_t lpi2c_master_edma_handle_t::tcds[2]

Two are allocated to provide enough room to align to 32-bytes.

28.6.3 Typedef Documentation

28.6.3.1 typedef void(* lpi2c_master_edma_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_edma_handle_t *handle, status_t completionStatus, void *userData)

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterCreateEDMAHandle().

Parameters

base	The LPI2C peripheral base address.
handle	Handle associated with the completed transfer.
completion-Status	Either #kStatus_Success or an error code describing how the transfer completed.
userData	Arbitrary pointer-sized value passed from the application.

28.6.4 Function Documentation

28.6.4.1 void LPI2C_MasterCreateEDMAHandle (LPI2C_Type * *base*, lpi2c_master_-edma_handle_t * *handle*, edma_handle_t * *rxDmaHandle*, edma_handle_t * *txDmaHandle*, lpi2c_master_edma_transfer_callback_t *callback*, void * *userData*)

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_MasterTransferAbortEDMA() API shall be called.

For devices where the LPI2C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

	base	The LPI2C peripheral base address.
out	*handle*	Pointer to the LPI2C master driver handle.
	rxDmaHandle	Handle for the eDMA receive channel. Created by the user prior to calling this function.
	txDmaHandle	Handle for the eDMA transmit channel. Created by the user prior to calling this function.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

28.6.4.2 status_t LPI2C_MasterTransferEDMA (LPI2C_Type * *base*, lpi2c_-master_edma_handle_t * *handle*, lpi2c_master_transfer_t * *transfer*)

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.
transfer	The pointer to the transfer descriptor.

Return values

#kStatus_Success	The transaction was started successfully.
kStatus_LPI2C_Busy	Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

28.6.4.3 status_t LPI2C_MasterTransferGetCountEDMA (LPI2C_Type * *base*, lpi2c_master_edma_handle_t * *handle*, size_t * *count*)

Parameters

	base	The LPI2C peripheral base address.
	handle	Pointer to the LPI2C master driver handle.
out	*count*	Number of bytes transferred so far by the non-blocking transaction.

Return values

#kStatus_Success	
#kStatus_NoTransferInProgress	There is not a DMA transaction currently in progress.

28.6.4.4 status_t LPI2C_MasterTransferAbortEDMA (LPI2C_Type * *base*, lpi2c_master_edma_handle_t * *handle*)

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the eDMA peripheral's IRQ priority.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.

Return values

#kStatus_Success	A transaction was successfully aborted.
kStatus_LPI2C_Idle	There is not a DMA transaction currently in progress.

28.7 LPI2C Slave Driver

28.7.1 Overview

Data Structures

- struct lpi2c_slave_config_t
  Structure with settings to initialize the LPI2C slave module. More...
- struct lpi2c_slave_transfer_t
  LPI2C slave transfer structure. More...
- struct lpi2c_slave_handle_t
  LPI2C slave handle structure. More...

Typedefs

- typedef void(∗ lpi2c_slave_transfer_callback_t)(LPI2C_Type ∗base, lpi2c_slave_transfer_t ∗transfer, void ∗userData)
  Slave event callback function pointer type.

Enumerations

- enum _lpi2c_slave_flags {
  kLPI2C_SlaveTxReadyFlag = LPI2C_SSR_TDF_MASK,
  kLPI2C_SlaveRxReadyFlag = LPI2C_SSR_RDF_MASK,
  kLPI2C_SlaveAddressValidFlag = LPI2C_SSR_AVF_MASK,
  kLPI2C_SlaveTransmitAckFlag = LPI2C_SSR_TAF_MASK,
  kLPI2C_SlaveRepeatedStartDetectFlag = LPI2C_SSR_RSF_MASK,
  kLPI2C_SlaveStopDetectFlag = LPI2C_SSR_SDF_MASK,
  kLPI2C_SlaveBitErrFlag = LPI2C_SSR_BEF_MASK,
  kLPI2C_SlaveFifoErrFlag = LPI2C_SSR_FEF_MASK,
  kLPI2C_SlaveAddressMatch0Flag = LPI2C_SSR_AM0F_MASK,
  kLPI2C_SlaveAddressMatch1Flag = LPI2C_SSR_AM1F_MASK,
  kLPI2C_SlaveGeneralCallFlag = LPI2C_SSR_GCF_MASK,
  kLPI2C_SlaveBusyFlag = LPI2C_SSR_SBF_MASK,
  kLPI2C_SlaveBusBusyFlag = LPI2C_SSR_BBF_MASK }
  LPI2C slave peripheral flags.
- enum lpi2c_slave_address_match_t {
  kLPI2C_MatchAddress0 = 0U,
  kLPI2C_MatchAddress0OrAddress1 = 2U,
  kLPI2C_MatchAddress0ThroughAddress1 = 6U }
  LPI2C slave address match options.
- enum lpi2c_slave_transfer_event_t {

kLPI2C_SlaveAddressMatchEvent = 0x01U,
kLPI2C_SlaveTransmitEvent = 0x02U,
kLPI2C_SlaveReceiveEvent = 0x04U,
kLPI2C_SlaveTransmitAckEvent = 0x08U,
kLPI2C_SlaveRepeatedStartEvent = 0x10U,
kLPI2C_SlaveCompletionEvent = 0x20U,
kLPI2C_SlaveAllEvents }
Set of events sent to the callback for non blocking slave transfers.

Slave initialization and deinitialization

- void LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t *slaveConfig)
  Provides a default configuration for the LPI2C slave peripheral.
- void LPI2C_SlaveInit (LPI2C_Type *base, const lpi2c_slave_config_t *slaveConfig, uint32_t sourceClock_Hz)
  Initializes the LPI2C slave peripheral.
- void LPI2C_SlaveDeinit (LPI2C_Type *base)
  Deinitializes the LPI2C slave peripheral.
- static void LPI2C_SlaveReset (LPI2C_Type *base)
  Performs a software reset of the LPI2C slave peripheral.
- static void LPI2C_SlaveEnable (LPI2C_Type *base, bool enable)
  Enables or disables the LPI2C module as slave.

Slave status

- static uint32_t LPI2C_SlaveGetStatusFlags (LPI2C_Type *base)
  Gets the LPI2C slave status flags.
- static void LPI2C_SlaveClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)
  Clears the LPI2C status flag state.

Slave interrupts

- static void LPI2C_SlaveEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Enables the LPI2C slave interrupt requests.
- static void LPI2C_SlaveDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Disables the LPI2C slave interrupt requests.
- static uint32_t LPI2C_SlaveGetEnabledInterrupts (LPI2C_Type *base)
  Returns the set of currently enabled LPI2C slave interrupt requests.

Slave DMA control

- static void LPI2C_SlaveEnableDMA (LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)
  Enables or disables the LPI2C slave peripheral DMA requests.

Slave bus operations

- static bool LPI2C_SlaveGetBusIdleState (LPI2C_Type *base)
  Returns whether the bus is idle.
- static void LPI2C_SlaveTransmitAck (LPI2C_Type *base, bool ackOrNack)
  Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.
- static uint32_t LPI2C_SlaveGetReceivedAddress (LPI2C_Type *base)
  Returns the slave address sent by the I2C master.
- status_t LPI2C_SlaveSend (LPI2C_Type *base, const void *txBuff, size_t txSize, size_t *actualTxSize)
  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_SlaveReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)
  Performs a polling receive transfer on the I2C bus.

Slave non-blocking

- void LPI2C_SlaveTransferCreateHandle (LPI2C_Type *base, lpi2c_slave_handle_t *handle, lpi2c_slave_transfer_callback_t callback, void *userData)
  Creates a new handle for the LPI2C slave non-blocking APIs.
- status_t LPI2C_SlaveTransferNonBlocking (LPI2C_Type *base, lpi2c_slave_handle_t *handle, uint32_t eventMask)
  Starts accepting slave transfers.
- status_t LPI2C_SlaveTransferGetCount (LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t *count)
  Gets the slave transfer status during a non-blocking transfer.
- void LPI2C_SlaveTransferAbort (LPI2C_Type *base, lpi2c_slave_handle_t *handle)
  Aborts the slave non-blocking transfers.

Slave IRQ handler

- void LPI2C_SlaveTransferHandleIRQ (LPI2C_Type *base, lpi2c_slave_handle_t *handle)
  Reusable routine to handle slave interrupts.

28.7.2 Data Structure Documentation

28.7.2.1 struct lpi2c_slave_config_t

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

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

Data Fields

- bool enableSlave
  Enable slave mode.
- uint8_t address0
  Slave's 7-bit address.
- uint8_t address1
  Alternate slave 7-bit address.
- lpi2c_slave_address_match_t addressMatchMode
  Address matching options.
- bool filterDozeEnable
  Enable digital glitch filter in doze mode.
- bool filterEnable
  Enable digital glitch filter.
- bool enableGeneralCall
  Enable general call address matching.
- bool ignoreAck
  Continue transfers after a NACK is detected.
- bool enableReceivedAddressRead
  Enable reading the address received address as the first byte of data.
- uint32_t sdaGlitchFilterWidth_ns
  Width in nanoseconds of the digital filter on the SDA signal.
- uint32_t sclGlitchFilterWidth_ns
  Width in nanoseconds of the digital filter on the SCL signal.
- uint32_t dataValidDelay_ns
  Width in nanoseconds of the data valid delay.
- uint32_t clockHoldTime_ns
  Width in nanoseconds of the clock hold time.
- bool enableAck
  Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted.
- bool enableTx
  Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.
- bool enableRx
  Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.
- bool enableAddress
  Enables SCL clock stretching when the address valid flag is asserted.

28.7.2.1.0.29 Field Documentation

28.7.2.1.0.29.1 bool lpi2c_slave_config_t::enableSlave

28.7.2.1.0.29.2 uint8_t lpi2c_slave_config_t::address0

28.7.2.1.0.29.3 uint8_t lpi2c_slave_config_t::address1

28.7.2.1.0.29.4 lpi2c_slave_address_match_t lpi2c_slave_config_t::addressMatchMode

28.7.2.1.0.29.5 bool lpi2c_slave_config_t::filterDozeEnable

28.7.2.1.0.29.6 bool lpi2c_slave_config_t::filterEnable

28.7.2.1.0.29.7 bool lpi2c_slave_config_t::enableGeneralCall

28.7.2.1.0.29.8 bool lpi2c_slave_config_t::enableAck

Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

28.7.2.1.0.29.9 bool lpi2c_slave_config_t::enableTx

28.7.2.1.0.29.10 bool lpi2c_slave_config_t::enableRx

28.7.2.1.0.29.11 bool lpi2c_slave_config_t::enableAddress

28.7.2.1.0.29.12 bool lpi2c_slave_config_t::ignoreAck

28.7.2.1.0.29.13 bool lpi2c_slave_config_t::enableReceivedAddressRead

28.7.2.1.0.29.14 uint32_t lpi2c_slave_config_t::sdaGlitchFilterWidth_ns

28.7.2.1.0.29.15 uint32_t lpi2c_slave_config_t::sclGlitchFilterWidth_ns

28.7.2.1.0.29.16 uint32_t lpi2c_slave_config_t::dataValidDelay_ns

28.7.2.1.0.29.17 uint32_t lpi2c_slave_config_t::clockHoldTime_ns

28.7.2.2 struct lpi2c_slave_transfer_t

Data Fields

- lpi2c_slave_transfer_event_t event
  Reason the callback is being invoked.
- uint8_t receivedAddress
  Matching address send by master.
- uint8_t * data
  Transfer buffer.
- size_t dataSize
  Transfer size.

- status_t completionStatus
  Success or error code describing how the transfer completed.
- size_t transferredCount
  Number of bytes actually transferred since start or last repeated start.

28.7.2.2.0.30 Field Documentation

28.7.2.2.0.30.1 lpi2c_slave_transfer_event_t lpi2c_slave_transfer_t::event

28.7.2.2.0.30.2 uint8_t lpi2c_slave_transfer_t::receivedAddress

28.7.2.2.0.30.3 status_t lpi2c_slave_transfer_t::completionStatus

Only applies for kLPI2C_SlaveCompletionEvent.

28.7.2.2.0.30.4 size_t lpi2c_slave_transfer_t::transferredCount

28.7.2.3 struct _lpi2c_slave_handle

Note

The contents of this structure are private and subject to change.

Data Fields

- lpi2c_slave_transfer_t transfer
  LPI2C slave transfer copy.
- bool isBusy
  Whether transfer is busy.
- bool wasTransmit
  Whether the last transfer was a transmit.
- uint32_t eventMask
  Mask of enabled events.
- uint32_t transferredCount
  Count of bytes transferred.
- lpi2c_slave_transfer_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback.

28.7.2.3.0.31 Field Documentation

28.7.2.3.0.31.1 lpi2c_slave_transfer_t lpi2c_slave_handle_t::transfer

28.7.2.3.0.31.2 bool lpi2c_slave_handle_t::isBusy

28.7.2.3.0.31.3 bool lpi2c_slave_handle_t::wasTransmit

28.7.2.3.0.31.4 uint32_t lpi2c_slave_handle_t::eventMask

28.7.2.3.0.31.5 uint32_t lpi2c_slave_handle_t::transferredCount

28.7.2.3.0.31.6 lpi2c_slave_transfer_callback_t lpi2c_slave_handle_t::callback

28.7.2.3.0.31.7 void* lpi2c_slave_handle_t::userData

28.7.3 Typedef Documentation

28.7.3.1 typedef void(* lpi2c_slave_transfer_callback_t)(LPI2C_Type *base, lpi2c_slave_transfer_t *transfer, void *userData)

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_-SlaveSetCallback() function after you have created a handle.

Parameters

base	Base address for the LPI2C instance on which the event occurred.
transfer	Pointer to transfer descriptor containing values passed to and/or from the callback.
userData	Arbitrary pointer-sized value passed from the application.

28.7.4 Enumeration Type Documentation

28.7.4.1 enum _lpi2c_slave_flags

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Note

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

Enumerator

kLPI2C_SlaveTxReadyFlag Transmit data flag.
kLPI2C_SlaveRxReadyFlag Receive data flag.
kLPI2C_SlaveAddressValidFlag Address valid flag.
kLPI2C_SlaveTransmitAckFlag Transmit ACK flag.
kLPI2C_SlaveRepeatedStartDetectFlag Repeated start detect flag.
kLPI2C_SlaveStopDetectFlag Stop detect flag.
kLPI2C_SlaveBitErrFlag Bit error flag.
kLPI2C_SlaveFifoErrFlag FIFO error flag.
kLPI2C_SlaveAddressMatch0Flag Address match 0 flag.
kLPI2C_SlaveAddressMatch1Flag Address match 1 flag.
kLPI2C_SlaveGeneralCallFlag General call flag.
kLPI2C_SlaveBusyFlag Master busy flag.
kLPI2C_SlaveBusBusyFlag Bus busy flag.

28.7.4.2 enum lpi2c_slave_address_match_t

Enumerator

kLPI2C_MatchAddress0 Match only address 0.

kLPI2C_MatchAddress0OrAddress1 Match either address 0 or address 1.
kLPI2C_MatchAddress0ThroughAddress1 Match a range of slave addresses from address 0 through address 1.

28.7.4.3 enum lpi2c_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 LPI2C_SlaveTransferNonBlocking() in order 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

kLPI2C_SlaveAddressMatchEvent Received the slave address after a start or repeated start.
kLPI2C_SlaveTransmitEvent Callback is requested to provide data to transmit (slave-transmitter role).
kLPI2C_SlaveReceiveEvent Callback is requested to provide a buffer in which to place received data (slave-receiver role).
kLPI2C_SlaveTransmitAckEvent Callback needs to either transmit an ACK or NACK.
kLPI2C_SlaveRepeatedStartEvent A repeated start was detected.
kLPI2C_SlaveCompletionEvent A stop was detected, completing the transfer.
kLPI2C_SlaveAllEvents Bit mask of all available events.

28.7.5 Function Documentation

28.7.5.1 void LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t * *slaveConfig*)

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave              = true;
slaveConfig->address0                 = 0U;
slaveConfig->address1                 = 0U;
slaveConfig->addressMatchMode         = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable         = true;
slaveConfig->filterEnable             = true;
slaveConfig->enableGeneralCall        = false;
slaveConfig->sclStall.enableAck       = false;
slaveConfig->sclStall.enableTx        = true;
slaveConfig->sclStall.enableRx        = true;
slaveConfig->sclStall.enableAddress   = true;
slaveConfig->ignoreAck                = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns  = 0; // TODO determine default width values
slaveConfig->sclGlitchFilterWidth_ns  = 0;
slaveConfig->dataValidDelay_ns        = 0;
slaveConfig->clockHoldTime_ns         = 0;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with LPI2C_SlaveInit(). Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

out	*slaveConfig*	User provided configuration structure that is set to default values. Refer to lpi2c_slave_config_t.

28.7.5.2 void LPI2C_SlaveInit (LPI2C_Type * *base*, const lpi2c_slave_config_t * *slaveConfig*, uint32_t *sourceClock_Hz*)

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

base	The LPI2C peripheral base address.
slaveConfig	User provided peripheral configuration. Use LPI2C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
sourceClock_Hz	Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

28.7.5.3 void LPI2C_SlaveDeinit (LPI2C_Type * *base*)

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

28.7.5.4 static void LPI2C_SlaveReset (LPI2C_Type * *base*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.

28.7.5.5 static void LPI2C_SlaveEnable (LPI2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.
enable	Pass true to enable or false to disable the specified LPI2C as slave.

28.7.5.6 static uint32_t LPI2C_SlaveGetStatusFlags (LPI2C_Type * *base*) [inline], [static]

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

State of the status flags:
- 1: related status flag is set.
- 0: related status flag is not set.

See Also

_lpi2c_slave_flags

28.7.5.7 static void LPI2C_SlaveClearStatusFlags (LPI2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

Attempts to clear other flags has no effect.

Parameters

base	The LPI2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_SlaveGetStatusFlags().

See Also

_lpi2c_slave_flags.

28.7.5.8 static void LPI2C_SlaveEnableInterrupts (LPI2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

28.7.5.9 static void LPI2C_SlaveDisableInterrupts (LPI2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

28.7.5.10 static uint32_t LPI2C_SlaveGetEnabledInterrupts (LPI2C_Type * *base*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_slave_flags enumerators OR'd together to indicate the set of enabled interrupts.

28.7.5.11 static void LPI2C_SlaveEnableDMA (LPI2C_Type * *base*, bool *enableAddressValid*, bool *enableRx*, bool *enableTx*) [inline], [static]

Parameters

base	The LPI2C peripheral base address.
enableAddress-Valid	Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
enableRx	Enable flag for the receive data DMA request. Pass true for enable, false for disable.
enableTx	Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

28.7.5.12 static bool LPI2C_SlaveGetBusIdleState (LPI2C_Type * *base*) [inline], [static]

Requires the slave mode to be enabled.

Parameters

base	The LPI2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

28.7.5.13 static void LPI2C_SlaveTransmitAck (LPI2C_Type * *base*, bool *ackOrNack*) [inline], [static]

Use this function to send an ACK or NAK when the kLPI2C_SlaveTransmitAckFlag is asserted. This only happens if you enable the sclStall.enableAck field of the lpi2c_slave_config_t configuration structure used to initialize the slave peripheral.

Parameters

base	The LPI2C peripheral base address.
ackOrNack	Pass true for an ACK or false for a NAK.

28.7.5.14 static uint32_t LPI2C_SlaveGetReceivedAddress (LPI2C_Type * *base*) [inline], [static]

This function should only be called if the kLPI2C_SlaveAddressValidFlag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

28.7.5.15 status_t LPI2C_SlaveSend (LPI2C_Type * *base*, const void * *txBuff*, size_t *txSize*, size_t * *actualTxSize*)

Parameters

	base	The LPI2C peripheral base address.
	txBuff	The pointer to the data to be transferred.
	txSize	The length in bytes of the data to be transferred.
out	*actualTxSize*	

Returns

Error or success status returned by API.

28.7.5.16 status_t LPI2C_SlaveReceive (LPI2C_Type * *base*, void * *rxBuff*, size_t *rxSize*, size_t * *actualRxSize*)

Parameters

	base	The LPI2C peripheral base address.
	rxBuff	The pointer to the data to be transferred.
	rxSize	The length in bytes of the data to be transferred.
out	*actualRxSize*	

Returns

Error or success status returned by API.

28.7.5.17 void LPI2C_SlaveTransferCreateHandle (LPI2C_Type * *base*, lpi2c_slave_handle_t * *handle*, lpi2c_slave_transfer_callback_t *callback*, void * *userData*)

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_SlaveTransferAbort() API shall be called.

Parameters

	base	The LPI2C peripheral base address.
out	*handle*	Pointer to the LPI2C slave driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

28.7.5.18 status_t LPI2C_SlaveTransferNonBlocking (LPI2C_Type * *base*, lpi2c_slave_handle_t * *handle*, uint32_t *eventMask*)

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_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 lpi2c_slave_transfer_event_t enumerators for the events you wish to receive. The kLPI2C_SlaveTransmitEvent and kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kLPI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.
eventMask	Bit mask formed by OR'ing together lpi2c_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 kLPI2C_SlaveAllEvents to enable all events.

Return values

#kStatus_Success	Slave transfers were successfully started.
kStatus_LPI2C_Busy	Slave transfers have already been started on this handle.

28.7.5.19 status_t LPI2C_SlaveTransferGetCount (LPI2C_Type * *base*, lpi2c_slave_handle_t * *handle*, size_t * *count*)

Parameters

	base	The LPI2C peripheral base address.
	handle	Pointer to i2c_slave_handle_t structure.
out	*count*	Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

#kStatus_Success	
#kStatus_NoTransferInProgress	

28.7.5.20 void LPI2C_SlaveTransferAbort (LPI2C_Type * *base*, lpi2c_slave_handle_t * *handle*)

Note

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

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.

Return values

#kStatus_Success	
kStatus_LPI2C_Idle	

28.7.5.21 void LPI2C_SlaveTransferHandleIRQ (LPI2C_Type * *base*, lpi2c_slave_handle_t * *handle*)

Note

This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.

28.8 LPI2C Slave DMA Driver

28.9 LPI2C µCOS/II Driver

28.9.1 Overview

Files

- file fsl_lpi2c_ucosii.h

Data Structures

- struct lpi2c_rtos_handle_t
  LPI2C FreeRTOS handle. More...

Driver version

- #define FSL_LPI2C_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPI2C uCOS II driver version 2.0.0.

LPI2C RTOS Operation

- status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t *handle, LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes LPI2C.
- status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t *handle)
  Deinitializes the LPI2C.
- status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t *handle, lpi2c_master_transfer_t *transfer)
  Performs I2C transfer.

28.9.2 Data Structure Documentation

28.9.2.1 struct lpi2c_rtos_handle_t

LPI2C uCOS III handle.

LPI2C uCOS II handle.

Data Fields

- LPI2C_Type * base
  LPI2C base address.
- lpi2c_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a trasfer.

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

28.9.3 Macro Definition Documentation

28.9.3.1 #define FSL_LPI2C_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

28.9.4 Function Documentation

28.9.4.1 status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t * *handle*, LPI2C_Type * *base*, const lpi2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the LPI2C instance to initialize.
masterConfig	Configuration structure to set-up LPI2C in master mode.
srcClock_Hz	Frequency of input clock of the LPI2C module.

Returns

status of the operation.

28.9.4.2 status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t * *handle*)

This function deinitializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle.

28.9.4.3 status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t * *handle*, lpi2c_master_transfer_t * *transfer*)

This function performs an I2C transfer using LPI2C module according to data given in the transfer structure.

Parameters

handle	The RTOS LPI2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

28.10 LPI2C µCOS/III Driver

28.10.1 Overview

Files

- file fsl_lpi2c_ucosiii.h

Data Structures

- struct lpi2c_rtos_handle_t
  LPI2C FreeRTOS handle. More...

Driver version

- #define FSL_LPI2C_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPI2C uCOS III driver version 2.0.0.

LPI2C RTOS Operation

- status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t *handle, LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes LPI2C.
- status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t *handle)
  Deinitializes the LPI2C.
- status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t *handle, lpi2c_master_transfer_t *transfer)
  Performs I2C transfer.

28.10.2 Data Structure Documentation

28.10.2.1 struct lpi2c_rtos_handle_t

LPI2C uCOS III handle.

LPI2C uCOS II handle.

Data Fields

- LPI2C_Type * base
  LPI2C base address.
- lpi2c_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.
- SemaphoreHandle_t mutex
  Mutex to lock the handle during a trasfer.

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

28.10.3 Macro Definition Documentation

28.10.3.1 #define FSL_LPI2C_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

28.10.4 Function Documentation

28.10.4.1 status_t LPI2C_RTOS_Init (lpi2c_rtos_handle_t * *handle*, LPI2C_Type * *base*, const lpi2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the LPI2C instance to initialize.
masterConfig	Configuration structure to set-up LPI2C in master mode.
srcClock_Hz	Frequency of input clock of the LPI2C module.

Returns

status of the operation.

28.10.4.2 status_t LPI2C_RTOS_Deinit (lpi2c_rtos_handle_t * *handle*)

This function deinitializes the LPI2C module and related RTOS context.

Parameters

handle	The RTOS LPI2C handle.

28.10.4.3 status_t LPI2C_RTOS_Transfer (lpi2c_rtos_handle_t * *handle*, lpi2c_master_transfer_t * *transfer*)

This function performs an I2C transfer using LPI2C module according to data given in the transfer structure.

Parameters

handle	The RTOS LPI2C handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 29
LPTMR: LPTMR Driver

29.1 Overview

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

Files

- file fsl_lptmr.h

Data Structures

- struct lptmr_config_t
  LPTMR config structure. More...

Enumerations

- enum lptmr_pin_select_t {
  kLPTMR_PinSelectInput_0 = 0x0U,
  kLPTMR_PinSelectInput_1 = 0x1U,
  kLPTMR_PinSelectInput_2 = 0x2U,
  kLPTMR_PinSelectInput_3 = 0x3U }
  LPTMR pin selection, used in pulse counter mode.
- enum lptmr_pin_polarity_t {
  kLPTMR_PinPolarityActiveHigh = 0x0U,
  kLPTMR_PinPolarityActiveLow = 0x1U }
  LPTMR pin polarity, used in pulse counter mode.
- enum lptmr_timer_mode_t {
  kLPTMR_TimerModeTimeCounter = 0x0U,
  kLPTMR_TimerModePulseCounter = 0x1U }
  LPTMR timer mode selection.
- enum lptmr_prescaler_glitch_value_t {

kLPTMR_Prescale_Glitch_0 = 0x0U,
kLPTMR_Prescale_Glitch_1 = 0x1U,
kLPTMR_Prescale_Glitch_2 = 0x2U,
kLPTMR_Prescale_Glitch_3 = 0x3U,
kLPTMR_Prescale_Glitch_4 = 0x4U,
kLPTMR_Prescale_Glitch_5 = 0x5U,
kLPTMR_Prescale_Glitch_6 = 0x6U,
kLPTMR_Prescale_Glitch_7 = 0x7U,
kLPTMR_Prescale_Glitch_8 = 0x8U,
kLPTMR_Prescale_Glitch_9 = 0x9U,
kLPTMR_Prescale_Glitch_10 = 0xAU,
kLPTMR_Prescale_Glitch_11 = 0xBU,
kLPTMR_Prescale_Glitch_12 = 0xCU,
kLPTMR_Prescale_Glitch_13 = 0xDU,
kLPTMR_Prescale_Glitch_14 = 0xEU,
kLPTMR_Prescale_Glitch_15 = 0xFU }
LPTMR prescaler/glitch filter values.

- enum lptmr_prescaler_clock_select_t {
  kLPTMR_PrescalerClock_0 = 0x0U,
  kLPTMR_PrescalerClock_1 = 0x1U,
  kLPTMR_PrescalerClock_2 = 0x2U,
  kLPTMR_PrescalerClock_3 = 0x3U }
  LPTMR prescaler/glitch filter clock select.
- enum lptmr_interrupt_enable_t { kLPTMR_TimerInterruptEnable = LPTMR_CSR_TIE_MASK }
  List of LPTMR interrupts.
- enum lptmr_status_flags_t { kLPTMR_TimerCompareFlag = LPTMR_CSR_TCF_MASK }
  List of 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)
  Ungate the LPTMR clock and configures the peripheral for basic operation.
- void LPTMR_Deinit (LPTMR_Type *base)
  Gate the LPTMR clock.
- void LPTMR_GetDefaultConfig (lptmr_config_t *config)
  Fill in the LPTMR config struct with the 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 counting.
- static void LPTMR_StopTimer (LPTMR_Type *base)
  Stops the timer counting.

29.2 Data Structure Documentation

29.2.1 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 config structure instance.

The config struct can be made const 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.

29.3 Enumeration Type Documentation

29.3.1 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.

29.3.2 enum lptmr_pin_polarity_t

Enumerator

kLPTMR_PinPolarityActiveHigh Pulse Counter input source is active-high.
kLPTMR_PinPolarityActiveLow Pulse Counter input source is active-low.

29.3.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.
kLPTMR_TimerModePulseCounter Pulse Counter mode.

29.3.4 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.

29.3.5 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.

29.3.6 enum lptmr_interrupt_enable_t

Enumerator

kLPTMR_TimerInterruptEnable Timer interrupt enable.

29.3.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

29.4 Function Documentation

29.4.1 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	Pointer to user's LPTMR config structure.

29.4.2 void LPTMR_Deinit (LPTMR_Type * *base*)

Parameters

base	LPTMR peripheral base address

29.4.3 void LPTMR_GetDefaultConfig (lptmr_config_t * *config*)

The default values are:

```
config->timerMode = kLPTMR_TimerModeTimeCounter;
config->pinSelect = kLPTMR_PinSelectInput_0;
config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
config->enableFreeRunning = false;
config->bypassPrescaler = true;
config->prescalerClockSource = kLPTMR_PrescalerClock_1;
config->value = kLPTMR_Prescale_Glitch_0;
```

Parameters

config	Pointer to user's LPTMR config structure.

29.4.4 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

29.4.5 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

29.4.6 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

29.4.7 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

29.4.8 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

29.4.9 static void LPTMR_SetTimerPeriod (LPTMR_Type * *base,* uint16_t *ticks*) [inline], [static]

Timers counts from 0 till 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. User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	LPTMR peripheral base address
ticks	Timer period in units of ticks

29.4.10 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

User can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	LPTMR peripheral base address

Returns

Current counter value in ticks

29.4.11 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 CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt will also be triggered if the timer interrupt is enabled.

Parameters

base	LPTMR peripheral base address

29.4.12 static void LPTMR_StopTimer (LPTMR_Type * *base*) [inline], [static]

This function stops the timer counting and resets the timer's counter register

Parameters

base	LPTMR peripheral base address

Chapter 30
LPUART: Low Power UART Driver

30.1 Overview

Modules

- LPUART Driver
- LPUART FreeRTOS Driver
- LPUART µCOS/II Driver
- LPUART µCOS/III Driver

30.2 LPUART Driver

30.2.1 Overview

The KSDK provides a peripheral driver for the Low Power UART (LPUART) module of Kinetis devices.

Typical use case

```
uint8_t ch;
LPUART_GetDefaultConfig(&user_config);
user_config.baudRate = 115200U;

LPUART_Init(LPUART1,&user_config,120000000U);

LPUART_SendDataPolling(LPUART1, txbuff, sizeof(txbuff));

while(1)
{
    LPUART_ReceiveDataPolling(LPUART1,&ch, 1);
    LPUART_SendDataPolling(LPUART1, &ch, 1);
}
```

Modules

- LPUART DMA Driver
- LPUART eDMA Driver

Files

- file fsl_lpuart.h

Data Structures

- struct lpuart_config_t
  LPUART configure 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,
  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) }
  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_stop_bit_count_t {
  kLPUART_OneStopBit = 0U,
  kLPUART_TwoStopBit = 1U }
  LPUART stop bit count.
- enum _lpuart_interrupt_enable {
  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) }
  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_RxActiveEdgeFlag,
kLPUART_RxActiveFlag }
LPUART status flags.

Driver version

- #define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  LPUART driver version 2.1.0.

Initialization and deinitialization

- void 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.
- void 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.

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 RX register.
- void LPUART_WriteBlocking (LPUART_Type *base, const uint8_t *data, size_t length)
  Writes to transmitter register using a blocking method.
- status_t LPUART_ReadBlocking (LPUART_Type *base, uint8_t *data, size_t length)
  Reads the RX 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)
  Abort the background transfer and uninstall 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)
  Get the number of bytes that have been written to LPUART TX 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)
  Get 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.

30.2.2 Data Structure Documentation

30.2.2.1 struct lpuart_config_t

Data Fields

- uint32_t baudRate_Bps
  LPUART baud rate.
- lpuart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- bool enableTx
  Enable TX.
- bool enableRx
  Enable RX.

30.2.2.2 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.

30.2.2.2.0.32 Field Documentation

30.2.2.2.0.32.1 uint8_t* lpuart_transfer_t::data

30.2.2.2.0.32.2 size_t lpuart_transfer_t::dataSize

30.2.2.3 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.

30.2.2.3.0.33 Field Documentation

30.2.2.3.0.33.1 uint8_t* volatile lpuart_handle_t::txData

30.2.2.3.0.33.2 volatile size_t lpuart_handle_t::txDataSize

30.2.2.3.0.33.3 size_t lpuart_handle_t::txDataSizeAll

30.2.2.3.0.33.4 uint8_t* volatile lpuart_handle_t::rxData

30.2.2.3.0.33.5 volatile size_t lpuart_handle_t::rxDataSize

30.2.2.3.0.33.6 size_t lpuart_handle_t::rxDataSizeAll

30.2.2.3.0.33.7 uint8_t* lpuart_handle_t::rxRingBuffer

30.2.2.3.0.33.8 size_t lpuart_handle_t::rxRingBufferSize

30.2.2.3.0.33.9 volatile uint16_t lpuart_handle_t::rxRingBufferHead

30.2.2.3.0.33.10 volatile uint16_t lpuart_handle_t::rxRingBufferTail

30.2.2.3.0.33.11 lpuart_transfer_callback_t lpuart_handle_t::callback

30.2.2.3.0.33.12 void* lpuart_handle_t::userData

30.2.2.3.0.33.13 volatile uint8_t lpuart_handle_t::txState

30.2.3 Macro Definition Documentation

30.2.3.1 #define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

30.2.4 Typedef Documentation

30.2.4.1 typedef void(* lpuart_transfer_callback_t)(LPUART_Type *base, lpuart_handle_t *handle, status_t status, void *userData)

30.2.5 Enumeration Type Documentation

30.2.5.1 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.

30.2.5.2 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.

30.2.5.3 enum lpuart_stop_bit_count_t

Enumerator

kLPUART_OneStopBit One stop bit.
kLPUART_TwoStopBit Two stop bits.

30.2.5.4 enum _lpuart_interrupt_enable

This structure contains the settings for all LPUART interrupt configurations.

Enumerator

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.

30.2.5.5 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_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.

30.2.6 Function Documentation

30.2.6.1 void 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_GetDefaultConfig() 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.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.

30.2.6.2 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.

30.2.6.3 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->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

config	Pointer to a configuration structure.

30.2.6.4 void 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.

30.2.6.5 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.

30.2.6.6 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_Flag-CannotClearManually	The flag can't be cleared by this function but it is cleared automatically by hardware.
kStatus_Success	Status in the mask are cleared.

30.2.6.7 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.

30.2.6.8 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.

30.2.6.9 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.

30.2.6.10 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.

30.2.6.11 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.

30.2.6.12 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.

30.2.6.13 static uint8_t LPUART_ReadByte (LPUART_Type * *base*) [inline], [static]

This function reads data from the TX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

base	LPUART peripheral base address.

Returns

Data read from data register.

30.2.6.14 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 then 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.

30.2.6.15 status_t LPUART_ReadBlocking (LPUART_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the RX register, waits for the RX register full or RX FIFO has data then 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_RxHardwareOverrun	Receiver overrun happened while receiving data.
kStatus_LPUART_NoiseError	Noise error happened while receiving data.
kStatus_LPUART_FramingError	Framing error happened while receiving data.
kStatus_LPUART_ParityError	Parity error happened while receiving data.
kStatus_Success	Successfully received all data.

30.2.6.16 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.

30.2.6.17 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 TX, 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, refer to 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.

30.2.6.18 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.

30.2.6.19 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.

30.2.6.20 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 still not sent out.

Parameters

base	LPUART peripheral base address.
handle	LPUART handle pointer.

30.2.6.21 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 interrupt method.

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`;

30.2.6.22 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, refer to 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_RxBusy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

30.2.6.23 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.

30.2.6.24 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_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

30.2.6.25 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.

30.2.6.26 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.

30.2.7 LPUART DMA Driver

30.2.7.1 Overview

Files

- file fsl_lpuart_dma.h

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)
  Get the number of bytes that have been written to LPUART TX register.
- status_t LPUART_TransferGetReceiveCountDMA (LPUART_Type ∗base, lpuart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

30.2.7.2 Data Structure Documentation

30.2.7.2.1 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.

30.2.7.2.1.1 Field Documentation

30.2.7.2.1.1.1 lpuart_dma_transfer_callback_t lpuart_dma_handle_t::callback

30.2.7.2.1.1.2 void* lpuart_dma_handle_t::userData

30.2.7.2.1.1.3 size_t lpuart_dma_handle_t::rxDataSizeAll

30.2.7.2.1.1.4 size_t lpuart_dma_handle_t::txDataSizeAll

30.2.7.2.1.1.5 dma_handle_t* lpuart_dma_handle_t::txDmaHandle

30.2.7.2.1.1.6 dma_handle_t* lpuart_dma_handle_t::rxDmaHandle

30.2.7.2.1.1.7 volatile uint8_t lpuart_dma_handle_t::txState

30.2.7.3 Typedef Documentation

30.2.7.3.1 typedef void(* lpuart_dma_transfer_callback_t)(LPUART_Type *base, lpuart_dma_handle_t *handle, status_t status, void *userData)

30.2.7.4 Function Documentation

30.2.7.4.1 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.

30.2.7.4.2 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.

30.2.7.4.3 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.

30.2.7.4.4 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

30.2.7.4.5 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

30.2.7.4.6 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_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

30.2.7.4.7 status_t LPUART_TransferGetReceiveCountDMA (LPUART_Type * *base*, lpuart_dma_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_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

30.2.8 LPUART eDMA Driver

30.2.8.1 Overview

Files

- file fsl_lpuart_edma.h

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)
  Get the number of bytes that have been written to LPUART TX register.
- status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type ∗base, lpuart_edma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

30.2.8.2 Data Structure Documentation

30.2.8.2.1 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.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

30.2.8.2.1.1 Field Documentation

30.2.8.2.1.1.1 lpuart_edma_transfer_callback_t lpuart_edma_handle_t::callback

30.2.8.2.1.1.2 void* lpuart_edma_handle_t::userData

30.2.8.2.1.1.3 size_t lpuart_edma_handle_t::rxDataSizeAll

30.2.8.2.1.1.4 size_t lpuart_edma_handle_t::txDataSizeAll

30.2.8.2.1.1.5 edma_handle_t* lpuart_edma_handle_t::txEdmaHandle

30.2.8.2.1.1.6 edma_handle_t* lpuart_edma_handle_t::rxEdmaHandle

30.2.8.2.1.1.7 volatile uint8_t lpuart_edma_handle_t::txState

30.2.8.3 Typedef Documentation

30.2.8.3.1 typedef void(* lpuart_edma_transfer_callback_t)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)

30.2.8.4 Function Documentation

30.2.8.4.1 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.

30.2.8.4.2 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.

30.2.8.4.3 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, refer to lpuart_transfer_t.

Return values

kStatus_Success	if succeed, others fail.
kStatus_LPUART_RxBusy	Previous transfer ongoing.
kStatus_InvalidArgument	Invalid argument.

30.2.8.4.4 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.

30.2.8.4.5 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.

30.2.8.4.6 status_t LPUART_TransferGetSendCountEDMA (LPUART_Type * *base*, lpuart_edma_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_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

30.2.8.4.7 status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type * *base*, lpuart_edma_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_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

30.3 LPUART FreeRTOS Driver

30.3.1 Overview

Files

- file fsl_lpuart_freertos.h

Driver version

- #define FSL_LPUART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPUART FreeRTOS driver version 2.0.0.

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const struct rtos_lpuart_config *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 background.
- int LPUART_RTOS_Receive (lpuart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

30.3.2 Macro Definition Documentation

30.3.2.1 #define FSL_LPUART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

30.3.3 Function Documentation

30.3.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t * *handle*, lpuart_handle_t * *t_handle*, const struct rtos_lpuart_config * *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

30.3.3.2 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.

30.3.3.3 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.

30.3.3.4 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.

30.4 LPUART µCOS/II Driver

30.4.1 Overview

Files

- file fsl_lpuart_ucosii.h

Driver version

- #define FSL_LPUART_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPUART µCOS-II driver version 2.0.0.

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const struct rtos_lpuart_config *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.

30.4.2 Macro Definition Documentation

30.4.2.1 #define FSL_LPUART_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

30.4.3 Function Documentation

30.4.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t * *handle*, lpuart_handle_t * *t_handle*, const struct rtos_lpuart_config * *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

30.4.3.2 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.

30.4.3.3 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.

30.4.3.4 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.

30.5 LPUART µCOS/III Driver

30.5.1 Overview

Files

- file fsl_lpuart_ucosiii.h

Driver version

- #define FSL_LPUART_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPUART µCOS-III driver version 2.0.0.

LPUART RTOS Operation

- int LPUART_RTOS_Init (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const struct rtos_lpuart_config *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)
  Send data in background.
- int LPUART_RTOS_Receive (lpuart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

30.5.2 Macro Definition Documentation

30.5.2.1 #define FSL_LPUART_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

30.5.3 Function Documentation

30.5.3.1 int LPUART_RTOS_Init (lpuart_rtos_handle_t * *handle*, lpuart_handle_t * *t_handle*, const struct rtos_lpuart_config * *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

30.5.3.2 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.

30.5.3.3 int LPUART_RTOS_Send (lpuart_rtos_handle_t * *handle,* const uint8_t * *buffer,* uint32_t *length*)

This function sends data. It is 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.

30.5.3.4 int LPUART_RTOS_Receive (lpuart_rtos_handle_t * *handle,* uint8_t * *buffer,* uint32_t *length,* size_t * *received*)

It is 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.

Chapter 31
LTC: LP Trusted Cryptography

31.1 Overview

The Kinetis SDK provides the Peripheral driver for the LP Trusted Cryptography (LTC) module of Kinetis devices. LP Trusted Cryptography is a set of cryptograhpic hardware accelerator engines that share common registers. LTC architecture can support AES, DES, 3DES, MDHA (SHA), RSA and ECC. Actual list of implemented cryptograhpic hardware accelerator engines depends on specific Kinetis microcontroller.

The driver comprises two sets of API functions.

In the first set, blocking synchronous APIs are provided, for all operations supported by LTC hardware. The LTC operations are complete (and results are made availabe for further usage) when a function returns. When called, these functions don't return until an LTC operation is complete. These functions use main CPU for simple polling loops to determine operation complete or error status and also for plaintext or ciphertext data movements. The driver functions are not re-entrant. These functions provide typical interface to upper layer or application software.

In the second set, DMA support for symmetric LTC processing is provided, for AES and DES engines. APIs in the second set use DMA for data movement to and from the LTC input and output FIFOs. By using these functions, main CPU is not used for plaintext or ciphertext data movements (DMA is used instead). Thus, CPU processing power can be used for other application tasks, at cost of decreased maximum data throughput (because of DMA module and transactions management overhead). These functions provide less typical interface, for applications that must offload main CPU while ciphertext or plaintext is being processed, at cost of longer cryptograhpic processing time.

31.2 LTC Driver Initialization and Configuration

LTC Driver is initialized by calling the LTC_Init() function, it enables the LTC module clock in the Kinetis SIM module. If AES or DES engine is used and the LTC module implementation features the LTC DPA Mask Seed register, seed the DPA mask generator by using the seed from a random number generator. The LTC_SetDpaMaskSeed() function is provided to set the DPA mask seed.

31.3 Comments about API usage in RTOS

LTC operations provided by this driver are not re-entrant. Thus, application software shall ensure the LTC module operation is not requested from different tasks or interrupt service routines while an operation is in progress.

31.4 Comments about API usage in interrupt handler

All APIs can be used from interrupt handler although execution time shall be considered (interrupt latency of equal and lower priority interrupts increases).

31.5 LTC Driver Examples

Simple examples

Initialize LTC after Power On Reset or reset cycle

```
~~~~~{.c}
    LTC_Init(LTC0);
    /* optionally initialize DPA mask seed register */
    LTC_SetDpaMaskSeed(randomNumber);
~~~~~
```

Encrypt plaintext by DES engine

```
~~~~~{.c}
    char plain[16];
    char cipher[16];

    char iv[LTC_DES_IV_SIZE];
    char key1[LTC_DES_KEY_SIZE];
    char key2[LTC_DES_KEY_SIZE];
    char key3[LTC_DES_KEY_SIZE];

    memcpy(plain, "Hello World!", 12);
    memcpy(iv, "initvect", LTC_DES_IV_SIZE);
    memcpy(key1, "mykey1aa", LTC_DES_KEY_SIZE);
    memcpy(key2, "mykey2bb", LTC_DES_KEY_SIZE);
    memcpy(key3, "mykey3cc", LTC_DES_KEY_SIZE);

    LTC_DES3_EncryptCbc(LTC0, plain, cipher, 16, iv, key1, key2, key3);
~~~~~
```

Encrypt ciphertext by AES engine

```
~~~~~{.c}
    char plain[16] = {0};
    char cipher[16];
    char iv[16] = {0};
    char key[16] = {0};

    memcpy(plain, "Hello World!", 12);
    memcpy(iv, "initvectorinitve", 16);
    memcpy(key, "__mykey1aa__^^..", 16);

    LTC_AES_EncryptCbc(LTC0, plain, cipher, 16, iv, key, 16);
~~~~~
```

Compute keyed hash by AES engine (CMAC)

```
~~~~~{.c}
   char message[] = "Hello World!";
   char key[16] = {0};
   char output[16];
   uint32_t szOutput = 16u;

   memcpy(key, "__mykey1aa__^^..", 16);
   LTC_HASH(LTC0, kLTC_Cmac, message, sizeof(message), key, 16, output, &szOutput);
~~~~~
```

Compute hash by MDHA engine (SHA-256)

```
~~~~~{.c}
   char message[] = "Hello World!";
   char output[32];
   uint32_t szOutput = 32u;

   LTC_HASH(LTC0, kLTC_Sha256, message, sizeof(message), NULL, output, &szOutput);
~~~~~
```

Compute modular integer exponentiation

```
~~~~~{.c}
    status_t status;
    const char bigA[] = "112233445566778899aabbccddeeff";
    const char bigN[] = "aabbaabbaabb112233445566778899aabbccddeefe";
    const char bigE[] = "065537";
    char A[256], E[256], N[256], res[256];
    uint16_t sizeA, sizeE, sizeN, sizeRes;

    /* Note LTC PKHA integer format is least significant byte at lowest address.
     * The _read_string() function converts the input string to LTC PKHA integer format
     * and writes sizeof() the integer to the size variable (sizeA, sizeE, sizeN).
     */
    _read_string(A, &sizeA, bigA);
    _read_string(E, &sizeE, bigN);
    _read_string(N, &sizeN, bigE);

    status = LTC_PKHA_ModExp(base, A, sizeA, N, sizeN, E, sizeE, res, &sizeRes,
      kLTC_PKHA_IntegerArith,
                             kLTC_PKHA_NormalValue, kLTC_PKHA_TimingEqualized);
~~~~~
```

Compute elliptic curve point multiplication

```
~~~~~{.c}
    status_t status;
    ltc_pkha_ecc_point_t B0, res0;
    uint8_t bx, by, resx, resy;
    uint8_t E[256];
    bool isPointOfInfinity;
    uint16_t resultSize, sizeE;

    /* Example carried out with 1-byte curve params and point coordinates. */
    uint8_t size = 1;
    uint8_t aCurveParam = 1;
    uint8_t bCurveParam = 0;

    bx = 9;
    by = 5;

    B0.X = &bx;
    B0.Y = &by;
    res0.X = &resx;
    res0.Y = &resy;

    /* Prime modulus of the field. */
    N[0] = 23;

    /* Note LTC PKHA integer has least significant byte at lowest address */

    /* Scalar multiplier */
    char ew[] = "0100"; /* 256 in decimal */
```

```
    _read_string(E, &sizeE, ew);

    status = LTC_PKHA_ECC_PointMul(LTC0, &B0, E, sizeE, N, NULL, &aCurveParam, &
      bCurveParam, size,
                                   kLTC_PKHA_TimingEqualized,
      kLTC_PKHA_IntegerArith, &res0, &isPointOfInfinity);
~~~~~
```

Modules

- LTC Blocking APIs
- LTC Non-blocking eDMA APIs

Files

- file fsl_ltc.h

Functions

- void LTC_Init (LTC_Type *base)
  Initializes the LTC driver.
- void LTC_Deinit (LTC_Type *base)
  Deinitializes the LTC driver.
- void LTC_SetDpaMaskSeed (LTC_Type *base, uint32_t mask)
  Sets the DPA Mask Seed register.

Driver version

- #define FSL_LTC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LTC driver version.

31.6 Macro Definition Documentation

31.6.1 #define FSL_LTC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

31.7 Function Documentation

31.7.1 void LTC_Init (LTC_Type * *base*)

This function initializes the LTC driver.

Parameters

base	LTC peripheral base address

31.7.2 void LTC_Deinit (LTC_Type * *base*)

This function deinitializes the LTC driver.

Parameters

base	LTC peripheral base address

31.7.3 void LTC_SetDpaMaskSeed (LTC_Type * *base,* uint32_t *mask*)

The DPA Mask Seed register reseeds the mask that provides resistance against DPA (differential power analysis) attacks on AES or DES keys.

Differential Power Analysis Mask (DPA) resistance uses a randomly changing mask that introduces "noise" into the power consumed by the AES or DES. This reduces the signal-to-noise ratio that differential power analysis attacks use to "guess" bits of the key. This randomly changing mask should be seeded at POR, and continues to provide DPA resistance from that point on. However, to provide even more DPA protection it is recommended that the DPA mask be reseeded after every 50,000 blocks have been processed. At that time, software can opt to write a new seed (preferably obtained from an RNG) into the DPA Mask Seed register (DPAMS), or software can opt to provide the new seed earlier or later, or not at all. DPA resistance continues even if the DPA mask is never reseeded.

Parameters

base	LTC peripheral base address
mask	The DPA mask seed.

31.8 LTC Blocking APIs

31.8.1 Overview

This section describes the programming interface of the LTC Synchronous Blocking functions

Modules

- LTC AES driver
- LTC DES driver
- LTC HASH driver
- LTC PKHA driver

31.8.2 LTC AES driver

31.8.2.1 Overview

This section describes the programming interface of the LTC AES driver.

Macros

- #define LTC_AES_BLOCK_SIZE 16
  AES block size in bytes.
- #define LTC_AES_IV_SIZE 16
  AES Input Vector size in bytes.
- #define LTC_AES_DecryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)
  AES CTR decrypt is mapped to the AES CTR generic operation.
- #define LTC_AES_EncryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)
  AES CTR encrypt is mapped to the AES CTR generic operation.

Enumerations

- enum ltc_aes_key_t {
  kLTC_EncryptKey = 0U,
  kLTC_DecryptKey = 1U }
  Type of AES key for ECB and CBC decrypt operations.

Functions

- status_t LTC_AES_GenerateDecryptKey (LTC_Type *base, const uint8_t *encryptKey, uint8_t *decryptKey, uint32_t keySize)
  Transforms an AES encrypt key (forward AES) into the decrypt key (inverse AES).
- status_t LTC_AES_EncryptEcb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *key, uint32_t keySize)
  Encrypts AES using the ECB block mode.
- status_t LTC_AES_DecryptEcb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)
  Decrypts AES using ECB block mode.
- status_t LTC_AES_EncryptCbc (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize)
  Encrypts AES using CBC block mode.
- status_t LTC_AES_DecryptCbc (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)
  Decrypts AES using CBC block mode.
- status_t LTC_AES_CryptCtr (LTC_Type *base, const uint8_t *input, uint8_t *output, uint32_t size, uint8_t counter[LTC_AES_BLOCK_SIZE], const uint8_t *key, uint32_t keySize, uint8_t

counterlast[LTC_AES_BLOCK_SIZE], uint32_t *szLeft)
Encrypts or decrypts AES using CTR block mode.

- status_t LTC_AES_EncryptTagGcm (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, uint8_t *tag, uint32_t tagSize)
  Encrypts AES and tags using GCM block mode.
- status_t LTC_AES_DecryptTagGcm (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, const uint8_t *tag, uint32_t tagSize)
  Decrypts AES and authenticates using GCM block mode.
- status_t LTC_AES_EncryptTagCcm (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, uint8_t *tag, uint32_t tagSize)
  Encrypts AES and tags using CCM block mode.
- status_t LTC_AES_DecryptTagCcm (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *iv, uint32_t ivSize, const uint8_t *aad, uint32_t aadSize, const uint8_t *key, uint32_t keySize, const uint8_t *tag, uint32_t tagSize)
  Decrypts AES and authenticates using CCM block mode.

31.8.2.2 Enumeration Type Documentation

31.8.2.2.1 enum ltc_aes_key_t

Enumerator

kLTC_EncryptKey Input key is an encrypt key.
kLTC_DecryptKey Input key is a decrypt key.

31.8.2.3 Function Documentation

31.8.2.3.1 status_t LTC_AES_GenerateDecryptKey (LTC_Type * *base,* const uint8_t * *encryptKey,* uint8_t * *decryptKey,* uint32_t *keySize*)

Transforms the AES encrypt key (forward AES) into the decrypt key (inverse AES). The key derived by this function can be used as a direct load decrypt key for AES ECB and CBC decryption operations (keyType argument).

Parameters

	base	LTC peripheral base address

	encryptKey	Input key for decrypt key transformation
out	*decryptKey*	Output key, the decrypt form of the AES key.
	keySize	Size of the input key and output key in bytes. Must be 16, 24, or 32.

Returns

Status from key generation operation

31.8.2.3.2 status_t LTC_AES_EncryptEcb (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t * *key,* uint32_t *keySize*)

Encrypts AES using the ECB block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

31.8.2.3.3 status_t LTC_AES_DecryptEcb (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t * *key,* uint32_t *keySize,* ltc_aes_key_t *keyType*)

Decrypts AES using ECB block mode.

Parameters

	base	LTC peripheral base address

	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	key	Input key.
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	keyType	Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.)

Returns

Status from decrypt operation

31.8.2.3.4 status_t LTC_AES_EncryptCbc (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *iv[LTC_AES_IV_SIZE],* const uint8_t * *key,* uint32_t *keySize*)

Parameters

	base	LTC peripheral base address
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	iv	Input initial vector to combine with the first input block.
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

31.8.2.3.5 status_t LTC_AES_DecryptCbc (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_AES_IV_SIZE],* const uint8_t * *key,* uint32_t *keySize,* ltc_aes_key_t *keyType*)

Parameters

	base	LTC peripheral base address
	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	iv	Input initial vector to combine with the first input block.
	key	Input key to use for decryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	keyType	Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.)

Returns

Status from decrypt operation

31.8.2.3.6 status_t LTC_AES_CryptCtr (LTC_Type * *base,* const uint8_t * *input,* uint8_t * *output,* uint32_t *size,* uint8_t *counter[LTC_AES_BLOCK_SIZE],* const uint8_t * *key,* uint32_t *keySize,* uint8_t *counterlast[LTC_AES_BLOCK_SIZE],* uint32_t * *szLeft*)

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

	base	LTC peripheral base address
	input	Input data for CTR block mode
out	*output*	Output data for CTR block mode
	size	Size of input and output data in bytes
in,out	*counter*	Input counter (updates on return)
	key	Input key to use for forward AES cipher

	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
out	*counterlast*	Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.
out	*szLeft*	Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns

Status from encrypt operation

31.8.2.3.7 status_t LTC_AES_EncryptTagGcm (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t * *iv,* uint32_t *ivSize,* const uint8_t * *aad,* uint32_t *aadSize,* const uint8_t * *key,* uint32_t *keySize,* uint8_t * *tag,* uint32_t *tagSize*)

Encrypts AES and optionally tags using GCM block mode. If plaintext is NULL, only the GHASH is calculated and output in the 'tag' field.

Parameters

	base	LTC peripheral base address
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text.
	size	Size of input and output data in bytes
	iv	Input initial vector
	ivSize	Size of the IV
	aad	Input additional authentication data
	aadSize	Input size in bytes of AAD
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
out	*tag*	Output hash tag. Set to NULL to skip tag processing.
	tagSize	Input size of the tag to generate, in bytes. Must be 4,8,12,13,14,15 or 16.

Returns

Status from encrypt operation

31.8.2.3.8 status_t LTC_AES_DecryptTagGcm (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t * *iv,* uint32_t *ivSize,* const uint8_t * *aad,* uint32_t *aadSize,* const uint8_t * *key,* uint32_t *keySize,* const uint8_t * *tag,* uint32_t *tagSize*)

Decrypts AES and optionally authenticates using GCM block mode. If ciphertext is NULL, only the GHASH is calculated and compared with the received GHASH in 'tag' field.

Parameters

	base	LTC peripheral base address
	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text.
	size	Size of input and output data in bytes
	iv	Input initial vector
	ivSize	Size of the IV
	aad	Input additional authentication data
	aadSize	Input size in bytes of AAD
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	tag	Input hash tag to compare. Set to NULL to skip tag processing.
	tagSize	Input size of the tag, in bytes. Must be 4, 8, 12, 13, 14, 15, or 16.

Returns

Status from decrypt operation

31.8.2.3.9 status_t LTC_AES_EncryptTagCcm (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t * *iv*, uint32_t *ivSize*, const uint8_t * *aad*, uint32_t *aadSize*, const uint8_t * *key*, uint32_t *keySize*, uint8_t * *tag*, uint32_t *tagSize*)

Encrypts AES and optionally tags using CCM block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text.
	size	Size of input and output data in bytes. Zero means authentication only.
	iv	Nonce

	ivSize	Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
	aad	Input additional authentication data. Can be NULL if aadSize is zero.
	aadSize	Input size in bytes of AAD. Zero means data mode only (authentication skipped).
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
out	*tag*	Generated output tag. Set to NULL to skip tag processing.
	tagSize	Input size of the tag to generate, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns

Status from encrypt operation

31.8.2.3.10 status_t LTC_AES_DecryptTagCcm (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t * *iv,* uint32_t *ivSize,* const uint8_t * *aad,* uint32_t *aadSize,* const uint8_t * *key,* uint32_t *keySize,* const uint8_t * *tag,* uint32_t *tagSize*)

Decrypts AES and optionally authenticates using CCM block mode.

Parameters

	base	LTC peripheral base address
	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text.
	size	Size of input and output data in bytes. Zero means authentication only.
	iv	Nonce
	ivSize	Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.
	aad	Input additional authentication data. Can be NULL if aadSize is zero.
	aadSize	Input size in bytes of AAD. Zero means data mode only (authentication skipped).

	key	Input key to use for decryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	tag	Received tag. Set to NULL to skip tag processing.
	tagSize	Input size of the received tag to compare with the computed tag, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16.

Returns

Status from decrypt operation

31.8.3 LTC DES driver

31.8.3.1 Overview

This section describes the programming interface of the LTC DES driver.

Macros

- #define LTC_DES_KEY_SIZE 8
  LTC DES key size - 64 bits.
- #define LTC_DES_IV_SIZE 8
  LTC DES IV size - 8 bytes.

Functions

- status_t LTC_DES_EncryptEcb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using ECB block mode.
- status_t LTC_DES_DecryptEcb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using ECB block mode.
- status_t LTC_DES_EncryptCbc (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using CBC block mode.
- status_t LTC_DES_DecryptCbc (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using CBC block mode.
- status_t LTC_DES_EncryptCfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using CFB block mode.
- status_t LTC_DES_DecryptCfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using CFB block mode.
- status_t LTC_DES_EncryptOfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using OFB block mode.
- status_t LTC_DES_DecryptOfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using OFB block mode.
- status_t LTC_DES2_EncryptEcb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Encrypts triple DES using ECB block mode with two keys.
- status_t LTC_DES2_DecryptEcb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Decrypts triple DES using ECB block mode with two keys.

- status_t LTC_DES2_EncryptCbc (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Encrypts triple DES using CBC block mode with two keys.
- status_t LTC_DES2_DecryptCbc (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Decrypts triple DES using CBC block mode with two keys.
- status_t LTC_DES2_EncryptCfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Encrypts triple DES using CFB block mode with two keys.
- status_t LTC_DES2_DecryptCfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Decrypts triple DES using CFB block mode with two keys.
- status_t LTC_DES2_EncryptOfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Encrypts triple DES using OFB block mode with two keys.
- status_t LTC_DES2_DecryptOfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Decrypts triple DES using OFB block mode with two keys.
- status_t LTC_DES3_EncryptEcb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Encrypts triple DES using ECB block mode with three keys.
- status_t LTC_DES3_DecryptEcb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Decrypts triple DES using ECB block mode with three keys.
- status_t LTC_DES3_EncryptCbc (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Encrypts triple DES using CBC block mode with three keys.
- status_t LTC_DES3_DecryptCbc (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Decrypts triple DES using CBC block mode with three keys.
- status_t LTC_DES3_EncryptCfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Encrypts triple DES using CFB block mode with three keys.
- status_t LTC_DES3_DecryptCfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Decrypts triple DES using CFB block mode with three keys.

- status_t LTC_DES3_EncryptOfb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Encrypts triple DES using OFB block mode with three keys.
- status_t LTC_DES3_DecryptOfb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])
  Decrypts triple DES using OFB block mode with three keys.

31.8.3.2 Macro Definition Documentation

31.8.3.2.1 #define LTC_DES_KEY_SIZE 8

31.8.3.3 Function Documentation

31.8.3.3.1 status_t LTC_DES_EncryptEcb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using ECB block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.2 status_t LTC_DES_DecryptEcb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using ECB block mode.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.3 status_t LTC_DES_EncryptCbc (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using CBC block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Ouput ciphertext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.4 status_t LTC_DES_DecryptCbc (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using CBC block mode.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.5 status_t LTC_DES_EncryptCfb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using CFB block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
	size	Size of input data in bytes
	iv	Input initial block.
	key	Input key to use for encryption
out	*ciphertext*	Output ciphertext

Returns

Status from encrypt/decrypt operation

31.8.3.3.6 status_t LTC_DES_DecryptCfb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using CFB block mode.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.7 status_t LTC_DES_EncryptOfb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using OFB block mode.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.8 status_t LTC_DES_DecryptOfb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using OFB block mode.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.8.3.3.9 status_t LTC_DES2_EncryptEcb (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using ECB block mode with two keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.10 status_t LTC_DES2_DecryptEcb (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using ECB block mode with two keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.11 status_t LTC_DES2_EncryptCbc (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CBC block mode with two keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.12 status_t LTC_DES2_DecryptCbc (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CBC block mode with two keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.13 status_t LTC_DES2_EncryptCfb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.14 status_t LTC_DES2_DecryptCfb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.15 status_t LTC_DES2_EncryptOfb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using OFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.16 status_t LTC_DES2_DecryptOfb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using OFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.17 status_t LTC_DES3_EncryptEcb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using ECB block mode with three keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.18 status_t LTC_DES3_DecryptEcb (LTC_Type * *base*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using ECB block mode with three keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.19 status_t LTC_DES3_EncryptCbc (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CBC block mode with three keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.20 status_t LTC_DES3_DecryptCbc (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE],* const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CBC block mode with three keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.21 status_t LTC_DES3_EncryptCfb (LTC_Type * *base,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE],* const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and ouput data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.22 status_t LTC_DES3_DecryptCfb (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE],* const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.23 status_t LTC_DES3_EncryptOfb (LTC_Type * *base*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using OFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.3.3.24 status_t LTC_DES3_DecryptOfb (LTC_Type * *base,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE],* const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using OFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.8.4 LTC HASH driver

31.8.4.1 Overview

This section describes the programming interface of the LTC HASH driver.

Macros

- #define LTC_HASH_CTX_SIZE 29
  LTC HASH Context size.

Typedefs

- typedef uint32_t ltc_hash_ctx_t [LTC_HASH_CTX_SIZE]
  Storage type used to save hash context.

Enumerations

- enum ltc_hash_algo_t {
  kLTC_XcbcMac = 0,
  kLTC_Cmac }
  Supported cryptographic block cipher functions for HASH creation.

Functions

- status_t LTC_HASH_Init (LTC_Type *base, ltc_hash_ctx_t *ctx, ltc_hash_algo_t algo, const uint8_t *key, uint32_t keySize)
  Initialize HASH context.
- status_t LTC_HASH_Update (ltc_hash_ctx_t *ctx, const uint8_t *input, uint32_t inputSize)
  Add data to current HASH.
- status_t LTC_HASH_Finish (ltc_hash_ctx_t *ctx, uint8_t *output, uint32_t *outputSize)
  Finalize hashing.
- status_t LTC_HASH (LTC_Type *base, ltc_hash_algo_t algo, const uint8_t *input, uint32_t inputSize, const uint8_t *key, uint32_t keySize, uint8_t *output, uint32_t *outputSize)
  Create HASH on given data.

31.8.4.2 Macro Definition Documentation

31.8.4.2.1 #define LTC_HASH_CTX_SIZE 29

31.8.4.3 Typedef Documentation

31.8.4.3.1 typedef uint32_t ltc_hash_ctx_t[LTC_HASH_CTX_SIZE]

31.8.4.4 Enumeration Type Documentation

31.8.4.4.1 enum ltc_hash_algo_t

Enumerator

kLTC_XcbcMac XCBC-MAC (AES engine)
kLTC_Cmac CMAC (AES engine)

31.8.4.5 Function Documentation

31.8.4.5.1 status_t LTC_HASH_Init (LTC_Type * *base,* ltc_hash_ctx_t * *ctx,* ltc_hash_algo_t *algo,* const uint8_t * *key,* uint32_t *keySize*)

This function initialize the HASH. Key shall be supplied if the underlaying algoritm is AES XCBC-MAC or CMAC. Key shall be NULL if the underlaying algoritm is SHA.

For XCBC-MAC, the key length must be 16. For CMAC, the key length can be the AES key lengths supported by AES engine. For MDHA the key length argument is ignored.

Parameters

	base	LTC peripheral base address
out	*ctx*	Output hash context
	algo	Underlaying algorithm to use for hash computation.
	key	Input key (NULL if underlaying algorithm is SHA)
	keySize	Size of input key in bytes

Returns

Status of initialization

31.8.4.5.2 status_t LTC_HASH_Update (ltc_hash_ctx_t * *ctx,* const uint8_t * *input,* uint32_t *inputSize*)

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed.

Parameters

in,out	*ctx*	HASH context
	input	Input data
	inputSize	Size of input data in bytes

Returns

Status of the hash update operation

31.8.4.5.3 status_t LTC_HASH_Finish (ltc_hash_ctx_t * *ctx*, uint8_t * *output*, uint32_t * *outputSize*)

Outputs the final hash and erases the context.

Parameters

in,out	*ctx*	Input hash context
out	*output*	Output hash data
out	*outputSize*	Output parameter storing the size of the output hash in bytes

Returns

Status of the hash finish operation

31.8.4.5.4 status_t LTC_HASH (LTC_Type * *base*, ltc_hash_algo_t *algo*, const uint8_t * *input*, uint32_t *inputSize*, const uint8_t * *key*, uint32_t *keySize*, uint8_t * *output*, uint32_t * *outputSize*)

Perform the full keyed HASH in one function call.

Parameters

	base	LTC peripheral base address
	algo	Block cipher algorithm to use for CMAC creation

	input	Input data
	inputSize	Size of input data in bytes
	key	Input key
	keySize	Size of input key in bytes
out	*output*	Output hash data
out	*outputSize*	Output parameter storing the size of the output hash in bytes

Returns

Status of the one call hash operation.

31.8.5 LTC PKHA driver

31.8.5.1 Overview

This section describes the programming interface of the LTC PKHA driver.

Data Structures

- struct ltc_pkha_ecc_point_t
  PKHA ECC point structure. More...

Enumerations

- enum ltc_pkha_timing_t
  Use of timing equalized version of a PKHA function.
- enum ltc_pkha_f2m_t {
  kLTC_PKHA_IntegerArith = 0U,
  kLTC_PKHA_F2mArith = 1U }
  Integer vs binary polynomial arithmetic selection.
- enum ltc_pkha_montgomery_form_t
  Montgomery or normal PKHA input format.

Functions

- int LTC_PKHA_CompareBigNum (const uint8_t *a, size_t sizeA, const uint8_t *b, size_t sizeB)
  Compare two PKHA big numbers.
- status_t LTC_PKHA_NormalToMontgomery (LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t *sizeB, uint8_t *R2, uint16_t *sizeR2, ltc_pkha_timing_t equalTime, ltc_pkha_f2m_t arithType)
  Converts from integer to Montgomery format.
- status_t LTC_PKHA_MontgomeryToNormal (LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *A, uint16_t *sizeA, uint8_t *B, uint16_t *sizeB, ltc_pkha_timing_t equalTime, ltc_pkha_f2m_t arithType)
  Converts from Montgomery format to int.
- status_t LTC_PKHA_ModAdd (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
  Performs modular addition - (A + B) mod N.
- status_t LTC_PKHA_ModSub1 (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize)
  Performs modular subtraction - (A - B) mod N.
- status_t LTC_PKHA_ModSub2 (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize)
  Performs modular subtraction - (B - A) mod N.
- status_t LTC_PKHA_ModMul (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha-

_f2m_t arithType, ltc_pkha_montgomery_form_t montIn, ltc_pkha_montgomery_form_t montOut, ltc_pkha_timing_t equalTime)
Performs modular multiplication - (A x B) mod N.

- status_t LTC_PKHA_ModExp (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, const uint8_t *E, uint16_t sizeE, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType, ltc_pkha_montgomery_form_t montIn, ltc_pkha_timing_t equalTime)
  Performs modular exponentiation - (A^E) mod N.
- status_t LTC_PKHA_ModRed (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
  Performs modular reduction - (A) mod N.
- status_t LTC_PKHA_ModInv (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
  Performs modular inversion - (A^-1) mod N.
- status_t LTC_PKHA_ModR2 (LTC_Type *base, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
  Computes integer Montgomery factor R^2 mod N.
- status_t LTC_PKHA_GCD (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *N, uint16_t sizeN, uint8_t *result, uint16_t *resultSize, ltc_pkha_f2m_t arithType)
  Calculates the greatest common divisor - GCD (A, N).
- status_t LTC_PKHA_PrimalityTest (LTC_Type *base, const uint8_t *A, uint16_t sizeA, const uint8_t *B, uint16_t sizeB, const uint8_t *N, uint16_t sizeN, bool *res)
  Executes Miller-Rabin primality test.
- status_t LTC_PKHA_ECC_PointAdd (LTC_Type *base, const ltc_pkha_ecc_point_t *A, const ltc_pkha_ecc_point_t *B, const uint8_t *N, const uint8_t *R2modN, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, ltc_pkha_f2m_t arithType, ltc_pkha_ecc_point_t *result)
  Adds elliptic curve points - A + B.
- status_t LTC_PKHA_ECC_PointDouble (LTC_Type *base, const ltc_pkha_ecc_point_t *B, const uint8_t *N, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, ltc_pkha_f2m_t arithType, ltc_pkha_ecc_point_t *result)
  Doubles elliptic curve points - B + B.
- status_t LTC_PKHA_ECC_PointMul (LTC_Type *base, const ltc_pkha_ecc_point_t *A, const uint8_t *E, uint8_t sizeE, const uint8_t *N, const uint8_t *R2modN, const uint8_t *aCurveParam, const uint8_t *bCurveParam, uint8_t size, ltc_pkha_timing_t equalTime, ltc_pkha_f2m_t arithType, ltc_pkha_ecc_point_t *result, bool *infinity)
  Multiplies an elliptic curve point by a scalar - E x (A0, A1).

31.8.5.2 Data Structure Documentation

31.8.5.2.1 struct ltc_pkha_ecc_point_t

Data Fields

- uint8_t * X
  X coordinate (affine)
- uint8_t * Y
  Y coordinate (affine)

31.8.5.3 Enumeration Type Documentation

31.8.5.3.1 enum ltc_pkha_timing_t

31.8.5.3.2 enum ltc_pkha_f2m_t

Enumerator

kLTC_PKHA_IntegerArith Use integer arithmetic.
kLTC_PKHA_F2mArith Use binary polynomial arithmetic.

31.8.5.3.3 enum ltc_pkha_montgomery_form_t

31.8.5.4 Function Documentation

31.8.5.4.1 int LTC_PKHA_CompareBigNum (const uint8_t * *a*, size_t *sizeA*, const uint8_t * *b*, size_t *sizeB*)

Compare two PKHA big numbers. Return 1 for a $>$ b, -1 for a $<$ b and 0 if they are same. PKHA big number is lsbyte first. Thus the comparison starts at msbyte which is the last member of tested arrays.

Parameters

a	First integer represented as an array of bytes, lsbyte first.
sizeA	Size in bytes of the first integer.
b	Second integer represented as an array of bytes, lsbyte first.
sizeB	Size in bytes of the second integer.

Returns

1 if a $>$ b.
-1 if a $<$ b.
0 if a = b.

31.8.5.4.2 status_t LTC_PKHA_NormalToMontgomery (LTC_Type * *base*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *A*, uint16_t * *sizeA*, uint8_t * *B*, uint16_t * *sizeB*, uint8_t * *R2*, uint16_t * *sizeR2*, ltc_pkha_timing_t *equalTime*, ltc_pkha_f2m_t *arithType*)

This function computes R2 mod N and optionally converts A or B into Montgomery format of A or B.

Parameters

	base	LTC peripheral base address
	N	modulus
	sizeN	size of N in bytes
in,out	*A*	The first input in non-Montgomery format. Output Montgomery format of the first input.
in,out	*sizeA*	pointer to size variable. On input it holds size of input A in bytes. On output it holds size of Montgomery format of A in bytes.
in,out	*B*	Second input in non-Montgomery format. Output Montgomery format of the second input.
in,out	*sizeB*	pointer to size variable. On input it holds size of input B in bytes. On output it holds size of Montgomery format of B in bytes.
out	*R2*	Output Montgomery factor R2 mod N.
out	*sizeR2*	pointer to size variable. On output it holds size of Montgomery factor R2 mod N in bytes.
	equalTime	Run the function time equalized or no timing equalization.
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.3 status_t LTC_PKHA_MontgomeryToNormal (LTC_Type * *base*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *A*, uint16_t * *sizeA*, uint8_t * *B*, uint16_t * *sizeB*, ltc_pkha_timing_t *equalTime*, ltc_pkha_f2m_t *arithType*)

This function converts Montgomery format of A or B into int A or B.

Parameters

	base	LTC peripheral base address
	N	modulus.
	sizeN	size of N modulus in bytes.

in,out	*A*	Input first number in Montgomery format. Output is non-Montgomery format.
in,out	*sizeA*	pointer to size variable. On input it holds size of the input A in bytes. On output it holds size of non-Montgomery A in bytes.
in,out	*B*	Input first number in Montgomery format. Output is non-Montgomery format.
in,out	*sizeB*	pointer to size variable. On input it holds size of the input B in bytes. On output it holds size of non-Montgomery B in bytes.
	equalTime	Run the function time equalized or no timing equalization.
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.4 status_t LTC_PKHA_ModAdd (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *B*, uint16_t *sizeB*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*)

This function performs modular addition of (A + B) mod N, with either integer or binary polynomial (F2m) inputs. In the F2m form, this function is equivalent to a bitwise XOR and it is functionally the same as subtraction.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	B	second addend (integer or binary polynomial)
	sizeB	Size of B in bytes
	N	modulus. For F2m operation this can be NULL, as N is ignored during F2m polynomial addition.
	sizeN	Size of N in bytes. This must be given for both integer and F2m polynomial additions.

out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.5 status_t LTC_PKHA_ModSub1 (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *B*, uint16_t *sizeB*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*)

This function performs modular subtraction of (A - B) mod N with integer inputs.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	B	second addend (integer or binary polynomial)
	sizeB	Size of B in bytes
	N	modulus
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes

Returns

Operation status.

31.8.5.4.6 status_t LTC_PKHA_ModSub2 (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *B*, uint16_t *sizeB*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*)

This function performs modular subtraction of (B - A) mod N, with integer inputs.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	B	second addend (integer or binary polynomial)
	sizeB	Size of B in bytes
	N	modulus
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes

Returns

Operation status.

31.8.5.4.7 status_t LTC_PKHA_ModMul (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *B*, uint16_t *sizeB*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*, ltc_pkha_montgomery_form_t *montIn*, ltc_pkha_montgomery_form_t *montOut*, ltc_pkha_timing_t *equalTime*)

This function performs modular multiplication with either integer or binary polynomial (F2m) inputs. It can optionally specify whether inputs and/or outputs will be in Montgomery form or not.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	B	second addend (integer or binary polynomial)
	sizeB	Size of B in bytes
	N	modulus.
	sizeN	Size of N in bytes

out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)
	montIn	Format of inputs
	montOut	Format of output
	equalTime	Run the function time equalized or no timing equalization. This argument is ignored for F2m modular multiplication.

Returns

Operation status.

31.8.5.4.8 status_t LTC_PKHA_ModExp (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *N*, uint16_t *sizeN*, const uint8_t * *E*, uint16_t *sizeE*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*, ltc_pkha_montgomery_form_t *montIn*, ltc_pkha_timing_t *equalTime*)

This function performs modular exponentiation with either integer or binary polynomial (F2m) inputs.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	N	modulus
	sizeN	Size of N in bytes
	E	exponent
	sizeE	Size of E in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	montIn	Format of A input (normal or Montgomery)
	arithType	Type of arithmetic to perform (integer or F2m)

	equalTime	Run the function time equalized or no timing equalization.

Returns

Operation status.

31.8.5.4.9 status_t LTC_PKHA_ModRed (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*)

This function performs modular reduction with either integer or binary polynomial (F2m) inputs.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	N	modulus
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.10 status_t LTC_PKHA_ModInv (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*)

This function performs modular inversion with either integer or binary polynomial (F2m) inputs.

Parameters

	base	LTC peripheral base address
	A	first addend (integer or binary polynomial)
	sizeA	Size of A in bytes
	N	modulus
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.11 status_t LTC_PKHA_ModR2 (LTC_Type * *base*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*)

This function computes a constant to assist in converting operands into the Montgomery residue system representation.

Parameters

	base	LTC peripheral base address
	N	modulus
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.12 status_t LTC_PKHA_GCD (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *N*, uint16_t *sizeN*, uint8_t * *result*, uint16_t * *resultSize*, ltc_pkha_f2m_t *arithType*)

This function calculates the greatest common divisor of two inputs with either integer or binary polynomial (F2m) inputs.

Parameters

	base	LTC peripheral base address
	A	first value (must be smaller than or equal to N)
	sizeA	Size of A in bytes
	N	second value (must be non-zero)
	sizeN	Size of N in bytes
out	*result*	Output array to store result of operation
out	*resultSize*	Output size of operation in bytes
	arithType	Type of arithmetic to perform (integer or F2m)

Returns

Operation status.

31.8.5.4.13 status_t LTC_PKHA_PrimalityTest (LTC_Type * *base*, const uint8_t * *A*, uint16_t *sizeA*, const uint8_t * *B*, uint16_t *sizeB*, const uint8_t * *N*, uint16_t *sizeN*, bool * *res*)

This function calculates whether or not a candidate prime number is likely to be a prime.

Parameters

	base	LTC peripheral base address
	A	initial random seed
	sizeA	Size of A in bytes
	B	number of trial runs
	sizeB	Size of B in bytes
	N	candidate prime integer
	sizeN	Size of N in bytes
out	*res*	True if the value is likely prime or false otherwise

Returns

Operation status.

31.8.5.4.14 status_t LTC_PKHA_ECC_PointAdd (LTC_Type * *base*, const ltc_pkha_ecc_point_t * *A*, const ltc_pkha_ecc_point_t * *B*, const uint8_t * *N*, const uint8_t * *R2modN*, const uint8_t * *aCurveParam*, const uint8_t * *bCurveParam*, uint8_t *size*, ltc_pkha_f2m_t *arithType*, ltc_pkha_ecc_point_t * *result*)

This function performs ECC point addition over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

	base	LTC peripheral base address
	A	Left-hand point
	B	Right-hand point
	N	Prime modulus of the field
	R2modN	NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
	aCurveParam	A parameter from curve equation
	bCurveParam	B parameter from curve equation (constant)
	size	Size in bytes of curve points and parameters
	arithType	Type of arithmetic to perform (integer or F2m)
out	*result*	Result point

Returns

Operation status.

31.8.5.4.15 status_t LTC_PKHA_ECC_PointDouble (LTC_Type * *base*, const ltc_pkha_ecc_point_t * *B*, const uint8_t * *N*, const uint8_t * *aCurveParam*, const uint8_t * *bCurveParam*, uint8_t *size*, ltc_pkha_f2m_t *arithType*, ltc_pkha_ecc_point_t * *result*)

This function performs ECC point doubling over a prime field (Fp) or binary field (F2m) using affine coordinates.

Parameters

	base	LTC peripheral base address
	B	Point to double
	N	Prime modulus of the field
	aCurveParam	A parameter from curve equation
	bCurveParam	B parameter from curve equation (constant)

	size	Size in bytes of curve points and parameters
	arithType	Type of arithmetic to perform (integer or F2m)
out	*result*	Result point

Returns

Operation status.

31.8.5.4.16 status_t LTC_PKHA_ECC_PointMul (LTC_Type * *base*, const ltc_pkha_ecc_point_t * *A*, const uint8_t * *E*, uint8_t *sizeE*, const uint8_t * *N*, const uint8_t * *R2modN*, const uint8_t * *aCurveParam*, const uint8_t * *bCurveParam*, uint8_t *size*, ltc_pkha_timing_t *equalTime*, ltc_pkha_f2m_t *arithType*, ltc_pkha_ecc_point_t * *result*, bool * *infinity*)

This function performs ECC point multiplication to multiply an ECC point by a scalar integer multiplier over a prime field (Fp) or a binary field (F2m).

Parameters

	base	LTC peripheral base address
	A	Point as multiplicand
	E	Scalar multiple
	sizeE	The size of E, in bytes
	N	Modulus, a prime number for the Fp field or Irreducible polynomial for F2m field.
	R2modN	NULL (the function computes R2modN internally) or pointer to pre-computed R2modN (obtained from LTC_PKHA_ModR2() function).
	aCurveParam	A parameter from curve equation
	bCurveParam	B parameter from curve equation (C parameter for operation over F2m).
	size	Size in bytes of curve points and parameters
	equalTime	Run the function time equalized or no timing equalization.
	arithType	Type of arithmetic to perform (integer or F2m)

out	*result*	Result point
out	*infinity*	Output true if the result is point of infinity, and false otherwise. Writing of this output will be ignored if the argument is NULL.

Returns

Operation status.

31.9 LTC Non-blocking eDMA APIs

31.9.1 Overview

This section describes the programming interface of the LTC EDMA Non Blocking functions

Modules

- LTC eDMA AES driver
- LTC eDMA DES driver

Data Structures

- struct ltc_edma_handle_t
  LTC EDMA handle. More...

Typedefs

- typedef void(∗ ltc_edma_callback_t)(LTC_Type ∗base, ltc_edma_handle_t ∗handle, status_t status, void ∗userData)
  LTC EDMA callback function.
- typedef status_t(∗ ltc_edma_state_machine_t)(LTC_Type ∗base, ltc_edma_handle_t ∗handle)
  LTC EDMA state machine function.

Functions

- void LTC_CreateHandleEDMA (LTC_Type ∗base, ltc_edma_handle_t ∗handle, ltc_edma_callback_t callback, void ∗userData, edma_handle_t ∗inputFifoEdmaHandle, edma_handle_t ∗outputFifoEdmaHandle)
  Init the LTC EDMA handle which is used in transcational functions.

31.9.2 Data Structure Documentation

31.9.2.1 struct _ltc_edma_handle

It is defined only for private usage inside LTC EDMA driver.

Data Fields

- ltc_edma_callback_t callback
  Callback function.
- void ∗ userData

LTC callback function parameter.

- edma_handle_t * inputFifoEdmaHandle
  The EDMA TX channel used.
- edma_handle_t * outputFifoEdmaHandle
  The EDMA RX channel used.
- ltc_edma_state_machine_t state_machine
  State machine.
- uint32_t state
  Internal state.
- const uint8_t * inData
  Input data.
- uint8_t * outData
  Output data.
- uint32_t size
  Size of input and output data in bytes.
- uint32_t modeReg
  LTC mode register.
- uint8_t * counter
  Input counter (updates on return)
- const uint8_t * key
  Input key to use for forward AES cipher.
- uint32_t keySize
  Size of the input key, in bytes.
- uint8_t * counterlast
  Output cipher of last counter, for chained CTR calls.
- uint32_t * szLeft
  Output number of bytes in left unused in counterlast block.
- uint32_t lastSize
  Last size.

31.9.2.1.0.1 Field Documentation

31.9.2.1.0.1.1 ltc_edma_callback_t ltc_edma_handle_t::callback

31.9.2.1.0.1.2 void* ltc_edma_handle_t::userData

31.9.2.1.0.1.3 edma_handle_t* ltc_edma_handle_t::inputFifoEdmaHandle

31.9.2.1.0.1.4 edma_handle_t* ltc_edma_handle_t::outputFifoEdmaHandle

31.9.2.1.0.1.5 ltc_edma_state_machine_t ltc_edma_handle_t::state_machine

31.9.2.1.0.1.6 uint32_t ltc_edma_handle_t::state

31.9.2.1.0.1.7 const uint8_t* ltc_edma_handle_t::inData

31.9.2.1.0.1.8 uint8_t* ltc_edma_handle_t::outData

31.9.2.1.0.1.9 uint32_t ltc_edma_handle_t::size

31.9.2.1.0.1.10 uint32_t ltc_edma_handle_t::modeReg

31.9.2.1.0.1.11 uint32_t ltc_edma_handle_t::keySize

Must be 16, 24, or 32.

31.9.2.1.0.1.12 uint8_t* ltc_edma_handle_t::counterlast

NULL can be passed if chained calls are not used.

31.9.2.1.0.1.13 uint32_t* ltc_edma_handle_t::szLeft

NULL can be passed if chained calls are not used.

31.9.2.1.0.1.14 uint32_t ltc_edma_handle_t::lastSize

31.9.3 Typedef Documentation

31.9.3.1 typedef void(* ltc_edma_callback_t)(LTC_Type *base, ltc_edma_handle_t *handle, status_t status, void *userData)

31.9.3.2 typedef status_t(* ltc_edma_state_machine_t)(LTC_Type *base, ltc_edma_handle_t *handle)

It is defined only for private usage inside LTC EDMA driver.

31.9.4 Function Documentation

31.9.4.1 void LTC_CreateHandleEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* ltc_edma_callback_t *callback,* void * *userData,* edma_handle_t * *inputFifoEdmaHandle,* edma_handle_t * *outputFifoEdmaHandle*)

Parameters

base	LTC module base address
handle	Pointer to ltc_edma_handle_t structure
callback	Callback function, NULL means no callback.
userData	Callback function parameter.
inputFifo-EdmaHandle	User requested EDMA handle for Input FIFO EDMA.
outputFifo-EdmaHandle	User requested EDMA handle for Output FIFO EDMA.

31.9.5 LTC eDMA AES driver

31.9.5.1 Overview

This section describes the programming interface of the LTC EDMA AES driver.

Macros

- #define LTC_AES_DecryptCtrEDMA(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtrEDMA(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft)
  AES CTR decrypt is mapped to the AES CTR generic operation.
- #define LTC_AES_EncryptCtrEDMA(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtrEDMA(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft)
  AES CTR encrypt is mapped to the AES CTR generic operation.

Functions

- status_t LTC_AES_EncryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *key, uint32_t keySize)
  Encrypts AES using the ECB block mode.
- status_t LTC_AES_DecryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)
  Decrypts AES using ECB block mode.
- status_t LTC_AES_EncryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize)
  Encrypts AES using CBC block mode.
- status_t LTC_AES_DecryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)
  Decrypts AES using CBC block mode.
- status_t LTC_AES_CryptCtrEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *input, uint8_t *output, uint32_t size, uint8_t counter[LTC_AES_BLOCK_SIZE], const uint8_t *key, uint32_t keySize, uint8_t counterlast[LTC_AES_BLOCK_SIZE], uint32_t *szLeft)
  Encrypts or decrypts AES using CTR block mode.

31.9.5.2 Function Documentation

31.9.5.2.1 status_t LTC_AES_EncryptEcbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t * *key,* uint32_t *keySize*)

Encrypts AES using the ECB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

31.9.5.2.2 status_t LTC_AES_DecryptEcbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t * *key*, uint32_t *keySize*, ltc_aes_key_t *keyType*)

Decrypts AES using ECB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	key	Input key.
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	keyType	Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.)

Returns

Status from decrypt operation

31.9.5.2.3 status_t LTC_AES_EncryptCbcEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *iv[LTC_AES_IV_SIZE],* const uint8_t * *key,* uint32_t *keySize*)

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plain text to encrypt
out	*ciphertext*	Output cipher text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	iv	Input initial vector to combine with the first input block.
	key	Input key to use for encryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.

Returns

Status from encrypt operation

31.9.5.2.4 status_t LTC_AES_DecryptCbcEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_AES_IV_SIZE]*, const uint8_t * *key*, uint32_t *keySize*, ltc_aes_key_t *keyType*)

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input cipher text to decrypt
out	*plaintext*	Output plain text
	size	Size of input and output data in bytes. Must be multiple of 16 bytes.
	iv	Input initial vector to combine with the first input block.
	key	Input key to use for decryption
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
	keyType	Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.)

Returns

Status from decrypt operation

31.9.5.2.5 status_t LTC_AES_CryptCtrEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *input,* uint8_t * *output,* uint32_t *size,* uint8_t *counter[LTC_AES_BLOCK_SIZE],* const uint8_t * *key,* uint32_t *keySize,* uint8_t *counterlast[LTC_AES_BLOCK_SIZE],* uint32_t * *szLeft*)

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	input	Input data for CTR block mode
out	*output*	Output data for CTR block mode
	size	Size of input and output data in bytes
in,out	*counter*	Input counter (updates on return)
	key	Input key to use for forward AES cipher
	keySize	Size of the input key, in bytes. Must be 16, 24, or 32.
out	*counterlast*	Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.
out	*szLeft*	Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns

Status from encrypt operation

31.9.6 LTC eDMA DES driver

31.9.6.1 Overview

This section describes the programming interface of the LTC EDMA DES driver.

Functions

- status_t LTC_DES_EncryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using ECB block mode.
- status_t LTC_DES_DecryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using ECB block mode.
- status_t LTC_DES_EncryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using CBC block mode.
- status_t LTC_DES_DecryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using CBC block mode.
- status_t LTC_DES_EncryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using CFB block mode.
- status_t LTC_DES_DecryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using CFB block mode.
- status_t LTC_DES_EncryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Encrypts DES using OFB block mode.
- status_t LTC_DES_DecryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key[LTC_DES_KEY_SIZE])
  Decrypts DES using OFB block mode.
- status_t LTC_DES2_EncryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Encrypts triple DES using ECB block mode with two keys.
- status_t LTC_DES2_DecryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])
  Decrypts triple DES using ECB block mode with two keys.
- status_t LTC_DES2_EncryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const

uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

Encrypts triple DES using CBC block mode with two keys.

- status_t LTC_DES2_DecryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

  Decrypts triple DES using CBC block mode with two keys.
- status_t LTC_DES2_EncryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

  Encrypts triple DES using CFB block mode with two keys.
- status_t LTC_DES2_DecryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

  Decrypts triple DES using CFB block mode with two keys.
- status_t LTC_DES2_EncryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

  Encrypts triple DES using OFB block mode with two keys.
- status_t LTC_DES2_DecryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE])

  Decrypts triple DES using OFB block mode with two keys.
- status_t LTC_DES3_EncryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Encrypts triple DES using ECB block mode with three keys.
- status_t LTC_DES3_DecryptEcbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Decrypts triple DES using ECB block mode with three keys.
- status_t LTC_DES3_EncryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Encrypts triple DES using CBC block mode with three keys.
- status_t LTC_DES3_DecryptCbcEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Decrypts triple DES using CBC block mode with three keys.
- status_t LTC_DES3_EncryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Encrypts triple DES using CFB block mode with three keys.
- status_t LTC_DES3_DecryptCfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const

uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

Decrypts triple DES using CFB block mode with three keys.

- status_t LTC_DES3_EncryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Encrypts triple DES using OFB block mode with three keys.
- status_t LTC_DES3_DecryptOfbEDMA (LTC_Type *base, ltc_edma_handle_t *handle, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_DES_IV_SIZE], const uint8_t key1[LTC_DES_KEY_SIZE], const uint8_t key2[LTC_DES_KEY_SIZE], const uint8_t key3[LTC_DES_KEY_SIZE])

  Decrypts triple DES using OFB block mode with three keys.

31.9.6.2 Function Documentation

31.9.6.2.1 status_t LTC_DES_EncryptEcbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using ECB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.2 status_t LTC_DES_DecryptEcbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using ECB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.3 status_t LTC_DES_EncryptCbcEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using CBC block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Ouput ciphertext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.4 status_t LTC_DES_DecryptCbcEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using CBC block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.5 status_t LTC_DES_EncryptCfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using CFB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
	size	Size of input data in bytes
	iv	Input initial block.
	key	Input key to use for encryption
out	*ciphertext*	Output ciphertext

Returns

Status from encrypt/decrypt operation

31.9.6.2.6 status_t LTC_DES_DecryptCfbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using CFB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.7 status_t LTC_DES_EncryptOfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key[LTC_DES_KEY_SIZE]*)

Encrypts DES using OFB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key	Input key to use for encryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.8 status_t LTC_DES_DecryptOfbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key[LTC_DES_KEY_SIZE]*)

Decrypts DES using OFB block mode.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key	Input key to use for decryption

Returns

Status from encrypt/decrypt operation

31.9.6.2.9 status_t LTC_DES2_EncryptEcbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using ECB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.10 status_t LTC_DES2_DecryptEcbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using ECB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.11 status_t LTC_DES2_EncryptCbcEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *plaintext,* uint8_t * *ciphertext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CBC block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle

	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.12 status_t LTC_DES2_DecryptCbcEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CBC block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.13 status_t LTC_DES2_EncryptCfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.14 status_t LTC_DES2_DecryptCfbEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial block.
	key1	First input key for key bundle

	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.15 status_t LTC_DES2_EncryptOfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using OFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.16 status_t LTC_DES2_DecryptOfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using OFB block mode with two keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.17 status_t LTC_DES3_EncryptEcbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using ECB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle

	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.18 status_t LTC_DES3_DecryptEcbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using ECB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes. Must be multiple of 8 bytes.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.19 status_t LTC_DES3_EncryptCbcEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CBC block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.20 status_t LTC_DES3_DecryptCbcEDMA (LTC_Type * *base,* ltc_edma_handle_t * *handle,* const uint8_t * *ciphertext,* uint8_t * *plaintext,* uint32_t *size,* const uint8_t *iv[LTC_DES_IV_SIZE],* const uint8_t *key1[LTC_DES_KEY_SIZE],* const uint8_t *key2[LTC_DES_KEY_SIZE],* const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CBC block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input initial vector to combine with the first plaintext block. The iv does not need to be secret, but it must be unpredictable.

	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.21 status_t LTC_DES3_EncryptCfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using CFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and ouput data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.22 status_t LTC_DES3_DecryptCfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using CFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input data in bytes
	iv	Input initial block.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.23 status_t LTC_DES3_EncryptOfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *plaintext*, uint8_t * *ciphertext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Encrypts triple DES using OFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	plaintext	Input plaintext to encrypt
out	*ciphertext*	Output ciphertext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.

	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

31.9.6.2.24 status_t LTC_DES3_DecryptOfbEDMA (LTC_Type * *base*, ltc_edma_handle_t * *handle*, const uint8_t * *ciphertext*, uint8_t * *plaintext*, uint32_t *size*, const uint8_t *iv[LTC_DES_IV_SIZE]*, const uint8_t *key1[LTC_DES_KEY_SIZE]*, const uint8_t *key2[LTC_DES_KEY_SIZE]*, const uint8_t *key3[LTC_DES_KEY_SIZE]*)

Decrypts triple DES using OFB block mode with three keys.

Parameters

	base	LTC peripheral base address
	handle	pointer to ltc_edma_handle_t structure which stores the transaction state.
	ciphertext	Input ciphertext to decrypt
out	*plaintext*	Output plaintext
	size	Size of input and output data in bytes
	iv	Input unique input vector. The OFB mode requires that the IV be unique for each execution of the mode under the given key.
	key1	First input key for key bundle
	key2	Second input key for key bundle
	key3	Third input key for key bundle

Returns

Status from encrypt/decrypt operation

Chapter 32
MPU: Memory Protection Unit

32.1 Overview

The MPU driver provides hardware access control for all memory references generated in the device. Use the MPU driver to program the region descriptors that define memory spaces and their access rights. After initialization, the MPU concurrently monitors the system bus transactions and evaluates the appropriateness.

32.2 Initialization and Deinitialize

To initialize the MPU module, call the MPU_Init() function and provide the user configuration data structure. This function sets the configuration of the MPU module automatically and enables the MPU module.

Note that the configuration start address, end address, the region valid value, and the debugger's access permission for the MPU region 0 cannot be changed.

This is example code to configure the MPU driver:

```
// Defines the MPU memory access permission configuration structure . //
 mpu_low_masters_access_rights_t mpuLowAccessRights =
{
     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_high_masters_access_rights_t mpuHighAccessRights =

{
      false,
      false,
      false,
      false,
      false,
      false,
      false,
      false
};

// Defines the MPU region configuration structure. //
 mpu_region_config_t mpuRegionConfig =
{
     kMPU_RegionNum00,
     0x0,
     0xffffffff,
     mpuLowAccessRights,
```

```
      mpuHighAccessRights,
      0,
      0
};

 // Defines the MPU user configuration structure. //
  mpu_config_t mpuUserConfig =
 {
      mpuRegionConfig,
      NULL
 };

 // Initializes the MPU region 0. //
 MPU_Init(MPU, &mpuUserConfig);
```

32.3 Basic Control Operations

MPU can be enabled/disabled for the entire memory protection region by calling the MPU_Enable(). To save the power for any unused special regions when the entire memory protection region is disabled, call the MPU_RegionEnable().

After MPU initialization, the MPU_SetRegionLowMasterAccessRights() and MPU_SetRegionHighMasterAccessRights() can be used to change the access rights for special master ports and for special region numbers. The MPU_SetRegionConfig can be used to set the whole region with the start/end address with access rights.

The MPU_GetHardwareInfo() API is provided to get the hardware information for the device. The MPU_GetSlavePortErrorStatus() API is provided to get the error status of a special slave port. When an error happens in this port, the MPU_GetDetailErrorAccessInfo() API is provided to get the detailed error information.

Files

- file fsl_mpu.h

Data Structures

- struct mpu_hardware_info_t
  MPU hardware basic information. More...
- struct mpu_access_err_info_t
  MPU detail error access information. More...
- struct mpu_low_masters_access_rights_t
  MPU access rights for low master master port 0 ~ port 3. More...
- struct mpu_high_masters_access_rights_t
  MPU access rights mode for high master port 4 ~ port 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_WORD_LOW_MASTER_SHIFT(n) (n * 6)

MPU low master bit shift.

- #define MPU_WORD_LOW_MASTER_MASK(n) (0x1Fu << MPU_WORD_LOW_MASTER_SHIFT(n))
  MPU low master bit mask.
- #define MPU_WORD_LOW_MASTER_WIDTH 5
  MPU low master bit width.
- #define MPU_WORD_LOW_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_WORD_LOW_MASTER_SHIFT(n))) & MPU_WORD_LOW_MASTER_MASK(n))
  MPU low master priority setting.
- #define MPU_LOW_MASTER_PE_SHIFT(n) (n * 6 + 5)
  MPU low master process enable bit shift.
- #define MPU_LOW_MASTER_PE_MASK(n) (0x1u << MPU_LOW_MASTER_PE_SHIFT(n))
  MPU low master process enable bit mask.
- #define MPU_WORD_MASTER_PE_WIDTH 1
  MPU low master process enable width.
- #define MPU_WORD_MASTER_PE(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_LOW_MASTER_PE_SHIFT(n))) & MPU_LOW_MASTER_PE_MASK(n))
  MPU low master process enable setting.
- #define MPU_WORD_HIGH_MASTER_SHIFT(n) (n * 2 + 24)
  MPU high master bit shift.
- #define MPU_WORD_HIGH_MASTER_MASK(n) (0x03u << MPU_WORD_HIGH_MASTER_SHIFT(n))
  MPU high master bit mask.
- #define MPU_WORD_HIGH_MASTER_WIDTH 2
  MPU high master bit width.
- #define MPU_WORD_HIGH_MASTER(n, x) (((uint32_t)(((uint32_t)(x)) << MPU_WORD_HIGH_MASTER_SHIFT(n))) & MPU_WORD_HIGH_MASTER_MASK(n))
  MPU high master priority setting.

Enumerations

- enum mpu_region_num_t
  MPU region number.
- enum mpu_master_t
  MPU master number.
- 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 = 4U,
  kMPU_Slave1 = 3U,
  kMPU_Slave2 = 2U,
  kMPU_Slave3 = 1U,
  kMPU_Slave4 = 0U }
  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 master port 0 ~ port 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 master port 0 ~ port 3.

Driver version

- #define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
MPU driver version 2.0.0.

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, mpu_region_num_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, mpu_region_num_t regionNum, uint32_t startAddr, uint32_t endAddr)
  Sets the region start and end address.
- void MPU_SetRegionLowMasterAccessRights (MPU_Type *base, mpu_region_num_t regionNum, mpu_master_t masterNum, const mpu_low_masters_access_rights_t *accessRights)
  Sets the MPU region access rights for low master port 0 ~ port 3.
- void MPU_SetRegionHighMasterAccessRights (MPU_Type *base, mpu_region_num_t regionNum, mpu_master_t masterNum, const mpu_high_masters_access_rights_t *accessRights)
  Sets the MPU region access rights for high master port 4 ~ port 7.
- 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.

32.4 Data Structure Documentation

32.4.1 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.

32.4.1.0.24.1 Field Documentation

32.4.1.0.24.1.1 uint8_t mpu_hardware_info_t::hardwareRevisionLevel

32.4.1.0.24.1.2 uint8_t mpu_hardware_info_t::slavePortsNumbers

32.4.1.0.24.1.3 mpu_region_total_num_t mpu_hardware_info_t::regionsNumbers

32.4.2 struct mpu_access_err_info_t

Data Fields

- mpu_master_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.

32.4.2.0.24.2 Field Documentation

32.4.2.0.24.2.1 mpu_master_t mpu_access_err_info_t::master

32.4.2.0.24.2.2 mpu_err_attributes_t mpu_access_err_info_t::attributes

32.4.2.0.24.2.3 mpu_err_access_type_t mpu_access_err_info_t::accessType

32.4.2.0.24.2.4 mpu_err_access_control_t mpu_access_err_info_t::accessControl

32.4.2.0.24.2.5 uint32_t mpu_access_err_info_t::address

32.4.3 struct mpu_low_masters_access_rights_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.

32.4.3.0.24.3 Field Documentation

32.4.3.0.24.3.1 mpu_supervisor_access_rights_t mpu_low_masters_access_rights_t::superAccessRights

32.4.3.0.24.3.2 mpu_user_access_rights_t mpu_low_masters_access_rights_t::userAccessRights

32.4.4 struct mpu_high_masters_access_rights_t

Data Fields

- bool writeEnable
  Enables or disables write permission.
- bool readEnable
  Enables or disables read permission.

32.4.4.0.24.4 Field Documentation

32.4.4.0.24.4.1 bool mpu_high_masters_access_rights_t::writeEnable

32.4.4.0.24.4.2 bool mpu_high_masters_access_rights_t::readEnable

32.4.5 struct mpu_region_config_t

This structure is used to configure the regionNum region. The accessRights1[0] ~ accessRights1[3] are used to configure the four low master numbers: master 0 ~ master 3. The accessRights2[0] ~ accessRights2[3] are used to configure the four high master numbers: master 4 ~ master 7. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note: MPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. MPU protection does not allow writes from the core to affect the "regionNum 0" start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantee the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is kMPU_RegionNum00.

Data Fields

- mpu_region_num_t regionNum
  MPU region number.
- uint32_t startAddress
  Memory region start address.
- uint32_t endAddress
  Memory region end address.
- mpu_low_masters_access_rights_t accessRights1 [4]
  Low masters access permission.
- mpu_high_masters_access_rights_t accessRights2 [4]
  High masters access permission.

32.4.5.0.24.5 Field Documentation

32.4.5.0.24.5.1 mpu_region_num_t mpu_region_config_t::regionNum

32.4.5.0.24.5.2 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.

32.4.5.0.24.5.3 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.

32.4.5.0.24.5.4 mpu_low_masters_access_rights_t mpu_region_config_t::accessRights1[4]

32.4.5.0.24.5.5 mpu_high_masters_access_rights_t mpu_region_config_t::accessRights2[4]

32.4.6 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.

32.4.6.0.24.6 Field Documentation

32.4.6.0.24.6.1 mpu_region_config_t mpu_config_t::regionConfig

32.4.6.0.24.6.2 struct _mpu_config* mpu_config_t::next

32.5 Macro Definition Documentation

32.5.1 #define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

32.5.2 #define MPU_WORD_LOW_MASTER_SHIFT(*n*) (n * 6)

32.5.3 #define MPU_WORD_LOW_MASTER_MASK(*n*) (0x1Fu << MPU_WORD_LOW_MASTER_SHIFT(n))

32.5.4 #define MPU_WORD_LOW_MASTER_WIDTH 5

32.5.5 #define MPU_WORD_LOW_MASTER(*n, x*) (((uint32_t)(((uint32_t)(x)) << MPU_WORD_LOW_MASTER_SHIFT(n))) & MPU_WORD_LOW_MASTER_MASK(n))

32.5.6 #define MPU_LOW_MASTER_PE_SHIFT(*n*) (n * 6 + 5)

32.5.7 #define MPU_LOW_MASTER_PE_MASK(*n*) (0x1u << MPU_LOW_MASTER_PE_SHIFT(n))

32.5.8 #define MPU_WORD_MASTER_PE_WIDTH 1

32.5.9 #define MPU_WORD_MASTER_PE(*n, x*) (((uint32_t)(((uint32_t)(x)) << MPU_LOW_MASTER_PE_SHIFT(n))) & MPU_LOW_MASTER_PE_MASK(n))

32.5.10 #define MPU_WORD_HIGH_MASTER_SHIFT(*n*) (n * 2 + 24)

32.5.11 #define MPU_WORD_HIGH_MASTER_MASK(*n*) (0x03u << MPU_WORD_HIGH_MASTER_SHIFT(n))

32.5.12 #define MPU_WORD_HIGH_MASTER_WIDTH 2

32.5.13 #define MPU_WORD_HIGH_MASTER(*n, x*) (((uint32_t)(((uint32_t)(x)) << MPU_WORD_HIGH_MASTER_SHIFT(n))) & MPU_WORD_HIGH_MASTER_MASK(n))

32.6 Enumeration Type Documentation

32.6.1 enum mpu_region_num_t

32.6.2 enum mpu_master_t

32.6.3 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.

32.6.4 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.

32.6.5 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.

32.6.6 enum mpu_err_access_type_t

Enumerator

kMPU_ErrTypeRead MPU error access type — read.
kMPU_ErrTypeWrite MPU error access type — write.

32.6.7 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.

32.6.8 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.

32.6.9 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.

32.7 Function Documentation

32.7.1 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.

32.7.2 void MPU_Deinit (MPU_Type * *base*)

Parameters

base	MPU peripheral base address.

32.7.3 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.

32.7.4 static void MPU_RegionEnable (MPU_Type * *base,* mpu_region_num_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.

32.7.5 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".

32.7.6 void MPU_SetRegionConfig (MPU_Type * *base,* const mpu_region_config_t * *regionConfig*)

Note: Due to the MPU protection, the kMPU_RegionNum00 does not allow writes from the 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".

32.7.7 void MPU_SetRegionAddr (MPU_Type * *base,* mpu_region_num_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 actual end address used by MPU is 31-modulo-32 byte address. Note: Due to the MPU protection, the startAddr and endAddr can't be changed by the core when regionNum is "kMPU_Region-Num00".

Parameters

base	MPU peripheral base address.
regionNum	MPU region number.
startAddr	Region start address.
endAddr	Region end address.

32.7.8 void MPU_SetRegionLowMasterAccessRights (MPU_Type * *base,* mpu_region_num_t *regionNum,* mpu_master_t *masterNum,* const mpu_low_masters_access_rights_t * *accessRights*)

This can be used to change the region access rights for any master port for any region.

Parameters

base	MPU peripheral base address.
regionNum	MPU region number.
masterNum	MPU master number. Should range from kMPU_Master0 ~ kMPU_Master3.
accessRights	The pointer to the MPU access rights configuration. See "mpu_low_masters_access_rights_t".

32.7.9 void MPU_SetRegionHighMasterAccessRights (MPU_Type * *base*, mpu_region_num_t *regionNum*, mpu_master_t *masterNum*, const mpu_high_masters_access_rights_t * *accessRights*)

This can be used to change the region access rights for any master port for any region.

Parameters

base	MPU peripheral base address.
regionNum	MPU region number.
masterNum	MPU master number. Should range from kMPU_Master4 ~ kMPU_Master7.
accessRights	The pointer to the MPU access rights configuration. See "mpu_high_masters_access_rights_t".

32.7.10 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.

32.7.11 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 33
Notification Framework

33.1 Overview

This section describes the programming interface of the Notifier driver.

33.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

The configuration transition includes 3 steps:

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.

The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.

1. After the "BEFORE" message is processed successfully, the system changes to the new configuration.
2. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This is an example to use the Notifier in the Power Manager application:

~~~~~{.c}

```
#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_PowerModeSwit
    ...
    ...
    /* Power mode switch
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex, kNOTIFIER_PolicyAgreement);
}
```

~~~~~{.c}

## 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 User can use this function to execute specific operations in configuration switch.*
- typedef status_t(∗ notifier_callback_t )(notifier_notification_block_t ∗notify, void ∗data)
  *Callback prototype.*

## Enumerations

- enum _notifier_status {
  kStatus_NOTIFIER_ErrorNotificationBefore,
  kStatus_NOTIFIER_ErrorNotificationAfter }
  *Notifier error codes.*
- enum notifier_policy_t {
  kNOTIFIER_PolicyAgreement,
  kNOTIFIER_PolicyForcible }
  *Notifier policies.*
- enum notifier_notification_type_t {
  kNOTIFIER_NotifyRecover = 0x00U,
  kNOTIFIER_NotifyBefore = 0x01U,
  kNOTIFIER_NotifyAfter = 0x02U }
  *Notification type.*
- enum notifier_callback_type_t {
  kNOTIFIER_CallbackBefore = 0x01U,
  kNOTIFIER_CallbackAfter = 0x02U,
  kNOTIFIER_CallbackBeforeAfter = 0x03U }
  *The callback type, indicates what kinds of notification the callback handles.*

## Functions

- status_t NOTIFIER_CreateHandle (notifier_handle_t ∗notifierHandle, notifier_user_config_t ∗∗configs, uint8_t configsNumber, notifier_callback_config_t ∗callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void ∗userData)
  *Create Notifier handle.*
~~~~~

- status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)
  Switch configuration according to a pre-defined structure.
- uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)
  This function returns the last failed notification callback.

33.3 Data Structure Documentation

33.3.1 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.

33.3.1.0.24.7 Field Documentation

33.3.1.0.24.7.1 notifier_user_config_t* notifier_notification_block_t::targetConfig

33.3.1.0.24.7.2 notifier_policy_t notifier_notification_block_t::policy

33.3.1.0.24.7.3 notifier_notification_type_t notifier_notification_block_t::notifyType

33.3.2 struct notifier_callback_config_t

This structure holds configuration of callbacks. Callbacks of this type are expected to be statically allocated. This structure contains following application-defined data: callback - pointer to the callback function callbackType - specifies when the callback is called callbackData - pointer to the data passed to the callback.

Data Fields

- notifier_callback_t callback
  Pointer to the callback function.
- notifier_callback_type_t callbackType
  Callback type.
- void * callbackData
  Pointer to the data passed to the callback.

33.3.2.0.24.8 Field Documentation

33.3.2.0.24.8.1 notifier_callback_t notifier_callback_config_t::callback

33.3.2.0.24.8.2 notifier_callback_type_t notifier_callback_config_t::callbackType

33.3.2.0.24.8.3 void* notifier_callback_config_t::callbackData

33.3.3 struct notifier_handle_t

Notifier handle structure. Contains data necessary for Notifier proper function. Stores references to registered configurations, callbacks, information about their numbers, user function, user data and other internal data. NOTIFIER_CreateHandle() must be called to intialize 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.

33.3.3.0.24.9 Field Documentation

33.3.3.0.24.9.1 notifier_user_config_t** notifier_handle_t::configsTable

33.3.3.0.24.9.2 uint8_t notifier_handle_t::configsNumber

33.3.3.0.24.9.3 notifier_callback_config_t* notifier_handle_t::callbacksTable

33.3.3.0.24.9.4 uint8_t notifier_handle_t::callbacksNumber

33.3.3.0.24.9.5 uint8_t notifier_handle_t::errorCallbackIndex

33.3.3.0.24.9.6 uint8_t notifier_handle_t::currentConfigIndex

33.3.3.0.24.9.7 notifier_user_function_t notifier_handle_t::userFunction

33.3.3.0.24.9.8 void* notifier_handle_t::userData

33.4 Typedef Documentation

33.4.1 typedef void notifier_user_config_t

Reference of user defined configuration is stored in an array, notifer switch between these configurations based on this array.

33.4.2 typedef status_t(* notifier_user_function_t)(notifier_user_config_t *targetConfig, void *userData)

Before and after this function execution, different notification will be sent to registered callbacks. If this function returns any error code, NOTIFIER_SwitchConfig() will exit.

Parameters

targetConfig	target Configuration.
userData	Refers to other specific data passed to user function.

Returns

An error code or kStatus_Success.

33.4.3 typedef status_t(* notifier_callback_t)(notifier_notification_block_t *notify, void *data)

Declaration of callback. It is common for registered callbacks. Reference to function of this type is part of notifier_callback_config_t callback configuration structure. Depending on callback type, function of this

prototype is called (see NOTIFIER_SwitchConfig()) before configuration switch, after it or in both cases to notify about the switch progress (see notifier_callback_type_t). When called, type of the notification is passed as parameter along with reference to the target configuration structure (see notifier_notification_block_t) and any data passed during the callback registration. When notified before configuration switch, depending on the configuration switch policy (see notifier_policy_t) the callback may deny the execution of user function by returning any error code different from kStatus_Success (see NOTIFIER_SwitchConfig()).

Parameters

notify	Notification block.
data	Callback data. Refers to the data passed during callback registration. Intended to pass any driver or application data such as internal state information.

Returns

An error code or kStatus_Success.

33.5 Enumeration Type Documentation

33.5.1 enum _notifier_status

Used as return value of Notifier functions.

Enumerator

kStatus_NOTIFIER_ErrorNotificationBefore Error occurs during send "BEFORE" notification.
kStatus_NOTIFIER_ErrorNotificationAfter Error occurs during send "AFTER" notification.

33.5.2 enum notifier_policy_t

Defines whether user function execution is forced or not. For kNOTIFIER_PolicyForcible, the user function is executed regardless of the callback results, while kNOTIFIER_PolicyAgreement policy is used to exit NOTIFIER_SwitchConfig() when any of the callbacks returns error code. See also NOTIFIER_SwitchConfig() description.

Enumerator

kNOTIFIER_PolicyAgreement NOTIFIER_SwitchConfig() method is exited when any of the callbacks returns error code.
kNOTIFIER_PolicyForcible user function is executed regardless of the results.

33.5.3 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.

33.5.4 enum notifier_callback_type_t

Used in the callback configuration structure (notifier_callback_config_t) to specify when the registered callback is called during configuration switch initiated by NOTIFIER_SwitchConfig(). Callback can be invoked in following situations:

- before the configuration switch (Callback return value can affect NOTIFIER_SwitchConfig() execution. Refer to the NOTIFIER_SwitchConfig() and notifier_policy_t documentation).
- after unsuccessful attempt to switch configuration
- after sucecessful configuration switch

Enumerator

kNOTIFIER_CallbackBefore Callback handles BEFORE notification.
kNOTIFIER_CallbackAfter Callback handles AFTER notification.
kNOTIFIER_CallbackBeforeAfter Callback handles BEFORE and AFTER notification.

33.6 Function Documentation

33.6.1 status_t NOTIFIER_CreateHandle (notifier_handle_t * *notifierHandle*, notifier_user_config_t ** *configs*, uint8_t *configsNumber*, notifier_callback_config_t * *callbacks*, uint8_t *callbacksNumber*, notifier_user_function_t *userFunction*, void * *userData*)

Parameters

notifierHandle	A pointer to notifier handle
configs	A pointer to an array with references to all configurations which is handled by the Notifier.

configsNumber	Number of configurations. Size of configs array.
callbacks	A pointer to an array of callback configurations. If there are no callbacks to register during Notifier initialization, use NULL value.
callbacks-Number	Number of registered callbacks. Size of callbacks array.
userFunction	user function.
userData	user data passed to user function.

Returns

An error code or kStatus_Success.

33.6.2 status_t NOTIFIER_SwitchConfig (notifier_handle_t * *notifierHandle*, uint8_t *configIndex*, notifier_policy_t *policy*)

This function sets the system to the target configuration. Before transition, the Notifier sends notifications to all callbacks registered to the callback table. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array. The same order is used for before and after switch notifications. The notifications before the configuration switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the configuration change, further execution of this function depends on the notifier policy: the configuration change is either forced (kNOTIFIER_PolicyForcible) or exited (kNOTIFIER_PolicyAgreement). When configuration change is forced, the result of the before switch notifications are ignored. If agreement is required, if any callback returns an error code then further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked The index of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and NOTIFIER_Get-ErrorCallback() can be used to get it. Regardless of the policies, if any callback returned an error code, an error code denoting in which phase the error occurred is returned when NOTIFIER_SwitchConfig() exits.

Parameters

notifierHandle	pointer to notifier handle
configIndex	Index of the target configuration.
policy	Transaction policy, kNOTIFIER_PolicyAgreement or kNOTIFIER_PolicyForcible.

Returns

An error code or kStatus_Success.

33.6.3 uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t * *notifierHandle*)

This function returns index of the last callback that failed during the configuration switch while the last NOTIFIER_SwitchConfig() was called. If the last NOTIFIER_SwitchConfig() call ended successfully value equal to callbacks number is returned. Returned value represents index in the array of static call-backs.

Parameters

notifierHandle	pointer to notifier handle

Returns

Callback index of last failed callback or value equal to callbacks count.

Chapter 34
PDB: Programmable Delay Block

34.1 Overview

The KSDK provides a peripheral driver for the Programmable Delay Block (PDB) module of Kinetis devices.

Overview

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 time with an interrupt. The counter increases automatically with the divided clock signal after it is triggered to start by an external trigger input or the software trigger. There are "milestones" for output trigger event. When the counter is equal to any of these "milestones", the corresponding trigger is generated and sent out to other modules. These "milestones" are for the following:

- Counter delay interrupt, which is the interrupt for the PDB module
- ADC pre-trigger to trigger the ADC conversion
- DAC interval trigger to trigger the DAC buffer and move the buffer read pointer
- Pulse-out triggers to generate a single of rising and falling edges, which can be assembled to a window.

The "milestone" values have a flexible load mode. To call the APIs to set these value is equivalent to writing data to their buffer. The loading event occurs as the load mode describes. This design ensures that all "milestones" can be updated at the same time.

Typical use case

Working as basic DPB counter with a PDB interrupt.

```
int main(void)
{
    // ...
    EnableIRQ(DEMO_PDB_IRQ_ID);

    // ...
    // Configures the PDB counter.
    PDB_GetDefaultConfig(&pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
       value is less than or equal to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
      kPDB_DelayInterruptEnable);
    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    while (1)
```

```
    {
        // ...
        g_PdbDelayInterruptFlag = false;
        PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
        while (!g_PdbDelayInterruptFlag)
        {
        }
    }
 }

void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    // ...
    g_PdbDelayInterruptFlag = true;
    PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
      kPDB_DelayEventFlag);
}
```

Working with an additional trigger. The ADC trigger is used as an example.

```
void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
      kPDB_DelayEventFlag);
    g_PdbDelayInterruptCounter++;
    g_PdbDelayInterruptFlag = true;
}

void DEMO_PDB_InitADC(void)
{
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Init(DEMO_PDB_ADC_INSTANCE, &adc16ConfigStruct);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, false);
    ADC16_DoAutoCalibration(DEMO_PDB_ADC_INSTANCE);
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, true);

    adc16ChannelConfigStruct.channelNumber = DEMO_PDB_ADC_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted = true; /* Enable the interrupt.
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE
    ADC16_SetChannelConfig(DEMO_PDB_ADC_INSTANCE, DEMO_PDB_ADC_CHANNEL_GROUP, &adc16ChannelConfigStruct);
}

void DEMO_PDB_ADC_IRQ_HANDLER_FUNCTION(void)
{
    uint32_t tmp32;

    tmp32 = ADC16_GetChannelConversionValue(DEMO_PDB_ADC_INSTANCE, DEMO_PDB_ADC_CHANNEL_GROUP); /* Read to
       clear COCO flag.
    g_AdcInterruptCounter++;
    g_AdcInterruptFlag = true;
}

int main(void)
{
    // ...

    EnableIRQ(DEMO_PDB_IRQ_ID);
    EnableIRQ(DEMO_PDB_ADC_IRQ_ID);
```

```
    // ...

    // Configures the PDB counter.
    PDB_GetDefaultConfig(&pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay value is less than or equal
       to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE, kPDB_DelayInterruptEnable);

    // Configures the ADC pre-trigger.
    pdbAdcPreTriggerConfigStruct.enablePreTriggerMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
    pdbAdcPreTriggerConfigStruct.enableOutputMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
    pdbAdcPreTriggerConfigStruct.enableBackToBackOperationMask = 0U;
    PDB_SetADCPreTriggerConfig(DEMO_PDB_INSTANCE, DEMO_PDB_ADC_TRIGGER_CHANNEL,
       &pdbAdcPreTriggerConfigStruct);
    PDB_SetADCPreTriggerDelayValue(DEMO_PDB_INSTANCE,
                                   DEMO_PDB_ADC_TRIGGER_CHANNEL, DEMO_PDB_ADC_PRETRIGGER_CHANNEL, 200U);
                        // The available pre-trigger delay value is less than or equal to the modulus
       value.

    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    // Configures the ADC.
    DEMO_PDB_InitADC();

    while (1)
    {
        g_PdbDelayInterruptFlag = false;
        g_AdcInterruptFlag = false;
        PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
        while ((!g_PdbDelayInterruptFlag) || (!g_AdcInterruptFlag))
        {
        }
        // ...
    }
}
```

Files

- file fsl_pdb.h

Data Structures

- struct pdb_config_t
  PDB module configuration. More...
- struct pdb_adc_pretrigger_config_t
  PDB ADC Pre-Trigger configuration. More...
- struct pdb_dac_trigger_config_t
  PDB DAC trigger configuration. More...

Enumerations

- enum _pdb_status_flags {
  kPDB_LoadOKFlag = PDB_SC_LDOK_MASK,
  kPDB_DelayEventFlag = PDB_SC_PDBIF_MASK }
  PDB flags.

- enum _pdb_adc_pretrigger_flags {
  kPDB_ADCPreTriggerChannel0Flag = PDB_S_CF(1U << 0),
  kPDB_ADCPreTriggerChannel1Flag = PDB_S_CF(1U << 1),
  kPDB_ADCPreTriggerChannel0ErrorFlag = PDB_S_ERR(1U << 0),
  kPDB_ADCPreTriggerChannel1ErrorFlag = PDB_S_ERR(1U << 1) }
  PDB ADC PreTrigger channel flags.
- enum _pdb_interrupt_enable {
  kPDB_SequenceErrorInterruptEnable = PDB_SC_PDBEIE_MASK,
  kPDB_DelayInterruptEnable = PDB_SC_PDBIE_MASK }
  PDB buffer interrupts.
- enum pdb_load_value_mode_t {
  kPDB_LoadValueImmediately = 0U,
  kPDB_LoadValueOnCounterOverflow = 1U,
  kPDB_LoadValueOnTriggerInput = 2U,
  kPDB_LoadValueOnCounterOverflowOrTriggerInput = 3U }
  PDB load value mode.
- enum pdb_prescaler_divider_t {
  kPDB_PrescalerDivider1 = 0U,
  kPDB_PrescalerDivider2 = 1U,
  kPDB_PrescalerDivider4 = 2U,
  kPDB_PrescalerDivider8 = 3U,
  kPDB_PrescalerDivider16 = 4U,
  kPDB_PrescalerDivider32 = 5U,
  kPDB_PrescalerDivider64 = 6U,
  kPDB_PrescalerDivider128 = 7U }
  Prescaler divider.
- enum pdb_divider_multiplication_factor_t {
  kPDB_DividerMultiplicationFactor1 = 0U,
  kPDB_DividerMultiplicationFactor10 = 1U,
  kPDB_DividerMultiplicationFactor20 = 2U,
  kPDB_DividerMultiplicationFactor40 = 3U }
  Multiplication factor select for prescaler.
- enum pdb_trigger_input_source_t {

kPDB_TriggerInput0 = 0U,
kPDB_TriggerInput1 = 1U,
kPDB_TriggerInput2 = 2U,
kPDB_TriggerInput3 = 3U,
kPDB_TriggerInput4 = 4U,
kPDB_TriggerInput5 = 5U,
kPDB_TriggerInput6 = 6U,
kPDB_TriggerInput7 = 7U,
kPDB_TriggerInput8 = 8U,
kPDB_TriggerInput9 = 9U,
kPDB_TriggerInput10 = 10U,
kPDB_TriggerInput11 = 11U,
kPDB_TriggerInput12 = 12U,
kPDB_TriggerInput13 = 13U,
kPDB_TriggerInput14 = 14U,
kPDB_TriggerSoftware = 15U }
Trigger input source.

Driver version

- #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  PDB driver version 2.0.1.

Initialization

- void PDB_Init (PDB_Type *base, const pdb_config_t *config)
  Initializes the PDB module.
- void PDB_Deinit (PDB_Type *base)
  De-initializes the PDB module.
- void PDB_GetDefaultConfig (pdb_config_t *config)
  Initializes the PDB user configure structure.
- static void PDB_Enable (PDB_Type *base, bool enable)
  Enables the PDB module.

Basic Counter

- static void PDB_DoSoftwareTrigger (PDB_Type *base)
  Triggers the PDB counter by software.
- static void PDB_DoLoadValues (PDB_Type *base)
  Loads the counter values.
- static void PDB_EnableDMA (PDB_Type *base, bool enable)
  Enables the DMA for the PDB module.
- static void PDB_EnableInterrupts (PDB_Type *base, uint32_t mask)
  Enables the interrupts for the PDB module.
- static void PDB_DisableInterrupts (PDB_Type *base, uint32_t mask)
  Disables the interrupts for the PDB module.
- static uint32_t PDB_GetStatusFlags (PDB_Type *base)
  Gets the status flags of the PDB module.

- static void PDB_ClearStatusFlags (PDB_Type *base, uint32_t mask)
  Clears the status flags of the PDB module.
- static void PDB_SetModulusValue (PDB_Type *base, uint32_t value)
  Specifies the period of the counter.
- static uint32_t PDB_GetCounterValue (PDB_Type *base)
  Gets the PDB counter's current value.
- static void PDB_SetCounterDelayValue (PDB_Type *base, uint32_t value)
  Sets the value for PDB counter delay event.

ADC Pre-Trigger

- static void PDB_SetADCPreTriggerConfig (PDB_Type *base, uint32_t channel, pdb_adc_pretrigger_config_t *config)
  Configures the ADC PreTrigger in PDB module.
- static void PDB_SetADCPreTriggerDelayValue (PDB_Type *base, uint32_t channel, uint32_t preChannel, uint32_t value)
  Sets the value for ADC Pre-Trigger delay event.
- static uint32_t PDB_GetADCPreTriggerStatusFlags (PDB_Type *base, uint32_t channel)
  Gets the ADC Pre-Trigger's status flags.
- static void PDB_ClearADCPreTriggerStatusFlags (PDB_Type *base, uint32_t channel, uint32_t mask)
  Clears the ADC Pre-Trigger's status flags.

Pulse-Out Trigger

- static void PDB_EnablePulseOutTrigger (PDB_Type *base, uint32_t channelMask, bool enable)
  Enables the pulse out trigger channels.
- static void PDB_SetPulseOutTriggerDelayValue (PDB_Type *base, uint32_t channel, uint32_t value1, uint32_t value2)
  Sets event values for pulse out trigger.

34.2 Data Structure Documentation

34.2.1 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.

34.2.1.0.24.10 Field Documentation

34.2.1.0.24.10.1 pdb_load_value_mode_t pdb_config_t::loadValueMode

34.2.1.0.24.10.2 pdb_prescaler_divider_t pdb_config_t::prescalerDivider

34.2.1.0.24.10.3 pdb_divider_multiplication_factor_t pdb_config_t::dividerMultiplicationFactor

34.2.1.0.24.10.4 pdb_trigger_input_source_t pdb_config_t::triggerInputSource

34.2.1.0.24.10.5 bool pdb_config_t::enableContinuousMode

34.2.2 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.

34.2.2.0.24.11 Field Documentation

34.2.2.0.24.11.1 uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask

34.2.2.0.24.11.2 uint32_t pdb_adc_pretrigger_config_t::enableOutputMask

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

34.2.2.0.24.11.3 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.

34.2.3 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.

34.2.3.0.24.12 Field Documentation

34.2.3.0.24.12.1 bool pdb_dac_trigger_config_t::enableExternalTriggerInput

34.2.3.0.24.12.2 bool pdb_dac_trigger_config_t::enableIntervalTrigger

34.3 Macro Definition Documentation

34.3.1 #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

34.4 Enumeration Type Documentation

34.4.1 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.

34.4.2 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.

34.4.3 enum _pdb_interrupt_enable

Enumerator

kPDB_SequenceErrorInterruptEnable PDB sequence error interrupt enable.
kPDB_DelayInterruptEnable PDB delay interrupt enable.

34.4.4 enum pdb_load_value_mode_t

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx_SC[LDOK]). These values are for:

- PDB counter (PDBx_MOD, PDBx_IDLY)
- ADC trigger (PDBx_CHnDLYm)

- DAC trigger (PDBx_DACINTx)
- CMP trigger (PDBx_POyDLY)

Enumerator

kPDB_LoadValueImmediately Load immediately after 1 is written to LDOK.
kPDB_LoadValueOnCounterOverflow Load when the PDB counter overflows (reaches the MOD register value).
kPDB_LoadValueOnTriggerInput Load a trigger input event is detected.
kPDB_LoadValueOnCounterOverflowOrTriggerInput Load either when the PDB counter overflows or a trigger input is detected.

34.4.5 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.

34.4.6 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.

34.4.7 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. Refer to 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.

34.5 Function Documentation

34.5.1 void PDB_Init (PDB_Type * *base,* const pdb_config_t * *config*)

This function is to make the initialization for PDB module. The operations includes are:

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

base	PDB peripheral base address.
config	Pointer to configuration structure. See "pdb_config_t".

34.5.2 void PDB_Deinit (PDB_Type * *base*)

Parameters

base	PDB peripheral base address.

34.5.3 void PDB_GetDefaultConfig (pdb_config_t * *config*)

This function initializes the user configure structure to default value. the default value are:

```
config->loadValueMode = kPDB_LoadValueImmediately;
config->prescalerDivider = kPDB_PrescalerDivider1;
config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1;
config->triggerInputSource = kPDB_TriggerSoftware;
config->enableContinuousMode = false;
```

Parameters

config	Pointer to configuration structure. See "pdb_config_t".

34.5.4 static void PDB_Enable (PDB_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the module or not.

34.5.5 static void PDB_DoSoftwareTrigger (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

34.5.6 static void PDB_DoLoadValues (PDB_Type * *base*) [inline], [static]

This function is to load the counter values from their internal buffer. See "pdb_load_value_mode_t" about PDB's load mode.

Parameters

base	PDB peripheral base address.

34.5.7 static void PDB_EnableDMA (PDB_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
enable	Enable the feature or not.

34.5.8 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".

34.5.9 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".

34.5.10 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".

34.5.11 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".

34.5.12 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.

34.5.13 static uint32_t PDB_GetCounterValue (PDB_Type * *base*) [inline], [static]

Parameters

base	PDB peripheral base address.

Returns

PDB counter's current value.

34.5.14 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.

34.5.15 static void PDB_SetADCPreTriggerConfig (PDB_Type * *base,* uint32_t *channel,* pdb_adc_pretrigger_config_t * *config*) [inline], [static]

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.
config	Pointer to configuration structure. See "pdb_adc_pretrigger_config_t".

34.5.16 static void PDB_SetADCPreTriggerDelayValue (PDB_Type * *base,* uint32_t *channel,* uint32_t *preChannel,* uint32_t *value*) [inline], [static]

This function is to set the value for ADC Pre-Trigger delay event. IT Specifies the delay value for the channel's corresponding pre-trigger. The pre-trigger asserts when the PDB counter is equal to the setting value here.

Parameters

base	PDB peripheral base address.
channel	Channel index for ADC instance.
preChannel	Channel group index for ADC instance.
value	Setting value for ADC Pre-Trigger delay event. 16-bit is available.

34.5.17 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".

34.5.18 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".

34.5.19 static void PDB_EnablePulseOutTrigger (PDB_Type * *base,* uint32_t *channelMask,* bool *enable*) [inline], [static]

Parameters

base	PDB peripheral base address.
channelMask	Channel mask value for multiple pulse out trigger channel.
enable	Enable the feature or not.

34.5.20 static void PDB_SetPulseOutTriggerDelayValue (PDB_Type * *base,* uint32_t *channel,* uint32_t *value1,* uint32_t *value2*) [inline], [static]

This function is used to set event values for pulse output trigger. These pulse output trigger delay values specify the delay for the PDB Pulse-Out. Pulse-Out goes high when the PDB counter is equal to the pulse output high value (value1). Pulse-Out goes low when the PDB counter is equal to the pulse output low value (value2).

Parameters

base	PDB peripheral base address.
channel	Channel index for pulse out trigger channel.
value1	Setting value for pulse out high.
value2	Setting value for pulse out low.

Chapter 35
PIT: Periodic Interrupt Timer Driver

35.1 Overview

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

Files

- file fsl_pit.h

Data Structures

- struct pit_config_t
  PIT config structure. More...

Enumerations

- enum pit_chnl_t {
  kPIT_Chnl_0 = 0U,
  kPIT_Chnl_1,
  kPIT_Chnl_2,
  kPIT_Chnl_3 }
  List of PIT channels.
- enum pit_interrupt_enable_t { kPIT_TimerInterruptEnable = PIT_TCTRL_TIE_MASK }
  List of PIT interrupts.
- enum pit_status_flags_t { kPIT_TimerFlag = PIT_TFLG_TIF_MASK }
  List of PIT status flags.

Driver version

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

Initialization and deinitialization

- void PIT_Init (PIT_Type *base, const pit_config_t *config)
  Ungates the PIT clock, enables the PIT module and configures the peripheral for basic operation.
- void PIT_Deinit (PIT_Type *base)
  Gate the PIT clock and disable the PIT module.
- static void PIT_GetDefaultConfig (pit_config_t *config)
  Fill in the PIT config struct with the default settings.

Interrupt Interface

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

Enables the selected PIT interrupts.
- static void PIT_DisableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)
  Disables the selected PIT interrupts.
- static uint32_t PIT_GetEnabledInterrupts (PIT_Type *base, pit_chnl_t channel)
  Gets the enabled PIT interrupts.

Status Interface

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

Read and Write the timer period

- static void PIT_SetTimerPeriod (PIT_Type *base, pit_chnl_t channel, uint32_t count)
  Sets the timer period in units of count.
- static uint32_t PIT_GetCurrentTimerCount (PIT_Type *base, pit_chnl_t channel)
  Reads the current timer counting value.

Timer Start and Stop

- static void PIT_StartTimer (PIT_Type *base, pit_chnl_t channel)
  Starts the timer counting.
- static void PIT_StopTimer (PIT_Type *base, pit_chnl_t channel)
  Stops the timer counting.

35.2 Data Structure Documentation

35.2.1 struct pit_config_t

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

The config struct can be made const so it resides in flash

Data Fields

- bool enableRunInDebug
  true: Timers run in debug mode; false: Timers stop in debug mode

35.3 Enumeration Type Documentation

35.3.1 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.

35.3.2 enum pit_interrupt_enable_t

Enumerator

kPIT_TimerInterruptEnable Timer interrupt enable.

35.3.3 enum pit_status_flags_t

Enumerator

kPIT_TimerFlag Timer flag.

35.4 Function Documentation

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

Note

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

Parameters

base	PIT peripheral base address
config	Pointer to user's PIT config structure

35.4.2 void PIT_Deinit (PIT_Type * *base*)

Parameters

base	PIT peripheral base address

35.4.3 static void PIT_GetDefaultConfig (pit_config_t * *config*) [inline], [static]

The default values are:

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

Parameters

config	Pointer to user's PIT config structure.

35.4.4 static void PIT_EnableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

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

35.4.5 static void PIT_DisableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	PIT peripheral base address
channel	Timer channel number

mask	The interrupts to disable. This is a logical OR of members of the enumeration pit_interrupt_enable_t

35.4.6 static uint32_t PIT_GetEnabledInterrupts (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

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

35.4.7 static uint32_t PIT_GetStatusFlags (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

Parameters

base	PIT peripheral base address
channel	Timer channel number

Returns

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

35.4.8 static void PIT_ClearStatusFlags (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

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

35.4.9 static void PIT_SetTimerPeriod (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *count*) [inline], [static]

Timers begin counting from the value set by this function until it reaches 0, then it will generate an interrupt and load this regiter value again. Writing a new value to this register will not restart the timer; instead the value will be loaded after the timer expires.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	PIT peripheral base address
channel	Timer channel number
count	Timer period in units of ticks

35.4.10 static uint32_t PIT_GetCurrentTimerCount (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

User can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	PIT peripheral base address

channel	Timer channel number

Returns

Current timer counting value in ticks

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

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

Parameters

base	PIT peripheral base address
channel	Timer channel number.

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

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

Parameters

base	PIT peripheral base address
channel	Timer channel number.

Chapter 36
PMC: Power Management Controller

36.1 Overview

The KSDK provides a Peripheral driver for the Power Management Controller (PMC) module of Kinetis devices. The PMC module contains internal voltage regulator, power on reset, low voltage detect system, and high voltage detect system.

Files

- file fsl_pmc.h

Data Structures

- struct pmc_low_volt_detect_config_t
  Low-Voltage Detect Configuration Structure. More...
- struct pmc_low_volt_warning_config_t
  Low-Voltage Warning Configuration Structure. More...

Driver version

- #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  PMC driver version.

Power Management Controller Control APIs

- void PMC_ConfigureLowVoltDetect (PMC_Type *base, const pmc_low_volt_detect_config_t *config)
  Configure the low voltage detect setting.
- static bool PMC_GetLowVoltDetectFlag (PMC_Type *base)
  Get Low-Voltage Detect Flag status.
- static void PMC_ClearLowVoltDetectFlag (PMC_Type *base)
  Acknowledge to clear the Low-Voltage Detect flag.
- void PMC_ConfigureLowVoltWarning (PMC_Type *base, const pmc_low_volt_warning_config_t *config)
  Configure the low voltage warning setting.
- static bool PMC_GetLowVoltWarningFlag (PMC_Type *base)
  Get Low-Voltage Warning Flag status.
- static void PMC_ClearLowVoltWarningFlag (PMC_Type *base)
  Acknowledge to Low-Voltage Warning flag.

36.2 Data Structure Documentation

36.2.1 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.

36.2.2 struct pmc_low_volt_warning_config_t

Data Fields

- bool enableInt
  Enable interrupt when low voltage warning.

36.3 Macro Definition Documentation

36.3.1 #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

36.4 Function Documentation

36.4.1 void PMC_ConfigureLowVoltDetect (PMC_Type * *base,* const pmc_low_volt_detect_config_t * *config*)

This function configures the low voltage detect setting, including the trip point voltage setting, enable interrupt or not, enable system reset or not.

Parameters

base	PMC peripheral base address.
config	Low-Voltage detect configuration structure.

36.4.2 static bool PMC_GetLowVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function reads the current LVDF status. If it returns 1, a low voltage event is detected.

Parameters

base	PMC peripheral base address.

Returns

Current low voltage detect flag
- true: Low-Voltage detected
- false: Low-Voltage not detected

36.4.3 static void PMC_ClearLowVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function acknowledges the low voltage detection errors (write 1 to clear LVDF).

Parameters

base	PMC peripheral base address.

36.4.4 void PMC_ConfigureLowVoltWarning (PMC_Type * *base,* const pmc_low_volt_warning_config_t * *config*)

This function configures the low voltage warning setting, including the trip point voltage setting and enable interrupt or not.

Parameters

base	PMC peripheral base address.
config	Low-Voltage warning configuration structure.

36.4.5 static bool PMC_GetLowVoltWarningFlag (PMC_Type * *base*) [inline], [static]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

base	PMC peripheral base address.

Returns

Current LVWF status
- true: Low-Voltage Warning Flag is set.
- false: the Low-Voltage Warning does not happen.

36.4.6 static void PMC_ClearLowVoltWarningFlag (PMC_Type * *base*) [inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

base	PMC peripheral base address.

Chapter 37
PORT: Port Control and Interrupts

37.1 Overview

The KSDK provides a driver for the Port Control and Interrupts (PORT) module of Kinetis devices.

37.2 Typical configuration case

37.2.1 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);
```

37.2.2 I2C PORT Configuration

```
/*  I2C pin PORTconfiguration
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainEnable,
    kPORT_LowDriveStrength,
    kPORT_MuxAlt5,
    kPORT_UnLockRegister,
};
PORT_SetPinConfig(PORTE,24u,&config);
PORT_SetPinConfig(PORTE,25u,&config);
```

Files

- file fsl_port.h

Data Structures

- struct port_pin_config_t
  PORT pin config structure. *More...*

Typical configuration case

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_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_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 }
  Pin mux selection.
- enum port_interrupt_t {
  kPORT_InterruptOrDMADisabled = 0x0U,
  kPORT_InterruptLogicZero = 0x8U,
  kPORT_InterruptRisingEdge = 0x9U,
  kPORT_InterruptFallingEdge = 0xAU,
  kPORT_InterruptEitherEdge = 0xBU,
  kPORT_InterruptLogicOne = 0xCU }
  Configures the interrupt generation condition.

Driver version

- #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  Version 2.0.1.

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.

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 pins' interrupt status flag.

37.3 Data Structure Documentation

37.3.1 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 driveStrength: 1
  fast/slow drive strength configure
- uint16_t mux: 3
  pin mux Configure

37.4 Macro Definition Documentation

37.4.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

37.5 Enumeration Type Documentation

37.5.1 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.

37.5.2 enum _port_slew_rate

Enumerator

kPORT_FastSlewRate fast slew rate is configured.
kPORT_SlowSlewRate slow slew rate is configured.

37.5.3 enum _port_passive_filter_enable

Enumerator

kPORT_PassiveFilterDisable fast slew rate is configured.
kPORT_PassiveFilterEnable slow slew rate is configured.

37.5.4 enum _port_drive_strength

Enumerator

kPORT_LowDriveStrength low drive strength is configured.
kPORT_HighDriveStrength high drive strength is configured.

37.5.5 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

37.5.6 enum port_interrupt_t

Enumerator

kPORT_InterruptOrDMADisabled Interrupt/DMA request is disabled.
kPORT_InterruptLogicZero Interrupt when logic zero.

kPORT_InterruptRisingEdge Interrupt on rising edge.
kPORT_InterruptFallingEdge Interrupt on falling edge.
kPORT_InterruptEitherEdge Interrupt on either edge.
kPORT_InterruptLogicOne Interrupt when logic one.

37.6 Function Documentation

37.6.1 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 configure structure.

37.6.2 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 pins' numbers macro.
config	PORT PCR register configure structure.

37.6.3 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 will be reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use in the case you just need to reset the pin mux

37.6.4 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).

37.6.5 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 17 have the interrupt.

37.6.6 static void PORT_ClearPinsInterruptFlags (PORT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	PORT peripheral base pointer.
mask	PORT pins' numbers macro.

Chapter 38
QSPI: Quad Serial Peripheral Interface Driver

38.1 Overview

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

38.2 Overview

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

Files

- file fsl_qspi.h

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, 0))
I2C driver version 2.0.0.

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_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.

38.3 Data Structure Documentation

38.3.1 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.

38.3.2 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.

38.3.3 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.

38.3.3.0.24.13 Field Documentation

38.3.3.0.24.13.1 uint8_t qspi_config_t::rxWatermark

38.3.3.0.24.13.2 uint32_t qspi_config_t::AHBbufferSize[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]

38.3.3.0.24.13.3 uint8_t qspi_config_t::AHBbufferMaster[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]

38.3.3.0.24.13.4 bool qspi_config_t::enableAHBbuffer3AllMaster

38.3.4 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.

38.3.4.0.24.14 Field Documentation

38.3.4.0.24.14.1 uint32_t qspi_flash_config_t::dataHoldTime

38.3.4.0.24.14.2 qspi_endianness_t qspi_flash_config_t::endian

38.3.4.0.24.14.3 bool qspi_flash_config_t::enableWordAddress

38.3.5 struct qspi_transfer_t

Data Fields

- uint32_t * data
  Pointer to data to transmit.
- size_t dataSize
  Bytes to be transmit.

38.4 Macro Definition Documentation

38.4.1 #define FSL_QSPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

38.5 Enumeration Type Documentation

38.5.1 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.

38.5.2 enum qspi_read_area_t

Enumerator

kQSPI_ReadAHB QSPI read from AHB buffer.
kQSPI_ReadIP QSPI read from IP FIFO.

38.5.3 enum qspi_command_seq_t

Enumerator

kQSPI_IPSeq IP command sequence.
kQSPI_BufferSeq Buffer command sequence.

38.5.4 enum qspi_fifo_t

Enumerator

kQSPI_TxFifo QSPI Tx FIFO.
kQSPI_RxFifo QSPI Rx FIFO.
kQSPI_AllFifo QSPI all FIFO, including Tx and Rx.

38.5.5 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

38.5.6 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.

38.5.7 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.

38.5.8 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.

38.5.9 enum _qspi_dma_enable

Enumerator

kQSPI_TxBufferFillDMAEnable Tx buffer fill DMA.
kQSPI_RxBufferDrainDMAEnable Rx buffer drain DMA.
kQSPI_AllDDMAEnable All DMA source.

38.5.10 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.

38.6 Function Documentation

38.6.1 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.

38.6.2 void QSPI_GetDefaultQspiConfig (qspi_config_t * *config*)

Parameters

config	QSPI configuration structure.

38.6.3 void QSPI_Deinit (QuadSPI_Type * *base*)

Clears the QSPI state and QSPI module registers.

Parameters

base	Pointer to QuadSPI Type.

38.6.4 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.

38.6.5 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.

38.6.6 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.

38.6.7 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.

38.6.8 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.

38.6.9 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.

38.6.10 static void QSPI_EnableInterrupts (QuadSPI_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	QSPI interrupt source.

38.6.11 static void QSPI_DisableInterrupts (QuadSPI_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
mask	QSPI interrupt source.

38.6.12 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.

38.6.13 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.

38.6.14 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.

38.6.15 static void QSPI_SetIPCommandAddress (QuadSPI_Type * *base,* uint32_t *addr*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
addr	IP command address.

38.6.16 static void QSPI_SetIPCommandSize (QuadSPI_Type * *base,* uint32_t *size*) [inline], [static]

Parameters

base	Pointer to QuadSPI Type.
size	IP command size.

38.6.17 void QSPI_ExecuteIPCommand (QuadSPI_Type * *base,* uint32_t *index*)

Parameters

base	Pointer to QuadSPI Type.
index	IP command located in which LUT table index.

38.6.18 void QSPI_ExecuteAHBCommand (QuadSPI_Type * *base,* uint32_t *index*)

Parameters

base	Pointer to QuadSPI Type.
index	AHB command located in which LUT table index.

38.6.19 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.

38.6.20 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.

38.6.21 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.

38.6.22 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.

38.6.23 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.

38.6.24 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

38.6.25 static void QSPI_WriteData (QuadSPI_Type * *base,* uint32_t *data*) [inline], [static]

Parameters

base	QSPI base pointer
data	The data bytes to send

38.6.26 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

38.6.27 uint32_t QSPI_ReadData (QuadSPI_Type * *base*)

Parameters

base	QSPI base pointer

Returns

The data in the FIFO.

38.6.28 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.

38.6.29 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.

38.7 QSPI eDMA Driver

38.7.1 Overview

Files

- file fsl_qspi_edma.h

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.

38.7.2 Data Structure Documentation

38.7.2.1 struct _qspi_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  eDMA handler for QSPI send
- size_t transferSize
  Bytes need to transfer.
- uint8_t 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.

38.7.2.1.0.15 Field Documentation

38.7.2.1.0.15.1 size_t qspi_edma_handle_t::transferSize

38.7.3 Function Documentation

38.7.3.1 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

38.7.3.2 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

38.7.3.3 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.

38.7.3.4 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.

38.7.3.5 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

38.7.3.6 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

38.7.3.7 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.

38.7.3.8 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 39
RCM: Reset Control Module Driver

39.1 Overview

The KSDK provides a Peripheral driver for the Reset Control Module (RCM) module of Kinetis devices.

Files

- file fsl_rcm.h

Data Structures

- struct rcm_reset_pin_filter_config_t
  Reset pin filter configuration. More...

Enumerations

- enum rcm_reset_source_t {
  kRCM_SourceLvd = RCM_SRS0_LVD_MASK,
  kRCM_SourceWdog = RCM_SRS0_WDOG_MASK,
  kRCM_SourcePin = RCM_SRS0_PIN_MASK,
  kRCM_SourcePor = RCM_SRS0_POR_MASK,
  kRCM_SourceLockup = RCM_SRS1_LOCKUP_MASK << 8U,
  kRCM_SourceSw = RCM_SRS1_SW_MASK,
  kRCM_SourceSackerr = RCM_SRS1_SACKERR_MASK << 8U }
  System Reset Source Name definitions.
- enum rcm_run_wait_filter_mode_t {
  kRCM_FilterDisable = 0U,
  kRCM_FilterBusClock = 1U,
  kRCM_FilterLpoClock = 2U }
  Reset pin filter select in Run and Wait modes.

Driver version

- #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  RCM driver version 2.0.0.

Reset Control Module APIs

- static uint32_t RCM_GetPreviousResetSources (RCM_Type *base)
  Gets the reset source status which caused a previous reset.
- void RCM_ConfigureResetPinFilter (RCM_Type *base, const rcm_reset_pin_filter_config_t *config)
  Configures the reset pin filter.

39.2 Data Structure Documentation

39.2.1 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.

39.2.1.0.0.16 Field Documentation

39.2.1.0.0.16.1 bool rcm_reset_pin_filter_config_t::enableFilterInStop

39.2.1.0.0.16.2 rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait

39.2.1.0.0.16.3 uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount

39.3 Macro Definition Documentation

39.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

39.4 Enumeration Type Documentation

39.4.1 enum rcm_reset_source_t

Enumerator

kRCM_SourceLvd low voltage detect reset
kRCM_SourceWdog Watchdog reset.
kRCM_SourcePin External pin reset.
kRCM_SourcePor Power on reset.
kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceSackerr Parameter could get all reset flags.

39.4.2 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.

39.5 Function Documentation

39.5.1 static uint32_t RCM_GetPreviousResetSources (RCM_Type * *base*) [inline], [static]

This function gets the current reset source status. Use source masks defined in the rcm_reset_source_t to get the desired source status.

Example:

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetPreviousResetSources(RCM) &
      kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (
      kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

base	RCM peripheral base address.

Returns

All reset source status bit map.

39.5.2 void RCM_ConfigureResetPinFilter (RCM_Type * *base,* const rcm_reset_pin_filter_config_t * *config*)

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

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

Chapter 40
RNGA: Random Number Generator Accelerator Driver

40.1 Overview

The Kinetis SDK provides Peripheral driver for the Random Number Generator Accelerator (RNGA) block of Kinetis devices.

40.2 RNGA Initialization

1. To initialize the RNGA module, call the RNGA_Init() function. This function automatically enables the RNGA module and its clock.
2. After calling the RNGA_Init() function, the RNGA is enabled and the counter starts working.
3. To disable the RNGA module, call the RNGA_Deinit() function.

40.3 Get random data from RNGA

1. RNGA_GetRandomData() function gets random data from the RNGA module.

40.4 RNGA Set/Get Working Mode

The RNGA works either in sleep mode or normal mode

1. RNGA_SetMode() function sets the RNGA mode.
2. RNGA_GetMode() function gets the RNGA working mode.

40.5 Seed RNGA

1. RNGA_Seed() function inputs an entropy value that the RNGA can use to seed the pseudo random algorithm.

This example code shows how to initialize and get random data from the RNGA driver:

```
{
    status_t       status;
    uint32_t       data;

    /* Initialize RNGA
    status = RNGA_Init(RNG);

    /* Read Random data
    status = RNGA_GetRandomData(RNG, data, sizeof(data));

    if(status == kStatus_Success)
    {
        /* Print data
        PRINTF("Random = 0x%X\r\n", i, data );
        PRINTF("Succeed.\r\n");
    }
    else
    {
```

```
        PRINTF("RNGA failed! (0x%x)\r\n", status);
    }

    /* Deinitialize RNGA
    RNGA_Deinit(RNG);
}
```

Note

It is important to note there is no known cryptographic proof showing this is a secure method of generating random data. In fact, there may be an attack against this random number generator if its output is used directly in a cryptographic application. The attack is based on the linearity of the internal shift registers. Therefore, it is highly recommended that this random data produced by this module be used as an entropy source to provide an input seed to a NIST-approved pseudo-random-number generator based on DES or SHA-1 and defined in NIST FIPS PUB 186-2 Appendix 3 and NIST FIPS PUB SP 800-90. The requirement is to maximize the entropy of this input seed. In order to do this, when data is extracted from RNGA as quickly as the hardware allows, there are about one or two bits of added entropy per 32-bit word. Any single bit of that word contains that entropy. Therefore, when used as an entropy source, a random number should be generated for each bit of entropy required, and the least significant bit (any bit would be equivalent) of each word retained. The remainder of each random number should then be discarded. Used this way, even with full knowledge of the internal state of RNGA and all prior random numbers, an attacker is not able to predict the values of the extracted bits. Other sources of entropy can be used along with RNGA to generate the seed to the pseudorandom algorithm. The more random sources combined to create the seed, the better. The following is a list of sources that can be easily combined with the output of this module:

- Current time using highest precision possible
- Real-time system inputs that can be characterized as "random"
- Other entropy supplied directly by the user

Files

- file fsl_rnga.h

Enumerations

- enum rnga_mode_t {
  kRNGA_ModeNormal = 0U,
  kRNGA_ModeSleep = 1U }
  RNGA working mode.

Functions

- void RNGA_Init (RNG_Type *base)
  Initializes the RNGA.
- void RNGA_Deinit (RNG_Type *base)
  Shuts down the RNGA.
- status_t RNGA_GetRandomData (RNG_Type *base, void *data, size_t data_size)

Gets random data.
- void RNGA_Seed (RNG_Type *base, uint32_t seed)
  Feeds the RNGA module.
- void RNGA_SetMode (RNG_Type *base, rnga_mode_t mode)
  Sets the RNGA in normal mode or sleep mode.
- rnga_mode_t RNGA_GetMode (RNG_Type *base)
  Gets the RNGA working mode.

Driver version

- #define FSL_RNGA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  RNGA driver version 2.0.1.

40.6 Macro Definition Documentation

40.6.1 #define FSL_RNGA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

40.7 Enumeration Type Documentation

40.7.1 enum rnga_mode_t

Enumerator

kRNGA_ModeNormal Normal Mode. The ring-oscillator clocks are active; RNGA generates entropy (randomness) from the clocks and stores it in shift registers.

kRNGA_ModeSleep Sleep Mode. The ring-oscillator clocks are inactive; RNGA does not generate entropy.

40.8 Function Documentation

40.8.1 void RNGA_Init (RNG_Type * *base*)

This function initializes the RNGA. When called, the RNGA entropy generation starts immediately.

Parameters

base	RNGA base address

40.8.2 void RNGA_Deinit (RNG_Type * *base*)

This function shuts down the RNGA.

Parameters

base	RNGA base address

40.8.3 status_t RNGA_GetRandomData (RNG_Type * *base,* void * *data,* size_t *data_size*)

This function gets random data from the RNGA.

Parameters

base	RNGA base address
data	pointer to user buffer to be filled by random data
data_size	size of data in bytes

Returns

RNGA status

40.8.4 void RNGA_Seed (RNG_Type * *base,* uint32_t *seed*)

This function inputs an entropy value that the RNGA uses to seed its pseudo-random algorithm.

Parameters

base	RNGA base address
seed	input seed value

40.8.5 void RNGA_SetMode (RNG_Type * *base,* rnga_mode_t *mode*)

This function sets the RNGA in sleep mode or normal mode.

Parameters

base	RNGA base address
mode	normal mode or sleep mode

40.8.6 rnga_mode_t RNGA_GetMode (RNG_Type * *base*)

This function gets the RNGA working mode.

Parameters

base	RNGA base address

Returns

normal mode or sleep mode

Chapter 41
RTC: Real Time Clock Driver

41.1 Overview

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

Files

- file fsl_rtc.h

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)
  Stop the timer and gate the RTC clock.
- void RTC_GetDefaultConfig (rtc_config_t *config)
  Fill 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.

41.2 Data Structure Documentation

41.2.1 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.

41.2.1.0.0.17 Field Documentation

41.2.1.0.0.17.1 uint16_t rtc_datetime_t::year

41.2.1.0.0.17.2 uint8_t rtc_datetime_t::month

41.2.1.0.0.17.3 uint8_t rtc_datetime_t::day

41.2.1.0.0.17.4 uint8_t rtc_datetime_t::hour

41.2.1.0.0.17.5 uint8_t rtc_datetime_t::minute

41.2.1.0.0.17.6 uint8_t rtc_datetime_t::second

41.2.2 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 32KHz 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.

41.3 Enumeration Type Documentation

41.3.1 enum rtc_interrupt_enable_t

Enumerator

kRTC_TimeInvalidInterruptEnable Time invalid interrupt.
kRTC_TimeOverflowInterruptEnable Time overflow interrupt.
kRTC_AlarmInterruptEnable Alarm interrupt.
kRTC_SecondsInterruptEnable Seconds interrupt.

41.3.2 enum rtc_status_flags_t

Enumerator

kRTC_TimeInvalidFlag Time invalid flag.
kRTC_TimeOverflowFlag Time overflow flag.
kRTC_AlarmFlag Alarm flag.

41.3.3 enum rtc_osc_cap_load_t

Enumerator

kRTC_Capacitor_2p 2pF capacitor load
kRTC_Capacitor_4p 4pF capacitor load
kRTC_Capacitor_8p 8pF capacitor load
kRTC_Capacitor_16p 16pF capacitor load

41.4 Function Documentation

41.4.1 void RTC_Init (RTC_Type * *base,* const rtc_config_t * *config*)

This function will issue 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 user's RTC config structure.

41.4.2 static void RTC_Deinit (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

41.4.3 void RTC_GetDefaultConfig (rtc_config_t * *config*)

The default values are:

```
config->wakeupSelect = false;
config->updateMode = false;
config->supervisorAccess = false;
config->compensationInterval = 0;
config->compensationTime = 0;
```

Parameters

config	Pointer to user's RTC config structure.

41.4.4 status_t RTC_SetDatetime (RTC_Type * *base*, const rtc_datetime_t * *datetime*)

The RTC counter must be stopped prior to calling this function as writes to the RTC seconds register will fail if the RTC counter is running.

Parameters

base	RTC peripheral base address
datetime	Pointer to structure where the date and time details to set are stored

Returns

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

41.4.5 void RTC_GetDatetime (RTC_Type * *base,* rtc_datetime_t * *datetime*)

Parameters

base	RTC peripheral base address
datetime	Pointer to structure where the date and time details are stored.

41.4.6 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 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

41.4.7 void RTC_GetAlarm (RTC_Type * *base*, rtc_datetime_t * *datetime*)

Parameters

base	RTC peripheral base address
datetime	Pointer to structure where the alarm date and time details are stored.

41.4.8 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

41.4.9 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

41.4.10 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

41.4.11 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

41.4.12 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

41.4.13 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

41.4.14 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

41.4.15 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

41.4.16 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 42
SAI: Serial Audio Interface

42.1 Overview

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

42.2 Overview

SAI driver includes functional APIs and transactional APIs.

Functional APIs are feature/property 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 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 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.

42.3 Typical use case

42.3.1 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)
    {
    }

    // ...
}
```

42.3.2 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

Files

- file fsl_sai.h

Data Structures

- struct sai_config_t
  SAI user configure 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.

Typical use case

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 }
The SAI interrupt enable flag.

- enum _sai_dma_enable_t { kSAI_FIFOWarningDMAEnable = I2S_TCSR_FWDE_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_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_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, 0))
  Version 2.1.0.

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 SAI Tx.
- void SAI_RxEnable (I2S_Type *base, bool enable)
  Enables/disables 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 SAI Tx interrupt requests.
- static void SAI_RxEnableInterrupts (I2S_Type *base, uint32_t mask)
  Enables SAI Rx interrupt requests.
- static void SAI_TxDisableInterrupts (I2S_Type *base, uint32_t mask)
  Disables SAI Tx interrupt requests.
- static void SAI_RxDisableInterrupts (I2S_Type *base, uint32_t mask)
  Disables SAI Rx interrupt requests.

DMA Control

- static void SAI_TxEnableDMA (I2S_Type *base, uint32_t mask, bool enable)
  Enables/disables SAI Tx DMA requests.
- static void SAI_RxEnableDMA (I2S_Type *base, uint32_t mask, bool enable)
  Enables/disables 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 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.

42.4 Data Structure Documentation

42.4.1 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.
- sai_mclk_source_t mclkSource
  Master Clock source.
- sai_bclk_source_t bclkSource
  Bit Clock source.
- sai_master_slave_t masterSlave
  Master or slave.

42.4.2 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 8/16/24/32bits.
- sai_mono_stereo_t stereo
  Mono or stereo.
- uint32_t masterClockHz
  Master clock frequency in Hz.
- uint8_t channel
  Data channel used in transfer.
- sai_protocol_t protocol
  Which audio protocol used.

42.4.2.0.0.18 Field Documentation

42.4.2.0.0.18.1 uint8_t sai_transfer_format_t::channel

42.4.3 struct sai_transfer_t

Data Fields

- uint8_t * data
  Data start address to transfer.
- size_t dataSize

Transfer size.

42.4.3.0.0.19 Field Documentation

42.4.3.0.0.19.1 uint8_t* sai_transfer_t::data

42.4.3.0.0.19.2 size_t sai_transfer_t::dataSize

42.4.4 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, 8/16/24/32bits.
- 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.

42.5 Macro Definition Documentation

42.5.1 #define SAI_XFER_QUEUE_SIZE (4)

42.6 Enumeration Type Documentation

42.6.1 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.

42.6.2 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.

42.6.3 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.
kSAI_Slave Slave mode.

42.6.4 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.
kSAI_MonoLeft Only left channel have sound.
kSAI_MonoRight Only Right channel have sound.

42.6.5 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.

42.6.6 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.

42.6.7 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.

42.6.8 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.

42.6.9 enum _sai_dma_enable_t

Enumerator

kSAI_FIFOWarningDMAEnable FIFO warning caused by the DMA request.

42.6.10 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_FIFOWarningFlag FIFO warning flag.

42.6.11 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.

42.6.12 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000Hz.
kSAI_SampleRate11025Hz Sample rate 11025Hz.
kSAI_SampleRate12KHz Sample rate 12000Hz.
kSAI_SampleRate16KHz Sample rate 16000Hz.
kSAI_SampleRate22050Hz Sample rate 22050Hz.
kSAI_SampleRate24KHz Sample rate 24000Hz.
kSAI_SampleRate32KHz Sample rate 32000Hz.
kSAI_SampleRate44100Hz Sample rate 44100Hz.
kSAI_SampleRate48KHz Sample rate 48000Hz.
kSAI_SampleRate96KHz Sample rate 96000Hz.

42.6.13 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.

42.7 Function Documentation

42.7.1 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 configure structure.

42.7.2 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 configure structure.

42.7.3 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(). Example:

```
sai_config_t config;
SAI_TxGetDefaultConfig(&config);
```

Parameters

config	pointer to master configuration structure

42.7.4 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(). Example:

```
sai_config_t config;
SAI_RxGetDefaultConfig(&config);
```

Parameters

config	pointer to master configuration structure

42.7.5 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

42.7.6 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

42.7.7 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

42.7.8 void SAI_TxEnable (I2S_Type * *base*, bool *enable*)

Parameters

base	SAI base pointer

enable	True means enable SAI Tx, false means disable.

42.7.9 void SAI_RxEnable (I2S_Type * *base,* bool *enable*)

Parameters

base	SAI base pointer
enable	True means enable SAI Rx, false means disable.

42.7.10 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.

42.7.11 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

42.7.12 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.

42.7.13 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 source if defined: • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

42.7.14 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 source if defined: • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

42.7.15 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 source if defined: • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

42.7.16 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 source if defined: • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

42.7.17 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 source if defined: • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

42.7.18 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 source if defined: • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
enable	True means enable DMA, false means disable DMA.

42.7.19 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 source if defined: • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
enable	True means enable DMA, false means disable DMA.

42.7.20 static uint32_t SAI_TxGetDataRegisterAddress (I2S_Type * *base,* uint32_t *channel*) [inline], [static]

This API is used to provide a transfer address for SAI DMA transfer configuration.

Parameters

base	SAI base pointer.
channel	Which data channel used.

Returns

data register address.

42.7.21 static uint32_t SAI_RxGetDataRegisterAddress (I2S_Type * *base,* uint32_t *channel*) [inline], [static]

This API is used to provide a transfer address for SAI DMA transfer configuration.

Parameters

base	SAI base pointer.
channel	Which data channel used.

Returns

data register address.

42.7.22 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 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.

42.7.23 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 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.

42.7.24 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 a audio word, usually 8/16/24/32 bits.
buffer	Pointer to the data to be written.
size	Bytes to be written.

42.7.25 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.

42.7.26 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 a audio word, usually 8/16/24/32 bits.
buffer	Pointer to the data to be read.
size	Bytes to be read.

42.7.27 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.

42.7.28 void SAI_TransferTxCreateHandle (I2S_Type * *base,* sai_handle_t * *handle,* sai_transfer_callback_t *callback,* void * *userData*)

This function initializes the Tx handle for SAI Tx transactional APIs. Call this function one time to get the handle initialized.

Parameters

base	SAI base pointer
handle	SAI handle pointer.
callback	pointer to user callback function
userData	user parameter passed to the callback function

42.7.29 void SAI_TransferRxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_callback_t *callback*, void * *userData*)

This function initializes the Rx handle for SAI Rx transactional APIs. Call this function one time to get the handle initialized.

Parameters

base	SAI base pointer.
handle	SAI handle pointer.
callback	pointer to user callback function
userData	user parameter passed to the callback function

42.7.30 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 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 to masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

42.7.31 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 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.

Returns

Status of this function. Return value is one of status_t.

42.7.32 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 sai_handle_t structure which stores the transfer state
xfer	pointer to 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.

42.7.33 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 sai_handle_t structure which stores the transfer state
xfer	pointer to 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.

42.7.34 status_t SAI_TransferGetSendCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer.
handle	pointer to sai_handle_t structure which stores the transfer state.
count	Bytes count sent.

Return values

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

42.7.35 status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

base	SAI base pointer.
handle	pointer to sai_handle_t structure which stores the transfer state.
count	Bytes count received.

Return values

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

42.7.36 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 sai_handle_t structure which stores the transfer state.

42.7.37 void SAI_TransferAbortReceive (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 sai_handle_t structure which stores the transfer state.

42.7.38 void SAI_TransferTxHandleIRQ (I2S_Type * *base,* sai_handle_t * *handle*)

Parameters

base	SAI base pointer.
handle	pointer to sai_handle_t structure.

42.7.39 void SAI_TransferRxHandleIRQ (I2S_Type * *base,* sai_handle_t * *handle*)

Parameters

base	SAI base pointer.
handle	pointer to sai_handle_t structure.

42.8 SAI DMA Driver

42.8.1 Overview

Files

- file fsl_sai_dma.h

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.

42.8.2 Data Structure Documentation

42.8.2.1 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.

42.8.2.1.0.20 Field Documentation

42.8.2.1.0.20.1 sai_transfer_t sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]

42.8.2.1.0.20.2 volatile uint8_t sai_dma_handle_t::queueUser

42.8.3 Function Documentation

42.8.3.1 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.

42.8.3.2 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.

42.8.3.3 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.

42.8.3.4 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.

42.8.3.5 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.

42.8.3.6 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.

42.8.3.7 void SAI_TransferAbortSendDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.

42.8.3.8 void SAI_TransferAbortReceiveDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*)

Parameters

base	SAI base pointer.
handle	SAI DMA handle pointer.

42.8.3.9 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.

42.8.3.10 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_NoTransferIn-Progress	There is not a non-blocking transaction currently in progress.

42.9 SAI eDMA Driver

42.9.1 Overview

Files

- file fsl_sai_edma.h

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.

42.9.2 Data Structure Documentation

42.9.2.1 struct _sai_edma_handle

Data Fields

- edma_handle_t * dmaHandle
  DMA handler for SAI send.
- 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.

42.9.2.1.0.21 Field Documentation

42.9.2.1.0.21.1 edma_tcd_t sai_edma_handle_t::tcd[SAI_XFER_QUEUE_SIZE+1U]

42.9.2.1.0.21.2 sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]

42.9.2.1.0.21.3 volatile uint8_t sai_edma_handle_t::queueUser

42.9.3 Function Documentation

42.9.3.1 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.

42.9.3.2 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.

42.9.3.3 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.

42.9.3.4 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.

42.9.3.5 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.

42.9.3.6 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.

42.9.3.7 void SAI_TransferAbortSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

Parameters

base	SAI base pointer.
handle	SAI eDMA handle pointer.

42.9.3.8 void SAI_TransferAbortReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

Parameters

base	SAI base pointer
handle	SAI eDMA handle pointer.

42.9.3.9 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.

42.9.3.10 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_NoTransferIn-Progress	There is no non-blocking transaction in progress.

Chapter 43
SDHC: Secured Digital Host Controller Driver

43.1 Overview

The KSDK provides a peripheral driver for the Secured Digital Host Controller (SDHC) module of Kinetis devices.

Typical use case

```
/* 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);
```

Files

- file fsl_sdhc.h

Data Structures

- struct sdhc_adma2_descriptor_t

Define 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
  Host descriptor. 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 max 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_DE-

SCRIPTOR_LENGTH_MASK)
The max value of LENGTH field in ADMA2's descriptor.

Typedefs

- typedef uint32_t sdhc_adma1_descriptor_t
  Define 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, 0U, 0U))
Driver version 2.0.0.

Initialization and deinitialization

- void SDHC_Init (SDHC_Type *base, const sdhc_config_t *config)
  SDHC module initialization function.
- void SDHC_Deinit (SDHC_Type *base)
  Deinitialize the SDHC.
- bool SDHC_Reset (SDHC_Type *base, uint32_t mask, uint32_t timeout)
  Reset 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)
  Set ADMA descriptor table configuration.

Interrupts

- static void SDHC_EnableInterruptStatus (SDHC_Type *base, uint32_t mask)
  Enable interrupt status.
- static void SDHC_DisableInterruptStatus (SDHC_Type *base, uint32_t mask)
  Disable interrupt status.
- static void SDHC_EnableInterruptSignal (SDHC_Type *base, uint32_t mask)
  Enable interrupts signal corresponding to the interrupt status flag.
- static void SDHC_DisableInterruptSignal (SDHC_Type *base, uint32_t mask)
  Disable interrupts signal corresponding to the interrupt status flag.

Status

- static uint32_t SDHC_GetInterruptStatusFlags (SDHC_Type *base)
  Get current interrupt status.
- static void SDHC_ClearInterruptStatusFlags (SDHC_Type *base, uint32_t mask)
  Clear specified interrupt status.
- static uint32_t SDHC_GetAutoCommand12ErrorStatusFlags (SDHC_Type *base)
  Get the status of auto command 12 error.
- static uint32_t SDHC_GetAdmaErrorStatusFlags (SDHC_Type *base)
  Get the status of ADMA error.
- static uint32_t SDHC_GetPresentStatusFlags (SDHC_Type *base)
  Get present status.

Bus Operations

- void SDHC_GetCapability (SDHC_Type *base, sdhc_capability_t *capability)
  Get the capability information.
- static void SDHC_EnableSdClock (SDHC_Type *base, bool enable)
  Enable or disable SD bus clock.
- uint32_t SDHC_SetSdClock (SDHC_Type *base, uint32_t srcClock_Hz, uint32_t busClock_Hz)
  Set SD bus clock frequency.
- bool SDHC_SetCardActive (SDHC_Type *base, uint32_t timeout)
  Send 80 clocks to the card to set it to be active state.
- static void SDHC_SetDataBusWidth (SDHC_Type *base, sdhc_data_bus_width_t width)
  Set the data transfer width.

- void SDHC_SetTransferConfig (SDHC_Type *base, const sdhc_transfer_config_t *config)
  Set card transfer-related configuration.
- static uint32_t SDHC_GetCommandResponse (SDHC_Type *base, uint32_t index)
  Get the command response.
- static void SDHC_WriteData (SDHC_Type *base, uint32_t data)
  Fill the the data port.
- static uint32_t SDHC_ReadData (SDHC_Type *base)
  Retrieve the data from the data port.
- static void SDHC_EnableWakeupEvent (SDHC_Type *base, uint32_t mask, bool enable)
  Enable or disable wakeup event in low power mode.
- static void SDHC_EnableCardDetectTest (SDHC_Type *base, bool enable)
  Enable or disable card detection level for test.
- static void SDHC_SetCardDetectTestLevel (SDHC_Type *base, bool high)
  Set card detection test level.
- void SDHC_EnableSdioControl (SDHC_Type *base, uint32_t mask, bool enable)
  Enable or disable SDIO card control.
- static void SDHC_SetContinueRequest (SDHC_Type *base)
  Restart a transaction which has stopped at the block gap for SDIO card.
- void SDHC_SetMmcBootConfig (SDHC_Type *base, const sdhc_boot_config_t *config)
  Configure the MMC boot feature.
- static void SDHC_SetForceEvent (SDHC_Type *base, uint32_t mask)
  Force to generate events according to the given mask.

Transactional

- status_t SDHC_TransferBlocking (SDHC_Type *base, uint32_t *admaTable, uint32_t admaTableWords, sdhc_transfer_t *transfer)
  Transfer command/data using blocking way.
- void SDHC_TransferCreateHandle (SDHC_Type *base, sdhc_handle_t *handle, const sdhc_transfer_callback_t *callback, void *userData)
  Create the SDHC handle.
- status_t SDHC_TransferNonBlocking (SDHC_Type *base, sdhc_handle_t *handle, uint32_t *admaTable, uint32_t admaTableWords, sdhc_transfer_t *transfer)
  Transfer command/data using interrupt and asynchronous way.
- void SDHC_TransferHandleIRQ (SDHC_Type *base, sdhc_handle_t *handle)
  IRQ handler for SDHC.

43.2 Data Structure Documentation

43.2.1 struct sdhc_adma2_descriptor_t

Data Fields

- uint32_t attribute
  The control and status field.
- const uint32_t * address
  The address field.

43.2.2 struct sdhc_capability_t

Define 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)

43.2.3 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)

43.2.4 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.

43.2.4.0.0.22 Field Documentation

43.2.4.0.0.22.1 sdhc_boot_mode_t sdhc_boot_config_t::bootMode

43.2.5 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.

43.2.6 struct sdhc_data_t

Define 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.

43.2.7 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.

43.2.8 struct sdhc_transfer_t

Data Fields

- sdhc_data_t * data
  Data to transfer.
- sdhc_command_t * command
  Command to send.

43.2.9 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.

43.2.10 struct _sdhc_handle

SDHC handle typedef.

Define the structure to save the SDHC state information and callback function. The detail interrupt status when send command or transfer data can be obtained from interruptFlags field by using 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.

43.2.11 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.

43.3 Macro Definition Documentation

43.3.1 #define FSL_SDHC_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 0U))

43.4 Typedef Documentation

43.4.1 typedef uint32_t sdhc_adma1_descriptor_t

43.4.2 typedef status_t(∗ sdhc_transfer_function_t)(SDHC_Type ∗base, sdhc_transfer_t ∗content)

43.5 Enumeration Type Documentation

43.5.1 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.

43.5.2 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.

43.5.3 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.

43.5.4 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.

43.5.5 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.

43.5.6 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.

43.5.7 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.

43.5.8 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.

43.5.9 enum _sdhc_adma_error_status_flag

Enumerator

kSDHC_AdmaLenghMismatchFlag Length mismatch error.
kSDHC_AdmaDescriptorErrorFlag Descriptor error.

43.5.10 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.

43.5.11 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.

43.5.12 enum sdhc_data_bus_width_t

Enumerator

kSDHC_DataBusWidth1Bit 1-bit mode
kSDHC_DataBusWidth4Bit 4-bit mode
kSDHC_DataBusWidth8Bit 8-bit mode

43.5.13 enum sdhc_endian_mode_t

Enumerator

kSDHC_EndianModeBig Big endian mode.
kSDHC_EndianModeHalfWordBig Half word big endian mode.
kSDHC_EndianModeLittle Little endian mode.

43.5.14 enum sdhc_dma_mode_t

Enumerator

kSDHC_DmaModeNo No DMA.
kSDHC_DmaModeAdma1 ADMA1 is selected.
kSDHC_DmaModeAdma2 ADMA2 is selected.

43.5.15 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.

43.5.16 enum sdhc_boot_mode_t

Enumerator

kSDHC_BootModeNormal Normal boot.
kSDHC_BootModeAlternative Alternative boot.

43.5.17 enum sdhc_command_type_t

Enumerator

kSDHC_CommandTypeNormal Normal command.
kSDHC_CommandTypeSuspend Suspend command.
kSDHC_CommandTypeResume Resume command.
kSDHC_CommandTypeAbort Abort command.

43.5.18 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.

43.5.19 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.

43.5.20 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.

43.6 Function Documentation

43.6.1 void SDHC_Init (SDHC_Type * *base*, const sdhc_config_t * *config*)

Configure the SDHC according to the user configuration.

Example:

```
sdhc_config_t config;
config.enableDat3AsCDPin = false;
config.endianMode = kSDHC_EndianModeLittle;
config.dmaMode = kSDHC_DmaModeAdma2;
config.readWatermarkLevel = 512U;
config.writeWatermarkLevel = 512U;
SDHC_Init(SDHC, &config);
```

Parameters

base	SDHC peripheral base address.
config	SDHC configuration information.

Return values

kStatus_Success	Operate successfully.

43.6.2 void SDHC_Deinit (SDHC_Type * *base*)

Parameters

base	SDHC peripheral base address.

43.6.3 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.

43.6.4 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.

43.6.5 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).

43.6.6 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).

43.6.7 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).

43.6.8 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).

43.6.9 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).

43.6.10 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).

43.6.11 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).

43.6.12 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).

43.6.13 static uint32_t SDHC_GetPresentStatusFlags (SDHC_Type * *base*) [inline], [static]

This function gets the present SDHC's status except for interrupt status and error status.

Parameters

base	SDHC peripheral base address.

Returns

Present SDHC's status flags mask(_sdhc_present_status_flag).

43.6.14 void SDHC_GetCapability (SDHC_Type * *base,* sdhc_capability_t * *capability*)

Parameters

base	SDHC peripheral base address.
capability	Structure to save capability information.

43.6.15 static void SDHC_EnableSdClock (SDHC_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	SDHC peripheral base address.
enable	True to enable, false to disable.

43.6.16 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.

43.6.17 bool SDHC_SetCardActive (SDHC_Type * *base*, uint32_t *timeout*)

This function must be called after each time the card is inserted to make card can receive 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.

43.6.18 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.

43.6.19 void SDHC_SetTransferConfig (SDHC_Type * *base*, const sdhc_transfer_config_t * *config*)

This function fills card transfer-related command argument/transfer flag/data size. Command and data will be 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.

43.6.20 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.

43.6.21 static void SDHC_WriteData (SDHC_Type * *base,* uint32_t *data*) [inline], [static]

This function is mainly 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.

43.6.22 static uint32_t SDHC_ReadData (SDHC_Type * *base*) [inline], [static]

This function is mainly used to implement the data transfer by Data Port instead of DMA.

Parameters

base	SDHC peripheral base address.

Returns

The data has been read.

43.6.23 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.

43.6.24 static void SDHC_EnableCardDetectTest (SDHC_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	SDHC peripheral base address.

enable	True to enable, false to disable.

43.6.25 static void SDHC_SetCardDetectTestLevel (SDHC_Type * *base*, bool *high*) [inline], [static]

This function set the card detection test level to indicate whether the card is inserted into SDHC when DAT[3]/ CD pin is selected as card detection pin. This function can also assert the pin logic when DAT[3]/CD pin is select as the card detection pin.

Parameters

base	SDHC peripheral base address.
high	True to set the card detect level to high.

43.6.26 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.

43.6.27 static void SDHC_SetContinueRequest (SDHC_Type * *base*) [inline], [static]

Parameters

base	SDHC peripheral base address.

43.6.28 void SDHC_SetMmcBootConfig (SDHC_Type * *base*, const sdhc_boot_config_t * *config*)

Example:

```
sdhc_boot_config_t bootConfig;
```

```
bootConfig.ackTimeoutCount = 4;
bootConfig.bootMode = kSDHC_BootModeNormal;
bootConfig.blockCount = 5;
bootConfig.enableBootAck = true;
bootConfig.enableBoot = true;
enableBoot.enableAutoStopAtBlockGap = true;
SDHC_SetMmcBootConfig(SDHC, &bootConfig);
```

Parameters

base	SDHC peripheral base address.
config	The MMC boot configuration information.

43.6.29 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).

43.6.30 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 got or SDHC encounters error by polling the status flag. Application must not call this API in multiple threads at the same time because of that this API doesn't support reentry mechanism.

Note

Needn't 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.

43.6.31 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.

43.6.32 status_t SDHC_TransferNonBlocking (SDHC_Type * *base*, sdhc_handle_t * *handle*, uint32_t * *admaTable*, uint32_t *admaTableWords*, sdhc_transfer_t * *transfer*)

This function send command and data and return immediately. It doesn't wait the transfer complete or encounter error. Application must not call this API in multiple threads at the same time because of that this API doesn't support reentry mechanism.

Note

Must 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.

43.6.33 void SDHC_TransferHandleIRQ (SDHC_Type * *base*, sdhc_handle_t * *handle*)

This function deals with IRQs on the given host controller.

Parameters

base	SDHC peripheral base address.
handle	SDHC handle.

Chapter 44
SDRAMC: Synchronous DRAM Controller Driver

44.1 Overview

The KSDK provides a peripheral driver for the Synchronous DRAM Controller block of Kinetis devices.

The SDRAM controller commands include the initialize 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.

44.2 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.
```

Typical use case

```
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;
```

Files

- file fsl_sdramc.h

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, 0, 0))
SDRAMC driver version 2.0.0.

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

- status_t 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 operation valid.

44.3 Data Structure Documentation

44.3.1 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.

44.3.1.0.0.23 Field Documentation

44.3.1.0.0.23.1 sdramc_block_selection_t sdramc_blockctl_config_t::block

44.3.1.0.0.23.2 sdramc_port_size_t sdramc_blockctl_config_t::portSize

44.3.1.0.0.23.3 sdramc_command_bit_location_t sdramc_blockctl_config_t::location

44.3.1.0.0.23.4 sdramc_latency_t sdramc_blockctl_config_t::latency

44.3.1.0.0.23.5 uint32_t sdramc_blockctl_config_t::address

44.3.1.0.0.23.6 uint32_t sdramc_blockctl_config_t::addressMask

44.3.2 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.

44.3.2.0.0.24 Field Documentation

44.3.2.0.0.24.1 sdramc_refresh_time_t sdramc_refresh_config_t::refreshTime

44.3.2.0.0.24.2 uint32_t sdramc_refresh_config_t::sdramRefreshRow

44.3.2.0.0.24.3 uint32_t sdramc_refresh_config_t::busClock_Hz

44.3.3 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.

44.3.3.0.0.25 Field Documentation

44.3.3.0.0.25.1 sdramc_refresh_config_t* sdramc_config_t::refreshConfig

44.3.3.0.0.25.2 sdramc_blockctl_config_t* sdramc_config_t::blockConfig

If both SDRAM blocks are used, use the two continuous blockConfig.

44.3.3.0.0.25.3 uint8_t sdramc_config_t::numBlockConfig

44.4 Macro Definition Documentation

44.4.1 #define FSL_SDRAMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

44.5 Enumeration Type Documentation

44.5.1 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.

44.5.2 enum sdramc_latency_t

The latency setting will 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 on 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.

44.5.3 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.

44.5.4 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.

44.5.5 enum sdramc_port_size_t

Enumerator

kSDRAMC_PortSize32Bit 32-Bit port size.
kSDRAMC_PortSize8Bit 8-Bit port size.
kSDRAMC_PortSize16Bit 16-Bit port size.

44.5.6 enum sdramc_block_selection_t

Enumerator

kSDRAMC_Block0 Select SDRAM block 0.
kSDRAMC_Block1 Select SDRAM block 1.

44.6 Function Documentation

44.6.1 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.

44.6.2 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.

44.6.3 status_t SDRAMC_SendCommand (SDRAM_Type * *base,* sdramc_block_selection_t *block,* sdramc_command_t *command*)

This function sends the command to SDRAM. There are precharge command, initialize MRS command, auto-refresh enable/disable command, and self-refresh enter/exit commands. Note the self-refresh enter/exit commands are all blocks setting and "block" are ignored. Ensure to set the right "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

Returns

Command execution status. All commands except the "initialize MRS command" and "precharge command" return kStatus_Success directly. For "initialize MRS command" and "precharge command" return kStatus_Success when the command success else return kStatus_Fail.

44.6.4 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.

44.6.5 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 operation valid, false disable the operation valid.

Chapter 45
SIM: System Integration Module Driver

45.1 Overview

The KSDK provides a peripheral driver for the System Integration Module (SIM) of Kinetis devices.

Files

- file fsl_sim.h

Data Structures

- struct sim_uid_t
  Unique ID. More...

Enumerations

- enum _sim_flash_mode {
  kSIM_FlashDisableInWait = SIM_FCFG1_FLASHDOZE_MASK,
  kSIM_FlashDisable = SIM_FCFG1_FLASHDIS_MASK }
  Flash enable mode.

Functions

- void SIM_GetUniqueId (sim_uid_t *uid)
  Get the unique identification register value.
- static void SIM_SetFlashMode (uint8_t mode)
  Set the flash enable mode.

Driver version

- #define FSL_SIM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Driver version 2.0.0.

45.2 Data Structure Documentation

45.2.1 struct sim_uid_t

Data Fields

- uint32_t MH
  UIDMH.
- uint32_t ML
  UIDML.

- uint32_t L
  UIDL.

45.2.1.0.0.26 Field Documentation

45.2.1.0.0.26.1 uint32_t sim_uid_t::MH

45.2.1.0.0.26.2 uint32_t sim_uid_t::ML

45.2.1.0.0.26.3 uint32_t sim_uid_t::L

45.3 Enumeration Type Documentation

45.3.1 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.
kSIM_FlashDisable Disable flash in normal mode.

45.4 Function Documentation

45.4.1 void SIM_GetUniqueId (sim_uid_t * *uid*)

Parameters

uid	Pointer to the structure to save the UID value.

45.4.2 static void SIM_SetFlashMode (uint8_t *mode*) [inline], [static]

Parameters

mode	The mode to set, see _sim_flash_mode for mode details.

Chapter 46
SLCD: Segment LCD Driver

46.1 Overview

The KSDK provides a peripheral driver for the Segment LCD (SLCD) module of Kinetis devices. The SLCD module is a CMOS charge pump voltage inverter that is designed for low voltage and low-power operation. SLCD is designed to generate the appropriate waveforms to drive multiplexed numeric, alphanumeric, or custom segment LCD panels. SLCD also has several timing and control settings that can be software-configured depending on the application's requirements. Timing and control consists of registers and control logic for the following:

1. LCD frame frequency
2. Duty cycle selection
3. Front plane/back plane selection and enabling
4. Blink modes and frequency
5. Operation in low-power modes

After the SLCD general initialization, the SLCD_SetBackPlanePhase(), SLCD_SetFrontPlaneSegments(), and SLCD_SetFrontPlaneOnePhase() are used to set the special back/front Plane to make SLCD display correctly. Then, the independent display control APIs, SLCD_StartDisplay() and SLCD_StopDisplay(), start and stop the SLCD display.

The SLCD_StartBlinkMode() and SLCD_StopBlinkMode() are provided for the runtime special blink mode control. To get the SLCD fault detection result, call the SLCD_GetFaultDetectCounter().

46.2 Typical use case

46.2.1 SLCD Initialization operation

```
    slcd_config_t configure = 0;
    slcd_clock_config_t clkConfig =
    {
        kSLCD_AlternateClk1,
        kSLCD_AltClkDivFactor1,
        kSLCD_ClkPrescaler00
#if FSL_FEATURE_SLCD_HAS_FAST_FRAME_RATE
        ,
        false
#endif
    };
    SLCD_GetDefaultConfig(&configure);
    configure.clkConfig.        = &clkConfig;
    configure.loadAdjust        = kSLCD_LowLoadOrIntermediateClkSrc
      ;
    configure.dutyCycle         = kSLCD_1Div4DutyCycle;
    configure.slcdlowPinEnabled = 0x1a44;
    configure.backPlanelowPin   = 0x0822;
    configure.faultConfig       = NULL;

    SLCD_Init(base, &configure);
```

Typical use case

```
SLCD_SetBackPlanePhase(base, 1, kSLCD_PhaseAActivate);
SLCD_SetBackPlanePhase(base, 5, kSLCD_PhaseBActivate);
SLCD_SetBackPlanePhase(base, 11, kSLCD_PhaseCActivate);

SLCD_SetFrontPlaneSegments(base, 0, (
  kSLCD_PhaseAActivate | kSLCD_PhaseBActivate));
SLCD_SetFrontPlaneSegments(base, 9, (
  kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));

SLCD_StartDisplay(base);
```

Files

- file fsl_slcd.h

Data Structures

- struct slcd_fault_detect_config_t
  SLCD fault frame detection configure structure. More...
- struct slcd_clock_config_t
  SLCD clock configure structure. More...
- struct slcd_config_t
  SLCD configure structure. More...

Enumerations

- enum slcd_power_supply_option_t {
  kSLCD_InternalVll3UseChargePump,
  kSLCD_ExternalVll3UseResistorBiasNetwork,
  kSLCD_ExteranlVll3UseChargePump,
  kSLCD_InternalVll1UseChargePump }
  SLCD power supply option.
- enum slcd_regulated_voltage_trim_t {
  kSLCD_RegulatedVolatgeTrim00 = 0U,
  kSLCD_RegulatedVolatgeTrim01,
  kSLCD_RegulatedVolatgeTrim02,
  kSLCD_RegulatedVolatgeTrim03,
  kSLCD_RegulatedVolatgeTrim04,
  kSLCD_RegulatedVolatgeTrim05,
  kSLCD_RegulatedVolatgeTrim06,
  kSLCD_RegulatedVolatgeTrim07,
  kSLCD_RegulatedVolatgeTrim08,
  kSLCD_RegulatedVolatgeTrim09,
  kSLCD_RegulatedVolatgeTrim10,
  kSLCD_RegulatedVolatgeTrim11,
  kSLCD_RegulatedVolatgeTrim12,
  kSLCD_RegulatedVolatgeTrim13,
  kSLCD_RegulatedVolatgeTrim14,
  kSLCD_RegulatedVolatgeTrim15 }
  SLCD regulated voltage trim parameter, be used to meet the desired contrast.

- enum slcd_load_adjust_t {
  kSLCD_LowLoadOrFastestClkSrc = 0U,
  kSLCD_LowLoadOrIntermediateClkSrc,
  kSLCD_HighLoadOrIntermediateClkSrc,
  kSLCD_HighLoadOrSlowestClkSrc }
  SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source.
- enum slcd_clock_src_t {
  kSLCD_DefaultClk = 0U,
  kSLCD_AlternateClk1 = 1U }
  SLCD clock source.
- enum slcd_alt_clock_div_t {
  kSLCD_AltClkDivFactor1 = 0U,
  kSLCD_AltClkDivFactor64,
  kSLCD_AltClkDivFactor256,
  kSLCD_AltClkDivFactor512 }
  SLCD alternate clock divider.
- enum slcd_clock_prescaler_t {
  kSLCD_ClkPrescaler00 = 0U,
  kSLCD_ClkPrescaler01,
  kSLCD_ClkPrescaler02,
  kSLCD_ClkPrescaler03,
  kSLCD_ClkPrescaler04,
  kSLCD_ClkPrescaler05,
  kSLCD_ClkPrescaler06,
  kSLCD_ClkPrescaler07 }
  SLCD clock prescaler to generate frame frequency.
- enum slcd_duty_cycle_t {
  kSLCD_1Div1DutyCycle = 0U,
  kSLCD_1Div2DutyCycle,
  kSLCD_1Div3DutyCycle,
  kSLCD_1Div4DutyCycle,
  kSLCD_1Div5DutyCycle,
  kSLCD_1Div6DutyCycle,
  kSLCD_1Div7DutyCycle,
  kSLCD_1Div8DutyCycle }
  SLCD duty cycle.
- enum slcd_phase_type_t {
  kSLCD_NoPhaseActivate = 0x00U,
  kSLCD_PhaseAActivate = 0x01U,
  kSLCD_PhaseBActivate = 0x02U,
  kSLCD_PhaseCActivate = 0x04U,
  kSLCD_PhaseDActivate = 0x08U,
  kSLCD_PhaseEActivate = 0x10U,
  kSLCD_PhaseFActivate = 0x20U,
  kSLCD_PhaseGActivate = 0x40U,

kSLCD_PhaseHActivate = 0x80U }
SLCD segment phase type.

- enum slcd_phase_index_t {
  kSLCD_PhaseAIndex = 0x0U,
  kSLCD_PhaseBIndex = 0x1U,
  kSLCD_PhaseCIndex = 0x2U,
  kSLCD_PhaseDIndex = 0x3U,
  kSLCD_PhaseEIndex = 0x4U,
  kSLCD_PhaseFIndex = 0x5U,
  kSLCD_PhaseGIndex = 0x6U,
  kSLCD_PhaseHIndex = 0x7U }
  SLCD segment phase bit index.
- enum slcd_display_mode_t {
  kSLCD_NormalMode = 0U,
  kSLCD_AlternateMode,
  kSLCD_BlankMode }
  SLCD display mode.
- enum slcd_blink_mode_t {
  kSLCD_BlankDisplayBlink = 0U,
  kSLCD_AltDisplayBlink }
  SLCD blink mode.
- enum slcd_blink_rate_t {
  kSLCD_BlinkRate00 = 0U,
  kSLCD_BlinkRate01,
  kSLCD_BlinkRate02,
  kSLCD_BlinkRate03,
  kSLCD_BlinkRate04,
  kSLCD_BlinkRate05,
  kSLCD_BlinkRate06,
  kSLCD_BlinkRate07 }
  SLCD blink rate.
- enum slcd_fault_detect_clock_prescaler_t {
  kSLCD_FaultSampleFreqDivider1 = 0U,
  kSLCD_FaultSampleFreqDivider2,
  kSLCD_FaultSampleFreqDivider4,
  kSLCD_FaultSampleFreqDivider8,
  kSLCD_FaultSampleFreqDivider16,
  kSLCD_FaultSampleFreqDivider32,
  kSLCD_FaultSampleFreqDivider64,
  kSLCD_FaultSampleFreqDivider128 }
  SLCD fault detect clock prescaler.
- enum slcd_fault_detect_sample_window_width_t {

kSLCD_FaultDetectWindowWidth4SampleClk = 0U,
kSLCD_FaultDetectWindowWidth8SampleClk,
kSLCD_FaultDetectWindowWidth16SampleClk,
kSLCD_FaultDetectWindowWidth32SampleClk,
kSLCD_FaultDetectWindowWidth64SampleClk,
kSLCD_FaultDetectWindowWidth128SampleClk,
kSLCD_FaultDetectWindowWidth256SampleClk,
kSLCD_FaultDetectWindowWidth512SampleClk }
SLCD fault detect sample window width.

- enum slcd_interrupt_enable_t { kSLCD_FaultDetectCompleteInterrupt = 1U }
  SLCD interrupt source.
- enum slcd_lowpower_behavior {
  kSLCD_EnabledInWaitStop = 0,
  kSLCD_EnabledInWaitOnly,
  kSLCD_EnabledInStopOnly,
  kSLCD_DisabledInWaitStop }
  SLCD behavior in low power mode.

Driver version

- #define FSL_SLCD_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SLCD driver version 2.0.0.

Initialization and deinitialization

- void SLCD_Init (LCD_Type *base, slcd_config_t *configure)
  Initializes the SLCD, ungates the module clock, initializes the power setting, enables all used plane pins, and sets with interrupt and work mode with configuration.
- void SLCD_Deinit (LCD_Type *base)
  Deinitializes the SLCD module, gates the module clock, disables an interrupt, and displays the SLCD.
- void SLCD_GetDefaultConfig (slcd_config_t *configure)
  Gets the SLCD default configuration structure.

Plane Setting and Display Control

- static void SLCD_StartDisplay (LCD_Type *base)
  Enables the SLCD controller, starts generate, and displays the front plane and back plane waveform.
- static void SLCD_StopDisplay (LCD_Type *base)
  Stops the SLCD controller.
- void SLCD_StartBlinkMode (LCD_Type *base, slcd_blink_mode_t mode, slcd_blink_rate_t rate)
  Starts the SLCD blink mode.
- static void SLCD_StopBlinkMode (LCD_Type *base)
  Stops the SLCD blink mode.
- static void SLCD_SetBackPlanePhase (LCD_Type *base, uint32_t pinIndx, slcd_phase_type_t phase)
  Sets the SLCD back plane pin phase.
- static void SLCD_SetFrontPlaneSegments (LCD_Type *base, uint32_t pinIndx, uint8_t operation)
  Sets the SLCD front plane segment operation for a front plane pin.

- static void SLCD_SetFrontPlaneOnePhase (LCD_Type *base, uint32_t pinIndx, slcd_phase_index_t phaseIndx, bool enable)
  Sets one SLCD front plane pin for one phase.
- static uint32_t SLCD_GetFaultDetectCounter (LCD_Type *base)
  Gets the SLCD fault detect counter.

Interrupts.

- void SLCD_EnableInterrupts (LCD_Type *base, uint32_t mask)
  Enables the SLCD interrupt.
- void SLCD_DisableInterrupts (LCD_Type *base, uint32_t mask)
  Disables the SLCD interrupt.
- uint32_t SLCD_GetInterruptStatus (LCD_Type *base)
  Gets the SLCD interrupt status flag.
- void SLCD_ClearInterruptStatus (LCD_Type *base, uint32_t mask)
  Clears the SLCD interrupt events status flag.

46.3 Data Structure Documentation

46.3.1 struct slcd_fault_detect_config_t

Data Fields

- bool faultDetectIntEnable
  Fault frame detection interrupt enable flag.
- bool faultDetectBackPlaneEnable
  True means the pin id fault detected is back plane otherwise front plane.
- uint8_t faultDetectPinIndex
  Fault detected pin id from 0 to 63.
- slcd_fault_detect_clock_prescaler_t faultPrescaler
  Fault detect clock prescaler.
- slcd_fault_detect_sample_window_width_t width
  Fault detect sample window width.

46.3.1.0.0.27 Field Documentation

46.3.1.0.0.27.1 bool slcd_fault_detect_config_t::faultDetectIntEnable

46.3.1.0.0.27.2 bool slcd_fault_detect_config_t::faultDetectBackPlaneEnable

46.3.1.0.0.27.3 uint8_t slcd_fault_detect_config_t::faultDetectPinIndex

46.3.1.0.0.27.4 slcd_fault_detect_clock_prescaler_t slcd_fault_detect_config_t::faultPrescaler

46.3.1.0.0.27.5 slcd_fault_detect_sample_window_width_t slcd_fault_detect_config_t::width

46.3.2 struct slcd_clock_config_t

Data Fields

- slcd_clock_src_t clkSource
  Clock source.
- slcd_alt_clock_div_t altClkDivider
  The divider to divide the alternate clock used for alternate clock source.
- slcd_clock_prescaler_t clkPrescaler
  Clock prescaler.

46.3.2.0.0.28 Field Documentation

46.3.2.0.0.28.1 slcd_clock_src_t slcd_clock_config_t::clkSource

"slcd_clock_src_t" is recommended to be used. The SLCD is optimized to operate using a 32.768kHz clock input.

46.3.2.0.0.28.2 slcd_alt_clock_div_t slcd_clock_config_t::altClkDivider

46.3.2.0.0.28.3 slcd_clock_prescaler_t slcd_clock_config_t::clkPrescaler

46.3.3 struct slcd_config_t

Data Fields

- slcd_power_supply_option_t powerSupply
  Power supply option.
- slcd_regulated_voltage_trim_t voltageTrim
  Regulated voltage trim used for the internal regulator VIREG to adjust to facilitate contrast control.
- slcd_clock_config_t * clkConfig
  Clock configure.
- slcd_display_mode_t displayMode
  SLCD display mode.
- slcd_load_adjust_t loadAdjust
  Load adjust to handle glass capacitance.
- slcd_duty_cycle_t dutyCycle

Duty cycle.
- slcd_lowpower_behavior lowPowerBehavior
  SLCD behavior in low power mode.
- uint32_t slcdLowPinEnabled
  Setting enabled SLCD pin 0 ~ pin 31.
- uint32_t slcdHighPinEnabled
  Setting enabled SLCD pin 32 ~ pin 63.
- uint32_t backPlaneLowPin
  Setting back plane pin 0 ~ pin 31.
- uint32_t backPlaneHighPin
  Setting back plane pin 32 ~ pin 63.
- slcd_fault_detect_config_t * faultConfig
  Fault frame detection configure.

46.3.3.0.0.29 Field Documentation

46.3.3.0.0.29.1 slcd_power_supply_option_t slcd_config_t::powerSupply

46.3.3.0.0.29.2 slcd_regulated_voltage_trim_t slcd_config_t::voltageTrim

46.3.3.0.0.29.3 slcd_clock_config_t* slcd_config_t::clkConfig

46.3.3.0.0.29.4 slcd_display_mode_t slcd_config_t::displayMode

46.3.3.0.0.29.5 slcd_load_adjust_t slcd_config_t::loadAdjust

46.3.3.0.0.29.6 slcd_duty_cycle_t slcd_config_t::dutyCycle

46.3.3.0.0.29.7 slcd_lowpower_behavior slcd_config_t::lowPowerBehavior

46.3.3.0.0.29.8 uint32_t slcd_config_t::slcdLowPinEnabled

Setting bit n to 1 means enable pin n.

46.3.3.0.0.29.9 uint32_t slcd_config_t::slcdHighPinEnabled

Setting bit n to 1 means enable pin (n + 32).

46.3.3.0.0.29.10 uint32_t slcd_config_t::backPlaneLowPin

Setting bit n to 1 means setting pin n as back plane. It should never have the same bit setting as the frontPlane Pin.

46.3.3.0.0.29.11 uint32_t slcd_config_t::backPlaneHighPin

Setting bit n to 1 means setting pin (n + 32) as back plane. It should never have the same bit setting as the frontPlane Pin.

46.3.3.0.0.29.12 slcd_fault_detect_config_t* slcd_config_t::faultConfig

If not requirement, set to NULL.

46.4 Macro Definition Documentation

46.4.1 #define FSL_SLCD_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

46.5 Enumeration Type Documentation

46.5.1 enum slcd_power_supply_option_t

Enumerator

kSLCD_InternalVll3UseChargePump VLL3 connected to VDD internally, charge pump is used to generate VLL1 and VLL2.

kSLCD_ExternalVll3UseResistorBiasNetwork VLL3 is driven externally and resistor bias network is used to generate VLL1 and VLL2.

kSLCD_ExteranlVll3UseChargePump VLL3 is driven externally and charge pump is used to generate VLL1 and VLL2.

kSLCD_InternalVll1UseChargePump VIREG is connected to VLL1 internally and charge pump is used to generate VLL2 and VLL3.

46.5.2 enum slcd_regulated_voltage_trim_t

Enumerator

kSLCD_RegulatedVolatgeTrim00 Increase the voltage to 0.91 V.
kSLCD_RegulatedVolatgeTrim01 Increase the voltage to 1.01 V.
kSLCD_RegulatedVolatgeTrim02 Increase the voltage to 0.96 V.
kSLCD_RegulatedVolatgeTrim03 Increase the voltage to 1.06 V.
kSLCD_RegulatedVolatgeTrim04 Increase the voltage to 0.93 V.
kSLCD_RegulatedVolatgeTrim05 Increase the voltage to 1.02 V.
kSLCD_RegulatedVolatgeTrim06 Increase the voltage to 0.98 V.
kSLCD_RegulatedVolatgeTrim07 Increase the voltage to 1.08 V.
kSLCD_RegulatedVolatgeTrim08 Increase the voltage to 0.92 V.
kSLCD_RegulatedVolatgeTrim09 Increase the voltage to 1.02 V.
kSLCD_RegulatedVolatgeTrim10 Increase the voltage to 0.97 V.
kSLCD_RegulatedVolatgeTrim11 Increase the voltage to 1.07 V.
kSLCD_RegulatedVolatgeTrim12 Increase the voltage to 0.94 V.
kSLCD_RegulatedVolatgeTrim13 Increase the voltage to 1.05 V.
kSLCD_RegulatedVolatgeTrim14 Increase the voltage to 0.99 V.
kSLCD_RegulatedVolatgeTrim15 Increase the voltage to 1.09 V.

46.5.3 enum slcd_load_adjust_t

Adjust the LCD glass capacitance if resistor bias network is enabled: kSLCD_LowLoadOrFastestClkSrc - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,V-

LL2,Vcap1 and Vcap2 pins) kSLCD_LowLoadOrIntermediateClkSrc - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) kSLCD-_HighLoadOrIntermediateClkSrc - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) kSLCD_HighLoadOrSlowestClkSrc - high load (LC-D glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: kSLCD_LowLoadOrFastestClkSrc - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) kSLCD_-LowLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 4000pF or 2000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOrIntermediateClkSrc - Intermediate clock source (LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) kSLCD_HighLoadOr-SlowestClkSrc - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

Enumerator

kSLCD_LowLoadOrFastestClkSrc Adjust in low load or selects fastest clock.
kSLCD_LowLoadOrIntermediateClkSrc Adjust in low load or selects intermediate clock.
kSLCD_HighLoadOrIntermediateClkSrc Adjust in high load or selects intermediate clock.
kSLCD_HighLoadOrSlowestClkSrc Adjust in high load or selects slowest clock.

46.5.4 enum slcd_clock_src_t

Enumerator

kSLCD_DefaultClk Select default clock ERCLK32K.
kSLCD_AlternateClk1 Select alternate clock source 1 : MCGIRCLK.

46.5.5 enum slcd_alt_clock_div_t

Enumerator

kSLCD_AltClkDivFactor1 No divide for alternate clock.
kSLCD_AltClkDivFactor64 Divide alternate clock with factor 64.
kSLCD_AltClkDivFactor256 Divide alternate clock with factor 256.
kSLCD_AltClkDivFactor512 Divide alternate clock with factor 512.

46.5.6 enum slcd_clock_prescaler_t

Enumerator

kSLCD_ClkPrescaler00 Prescaler 0.
kSLCD_ClkPrescaler01 Prescaler 1.

kSLCD_ClkPrescaler02 Prescaler 2.
kSLCD_ClkPrescaler03 Prescaler 3.
kSLCD_ClkPrescaler04 Prescaler 4.
kSLCD_ClkPrescaler05 Prescaler 5.
kSLCD_ClkPrescaler06 Prescaler 6.
kSLCD_ClkPrescaler07 Prescaler 7.

46.5.7 enum slcd_duty_cycle_t

Enumerator

kSLCD_1Div1DutyCycle LCD use 1 BP 1/1 duty cycle.
kSLCD_1Div2DutyCycle LCD use 2 BP 1/2 duty cycle.
kSLCD_1Div3DutyCycle LCD use 3 BP 1/3 duty cycle.
kSLCD_1Div4DutyCycle LCD use 4 BP 1/4 duty cycle.
kSLCD_1Div5DutyCycle LCD use 5 BP 1/5 duty cycle.
kSLCD_1Div6DutyCycle LCD use 6 BP 1/6 duty cycle.
kSLCD_1Div7DutyCycle LCD use 7 BP 1/7 duty cycle.
kSLCD_1Div8DutyCycle LCD use 8 BP 1/8 duty cycle.

46.5.8 enum slcd_phase_type_t

Enumerator

kSLCD_NoPhaseActivate LCD wareform no phase activates.
kSLCD_PhaseAActivate LCD waveform phase A activates.
kSLCD_PhaseBActivate LCD waveform phase B activates.
kSLCD_PhaseCActivate LCD waveform phase C activates.
kSLCD_PhaseDActivate LCD waveform phase D activates.
kSLCD_PhaseEActivate LCD waveform phase E activates.
kSLCD_PhaseFActivate LCD waveform phase F activates.
kSLCD_PhaseGActivate LCD waveform phase G activates.
kSLCD_PhaseHActivate LCD waveform phase H activates.

46.5.9 enum slcd_phase_index_t

Enumerator

kSLCD_PhaseAIndex LCD phase A bit index.
kSLCD_PhaseBIndex LCD phase B bit index.
kSLCD_PhaseCIndex LCD phase C bit index.

kSLCD_PhaseDIndex LCD phase D bit index.
kSLCD_PhaseEIndex LCD phase E bit index.
kSLCD_PhaseFIndex LCD phase F bit index.
kSLCD_PhaseGIndex LCD phase G bit index.
kSLCD_PhaseHIndex LCD phase H bit index.

46.5.10 enum slcd_display_mode_t

Enumerator

kSLCD_NormalMode LCD Normal display mode.
kSLCD_AlternateMode LCD Alternate display mode. For four back planes or less.
kSLCD_BlankMode LCD Blank display mode.

46.5.11 enum slcd_blink_mode_t

Enumerator

kSLCD_BlankDisplayBlink Display blank during the blink period.
kSLCD_AltDisplayBlink Display alternate display during the blink period if duty cycle is lower than 5.

46.5.12 enum slcd_blink_rate_t

Enumerator

kSLCD_BlinkRate00 SLCD blink rate is LCD clock/((2^12)).
kSLCD_BlinkRate01 SLCD blink rate is LCD clock/((2^13)).
kSLCD_BlinkRate02 SLCD blink rate is LCD clock/((2^14)).
kSLCD_BlinkRate03 SLCD blink rate is LCD clock/((2^15)).
kSLCD_BlinkRate04 SLCD blink rate is LCD clock/((2^16)).
kSLCD_BlinkRate05 SLCD blink rate is LCD clock/((2^17)).
kSLCD_BlinkRate06 SLCD blink rate is LCD clock/((2^18)).
kSLCD_BlinkRate07 SLCD blink rate is LCD clock/((2^19)).

46.5.13 enum slcd_fault_detect_clock_prescaler_t

Enumerator

kSLCD_FaultSampleFreqDivider1 Fault detect sample clock frequency is 1/1 bus clock.

kSLCD_FaultSampleFreqDivider2 Fault detect sample clock frequency is 1/2 bus clock.
kSLCD_FaultSampleFreqDivider4 Fault detect sample clock frequency is 1/4 bus clock.
kSLCD_FaultSampleFreqDivider8 Fault detect sample clock frequency is 1/8 bus clock.
kSLCD_FaultSampleFreqDivider16 Fault detect sample clock frequency is 1/16 bus clock.
kSLCD_FaultSampleFreqDivider32 Fault detect sample clock frequency is 1/32 bus clock.
kSLCD_FaultSampleFreqDivider64 Fault detect sample clock frequency is 1/64 bus clock.
kSLCD_FaultSampleFreqDivider128 Fault detect sample clock frequency is 1/128 bus clock.

46.5.14 enum slcd_fault_detect_sample_window_width_t

Enumerator

kSLCD_FaultDetectWindowWidth4SampleClk Sample window width is 4 sample clock cycles.
kSLCD_FaultDetectWindowWidth8SampleClk Sample window width is 8 sample clock cycles.
kSLCD_FaultDetectWindowWidth16SampleClk Sample window width is 16 sample clock cycles.
kSLCD_FaultDetectWindowWidth32SampleClk Sample window width is 32 sample clock cycles.
kSLCD_FaultDetectWindowWidth64SampleClk Sample window width is 64 sample clock cycles.
kSLCD_FaultDetectWindowWidth128SampleClk Sample window width is 128 sample clock cycles.
kSLCD_FaultDetectWindowWidth256SampleClk Sample window width is 256 sample clock cycles.
kSLCD_FaultDetectWindowWidth512SampleClk Sample window width is 512 sample clock cycles.

46.5.15 enum slcd_interrupt_enable_t

Enumerator

kSLCD_FaultDetectCompleteInterrupt SLCD fault detection complete interrupt source.

46.5.16 enum slcd_lowpower_behavior

Enumerator

kSLCD_EnabledInWaitStop SLCD works in wait and stop mode.
kSLCD_EnabledInWaitOnly SLCD works in wait mode and is disabled in stop mode.
kSLCD_EnabledInStopOnly SLCD works in stop mode and is disabled in wait mode.
kSLCD_DisabledInWaitStop SLCD is disabled in stop mode and wait mode.

46.6 Function Documentation

46.6.1 void SLCD_Init (LCD_Type * *base*, slcd_config_t * *configure*)

Parameters

base	SLCD peripheral base address.
configure	SLCD configuration pointer. For the configuration structure, many parameters have the default setting and the SLCD_Getdefaultconfig() is provided to get them. Use it verified for their applications. The others have no default settings such as "clkConfig" and must be provided by the application before calling the SLCD_Init() API.

46.6.2 void SLCD_Deinit (LCD_Type * *base*)

Parameters

base	SLCD peripheral base address.

46.6.3 void SLCD_GetDefaultConfig (slcd_config_t * *configure*)

The purpose of this API is to get default parameters of the configuration structure for the SLCD_Init(). Use these initialized parameters unchanged in SLCD_Init(), or modify some fields of the structure before the calling SLCD_Init(). All default parameters of the configure structure are listed:

```
config.displayMode        = kSLCD_NormalMode; // SLCD normal mode
config.powerSupply        = kSLCD_InternalVll3UseChargePump;  // Use charge
       pump internal VLL3
config.voltageTrim        = kSLCD_RegulatedVolatgeTrim00;
config.lowPowerBehavior   = kSLCD_EnabledInWaitStop;  // Work on low power mode
config.interruptSrc       = 0;                // No interrupt source is enabled
config.faultConfig        = NULL;             // Fault detection is disabled
config.frameFreqIntEnable =  false;
```

Parameters

configure	The SLCD configuration structure pointer.

46.6.4 static void SLCD_StartDisplay (LCD_Type * *base*) [inline], [static]

Parameters

base	SLCD peripheral base address.

46.6.5 static void SLCD_StopDisplay (LCD_Type * *base*) [inline], [static]

There is no waveform generator and all enabled pins only output a low value.

Parameters

base	SLCD peripheral base address.

46.6.6 void SLCD_StartBlinkMode (LCD_Type * *base*, slcd_blink_mode_t *mode*, slcd_blink_rate_t *rate*)

Parameters

base	SLCD peripheral base address.
mode	SLCD blink mode.
rate	SLCD blink rate.

46.6.7 static void SLCD_StopBlinkMode (LCD_Type * *base*) [inline], [static]

Parameters

base	SLCD peripheral base address.

46.6.8 static void SLCD_SetBackPlanePhase (LCD_Type * *base*, uint32_t *pinIndx*, slcd_phase_type_t *phase*) [inline], [static]

This function sets the SLCD back plane pin phase. "kSLCD_PhaseXActivate" setting means the phase X is active for the back plane pin. "kSLCD_NoPhaseActivate" setting means there is no phase active for the back plane pin. register value. For example, set the back plane pin 20 for phase A:

```
SLCD_SetBackPlanePhase(LCD, 20, kSLCD_PhaseAActivate);
```

Parameters

base	SLCD peripheral base address.
pinIndx	SLCD back plane pin index. Range from 0 to 63.
phase	The phase activates for the back plane pin.

46.6.9 static void SLCD_SetFrontPlaneSegments (LCD_Type * *base*, uint32_t *pinIndx*, uint8_t *operation*) [inline], [static]

This function sets the SLCD front plane segment on or off operation. Each bit turns on or off the segments associated with the front plane pin in the following pattern: HGFEDCBA (most significant bit controls segment H and least significant bit controls segment A). For example, turn on the front plane pin 20 for phase B and phase C:

```
SLCD_SetFrontPlaneSegments(LCD, 20, (
      kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));
```

Parameters

base	SLCD peripheral base address.
pinIndx	SLCD back plane pin index. Range from 0 to 63.
operation	The operation for the segment on the front plane pin. This is a logical OR of the enumeration :: slcd_phase_type_t.

46.6.10 static void SLCD_SetFrontPlaneOnePhase (LCD_Type * *base*, uint32_t *pinIndx*, slcd_phase_index_t *phaseIndx*, bool *enable*) [inline], [static]

This function can be used to set one phase on or off for the front plane pin. It can be call many times to set the plane pin for different phase indexes. For example, turn on the front plane pin 20 for phase B and phase C:

```
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseBIndex, true);
SLCD_SetFrontPlaneOnePhase(LCD, 20, kSLCD_PhaseCIndex, true);
```

Parameters

base	SLCD peripheral base address.
pinIndx	SLCD back plane pin index. Range from 0 to 63.
phaseIndx	The phase bit index slcd_phase_index_t.
enable	True to turn on the segment for phaseIndx phase false to turn off the segment for phaseIndx phase.

46.6.11 static uint32_t SLCD_GetFaultDetectCounter (LCD_Type * *base*) [inline], [static]

This function gets the number of samples inside the fault detection sample window.

Parameters

base	SLCD peripheral base address.

Returns

The fault detect counter. The maximum return value is 255. If the maximum 255 returns, the overflow may happen. Reconfigure the fault detect sample window and fault detect clock prescaler for proper sampling.

46.6.12 void SLCD_EnableInterrupts (LCD_Type * *base*, uint32_t *mask*)

For example, to enable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD_EnableInterrupts(LCD,
      kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);
```

Parameters

base	SLCD peripheral base address.
mask	SLCD interrupts to enable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

46.6.13 void SLCD_DisableInterrupts (LCD_Type * *base*, uint32_t *mask*)

For example, to disable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
SLCD_DisableInterrupts(LCD,
      kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);
```

Parameters

base	SLCD peripheral base address.
mask	SLCD interrupts to disable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

46.6.14 uint32_t SLCD_GetInterruptStatus (LCD_Type * *base*)

Parameters

base	SLCD peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

46.6.15 void SLCD_ClearInterruptStatus (LCD_Type * *base*, uint32_t *mask*)

Parameters

base	SLCD peripheral base address.
mask	SLCD interrupt source to be cleared. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

Chapter 47
SMC: System Mode Controller Driver

47.1 Overview

The KSDK provides a Peripheral driver for the System Mode Controller (SMC) module of Kinetis devices. The SMC module is responsible for sequencing the system into and out of all low-power Stop and Run modes

API functions are provided for configuring the system working in a dedicated power mode. For different power modes, function `SMC_SetPowerModexxx` accepts different parameters. System power mode state transitions are not available for between power modes. For details about available transitions, see the Power mode transitions section in the SoC reference manual.

Files

- file fsl_smc.h

Enumerations

- enum smc_power_mode_protection_t {
  kSMC_AllowPowerModeVlp = SMC_PMPROT_AVLP_MASK,
  kSMC_AllowPowerModeAll }
  Power Modes Protection.
- enum smc_power_state_t {
  kSMC_PowerStateRun = 0x01U << 0U,
  kSMC_PowerStateStop = 0x01U << 1U,
  kSMC_PowerStateVlpr = 0x01U << 2U,
  kSMC_PowerStateVlpw = 0x01U << 3U,
  kSMC_PowerStateVlps = 0x01U << 4U }
  Power Modes in PMSTAT.
- enum smc_run_mode_t {
  kSMC_RunNormal = 0U,
  kSMC_RunVlpr = 2U }
  Run mode definition.
- enum smc_stop_mode_t {
  kSMC_StopNormal = 0U,
  kSMC_StopVlps = 2U }
  Stop mode definition.
- enum smc_partial_stop_option_t {
  kSMC_PartialStop = 0U,
  kSMC_PartialStop1 = 1U,
  kSMC_PartialStop2 = 2U }
  Partial STOP option.
- enum _smc_status { kStatus_SMC_StopAbort = MAKE_STATUS(kStatusGroup_POWER, 0) }

SMC configuration status.

Driver version

- #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))
  SMC driver version 2.0.1.

System mode controller APIs

- static void SMC_SetPowerModeProtection (SMC_Type ∗base, uint8_t allowedModes)
  Configures all power mode protection settings.
- static smc_power_state_t SMC_GetPowerModeState (SMC_Type ∗base)
  Gets the current power mode status.
- status_t SMC_SetPowerModeRun (SMC_Type ∗base)
  Configure the system to RUN power mode.
- status_t SMC_SetPowerModeWait (SMC_Type ∗base)
  Configure the system to WAIT power mode.
- status_t SMC_SetPowerModeStop (SMC_Type ∗base, smc_partial_stop_option_t option)
  Configure the system to Stop power mode.
- status_t SMC_SetPowerModeVlpr (SMC_Type ∗base)
  Configure the system to VLPR power mode.
- status_t SMC_SetPowerModeVlpw (SMC_Type ∗base)
  Configure the system to VLPW power mode.
- status_t SMC_SetPowerModeVlps (SMC_Type ∗base)
  Configure the system to VLPS power mode.

47.2 Macro Definition Documentation

47.2.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

47.3 Enumeration Type Documentation

47.3.1 enum smc_power_mode_protection_t

Enumerator

kSMC_AllowPowerModeVlp Allow Very-Low-Power Mode.
kSMC_AllowPowerModeAll Allow all power mode.

47.3.2 enum smc_power_state_t

Enumerator

kSMC_PowerStateRun 0000_0001 - Current power mode is RUN
kSMC_PowerStateStop 0000_0010 - Current power mode is STOP
kSMC_PowerStateVlpr 0000_0100 - Current power mode is VLPR
kSMC_PowerStateVlpw 0000_1000 - Current power mode is VLPW
kSMC_PowerStateVlps 0001_0000 - Current power mode is VLPS

47.3.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal normal RUN mode.
kSMC_RunVlpr Very-Low-Power RUN mode.

47.3.4 enum smc_stop_mode_t

Enumerator

kSMC_StopNormal Normal STOP mode.
kSMC_StopVlps Very-Low-Power STOP mode.

47.3.5 enum smc_partial_stop_option_t

Enumerator

kSMC_PartialStop STOP - Normal Stop mode.
kSMC_PartialStop1 Partial Stop with both system and bus clocks disabled.
kSMC_PartialStop2 Partial Stop with system clock disabled and bus clock enabled.

47.3.6 enum _smc_status

Enumerator

kStatus_SMC_StopAbort Entering Stop mode is abort.

47.4 Function Documentation

47.4.1 static void SMC_SetPowerModeProtection (SMC_Type * *base,* uint8_t *allowedModes*) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the smc_power_mode_protection_t. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map, for example, to allow LLS and VLLS, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps). To allow all modes, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll).

Parameters

base	SMC peripheral base address.
allowedModes	Bitmap of the allowed power modes.

47.4.2 static smc_power_state_t SMC_GetPowerModeState (SMC_Type * *base*) [inline], [static]

This function returns the current power mode stat. Once application switches the power mode, it should always check the stat to check whether it runs into the specified mode or not. An application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc_power_state_t for information about the power stat.

Parameters

base	SMC peripheral base address.

Returns

Current power mode status.

47.4.3 status_t SMC_SetPowerModeRun (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

47.4.4 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

47.4.5 status_t SMC_SetPowerModeStop (SMC_Type * *base*, smc_partial_stop_option_t *option*)

Parameters

base	SMC peripheral base address.
option	Partial Stop mode option.

Returns

SMC configuration error code.

47.4.6 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

47.4.7 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

47.4.8 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

base	SMC peripheral base address.

Returns

SMC configuration error code.

Chapter 48
SPI: Serial Peripheral Interface Driver

48.1 Overview

Modules

- SPI Driver

48.2 Overview

- SPI FreeRTOS driver
- SPI µCOS/II driver
- SPI µCOS/III driver

48.3 SPI Driver

48.3.1 Overview

48.3.2 Overview

SPI driver includes functional APIs and transactional APIs.

Functional APIs are feature/property target low level APIs. Functional APIs can be used for SPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the SPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. SPI 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 spi_handle_t as the first parameter. Initialize the handle by calling the SPI_MasterTransferCreateHandle() or SPI_SlaveTransferCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions SPI_MasterTransferNonBlocking() and SPI_SlaveTransferNonBlocking() set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_SPI_Idle status.

48.3.3 Typical use case

48.3.3.1 SPI master transfer using an interrupt method

```
#define BUFFER_LEN (64)
spi_master_handle_t spiHandle;
spi_master_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished = false;

const uint8_t sendData[BUFFER_LEN] = [......];
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_master_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    SPI_MasterGetDefaultConfig(&masterConfig);

    SPI_MasterInit(SPI0, &masterConfig);
    SPI_MasterTransferCreateHandle(SPI0, &spiHandle, SPI_UserCallback, NULL);

    // Prepare to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
    xfer.dataSize = BUFFER_LEN;
```

```
    // Send out.
    SPI_MasterTransferNonBlocking(SPI0, &spiHandle, &xfer);

    // Wait send finished.
    while (!isFinished)
    {
    }

    // ...
}
```

48.3.3.2 SPI Send/receive using a DMA method

```
#define BUFFER_LEN (64)
spi_dma_handle_t spiHandle;
dma_handle_t g_spiTxDmaHandle;
dma_handle_t g_spiRxDmaHandle;
spi_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished;
uint8_t sendData[BUFFER_LEN] = ...;
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    SPI_MasterGetDefaultConfig(&masterConfig);
    SPI_MasterInit(SPI0, &masterConfig);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SPI_TX_DMA_CHANNEL, SPI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SPI_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, SPI_RX_DMA_CHANNEL, SPI_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SPI_RX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle.
    DMA_CreateHandle(&g_spiTxDmaHandle, DMA0, SPI_TX_DMA_CHANNEL);
    DMA_CreateHandle(&g_spiRxDmaHandle, DMA0, SPI_RX_DMA_CHANNEL);

    SPI_MasterTransferCreateHandleDMA(SPI0, spiHandle, &g_spiTxDmaHandle, &g_spiRxDmaHandle,
       SPI_UserCallback, NULL);

    // Prepares to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
    xfer.dataSize = BUFFER_LEN;

    // Sends out.
    SPI_MasterTransferDMA(SPI0, &spiHandle, &xfer);

    // Waits for send to complete.
    while (!isFinished)
    {
    }
```

```
    // ...
}
```

Modules

- SPI DMA Driver

Files

- file fsl_spi.h

Data Structures

- struct spi_master_config_t
  SPI master user configure structure. More...
- struct spi_slave_config_t
  SPI slave user configure structure. More...
- struct spi_transfer_t
  SPI transfer structure. More...
- struct spi_master_handle_t
  SPI transfer handle structure. More...

Macros

- #define SPI_DUMMYDATA (0xFFU)
  SPI dummy transfer data, the data is sent while txBuff is NULL.

Typedefs

- typedef spi_master_handle_t spi_slave_handle_t
  Slave handle is the same with master handle.
- typedef void(∗ spi_master_callback_t)(SPI_Type ∗base, spi_master_handle_t ∗handle, status_t status, void ∗userData)
  SPI master callback for finished transmit.
- typedef void(∗ spi_slave_callback_t)(SPI_Type ∗base, spi_slave_handle_t ∗handle, status_t status, void ∗userData)
  SPI master callback for finished transmit.

Enumerations

- enum _spi_status {
  kStatus_SPI_Busy = MAKE_STATUS(kStatusGroup_SPI, 0),
  kStatus_SPI_Idle = MAKE_STATUS(kStatusGroup_SPI, 1),

kStatus_SPI_Error = MAKE_STATUS(kStatusGroup_SPI, 2) }
Return status for the SPI driver.

- enum spi_clock_polarity_t {
  kSPI_ClockPolarityActiveHigh = 0x0U,
  kSPI_ClockPolarityActiveLow }
  SPI clock polarity configuration.
- enum spi_clock_phase_t {
  kSPI_ClockPhaseFirstEdge = 0x0U,
  kSPI_ClockPhaseSecondEdge }
  SPI clock phase configuration.
- enum spi_shift_direction_t {
  kSPI_MsbFirst = 0x0U,
  kSPI_LsbFirst }
  SPI data shifter direction options.
- enum spi_ss_output_mode_t {
  kSPI_SlaveSelectAsGpio = 0x0U,
  kSPI_SlaveSelectFaultInput = 0x2U,
  kSPI_SlaveSelectAutomaticOutput = 0x3U }
  SPI slave select output mode options.
- enum spi_pin_mode_t {
  kSPI_PinModeNormal = 0x0U,
  kSPI_PinModeInput = 0x1U,
  kSPI_PinModeOutput = 0x3U }
  SPI pin mode options.
- enum spi_data_bitcount_mode_t {
  kSPI_8BitMode = 0x0U,
  kSPI_16BitMode }
  SPI data length mode options.
- enum _spi_interrupt_enable {
  kSPI_RxFullAndModfInterruptEnable = 0x1U,
  kSPI_TxEmptyInterruptEnable = 0x2U,
  kSPI_MatchInterruptEnable = 0x4U }
  SPI interrupt sources.
- enum _spi_flags {
  kSPI_RxBufferFullFlag = SPI_S_SPRF_MASK,
  kSPI_MatchFlag = SPI_S_SPMF_MASK,
  kSPI_TxBufferEmptyFlag = SPI_S_SPTEF_MASK,
  kSPI_ModeFaultFlag = SPI_S_MODF_MASK }
  SPI status flags.

Driver version

- #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SPI driver version 2.0.0.

Initialization and deinitialization

- void SPI_MasterGetDefaultConfig (spi_master_config_t *config)
  Sets the SPI master configuration structure to default values.
- void SPI_MasterInit (SPI_Type *base, const spi_master_config_t *config, uint32_t srcClock_Hz)
  Initializes the SPI with master configuration.
- void SPI_SlaveGetDefaultConfig (spi_slave_config_t *config)
  Sets the SPI slave configuration structure to default values.
- void SPI_SlaveInit (SPI_Type *base, const spi_slave_config_t *config)
  Initializes the SPI with slave configuration.
- void SPI_Deinit (SPI_Type *base)
  De-initializes the SPI.
- static void SPI_Enable (I2C_Type *base, bool enable)
  Enables or disables the SPI.

Status

- uint32_t SPI_GetStatusFlags (SPI_Type *base)
  Gets the status flag.

Interrupts

- void SPI_EnableInterrupts (SPI_Type *base, uint32_t mask)
  Enables the interrupt for the SPI.
- void SPI_DisableInterrupts (SPI_Type *base, uint32_t mask)
  Disables the interrupt for the SPI.

DMA Control

- static uint32_t SPI_GetDataRegisterAddress (SPI_Type *base)
  Gets the SPI tx/rx data register address.

Bus Operations

- void SPI_MasterSetBaudRate (SPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the baud rate for SPI transfer.
- static void SPI_SetMatchData (SPI_Type *base, uint32_t matchData)
  Sets the match data for SPI.
- void SPI_WriteBlocking (SPI_Type *base, uint8_t *buffer, size_t size)
  Sends a buffer of data bytes using a blocking method.
- void SPI_WriteData (SPI_Type *base, uint16_t data)
  Writes a data into the SPI data register.
- uint16_t SPI_ReadData (SPI_Type *base)
  Gets a data from the SPI data register.

Transactional

- void SPI_MasterTransferCreateHandle (SPI_Type *base, spi_master_handle_t *handle, spi_master_callback_t callback, void *userData)
  Initializes the SPI master handle.
- status_t SPI_MasterTransferBlocking (SPI_Type *base, spi_transfer_t *xfer)
  Transfers a block of data using a polling method.
- status_t SPI_MasterTransferNonBlocking (SPI_Type *base, spi_master_handle_t *handle, spi_transfer_t *xfer)
  Performs a non-blocking SPI interrupt transfer.
- status_t SPI_MasterTransferGetCount (SPI_Type *base, spi_master_handle_t *handle, size_t *count)
  Gets the bytes of the SPI interrupt transferred.
- void SPI_MasterTransferAbort (SPI_Type *base, spi_master_handle_t *handle)
  Aborts an SPI transfer using interrupt.
- void SPI_MasterTransferHandleIRQ (SPI_Type *base, spi_master_handle_t *handle)
  Interrupts the handler for the SPI.
- static void SPI_SlaveTransferCreateHandle (SPI_Type *base, spi_slave_handle_t *handle, spi_slave_callback_t callback, void *userData)
  Initializes the SPI slave handle.
- static status_t SPI_SlaveTransferNonBlocking (SPI_Type *base, spi_slave_handle_t *handle, spi_transfer_t *xfer)
  Performs a non-blocking SPI slave interrupt transfer.
- static status_t SPI_SlaveTransferGetCount (SPI_Type *base, spi_slave_handle_t *handle, size_t *count)
  Gets the bytes of the SPI interrupt transferred.
- static void SPI_SlaveTransferAbort (SPI_Type *base, spi_slave_handle_t *handle)
  Aborts an SPI slave transfer using interrupt.
- static void SPI_SlaveTransferHandleIRQ (SPI_Type *base, spi_slave_handle_t *handle)
  Interrupts a handler for the SPI slave.

48.3.4 Data Structure Documentation

48.3.4.1 struct spi_master_config_t

Data Fields

- bool enableMaster
  Enable SPI at initialization time.
- bool enableStopInWaitMode
  SPI stop in wait mode.
- spi_clock_polarity_t polarity
  Clock polarity.
- spi_clock_phase_t phase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.
- spi_ss_output_mode_t outputMode

SS pin setting.
- spi_pin_mode_t pinMode
  SPI pin mode select.
- uint32_t baudRate_Bps
  Baud Rate for SPI in Hz.

48.3.4.2 struct spi_slave_config_t

Data Fields

- bool enableSlave
  Enable SPI at initialization time.
- bool enableStopInWaitMode
  SPI stop in wait mode.
- spi_clock_polarity_t polarity
  Clock polarity.
- spi_clock_phase_t phase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.

48.3.4.3 struct spi_transfer_t

Data Fields

- uint8_t * txData
  Send buffer.
- uint8_t * rxData
  Receive buffer.
- size_t dataSize
  Transfer bytes.
- uint32_t flags
  SPI control flag, useless to SPI.

48.3.4.3.0.30 Field Documentation

48.3.4.3.0.30.1 uint32_t spi_transfer_t::flags

48.3.4.4 struct _spi_master_handle

Data Fields

- uint8_t *volatile txData
  Transfer buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t txRemainingBytes
  Send data remaining in bytes.
- volatile size_t rxRemainingBytes

Receive data remaining in bytes.
- volatile uint32_t state
  SPI internal state.
- size_t transferSize
  Bytes to be transferred.
- uint8_t bytePerFrame
  SPI mode, 2bytes or 1byte in a frame.
- uint8_t watermark
  Watermark value for SPI transfer.
- spi_master_callback_t callback
  SPI callback.
- void * userData
  Callback parameter.

48.3.5 Macro Definition Documentation

48.3.5.1 #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

48.3.5.2 #define SPI_DUMMYDATA (0xFFU)

48.3.6 Enumeration Type Documentation

48.3.6.1 enum _spi_status

Enumerator

kStatus_SPI_Busy SPI bus is busy.
kStatus_SPI_Idle SPI is idle.
kStatus_SPI_Error SPI error.

48.3.6.2 enum spi_clock_polarity_t

Enumerator

kSPI_ClockPolarityActiveHigh Active-high SPI clock (idles low).
kSPI_ClockPolarityActiveLow Active-low SPI clock (idles high).

48.3.6.3 enum spi_clock_phase_t

Enumerator

kSPI_ClockPhaseFirstEdge First edge on SPSCK occurs at the middle of the first cycle of a data transfer.
kSPI_ClockPhaseSecondEdge First edge on SPSCK occurs at the start of the first cycle of a data transfer.

48.3.6.4 enum spi_shift_direction_t

Enumerator

kSPI_MsbFirst Data transfers start with most significant bit.
kSPI_LsbFirst Data transfers start with least significant bit.

48.3.6.5 enum spi_ss_output_mode_t

Enumerator

kSPI_SlaveSelectAsGpio Slave select pin configured as GPIO.
kSPI_SlaveSelectFaultInput Slave select pin configured for fault detection.
kSPI_SlaveSelectAutomaticOutput Slave select pin configured for automatic SPI output.

48.3.6.6 enum spi_pin_mode_t

Enumerator

kSPI_PinModeNormal Pins operate in normal, single-direction mode.
kSPI_PinModeInput Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input.
kSPI_PinModeOutput Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output.

48.3.6.7 enum spi_data_bitcount_mode_t

Enumerator

kSPI_8BitMode 8-bit data transmission mode
kSPI_16BitMode 16-bit data transmission mode

48.3.6.8 enum _spi_interrupt_enable

Enumerator

kSPI_RxFullAndModfInterruptEnable Receive buffer full (SPRF) and mode fault (MODF) interrupt.
kSPI_TxEmptyInterruptEnable Transmit buffer empty interrupt.
kSPI_MatchInterruptEnable Match interrupt.

48.3.6.9 enum _spi_flags

Enumerator

kSPI_RxBufferFullFlag Read buffer full flag.
kSPI_MatchFlag Match flag.
kSPI_TxBufferEmptyFlag Transmit buffer empty flag.
kSPI_ModeFaultFlag Mode fault flag.

48.3.7 Function Documentation

48.3.7.1 void SPI_MasterGetDefaultConfig (spi_master_config_t * *config*)

The purpose of this API is to get the configuration structure initialized for use in SPI_MasterInit(). User may use the initialized structure unchanged in SPI_MasterInit(), or modify some fields of the structure before calling SPI_MasterInit(). After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

Parameters

config	pointer to master config structure

48.3.7.2 void SPI_MasterInit (SPI_Type * *base*, const spi_master_config_t * *config*, uint32_t *srcClock_Hz*)

The configuration structure can be filled by user from scratch, or be set with default values by SPI_MasterGetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
.baudRate_Bps = 400000,
...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

base	SPI base pointer

config	pointer to master configuration structure
srcClock_Hz	Source clock frequency.

48.3.7.3 void SPI_SlaveGetDefaultConfig (spi_slave_config_t * *config*)

The purpose of this API is to get the configuration structure initialized for use in SPI_SlaveInit(). Modify some fields of the structure before calling SPI_SlaveInit(). Example:

```
spi_slave_config_t config;
SPI_SlaveGetDefaultConfig(&config);
```

Parameters

config	pointer to slave configuration structure

48.3.7.4 void SPI_SlaveInit (SPI_Type * *base*, const spi_slave_config_t * *config*)

The configuration structure can be filled by user from scratch or be set with default values by SPI_SlaveGetDefaultConfig(). After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {
.polarity = kSPIClockPolarity_ActiveHigh;
.phase = kSPIClockPhase_FirstEdge;
.direction = kSPIMsbFirst;
...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

base	SPI base pointer
config	pointer to master configuration structure

48.3.7.5 void SPI_Deinit (SPI_Type * *base*)

Calling this API resets the SPI module, gates the SPI clock. The SPI module can't work unless calling the SPI_MasterInit/SPI_SlaveInit to initialize module.

Parameters

base	SPI base pointer

48.3.7.6 static void SPI_Enable (I2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SPI base pointer
enable	pass true to enable module, false to disable module

48.3.7.7 uint32_t SPI_GetStatusFlags (SPI_Type * *base*)

Parameters

base	SPI base pointer

Returns

SPI Status, use status flag to AND _spi_flags could get the related status.

48.3.7.8 void SPI_EnableInterrupts (SPI_Type * *base*, uint32_t *mask*)

Parameters

base	SPI base pointer
mask	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxFullAndModfInterruptEnable • kSPI_TxEmptyInterruptEnable • kSPI_MatchInterruptEnable • kSPI_RxFifoNearFullInterruptEnable • kSPI_TxFifoNearEmptyInterruptEnable

48.3.7.9 void SPI_DisableInterrupts (SPI_Type * *base*, uint32_t *mask*)

Parameters

base	SPI base pointer
mask	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxFullAndModfInterruptEnable • kSPI_TxEmptyInterruptEnable • kSPI_MatchInterruptEnable • kSPI_RxFifoNearFullInterruptEnable • kSPI_TxFifoNearEmptyInterruptEnable

48.3.7.10 static uint32_t SPI_GetDataRegisterAddress (SPI_Type * *base*) [inline], [static]

This API is used to provide a transfer address for the SPI DMA transfer configuration.

Parameters

base	SPI base pointer

Returns

data register address

48.3.7.11 void SPI_MasterSetBaudRate (SPI_Type * *base,* uint32_t *baudRate_Bps,* uint32_t *srcClock_Hz*)

This is only used in master.

Parameters

base	SPI base pointer
baudRate_Bps	baud rate needed in Hz.
srcClock_Hz	SPI source clock frequency in Hz.

48.3.7.12 static void SPI_SetMatchData (SPI_Type * *base,* uint32_t *matchData*) [inline], [static]

The match data is a hardware comparison value. When the value received in the SPI receive data buffer equals the hardware comparison value, the SPI Match Flag in the S register (S[SPMF]) sets. This can also generate an interrupt if the enable bit sets.

Parameters

base	SPI base pointer
matchData	Match data.

48.3.7.13 void SPI_WriteBlocking (SPI_Type * *base*, uint8_t * *buffer*, size_t *size*)

Note

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

Parameters

base	SPI base pointer
buffer	The data bytes to send
size	The number of data bytes to send

48.3.7.14 void SPI_WriteData (SPI_Type * *base*, uint16_t *data*)

Parameters

base	SPI base pointer
data	needs to be write.

48.3.7.15 uint16_t SPI_ReadData (SPI_Type * *base*)

Parameters

base	SPI base pointer

Returns

Data in the register.

48.3.7.16 void SPI_MasterTransferCreateHandle (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_master_callback_t *callback*, void * *userData*)

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

Parameters

base	SPI peripheral base address.
handle	SPI handle pointer.
callback	Callback function.
userData	User data.

48.3.7.17 status_t SPI_MasterTransferBlocking (SPI_Type * *base*, spi_transfer_t * *xfer*)

Parameters

base	SPI base pointer
xfer	pointer to spi_xfer_config_t structure

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.

48.3.7.18 status_t SPI_MasterTransferNonBlocking (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_transfer_t * *xfer*)

Note

The API immediately returns after transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.
If using the SPI with FIFO for the interrupt transfer, the transfer size is the integer times of the watermark. Otherwise, the last data may be lost because it cannot generate an interrupt request. Users can also call the functional API to get the last received data.

Parameters

base	SPI peripheral base address.
handle	pointer to spi_master_handle_t structure which stores the transfer state

xfer	pointer to spi_xfer_config_t structure

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_SPI_Busy	SPI is not idle, is running another transfer.

48.3.7.19 status_t SPI_MasterTransferGetCount (SPI_Type * *base*, spi_master_handle_t * *handle*, size_t * *count*)

Parameters

base	SPI peripheral base address.
handle	Pointer to SPI transfer handle, this should be a static variable.
count	Transferred bytes of SPI master.

Return values

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

48.3.7.20 void SPI_MasterTransferAbort (SPI_Type * *base*, spi_master_handle_t * *handle*)

Parameters

base	SPI peripheral base address.
handle	Pointer to SPI transfer handle, this should be a static variable.

48.3.7.21 void SPI_MasterTransferHandleIRQ (SPI_Type * *base*, spi_master_handle_t * *handle*)

Parameters

base	SPI peripheral base address.
handle	pointer to spi_master_handle_t structure which stores the transfer state.

48.3.7.22 static void SPI_SlaveTransferCreateHandle (SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_slave_callback_t *callback*, void * *userData*) [inline], [static]

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

Parameters

base	SPI peripheral base address.
handle	SPI handle pointer.
callback	Callback function.
userData	User data.

48.3.7.23 static status_t SPI_SlaveTransferNonBlocking (SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_transfer_t * *xfer*) [inline], [static]

Note

The API returns immediately after the transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.
If using the SPI with FIFO for the interrupt transfer, the transfer size is the integer times the watermark. Otherwise, the last data may be lost because it cannot generate an interrupt request. Call the functional API to get the last several receive data.

Parameters

base	SPI peripheral base address.
handle	pointer to spi_master_handle_t structure which stores the transfer state
xfer	pointer to spi_xfer_config_t structure

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_SPI_Busy	SPI is not idle, is running another transfer.

48.3.7.24 static status_t SPI_SlaveTransferGetCount (SPI_Type * *base*, spi_slave_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

base	SPI peripheral base address.
handle	Pointer to SPI transfer handle, this should be a static variable.
count	Transferred bytes of SPI slave.

Return values

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

48.3.7.25 static void SPI_SlaveTransferAbort (SPI_Type * *base*, spi_slave_handle_t * *handle*) [inline], [static]

Parameters

base	SPI peripheral base address.
handle	Pointer to SPI transfer handle, this should be a static variable.

48.3.7.26 static void SPI_SlaveTransferHandleIRQ (SPI_Type * *base*, spi_slave_handle_t * *handle*) [inline], [static]

Parameters

base	SPI peripheral base address.
handle	pointer to spi_slave_handle_t structure which stores the transfer state

48.3.8 SPI DMA Driver

48.3.8.1 Overview

This section describes the programming interface of the SPI DMA driver.

Files

- file fsl_spi_dma.h

Data Structures

- struct spi_dma_handle_t
  SPI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(* spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)
  SPI DMA callback called at the end of transfer.

DMA Transactional

- void SPI_MasterTransferCreateHandleDMA (SPI_Type *base, spi_dma_handle_t *handle, spi_dma_callback_t callback, void *userData, dma_handle_t *txHandle, dma_handle_t *rxHandle)
  Initialize the SPI master DMA handle.
- status_t SPI_MasterTransferDMA (SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t *xfer)
  Perform a non-blocking SPI transfer using DMA.
- void SPI_MasterTransferAbortDMA (SPI_Type *base, spi_dma_handle_t *handle)
  Abort a SPI transfer using DMA.
- status_t SPI_MasterTransferGetCountDMA (SPI_Type *base, spi_dma_handle_t *handle, size_t *count)
  Get the transferred bytes for SPI slave DMA.
- static void SPI_SlaveTransferCreateHandleDMA (SPI_Type *base, spi_dma_handle_t *handle, spi_dma_callback_t callback, void *userData, dma_handle_t *txHandle, dma_handle_t *rxHandle)
  Initialize the SPI slave DMA handle.
- static status_t SPI_SlaveTransferDMA (SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t *xfer)
  Perform a non-blocking SPI transfer using DMA.
- static void SPI_SlaveTransferAbortDMA (SPI_Type *base, spi_dma_handle_t *handle)
  Abort a SPI transfer using DMA.
- static status_t SPI_SlaveTransferGetCountDMA (SPI_Type *base, spi_dma_handle_t *handle, size_t *count)
  Get the transferred bytes for SPI slave DMA.

48.3.8.2 Data Structure Documentation

48.3.8.2.1 struct _spi_dma_handle

Data Fields

- bool txInProgress
  Send transfer finished.
- bool rxInProgress
  Receive transfer finished.
- dma_handle_t * txHandle
  DMA handler for SPI send.
- dma_handle_t * rxHandle
  DMA handler for SPI receive.
- uint8_t bytesPerFrame
  Bytes in a frame for SPI tranfer.
- spi_dma_callback_t callback
  Callback for SPI DMA transfer.
- void * userData
  User Data for SPI DMA callback.
- uint32_t state
  Internal state of SPI DMA transfer.
- size_t transferSize
  Bytes need to be transfer.

48.3.8.3 Typedef Documentation

48.3.8.3.1 typedef void(* spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)

48.3.8.4 Function Documentation

48.3.8.4.1 void SPI_MasterTransferCreateHandleDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_dma_callback_t *callback*, void * *userData*, dma_handle_t * *txHandle*, dma_handle_t * *rxHandle*)

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

Parameters

base	SPI peripheral base address.

handle	SPI handle pointer.
callback	User callback function called at the end of a transfer.
userData	User data for callback.
txHandle	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
rxHandle	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

48.3.8.4.2 status_t SPI_MasterTransferDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_transfer_t * *xfer*)

Note

This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.
xfer	Pointer to dma transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_SPI_Busy	SPI is not idle, is running another transfer.

48.3.8.4.3 void SPI_MasterTransferAbortDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*)

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.

48.3.8.4.4 status_t SPI_MasterTransferGetCountDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.
count	Transferred bytes.

Return values

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

48.3.8.4.5 static void SPI_SlaveTransferCreateHandleDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_dma_callback_t *callback*, void * *userData*, dma_handle_t * *txHandle*, dma_handle_t * *rxHandle*) [inline], [static]

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

Parameters

base	SPI peripheral base address.
handle	SPI handle pointer.
callback	User callback function called at the end of a transfer.
userData	User data for callback.
txHandle	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
rxHandle	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

48.3.8.4.6 static status_t SPI_SlaveTransferDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_transfer_t * *xfer*) [inline], [static]

Note

This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.
xfer	Pointer to dma transfer structure.

Return values

kStatus_Success	Successfully start a transfer.
kStatus_InvalidArgument	Input argument is invalid.
kStatus_SPI_Busy	SPI is not idle, is running another transfer.

48.3.8.4.7 static void SPI_SlaveTransferAbortDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*) [inline], [static]

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.

48.3.8.4.8 static status_t SPI_SlaveTransferGetCountDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

base	SPI peripheral base address.
handle	SPI DMA handle pointer.
count	Transferred bytes.

Return values

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

48.4 SPI FreeRTOS driver

48.4.1 Overview

This section describes the programming interface of the SPI FreeRTOS driver.

Files

- file fsl_spi_freertos.h

Data Structures

- struct spi_rtos_handle_t
  SPI FreeRTOS handle. More...

Driver version

- #define FSL_SPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SPI FreeRTOS driver version 2.0.0.

SPI RTOS Operation

- status_t SPI_RTOS_Init (spi_rtos_handle_t *handle, SPI_Type *base, const spi_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes SPI.
- status_t SPI_RTOS_Deinit (spi_rtos_handle_t *handle)
  Deinitializes the SPI.
- status_t SPI_RTOS_Transfer (spi_rtos_handle_t *handle, spi_transfer_t *transfer)
  Performs SPI transfer.

48.4.2 Data Structure Documentation

48.4.2.1 struct spi_rtos_handle_t

SPI RTOS handle.

Data Fields

- SPI_Type * base
  SPI base address.
- spi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.

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

48.4.3 Macro Definition Documentation

48.4.3.1 #define FSL_SPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

48.4.4 Function Documentation

48.4.4.1 status_t SPI_RTOS_Init (spi_rtos_handle_t * *handle*, SPI_Type * *base*, const spi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the SPI module and related RTOS context.

Parameters

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

Returns

status of the operation.

48.4.4.2 status_t SPI_RTOS_Deinit (spi_rtos_handle_t * *handle*)

This function deinitializes the SPI module and related RTOS context.

Parameters

handle	The RTOS SPI handle.

48.4.4.3 status_t SPI_RTOS_Transfer (spi_rtos_handle_t * *handle,* spi_transfer_t * *transfer*)

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

Parameters

handle	The RTOS SPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

48.5 SPI µCOS/II driver

48.5.1 Overview

This section describes the programming interface of the SPI µCOS/II driver.

Files

- file fsl_spi_ucosii.h

Data Structures

- struct spi_rtos_handle_t
  SPI FreeRTOS handle. More...

Driver version

- #define FSL_SPI_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SPI uCOS II driver version 2.0.0.

SPI RTOS Operation

- status_t SPI_RTOS_Init (spi_rtos_handle_t *handle, SPI_Type *base, const spi_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes SPI.
- status_t SPI_RTOS_Deinit (spi_rtos_handle_t *handle)
  Deinitializes the SPI.
- status_t SPI_RTOS_Transfer (spi_rtos_handle_t *handle, spi_transfer_t *transfer)
  Performs SPI transfer.

48.5.2 Data Structure Documentation

48.5.2.1 struct spi_rtos_handle_t

SPI RTOS handle.

Data Fields

- SPI_Type * base
  SPI base address.
- spi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.

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

48.5.3 Macro Definition Documentation

48.5.3.1 #define FSL_SPI_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

48.5.4 Function Documentation

48.5.4.1 status_t SPI_RTOS_Init (spi_rtos_handle_t * *handle*, SPI_Type * *base*, const spi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the SPI module and related RTOS context.

Parameters

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

Returns

status of the operation.

48.5.4.2 status_t SPI_RTOS_Deinit (spi_rtos_handle_t * *handle*)

This function deinitializes the SPI module and related RTOS context.

Parameters

handle	The RTOS SPI handle.

48.5.4.3 status_t SPI_RTOS_Transfer (spi_rtos_handle_t * *handle,* spi_transfer_t * *transfer*)

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

Parameters

handle	The RTOS SPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

48.6 SPI µCOS/III driver

48.6.1 Overview

This section describes the programming interface of the SPI µCOS/III driver.

Files

- file fsl_spi_ucosiii.h

Data Structures

- struct spi_rtos_handle_t
  SPI FreeRTOS handle. More...

Driver version

- #define FSL_SPI_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SPI uCOS III driver version 2.0.0.

SPI RTOS Operation

- status_t SPI_RTOS_Init (spi_rtos_handle_t *handle, SPI_Type *base, const spi_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes SPI.
- status_t SPI_RTOS_Deinit (spi_rtos_handle_t *handle)
  Deinitializes the SPI.
- status_t SPI_RTOS_Transfer (spi_rtos_handle_t *handle, spi_transfer_t *transfer)
  Performs SPI transfer.

48.6.2 Data Structure Documentation

48.6.2.1 struct spi_rtos_handle_t

SPI RTOS handle.

Data Fields

- SPI_Type * base
  SPI base address.
- spi_master_handle_t drv_handle
  Handle of the underlying driver, treated as opaque by the RTOS layer.

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

48.6.3 Macro Definition Documentation

48.6.3.1 #define FSL_SPI_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

48.6.4 Function Documentation

48.6.4.1 status_t SPI_RTOS_Init (spi_rtos_handle_t * *handle*, SPI_Type * *base*, const spi_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the SPI module and related RTOS context.

Parameters

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

Returns

status of the operation.

48.6.4.2 status_t SPI_RTOS_Deinit (spi_rtos_handle_t * *handle*)

This function deinitializes the SPI module and related RTOS context.

Parameters

handle	The RTOS SPI handle.

48.6.4.3 status_t SPI_RTOS_Transfer (spi_rtos_handle_t * *handle*, spi_transfer_t * *transfer*)

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

Parameters

handle	The RTOS SPI handle.
transfer	Structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 49
Smart Card

49.1 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_ncn8025 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.

49.2 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.

49.3 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 NCN8025 Driver
- Smart Card UART Driver
- Smart Card µCOS/II Driver
- Smart Card µCOS/III Driver

Files

- file fsl_smartcard.h

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 miliseconds)
  Time Delay function used to passive waiting using RTOS [ms].

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_SMARTCARD, 3),
  kStatus_SMARTCARD_Timeout = MAKE_STATUS(kStatusGroup_SMARTCARD, 4),
  kStatus_SMARTCARD_Initialized = MAKE_STATUS(kStatusGroup_SMARTCARD, 5),
  kStatus_SMARTCARD_PhyInitialized = MAKE_STATUS(kStatusGroup_SMARTCARD, 6),
  kStatus_SMARTCARD_CardNotActivated = MAKE_STATUS(kStatusGroup_SMARTCARD, 7),
  kStatus_SMARTCARD_InvalidInput = MAKE_STATUS(kStatusGroup_SMARTCARD, 8),
  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, 0))
  Smart card driver version 2.1.0.

49.4 Data Structure Documentation

49.4.1 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_Inverse-Convention for inverse convention.

49.4.1.0.0.1 Field Documentation

49.4.1.0.0.1.1 uint8_t smartcard_card_params_t::modeNegotiable

49.4.2 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.

49.4.3 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.

49.4.4 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.

49.4.4.0.0.2 Field Documentation

49.4.4.0.0.2.1 smartcard_direction_t smartcard_xfer_t::direction

(RX/TX)

49.4.4.0.0.2.2 uint8_t* smartcard_xfer_t::buff

49.4.4.0.0.2.3 size_t smartcard_xfer_t::size

49.4.5 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.

49.4.5.0.0.3 Field Documentation

49.4.5.0.0.3.1 smartcard_direction_t smartcard_context_t::direction

(RX/TX)

49.4.5.0.0.3.2 uint8_t* smartcard_context_t::xBuff

49.4.5.0.0.3.3 volatile size_t smartcard_context_t::xSize

49.4.5.0.0.3.4 volatile bool smartcard_context_t::xIsBusy

49.4.5.0.0.3.5 uint8_t smartcard_context_t::txFifoEntryCount

49.4.5.0.0.3.6 smartcard_interface_callback_t smartcard_context_t::interfaceCallback

49.4.5.0.0.3.7 smartcard_transfer_callback_t smartcard_context_t::transferCallback

49.4.5.0.0.3.8 void* smartcard_context_t::interfaceCallbackParam

49.4.5.0.0.3.9 void* smartcard_context_t::transferCallbackParam

49.4.5.0.0.3.10 smartcard_time_delay_t smartcard_context_t::timeDelay

49.4.5.0.0.3.11 smartcard_reset_type_t smartcard_context_t::resetType

49.5 Macro Definition Documentation

49.5.1 #define FSL_SMARTCARD_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

49.6 Enumeration Type Documentation

49.6.1 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.

49.6.2 enum smartcard_control_t

49.6.3 enum smartcard_direction_t

49.7 Smart Card EMVSIM Driver

49.7.1 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.

Files

- file fsl_smartcard_emvsim.h

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 (180u)
  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)
  Fill in smartcard_card_params structure with default values according 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 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 EMVSIM module as per different user request.
- void SMARTCARD_EMVSIM_IRQHandler (EMVSIM_Type *base, smartcard_context_t *context)
  Handles EMVSIM module interrupts.

49.7.2 Enumeration Type Documentation

49.7.2.1 enum emvsim_gpc_clock_select_t

Enumerator

kEMVSIM_GPCClockDisable disabled
kEMVSIM_GPCCardClock card clock
kEMVSIM_GPCRxClock receive clock
kEMVSIM_GPCTxClock transmit ETU clock

49.7.2.2 enum emvsim_presence_detect_edge_t

Enumerator

kEMVSIM_DetectOnFallingEdge presence detect on falling edge
kEMVSIM_DetectOnRisingEdge presence detect on rising edge

49.7.2.3 enum emvsim_presence_detect_status_t

Enumerator

kEMVSIM_DetectPinIsLow presence detect pin is logic low
kEMVSIM_DetectPinIsHigh presence detect pin is logic high

49.7.3 Function Documentation

49.7.3.1 void SMARTCARD_EMVSIM_GetDefaultConfig (smartcard_card_params_t * *cardParams*)

Fill in smartcard_card_params structure with default values according EMV 4.3 specification.

Parameters

cardParams	The configuration structure of type smartcard_interface_config_t. Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.

49.7.3.2 status_t SMARTCARD_EMVSIM_Init (EMVSIM_Type * *base*, smartcard_context_t * *context*, uint32_t *srcClock_Hz*)

This function Un-gate EMVSIM clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core and initialize driver context.

Parameters

base	The EMVSIM 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.

49.7.3.3 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.

Parameters

base	The EMVSIM module base address.

49.7.3.4 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.

49.7.3.5 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.

49.7.3.6 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 non-blocking (asynchronous) transfer function, the application must get the transfer status to check if 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 smart card transfer structure where are linked buffers and sizes.

Returns

An error code or kStatus_SMARTCARD_Success.

49.7.3.7 status_t SMARTCARD_EMVSIM_Control (EMVSIM_Type * *base*, smartcard_context_t * *context*, smartcard_control_t *control*, uint32_t *param*)

Controls EMVSIM module as per different user request.

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.

49.7.3.8 void SMARTCARD_EMVSIM_IRQHandler (EMVSIM_Type * *base*, smartcard_context_t * *context*)

Handles EMVSIM module interrupts.

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a smart card driver context structure.

49.8 Smart Card FreeRTOS Driver

49.8.1 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_UART_Control(base, context, control, 0)
  Common smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_UART_TransferNonBlocking(base, context, xfer)
  Common smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_UART_Init(base, context, sourceClockHz)
  Common smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_UART_Deinit(base)
  Common smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_UART_GetTransferRemainingBytes(base, context)
  Common smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_UART_GetDefaultConfig(cardParams)
  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 as per different user request.

- 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.

49.8.2 Data Structure Documentation

49.8.2.1 struct rtos_smartcard_context_t

Runtime RTOS Smart card driver context.

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- EventGroupHandle_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.

49.8.2.1.0.4 Field Documentation

49.8.2.1.0.4.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

49.8.3 Macro Definition Documentation

49.8.3.1 #define SMARTCARD_Control(*base, context, control, param*) SMARTCARD_UART_Control(base, context, control, 0)

Common smart card API macro

49.8.4 Function Documentation

49.8.4.1 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. Initialize 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

An zero in Success or error code.

49.8.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

Deactivates also smart card PHY interface, stops smart card clocks. Free all synchronization objects allocated in RTOS smart card context.

Parameters

ctx	The smart card RTOS state.

Returns

An zero in Success or error code.

49.8.4.3 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 function AbortTransfer.

Parameters

ctx	A pointer to the RTOS smart card driver context.
xfer	smart card transfer structure.

Returns

An zero in Success or error code.

49.8.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t * *ctx*)

Task waits on a transfer finish event. Don't initialize the transfer. Instead, wait for transfer callback. Can be used while waiting on initial TS character.

Parameters

ctx	A pointer to the RTOS smart card driver context.

Returns

A zero in Success or error code.

49.8.4.5 int SMARTCARD_RTOS_Control (rtos_smartcard_context_t * *ctx*, smartcard_control_t *control*, uint32_t *param*)

Controls the smart card module as per different user request.

Parameters

ctx	The smart card RTOS context pointer.
control	Control type
param	Integer value of specific to control command.

Returns

An zero in Success or error code.

49.8.4.6 int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t * *ctx*, smartcard_interface_control_t *control*, uint32_t *param*)

Controls Smart card module as per different user request.

Parameters

ctx	The smart card RTOS context pointer.
control	Control type
param	Integer value of specific to control command.

Returns

An zero in Success or error code.

49.8.4.7 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

An zero in Success or error code.

49.8.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The smart card RTOS driver context structure.

Activates the Smart card interface.

Returns

An zero in Success or error code.

49.9 Smart Card PHY EMVSIM Driver

49.9.1 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.

Files

- file fsl_smartcard_phy_emvsim.h

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  SMARTCARD define which specify adjustment number of clock cycles during which ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  SMARTCARD define which specify adjustment number of clock cycles until initial 'TS' character has to be received.

Functions

- void SMARTCARD_PHY_EMVSIM_GetDefaultConfig (smartcard_interface_config_t *config)
  Fill in 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 an SMARTCARD interface for operation.
- void SMARTCARD_PHY_EMVSIM_Deinit (EMVSIM_Type *base, const smartcard_interface_config_t *config)
  De-initializes an SMARTCARD interface.
- 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 SMARTCARD interface IC.

49.9.2 Function Documentation

49.9.2.1 void SMARTCARD_PHY_EMVSIM_GetDefaultConfig (smartcard_interface_config_t * *config*)

Fill in smartcardInterfaceConfig structure with default values.

Parameters

config	The user configuration structure of type smartcard_interface_config_t. Function fill in members: clockToResetDelay = 40000, vcc = kSmartcardVoltageClassB3_3V with default values.

49.9.2.2 status_t SMARTCARD_PHY_EMVSIM_Init (EMVSIM_Type * *base*, const smartcard_interface_config_t * *config*, uint32_t *srcClock_Hz*)

Configures an SMARTCARD interface for operation.

Parameters

base	The SMARTCARD 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	SMARTCARD clock generation module source clock.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

49.9.2.3 void SMARTCARD_PHY_EMVSIM_Deinit (EMVSIM_Type * *base*, const smartcard_interface_config_t * *config*)

Stops SMARTCARD clock and disable VCC.

Parameters

base	Smartcard peripheral module base address.
config	SMARTCARD configuration structure.

49.9.2.4 status_t SMARTCARD_PHY_EMVSIM_Activate (EMVSIM_Type * *base*, smartcard_context_t * *context*, smartcard_reset_type_t *resetType*)

Activates the smart card IC.

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a SMARTCARD 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.

49.9.2.5 status_t SMARTCARD_PHY_EMVSIM_Deactivate (EMVSIM_Type * *base*, smartcard_context_t * *context*)

De-activates the smart card IC.

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a smartcard driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

49.9.2.6 status_t SMARTCARD_PHY_EMVSIM_Control (EMVSIM_Type * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Controls SMARTCARD interface IC.

Parameters

base	The EMVSIM peripheral base address.
context	A pointer to a SMARTCARD 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.

49.10 Smart Card PHY GPIO Driver

49.10.1 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.

Files

- file fsl_smartcard_phy_gpio.h

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  SMARTCARD define which specify adjustment number of clock cycles during which ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  SMARTCARD define which specify adjustment number of clock cycles until initial 'TS' character has to be received.

Functions

- void SMARTCARD_PHY_GPIO_GetDefaultConfig (smartcard_interface_config_t ∗config)
  Fill in config structure with default values.
- status_t SMARTCARD_PHY_GPIO_Init (UART_Type ∗base, smartcard_interface_config_t const ∗config, uint32_t srcClock_Hz)
  Initializes an SMARTCARD interface instance for operation.
- void SMARTCARD_PHY_GPIO_Deinit (UART_Type ∗base, smartcard_interface_config_t ∗config)
  De-initializes an SMARTCARD interface.
- 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 SMARTCARD interface IC.

49.10.2 Function Documentation

49.10.2.1 void SMARTCARD_PHY_GPIO_GetDefaultConfig (smartcard_interface_config_t * *config*)

Controls SMARTCARD interface IC.

Parameters

config	The smartcard 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.

49.10.2.2 status_t SMARTCARD_PHY_GPIO_Init (UART_Type * *base*, smartcard_interface_config_t const * *config*, uint32_t *srcClock_Hz*)

Initializes an SMARTCARD interface instance for operation.

Parameters

base	The SMARTCARD peripheral module 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_GPIO_GetDefaultConfig().
srcClock_Hz	Smartcard clock generation module source clock.

Return values

kStatus_SMARTCARD_Success	or kStatus_SMARTCARD_OtherError in case of error.

49.10.2.3 void SMARTCARD_PHY_GPIO_Deinit (UART_Type * *base*, smartcard_interface_config_t * *config*)

Stops smartcard clock and disable VCC.

Parameters

base	The SMARTCARD peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t.

49.10.2.4 status_t SMARTCARD_PHY_GPIO_Activate (UART_Type * *base*, smartcard_context_t * *context*, smartcard_reset_type_t *resetType*)

Activates the smart card IC.

Parameters

base	The SMARTCARD peripheral module base address.
context	A pointer to a smartcard 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.

49.10.2.5 status_t SMARTCARD_PHY_GPIO_Deactivate (UART_Type * *base*, smartcard_context_t * *context*)

De-activates the smart card IC.

Parameters

base	The SMARTCARD peripheral module base address.
context	A pointer to a smartcard driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

49.10.2.6 status_t SMARTCARD_PHY_GPIO_Control (UART_Type * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

Controls SMARTCARD interface IC.

Parameters

base	The SMARTCARD peripheral module base address.
context	A pointer to a smartcard 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.

49.11 Smart Card PHY NCN8025 Driver

49.11.1 Overview

The Smart Card interface NCN8025 driver handles the external interface chip NCN8025 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.

Files

- file fsl_smartcard_phy_ncn8025.h

Macros

- #define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)
  SMARTCARD define which specify adjustment number of clock cycles during which ATR string has to be received.
- #define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)
  SMARTCARD define which specify adjustment number of clock cycles until initial 'TS' character has to be received.
- #define SMARTCARD_NCN8025_STATUS_PRES (0x01u)
  Masks for NCN8025 status register.
- #define SMARTCARD_NCN8025_STATUS_ACTIVE (0x02u)
  SMARTCARD phy NCN8025 smartcard active status.
- #define SMARTCARD_NCN8025_STATUS_FAULTY (0x04u)
  SMARTCARD phy NCN8025 smartcard faulty status.
- #define SMARTCARD_NCN8025_STATUS_CARD_REMOVED (0x08u)
  SMARTCARD phy NCN8025 smartcard removed status.
- #define SMARTCARD_NCN8025_STATUS_CARD_DEACTIVATED (0x10u)
  SMARTCARD phy NCN8025 smartcard deactivated status.

Functions

- void SMARTCARD_PHY_NCN8025_GetDefaultConfig (smartcard_interface_config_t *config)
  Fill in config structure with default values.
- status_t SMARTCARD_PHY_NCN8025_Init (void *base, smartcard_interface_config_t const *config, uint32_t srcClock_Hz)
  Initializes an SMARTCARD interface instance for operation.
- void SMARTCARD_PHY_NCN8025_Deinit (void *base, smartcard_interface_config_t *config)
  De-initializes an SMARTCARD interface.
- 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 SMARTCARD interface IC.
- void SMARTCARD_PHY_NCN8025_IRQHandler (void *base, smartcard_context_t *context)
  SMARTCARD interface IC IRQ ISR.

49.11.2 Macro Definition Documentation

49.11.2.1 #define SMARTCARD_NCN8025_STATUS_PRES (0x01u)

SMARTCARD phy NCN8025 smartcard present status

49.11.3 Function Documentation

49.11.3.1 void SMARTCARD_PHY_NCN8025_GetDefaultConfig (smartcard_interface_config_t * *config*)

Fill in config structure with default values.

Parameters

config	The smartcard 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.

49.11.3.2 status_t SMARTCARD_PHY_NCN8025_Init (void * *base*, smartcard_interface_config_t const * *config*, uint32_t *srcClock_Hz*)

Initializes an SMARTCARD interface instance for operation.

Parameters

base	The SMARTCARD 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_NCN8025_GetDefaultConfig().
srcClock_Hz	Smartcard clock generation module source clock.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

49.11.3.3 void SMARTCARD_PHY_NCN8025_Deinit (void * *base*, smartcard_interface_config_t * *config*)

Stops smartcard clock and disable VCC.

Parameters

base	The SMARTCARD peripheral module base address.
config	The user configuration structure of type smartcard_interface_config_t.

49.11.3.4 status_t SMARTCARD_PHY_NCN8025_Activate (void * *base*, smartcard_context_t * *context*, smartcard_reset_type_t *resetType*)

Activates the smart card IC.

Parameters

base	The SMARTCARD peripheral module base address.
context	A pointer to a smartcard 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.

49.11.3.5 status_t SMARTCARD_PHY_NCN8025_Deactivate (void * *base*, smartcard_context_t * *context*)

De-activates the smart card IC.

Parameters

base	The SMARTCARD peripheral module base address.

context	A pointer to a smartcard driver context structure.

Return values

kStatus_SMARTCARD_-Success	or kStatus_SMARTCARD_OtherError in case of error.

49.11.3.6 status_t SMARTCARD_PHY_NCN8025_Control (void * *base*, smartcard_context_t * *context*, smartcard_interface_control_t *control*, uint32_t *param*)

De-activates the smart card IC.

Parameters

base	The SMARTCARD peripheral module base address.
context	A pointer to a smartcard 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.

49.11.3.7 void SMARTCARD_PHY_NCN8025_IRQHandler (void * *base*, smartcard_context_t * *context*)

SMARTCARD interface IC IRQ ISR.

Parameters

base	The SMARTCARD peripheral module base address.
context	The smartcard context pointer.

49.12 Smart Card UART Driver

49.12.1 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.

Files

- file fsl_smartcard_uart.h

Macros

- #define SMARTCARD_EMV_RX_NACK_THRESHOLD (5u)
  EMV RX NACK interrupt generation threshold.
- #define SMARTCARD_EMV_TX_NACK_THRESHOLD (3u)
  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)
  Fill in smartcard_card_params structure with default values according EMV 4.3 specification.
- status_t SMARTCARD_UART_Init (UART_Type *base, smartcard_context_t *context, uint32_t srcClock_Hz)
  Initializes an UART peripheral for 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)
  Transfer data using interrupts.
- status_t SMARTCARD_UART_Control (UART_Type *base, smartcard_context_t *context, smartcard_control_t control, uint32_t param)
  Controls UART module as per different user request.
- 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.

49.12.2 Function Documentation

49.12.2.1 void SMARTCARD_UART_GetDefaultConfig (smartcard_card_params_t * *cardParams*)

SMARTCARD interface IC IRQ ISR.

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.

49.12.2.2 status_t SMARTCARD_UART_Init (UART_Type * *base*, smartcard_context_t * *context*, uint32_t *srcClock_Hz*)

This function Un-gate UART clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core and initialize 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.

49.12.2.3 void SMARTCARD_UART_Deinit (UART_Type * *base*)

SMARTCARD interface IC IRQ ISR.

Parameters

base	The UART peripheral base address.

49.12.2.4 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.

49.12.2.5 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.

49.12.2.6 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 are linked buffers and sizes.

Returns

An error code or kStatus_SMARTCARD_Success.

49.12.2.7 status_t SMARTCARD_UART_Control (UART_Type * *base*, smartcard_context_t * *context*, smartcard_control_t *control*, uint32_t *param*)

Controls UART module as per different user request.

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 of specific to control command.

return An kStatus_SMARTCARD_OtherError in case of error return kStatus_SMARTCARD_Success in success

49.12.2.8 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. Smart card driver requires this function to call when UART interrupt occurs.

Parameters

base	The UART peripheral base address.
context	A pointer to a smart card driver context structure.

49.12.2.9 void SMARTCARD_UART_ErrIRQHandler (UART_Type * *base*, smartcard_context_t * *context*)

This function handles error conditions during transfer.

Parameters

base	The UART peripheral base address.
context	A pointer to a smart card driver context structure.

49.12.2.10 void SMARTCARD_UART_TSExpiryCallback (UART_Type * *base*, smartcard_context_t * *context*)

Handles initial TS character timer time-out event.

Parameters

base	The UART peripheral base address.
context	A pointer to a smart card driver context structure.

49.13 Smart Card µCOS/II Driver

49.13.1 Overview

Files

- file fsl_smartcard_ucosii.h

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_UART_Control(base, context, control, 0)
  Common Smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_UART_TransferNonBlocking(base, context, xfer)
  Common Smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_UART_Init(base, context, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_UART_Deinit(base)
  Common Smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_UART_GetTransferRemainingBytes(base, context)
  Common Smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_UART_GetDefaultConfig(cardParams)
  Common Smart card API macro.

Functions

- int SMARTCARD_RTOS_Init (void *base, rtos_smartcard_context_t *ctx, uint32_t sourceClockHz)
  Initializes an 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 transfer is finished.
- int SMARTCARD_RTOS_Control (rtos_smartcard_context_t *ctx, smartcard_control_t control, uint32_t param)
  Controls Smart card module as per different user request.
- 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.

49.13.2 Data Structure Documentation

49.13.2.1 struct rtos_smartcard_context_t

Runtime RTOS Smart card driver context.

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- EventGroupHandle_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.

49.13.2.1.0.5 Field Documentation

49.13.2.1.0.5.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

49.13.3 Macro Definition Documentation

49.13.3.1 #define SMARTCARD_Control(*base, context, control, param*) SMARTCARD_UART_Control(base, context, control, 0)

Common Smart card API macro

49.13.4 Function Documentation

49.13.4.1 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. Initialize 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

An zero in Success or error code.

49.13.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

Deactivates also Smart card PHY interface, stops Smart card clocks. Free all synchronization objects allocated in RTOS Smart card context.

Parameters

ctx	The Smart card RTOS state.

Returns

An zero in Success or error code.

49.13.4.3 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. User can cancel this transfer by calling function AbortTransfer.

Parameters

ctx	A pointer to the RTOS Smart card driver context.
xfer	Smart card transfer structure.

Returns

An zero in Success or error code.

49.13.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t * *ctx*)

Task waits on the transfer finish event. Don't initialize transfer, just wait for transfer callback. Can be used while waiting on initial TS character.

Parameters

ctx	A pointer to the RTOS Smart card driver context.

Returns

An zero in Success or error code.

49.13.4.5 int SMARTCARD_RTOS_Control (rtos_smartcard_context_t * *ctx,* smartcard_control_t *control,* uint32_t *param*)

Controls Smart card module as per different user request.

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value of specific to control command.

Returns

An zero in Success or error code.

49.13.4.6 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.

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value of specific to control command.

Returns

An zero in Success or error code.

49.13.4.7 int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t * *ctx*, smartcard_reset_type_t *resetType*)

Activates the Smart card interface.

Parameters

ctx	The Smart card RTOS driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Returns

An zero in Success or error code.

49.13.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The Smart card RTOS driver context structure.

Deactivates the Smart card interface.

Returns

An zero in Success or error code.

49.14 Smart Card µCOS/III Driver

49.14.1 Overview

Files

- file fsl_smartcard_ucosiii.h

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_UART_Control(base, context, control, 0)
  Common Smart card driver API defines.
- #define SMARTCARD_Transfer(base, context, xfer) SMARTCARD_UART_TransferNonBlocking(base, context, xfer)
  Common Smart card API macro.
- #define SMARTCARD_Init(base, context, sourceClockHz) SMARTCARD_UART_Init(base, context, sourceClockHz)
  Common Smart card API macro.
- #define SMARTCARD_Deinit(base) SMARTCARD_UART_Deinit(base)
  Common Smart card API macro.
- #define SMARTCARD_GetTransferRemainingBytes(base, context) SMARTCARD_UART_GetTransferRemainingBytes(base, context)
  Common Smart card API macro.
- #define SMARTCARD_GetDefaultConfig(cardParams) SMARTCARD_UART_GetDefaultConfig(cardParams)
  Common Smart card API macro.

Functions

- int SMARTCARD_RTOS_Init (void *base, rtos_smartcard_context_t *ctx, uint32_t sourceClockHz)
  Initializes an 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 transfer is finished.
- int SMARTCARD_RTOS_Control (rtos_smartcard_context_t *ctx, smartcard_control_t control, uint32_t param)
  Controls Smart card module as per different user request.
- int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t *ctx, smartcard_interface_control_t control, uint32_t param)
  Controls 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.

49.14.2 Data Structure Documentation

49.14.2.1 struct rtos_smartcard_context_t

Runtime RTOS Smart card driver context.

Data Fields

- SemaphoreHandle_t x_sem
  RTOS unique access assurance object.
- EventGroupHandle_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.

49.14.2.1.0.6 Field Documentation

49.14.2.1.0.6.1 smartcard_context_t rtos_smartcard_context_t::x_context

Transactional layer state.

49.14.3 Macro Definition Documentation

49.14.3.1 #define SMARTCARD_Control(*base, context, control, param*) SMARTCARD_UART_Control(base, context, control, 0)

Common Smart card API macro

49.14.4 Function Documentation

49.14.4.1 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. Initialize 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

An zero in Success or error code.

49.14.4.2 int SMARTCARD_RTOS_Deinit (rtos_smartcard_context_t * *ctx*)

Deactivates also Smart card PHY interface, stops Smart card clocks. Free all synchronization objects allocated in RTOS Smart card context.

Parameters

ctx	The Smart card RTOS state.

Returns

An zero in Success or error code.

49.14.4.3 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 function AbortTransfer.

Parameters

ctx	A pointer to the RTOS Smart card driver context.
xfer	Smart card transfer structure.

Returns

An zero in Success or error code.

49.14.4.4 int SMARTCARD_RTOS_WaitForXevent (rtos_smartcard_context_t * *ctx*)

Task waits on the transfer finish event. Don't initialize transfer, just wait for transfer callback. Can be used while waiting on initial TS character.

Parameters

ctx	A pointer to the RTOS Smart card driver context.

Returns

An zero in Success or error code.

49.14.4.5 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 of specific to control command.

Returns

An zero in Success or error code.

49.14.4.6 int SMARTCARD_RTOS_PHY_Control (rtos_smartcard_context_t * *ctx*, smartcard_interface_control_t *control*, uint32_t *param*)

Controls Smart card module as per different user request.

Parameters

ctx	The Smart card RTOS context pointer.
control	Control type
param	Integer value of specific to control command.

Returns

An zero in Success or error code.

49.14.4.7 int SMARTCARD_RTOS_PHY_Activate (rtos_smartcard_context_t * *ctx*, smartcard_reset_type_t *resetType*)

Activates the Smart card interface.

Parameters

ctx	The Smart card RTOS driver context structure.
resetType	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Returns

An zero in Success or error code.

49.14.4.8 int SMARTCARD_RTOS_PHY_Deactivate (rtos_smartcard_context_t * *ctx*)

Parameters

ctx	The Smart card RTOS driver context structure.

Deactivates the Smart card interface.

Returns

An zero in Success or error code.

Chapter 50
TPM: Timer PWM Module

50.1 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 SoC's, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decode modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoC's.

The function TPM_Init() initializes the TPM with specified configurations. The function TPM_GetDefaultConfig() gets the default configurations. On some SoC's, 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_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 SoC's. 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 SoC's. 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

SoC's. 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.

50.2 Typical use case

50.2.1 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);
    }
}
```

Files

- file fsl_tpm.h

Data Structures

- struct tpm_chnl_pwm_signal_param_t
  Options to configure a TPM channel's PWM signal. 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 }
  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_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_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, 1))
  Version 2.0.1.

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.

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

- uint32_t TPM_GetStatusFlags (TPM_Type *base)
  Gets the TPM status flags.
- 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.

50.3 Data Structure Documentation

50.3.1 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)...

50.3.1.0.0.7 Field Documentation

50.3.1.0.0.7.1 tpm_chnl_t tpm_chnl_pwm_signal_param_t::chnlNumber

In combined mode (available in some SoC's, this represents the channel pair number

50.3.1.0.0.7.2 uint8_t tpm_chnl_pwm_signal_param_t::dutyCyclePercent

100=always active signal (100% duty cycle)

50.3.2 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.
- 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

50.4 Enumeration Type Documentation

50.4.1 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.

50.4.2 enum tpm_pwm_mode_t

Enumerator

kTPM_EdgeAlignedPwm Edge aligned PWM.
kTPM_CenterAlignedPwm Center aligned PWM.

50.4.3 enum tpm_pwm_level_select_t

Enumerator

kTPM_NoPwmSignal No PWM output on pin.
kTPM_LowTrue Low true pulses.
kTPM_HighTrue High true pulses.

50.4.4 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.

50.4.5 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.

50.4.6 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.

50.4.7 enum tpm_clock_source_t

Enumerator

kTPM_SystemClock System clock.
kTPM_ExternalClock External clock.

50.4.8 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.

50.4.9 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.

50.4.10 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.

50.5 Function Documentation

50.5.1 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.

50.5.2 void TPM_Deinit (TPM_Type * *base*)

Parameters

base	TPM peripheral base address

50.5.3 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.

50.5.4 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

50.5.5 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)

50.5.6 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.

50.5.7 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

50.5.8 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.

50.5.9 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

50.5.10 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

50.5.11 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

50.5.12 uint32_t TPM_GetStatusFlags (TPM_Type * *base*)

Parameters

base	TPM peripheral base address

Returns

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

50.5.13 void TPM_ClearStatusFlags (TPM_Type * *base*, uint32_t *mask*)

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

50.5.14 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

50.5.15 static void TPM_StopTimer (TPM_Type * *base*) [inline], [static]

Parameters

base	TPM peripheral base address

Chapter 51
TRNG: True Random Number Generator

51.1 Overview

The Kinetis SDK provides the peripheral driver for the True Random Number Generator (TRNG) module of Kinetis devices.

The True Random Number Generator is hardware accelerator module that generates a 512-bit entropy as needed by an entropy consuming module or by other post processing functions. A typical entropy consumer is a pseudo random number generator (PRNG) which can be implemented to achieve both true randomness and cryptographic strength random numbers using the TRNG output as its entropy seed. The entropy generated by a TRNG is intended for direct use by functions that generate secret keys, per-message secrets, random challenges, and other similar quantities used in cryptographic algorithms.

51.2 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.

51.3 Get random data from TRNG

1. TRNG_GetRandomData() function gets random data from the TRNG module.

This example code shows how to initialize and get random data from the TRNG driver:

```
{
    trng_user_config_t  trngConfig;
    status_t            status;
    uint32_t            data;

    /* Initialize TRNG configuration structure to default.
    TRNG_InitUserConfigDefault(&trngConfig);

    /* Initialize TRNG
    status = TRNG_Init(TRNG0, &trngConfig);

    if (status == kStatus_Success)
    {
        /* Read Random data
        if((status = TRNG_GetRandomData(TRNG0, data, sizeof(data))) == kStatus_TRNG_Success)
        {
            /* Print data
            PRINTF("Random = 0x%X\r\n", i, data );

            PRINTF("Succeed.\r\n");
        }
```

```
        else
        {
            PRINTF("TRNG failed! (0x%x)\r\n", status);
        }

        /* Deinitialize TRNG
        TRNG_Deinit(TRNG0);
    }
    else
    {
        PRINTF("TRNG initialization failed!\r\n");
    }
}
```

Files

- file fsl_trng.h

Data Structures

- struct trng_statistical_check_limit_t
  Data structure for definition of statistical check limits. More...
- struct trng_config_t
  Data structure for the TRNG initialization. More...

Enumerations

- enum trng_sample_mode_t {
  kTRNG_SampleModeVonNeumann = 0U,
  kTRNG_SampleModeRaw = 1U,
  kTRNG_SampleModeVonNeumannRaw }
  TRNG sample mode.
- enum trng_clock_mode_t {
  kTRNG_ClockModeRingOscillator = 0U,
  kTRNG_ClockModeSystem = 1U }
  TRNG clock mode.
- enum trng_ring_osc_div_t {
  kTRNG_RingOscDiv0 = 0U,
  kTRNG_RingOscDiv2 = 1U,
  kTRNG_RingOscDiv4 = 2U,
  kTRNG_RingOscDiv8 = 3U }
  TRNG ring oscillator divide.

Functions

- status_t TRNG_GetDefaultConfig (trng_config_t *userConfig)
  Initializes user configuration structure to default.
- 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, 0))
  TRNG driver version 2.0.0.

51.4 Data Structure Documentation

51.4.1 struct trng_statistical_check_limit_t

Used by trng_config_t.

Data Fields

- uint32_t maximum
  Maximum limit.
- uint32_t minimum
  Minimum limit.

51.4.1.0.0.8 Field Documentation

51.4.1.0.0.8.1 uint32_t trng_statistical_check_limit_t::maximum

51.4.1.0.0.8.2 uint32_t trng_statistical_check_limit_t::minimum

51.4.2 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.

51.4.2.0.0.9 Field Documentation

51.4.2.0.0.9.1 bool trng_config_t::lock

51.4.2.0.0.9.2 trng_clock_mode_t trng_config_t::clockMode

51.4.2.0.0.9.3 trng_ring_osc_div_t trng_config_t::ringOscDiv

51.4.2.0.0.9.4 trng_sample_mode_t trng_config_t::sampleMode

51.4.2.0.0.9.5 uint16_t trng_config_t::entropyDelay

Defines the length (in system clocks) of each Entropy sample taken.

51.4.2.0.0.9.6 uint16_t trng_config_t::sampleSize

Defines the total number of Entropy samples that will be taken during Entropy generation.

51.4.2.0.0.9.7 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.

51.4.2.0.0.9.8 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.

51.4.2.0.0.9.9 uint8_t trng_config_t::longRunMaxLimit

51.4.2.0.0.9.10 trng_statistical_check_limit_t trng_config_t::monobitLimit

51.4.2.0.0.9.11 trng_statistical_check_limit_t trng_config_t::runBit1Limit

51.4.2.0.0.9.12 trng_statistical_check_limit_t trng_config_t::runBit2Limit

51.4.2.0.0.9.13 trng_statistical_check_limit_t trng_config_t::runBit3Limit

51.4.2.0.0.9.14 trng_statistical_check_limit_t trng_config_t::runBit4Limit

51.4.2.0.0.9.15 trng_statistical_check_limit_t trng_config_t::runBit5Limit

51.4.2.0.0.9.16 trng_statistical_check_limit_t trng_config_t::runBit6PlusLimit

51.4.2.0.0.9.17 trng_statistical_check_limit_t trng_config_t::pokerLimit

51.4.2.0.0.9.18 trng_statistical_check_limit_t trng_config_t::frequencyCountLimit

51.5 Macro Definition Documentation

51.5.1 #define FSL_TRNG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

51.6 Enumeration Type Documentation

51.6.1 enum trng_sample_mode_t

Used by trng_config_t.

Enumerator

kTRNG_SampleModeVonNeumann Use Von Neumann data into both Entropy shifter and Statistical Checker.

kTRNG_SampleModeRaw Use raw data into both Entropy shifter and Statistical Checker.

kTRNG_SampleModeVonNeumannRaw Use Von Neumann data into Entropy shifter. Use raw data into Statistical Checker.

51.6.2 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.

51.6.3 enum trng_ring_osc_div_t

Used by trng_config_t.

Enumerator

kTRNG_RingOscDiv0 Ring oscillator with no divide (default).
kTRNG_RingOscDiv2 Ring oscillator divided-by-2.
kTRNG_RingOscDiv4 Ring oscillator divided-by-4.
kTRNG_RingOscDiv8 Ring oscillator divided-by-8.

51.7 Function Documentation

51.7.1 status_t TRNG_GetDefaultConfig (trng_config_t * *userConfig*)

This function initializes the configure structure to default value. the default value are:

```
user_config->lock = 0;
user_config->clockMode = kTRNG_ClockModeRingOscillator;
user_config->ringOscDiv = kTRNG_RingOscDiv0;  Or  to other kTRNG_RingOscDiv[2|8] depending
       on 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.

51.7.2 status_t TRNG_Init (TRNG_Type * *base,* const trng_config_t * *userConfig*)

This function initializes the TRNG. When called, the TRNG entropy generation starts immediately.

Parameters

base	TRNG base address
userConfig	Pointer to initialize configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

51.7.3 void TRNG_Deinit (TRNG_Type * *base*)

This function shuts down the TRNG.

Parameters

base	TRNG base address

51.7.4 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 52
TSI: Touch Sensing Input

52.1 Overview

Modules

- TSIv2 Driver
- TSIv4 Driver

52.2 TSIv2 Driver

52.2.1 Overview

The KSDK provides a driver for the Touch Sensing Input (TSI) module of Kinetis devices.

Typical use case

```
TSI_Init(TSI0);
TSI_Configure(TSI0, &user_config);
TSI_EnableChannel(TSI0, channelMask);
TSI_EnableInterrupts(TSI0, kTSI_GlobalInterruptEnable |
      kTSI_EndOfScanInterruptEnable);

TSI_EnablePeriodicalScan(TSI0);
TSI_EnableModule(TSI0);
while(1);
```

Files

- file fsl_tsi_v2.h

Data Structures

- struct tsi_calibration_data_t
  TSI calibration data storage. *More...*
- struct tsi_config_t
  TSI configuration structure. *More...*

Enumerations

- enum tsi_n_consecutive_scans_t {
  kTSI_ConsecutiveScansNumber_1time = 0U,
  kTSI_ConsecutiveScansNumber_2time = 1U,
  kTSI_ConsecutiveScansNumber_3time = 2U,
  kTSI_ConsecutiveScansNumber_4time = 3U,
  kTSI_ConsecutiveScansNumber_5time = 4U,
  kTSI_ConsecutiveScansNumber_6time = 5U,
  kTSI_ConsecutiveScansNumber_7time = 6U,
  kTSI_ConsecutiveScansNumber_8time = 7U,
  kTSI_ConsecutiveScansNumber_9time = 8U,
  kTSI_ConsecutiveScansNumber_10time = 9U,
  kTSI_ConsecutiveScansNumber_11time = 10U,
  kTSI_ConsecutiveScansNumber_12time = 11U,
  kTSI_ConsecutiveScansNumber_13time = 12U,
  kTSI_ConsecutiveScansNumber_14time = 13U,
  kTSI_ConsecutiveScansNumber_15time = 14U,
  kTSI_ConsecutiveScansNumber_16time = 15U,
  kTSI_ConsecutiveScansNumber_17time = 16U,
  kTSI_ConsecutiveScansNumber_18time = 17U,
  kTSI_ConsecutiveScansNumber_19time = 18U,
  kTSI_ConsecutiveScansNumber_20time = 19U,
  kTSI_ConsecutiveScansNumber_21time = 20U,
  kTSI_ConsecutiveScansNumber_22time = 21U,
  kTSI_ConsecutiveScansNumber_23time = 22U,
  kTSI_ConsecutiveScansNumber_24time = 23U,
  kTSI_ConsecutiveScansNumber_25time = 24U,
  kTSI_ConsecutiveScansNumber_26time = 25U,
  kTSI_ConsecutiveScansNumber_27time = 26U,
  kTSI_ConsecutiveScansNumber_28time = 27U,
  kTSI_ConsecutiveScansNumber_29time = 28U,
  kTSI_ConsecutiveScansNumber_30time = 29U,
  kTSI_ConsecutiveScansNumber_31time = 30U,
  kTSI_ConsecutiveScansNumber_32time = 31U,
  kTSI_ConsecutiveScansNumber_1time = 0U,
  kTSI_ConsecutiveScansNumber_2time = 1U,
  kTSI_ConsecutiveScansNumber_3time = 2U,
  kTSI_ConsecutiveScansNumber_4time = 3U,
  kTSI_ConsecutiveScansNumber_5time = 4U,
  kTSI_ConsecutiveScansNumber_6time = 5U,
  kTSI_ConsecutiveScansNumber_7time = 6U,
  kTSI_ConsecutiveScansNumber_8time = 7U,
  kTSI_ConsecutiveScansNumber_9time = 8U,
  kTSI_ConsecutiveScansNumber_10time = 9U,
  kTSI_ConsecutiveScansNumber_11time = 10U,
  kTSI_ConsecutiveScansNumber_12time = 11U,
  kTSI_ConsecutiveScansNumber_13time = 12U,
  kTSI_ConsecutiveScansNumber_14time = 13U,

kTSI_ConsecutiveScansNumber_32time = 31U }
TSI number of scan intervals for each electrode.

- enum tsi_electrode_osc_prescaler_t {
kTSI_ElecOscPrescaler_1div = 0U,
kTSI_ElecOscPrescaler_2div = 1U,
kTSI_ElecOscPrescaler_4div = 2U,
kTSI_ElecOscPrescaler_8div = 3U,
kTSI_ElecOscPrescaler_16div = 4U,
kTSI_ElecOscPrescaler_32div = 5U,
kTSI_ElecOscPrescaler_64div = 6U,
kTSI_ElecOscPrescaler_128div = 7U,
kTSI_ElecOscPrescaler_1div = 0U,
kTSI_ElecOscPrescaler_2div = 1U,
kTSI_ElecOscPrescaler_4div = 2U,
kTSI_ElecOscPrescaler_8div = 3U,
kTSI_ElecOscPrescaler_16div = 4U,
kTSI_ElecOscPrescaler_32div = 5U,
kTSI_ElecOscPrescaler_64div = 6U,
kTSI_ElecOscPrescaler_128div = 7U }
TSI electrode oscillator prescaler.
- enum tsi_low_power_clock_source_t {
kTSI_LowPowerClockSource_LPOCLK = 0U,
kTSI_LowPowerClockSource_VLPOSCCLK = 1U }
TSI low power mode clock source.
- enum tsi_low_power_scan_interval_t {
kTSI_LowPowerInterval_1ms = 0U,
kTSI_LowPowerInterval_5ms = 1U,
kTSI_LowPowerInterval_10ms = 2U,
kTSI_LowPowerInterval_15ms = 3U,
kTSI_LowPowerInterval_20ms = 4U,
kTSI_LowPowerInterval_30ms = 5U,
kTSI_LowPowerInterval_40ms = 6U,
kTSI_LowPowerInterval_50ms = 7U,
kTSI_LowPowerInterval_75ms = 8U,
kTSI_LowPowerInterval_100ms = 9U,
kTSI_LowPowerInterval_125ms = 10U,
kTSI_LowPowerInterval_150ms = 11U,
kTSI_LowPowerInterval_200ms = 12U,
kTSI_LowPowerInterval_300ms = 13U,
kTSI_LowPowerInterval_400ms = 14U,
kTSI_LowPowerInterval_500ms = 15U }
TSI low power scan intervals.
- enum tsi_reference_osc_charge_current_t {

kTSI_RefOscChargeCurrent_2uA = 0U,
kTSI_RefOscChargeCurrent_4uA = 1U,
kTSI_RefOscChargeCurrent_6uA = 2U,
kTSI_RefOscChargeCurrent_8uA = 3U,
kTSI_RefOscChargeCurrent_10uA = 4U,
kTSI_RefOscChargeCurrent_12uA = 5U,
kTSI_RefOscChargeCurrent_14uA = 6U,
kTSI_RefOscChargeCurrent_16uA = 7U,
kTSI_RefOscChargeCurrent_18uA = 8U,
kTSI_RefOscChargeCurrent_20uA = 9U,
kTSI_RefOscChargeCurrent_22uA = 10U,
kTSI_RefOscChargeCurrent_24uA = 11U,
kTSI_RefOscChargeCurrent_26uA = 12U,
kTSI_RefOscChargeCurrent_28uA = 13U,
kTSI_RefOscChargeCurrent_30uA = 14U,
kTSI_RefOscChargeCurrent_32uA = 15U,
kTSI_RefOscChargeCurrent_500nA = 0U,
kTSI_RefOscChargeCurrent_1uA = 1U,
kTSI_RefOscChargeCurrent_2uA = 2U,
kTSI_RefOscChargeCurrent_4uA = 3U,
kTSI_RefOscChargeCurrent_8uA = 4U,
kTSI_RefOscChargeCurrent_16uA = 5U,
kTSI_RefOscChargeCurrent_32uA = 6U,
kTSI_RefOscChargeCurrent_64uA = 7U }

TSI Reference oscillator charge current select.

- enum tsi_external_osc_charge_current_t {

kTSI_ExtOscChargeCurrent_2uA = 0U,
kTSI_ExtOscChargeCurrent_4uA = 1U,
kTSI_ExtOscChargeCurrent_6uA = 2U,
kTSI_ExtOscChargeCurrent_8uA = 3U,
kTSI_ExtOscChargeCurrent_10uA = 4U,
kTSI_ExtOscChargeCurrent_12uA = 5U,
kTSI_ExtOscChargeCurrent_14uA = 6U,
kTSI_ExtOscChargeCurrent_16uA = 7U,
kTSI_ExtOscChargeCurrent_18uA = 8U,
kTSI_ExtOscChargeCurrent_20uA = 9U,
kTSI_ExtOscChargeCurrent_22uA = 10U,
kTSI_ExtOscChargeCurrent_24uA = 11U,
kTSI_ExtOscChargeCurrent_26uA = 12U,
kTSI_ExtOscChargeCurrent_28uA = 13U,
kTSI_ExtOscChargeCurrent_30uA = 14U,
kTSI_ExtOscChargeCurrent_32uA = 15U,
kTSI_ExtOscChargeCurrent_500nA = 0U,
kTSI_ExtOscChargeCurrent_1uA = 1U,
kTSI_ExtOscChargeCurrent_2uA = 2U,
kTSI_ExtOscChargeCurrent_4uA = 3U,
kTSI_ExtOscChargeCurrent_8uA = 4U,
kTSI_ExtOscChargeCurrent_16uA = 5U,
kTSI_ExtOscChargeCurrent_32uA = 6U,
kTSI_ExtOscChargeCurrent_64uA = 7U }
TSI External oscillator charge current select.

- enum tsi_active_mode_clock_source_t {
  kTSI_ActiveClkSource_LPOSCCLK = 0U,
  kTSI_ActiveClkSource_MCGIRCLK = 1U,
  kTSI_ActiveClkSource_OSCERCLK = 2U }
  TSI Active mode clock source.
- enum tsi_active_mode_prescaler_t {
  kTSI_ActiveModePrescaler_1div = 0U,
  kTSI_ActiveModePrescaler_2div = 1U,
  kTSI_ActiveModePrescaler_4div = 2U,
  kTSI_ActiveModePrescaler_8div = 3U,
  kTSI_ActiveModePrescaler_16div = 4U,
  kTSI_ActiveModePrescaler_32div = 5U,
  kTSI_ActiveModePrescaler_64div = 6U,
  kTSI_ActiveModePrescaler_128div = 7U }
  TSI active mode prescaler.
- enum tsi_status_flags_t {

kTSI_EndOfScanFlag = TSI_GENCS_EOSF_MASK,
kTSI_OutOfRangeFlag = TSI_GENCS_OUTRGF_MASK,
kTSI_ExternalElectrodeErrorFlag = TSI_GENCS_EXTERF_MASK,
kTSI_OverrunErrorFlag = TSI_GENCS_OVRF_MASK,
kTSI_EndOfScanFlag = TSI_GENCS_EOSF_MASK,
kTSI_OutOfRangeFlag = TSI_GENCS_OUTRGF_MASK }
TSI status flags.

- enum tsi_interrupt_enable_t {
kTSI_GlobalInterruptEnable = 1U,
kTSI_OutOfRangeInterruptEnable = 2U,
kTSI_EndOfScanInterruptEnable = 4U,
kTSI_ErrorInterrruptEnable = 8U,
kTSI_GlobalInterruptEnable = 1U,
kTSI_OutOfRangeInterruptEnable = 2U,
kTSI_EndOfScanInterruptEnable = 4U }
TSI feature interrupt source.

Functions

- void TSI_Init (TSI_Type *base, const tsi_config_t *config)
Initializes hardware.
- void TSI_Deinit (TSI_Type *base)
De-initializes hardware.
- void TSI_GetNormalModeDefaultConfig (tsi_config_t *userConfig)
Gets TSI normal mode user configuration structure.
- void TSI_GetLowPowerModeDefaultConfig (tsi_config_t *userConfig)
Gets the TSI low power mode default user configuration structure.
- void TSI_Calibrate (TSI_Type *base, tsi_calibration_data_t *calBuff)
Hardware calibration.
- void TSI_EnableInterrupts (TSI_Type *base, uint32_t mask)
Enables the TSI interrupt requests.
- void TSI_DisableInterrupts (TSI_Type *base, uint32_t mask)
Disables the TSI interrupt requests.
- static uint32_t TSI_GetStatusFlags (TSI_Type *base)
Gets the interrupt flags.
- void TSI_ClearStatusFlags (TSI_Type *base, uint32_t mask)
Clears the interrupt flags.
- static uint32_t TSI_GetScanTriggerMode (TSI_Type *base)
Gets the TSI scan trigger mode.
- static bool TSI_IsScanInProgress (TSI_Type *base)
Gets the scan in progress flag.
- static void TSI_SetElectrodeOSCPrescaler (TSI_Type *base, tsi_electrode_osc_prescaler_-t prescaler)
Sets the electrode oscillator prescaler.
- static void TSI_SetNumberOfScans (TSI_Type *base, tsi_n_consecutive_scans_t number)
Sets the number of scans (NSCN).
- static void TSI_EnableModule (TSI_Type *base, bool enable)
Enables/disables the TSI module.

- static void TSI_EnableLowPower (TSI_Type *base, bool enable)
  Enables/disables the TSI module in low power stop mode.
- static void TSI_EnablePeriodicalScan (TSI_Type *base, bool enable)
  Enables/disables the periodical (hardware) trigger scan.
- static void TSI_StartSoftwareTrigger (TSI_Type *base)
  Starts a measurement (trigger a new measurement).
- static void TSI_SetLowPowerScanInterval (TSI_Type *base, tsi_low_power_scan_interval_t interval)
  Sets a low power scan interval.
- static void TSI_SetLowPowerClock (TSI_Type *base, uint32_t clock)
  Sets a low power clock.
- static void TSI_SetReferenceChargeCurrent (TSI_Type *base, tsi_reference_osc_charge_current_t current)
  Sets the reference oscillator charge current.
- static void TSI_SetElectrodeChargeCurrent (TSI_Type *base, tsi_external_osc_charge_current_t current)
  Sets the electrode charge current.
- static void TSI_SetScanModulo (TSI_Type *base, uint32_t modulo)
  Sets the scan modulo value.
- static void TSI_SetActiveModeSource (TSI_Type *base, uint32_t source)
  Sets the active mode source.
- static void TSI_SetActiveModePrescaler (TSI_Type *base, tsi_active_mode_prescaler_t prescaler)
  Sets the active mode prescaler.
- static void TSI_SetLowPowerChannel (TSI_Type *base, uint16_t channel)
  Sets the low power channel.
- static uint32_t TSI_GetLowPowerChannel (TSI_Type *base)
  Gets the enabled channel in low power modes.
- static void TSI_EnableChannel (TSI_Type *base, uint16_t channel, bool enable)
  Enables/disables a channel.
- static void TSI_EnableChannels (TSI_Type *base, uint16_t channelsMask, bool enable)
  Enables/disables channels.
- static bool TSI_IsChannelEnabled (TSI_Type *base, uint16_t channel)
  Returns if a channel is enabled.
- static uint16_t TSI_GetEnabledChannels (TSI_Type *base)
  Returns the mask of enabled channels.
- static uint16_t TSI_GetWakeUpChannelCounter (TSI_Type *base)
  Gets the wake up channel counter for low-power mode usage.
- static uint16_t TSI_GetNormalModeCounter (TSI_Type *base, uint16_t channel)
  Gets the TSI conversion counter of a specific channel in normal mode.
- static void TSI_SetLowThreshold (TSI_Type *base, uint16_t low_threshold)
  Sets a low threshold.
- static void TSI_SetHighThreshold (TSI_Type *base, uint16_t high_threshold)
  Sets a high threshold.

Driver version

- #define FSL_TSI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  TSI driver version 2.0.0.

52.2.2 Data Structure Documentation

52.2.2.1 struct tsi_calibration_data_t

Data Fields

- uint16_t calibratedData [FSL_FEATURE_TSI_CHANNEL_COUNT]
  TSI calibration data storage buffer.

52.2.2.2 struct tsi_config_t

This structure contains the settings for the most common TSI configurations including the TSI module charge currents, number of scans, thresholds, and so on.

Data Fields

- uint16_t thresh
  High threshold.
- uint16_t thresl
  Low threshold.
- tsi_low_power_clock_source_t lpclks
  Low power clock.
- tsi_low_power_scan_interval_t lpscnitv
  Low power scan interval.
- tsi_active_mode_clock_source_t amclks
  Active mode clock source.
- tsi_active_mode_prescaler_t ampsc
  Active mode prescaler.
- tsi_electrode_osc_prescaler_t ps
  Electrode Oscillator Prescaler.
- tsi_external_osc_charge_current_t extchrg
  External Oscillator charge current.
- tsi_reference_osc_charge_current_t refchrg
  Reference Oscillator charge current.
- tsi_n_consecutive_scans_t nscn
  Number of scans.
- tsi_electrode_osc_prescaler_t prescaler
  Prescaler.
- tsi_analog_mode_t mode
  TSI mode of operation.
- tsi_osc_voltage_rails_t dvolt
  Oscillator's voltage rails.
- tsi_series_resistor_t resistor
  Series resistance value.
- tsi_filter_bits_t filter
  Noise mode filter bits.

52.2.2.2.0.10 Field Documentation

52.2.2.2.0.10.1 uint16_t tsi_config_t::thresh

52.2.2.2.0.10.2 uint16_t tsi_config_t::thresl

52.2.2.2.0.10.3 tsi_low_power_clock_source_t tsi_config_t::lpclks

52.2.2.2.0.10.4 tsi_low_power_scan_interval_t tsi_config_t::lpscnitv

52.2.2.2.0.10.5 tsi_active_mode_clock_source_t tsi_config_t::amclks

52.2.2.2.0.10.6 tsi_active_mode_prescaler_t tsi_config_t::ampsc

52.2.2.2.0.10.7 tsi_external_osc_charge_current_t tsi_config_t::extchrg

Electrode charge current.

52.2.2.2.0.10.8 tsi_reference_osc_charge_current_t tsi_config_t::refchrg

Reference charge current.

52.2.2.2.0.10.9 tsi_n_consecutive_scans_t tsi_config_t::nscn

52.2.2.2.0.10.10 tsi_analog_mode_t tsi_config_t::mode

52.2.2.2.0.10.11 tsi_osc_voltage_rails_t tsi_config_t::dvolt

52.2.3 Macro Definition Documentation

52.2.3.1 #define FSL_TSI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

52.2.4 Enumeration Type Documentation

52.2.4.1 enum tsi_n_consecutive_scans_t

These constants define the TSI number of consecutive scans in a TSI instance for each electrode.

Enumerator

kTSI_ConsecutiveScansNumber_1time	Once per electrode.
kTSI_ConsecutiveScansNumber_2time	Twice per electrode.
kTSI_ConsecutiveScansNumber_3time	3 times consecutive scan
kTSI_ConsecutiveScansNumber_4time	4 times consecutive scan
kTSI_ConsecutiveScansNumber_5time	5 times consecutive scan
kTSI_ConsecutiveScansNumber_6time	6 times consecutive scan
kTSI_ConsecutiveScansNumber_7time	7 times consecutive scan
kTSI_ConsecutiveScansNumber_8time	8 times consecutive scan
kTSI_ConsecutiveScansNumber_9time	9 times consecutive scan

kTSI_ConsecutiveScansNumber_10time 10 times consecutive scan
kTSI_ConsecutiveScansNumber_11time 11 times consecutive scan
kTSI_ConsecutiveScansNumber_12time 12 times consecutive scan
kTSI_ConsecutiveScansNumber_13time 13 times consecutive scan
kTSI_ConsecutiveScansNumber_14time 14 times consecutive scan
kTSI_ConsecutiveScansNumber_15time 15 times consecutive scan
kTSI_ConsecutiveScansNumber_16time 16 times consecutive scan
kTSI_ConsecutiveScansNumber_17time 17 times consecutive scan
kTSI_ConsecutiveScansNumber_18time 18 times consecutive scan
kTSI_ConsecutiveScansNumber_19time 19 times consecutive scan
kTSI_ConsecutiveScansNumber_20time 20 times consecutive scan
kTSI_ConsecutiveScansNumber_21time 21 times consecutive scan
kTSI_ConsecutiveScansNumber_22time 22 times consecutive scan
kTSI_ConsecutiveScansNumber_23time 23 times consecutive scan
kTSI_ConsecutiveScansNumber_24time 24 times consecutive scan
kTSI_ConsecutiveScansNumber_25time 25 times consecutive scan
kTSI_ConsecutiveScansNumber_26time 26 times consecutive scan
kTSI_ConsecutiveScansNumber_27time 27 times consecutive scan
kTSI_ConsecutiveScansNumber_28time 28 times consecutive scan
kTSI_ConsecutiveScansNumber_29time 29 times consecutive scan
kTSI_ConsecutiveScansNumber_30time 30 times consecutive scan
kTSI_ConsecutiveScansNumber_31time 31 times consecutive scan
kTSI_ConsecutiveScansNumber_32time 32 times consecutive scan
kTSI_ConsecutiveScansNumber_1time Once per electrode.
kTSI_ConsecutiveScansNumber_2time Twice per electrode.
kTSI_ConsecutiveScansNumber_3time 3 times consecutive scan
kTSI_ConsecutiveScansNumber_4time 4 times consecutive scan
kTSI_ConsecutiveScansNumber_5time 5 times consecutive scan
kTSI_ConsecutiveScansNumber_6time 6 times consecutive scan
kTSI_ConsecutiveScansNumber_7time 7 times consecutive scan
kTSI_ConsecutiveScansNumber_8time 8 times consecutive scan
kTSI_ConsecutiveScansNumber_9time 9 times consecutive scan
kTSI_ConsecutiveScansNumber_10time 10 times consecutive scan
kTSI_ConsecutiveScansNumber_11time 11 times consecutive scan
kTSI_ConsecutiveScansNumber_12time 12 times consecutive scan
kTSI_ConsecutiveScansNumber_13time 13 times consecutive scan
kTSI_ConsecutiveScansNumber_14time 14 times consecutive scan
kTSI_ConsecutiveScansNumber_15time 15 times consecutive scan
kTSI_ConsecutiveScansNumber_16time 16 times consecutive scan
kTSI_ConsecutiveScansNumber_17time 17 times consecutive scan
kTSI_ConsecutiveScansNumber_18time 18 times consecutive scan
kTSI_ConsecutiveScansNumber_19time 19 times consecutive scan
kTSI_ConsecutiveScansNumber_20time 20 times consecutive scan
kTSI_ConsecutiveScansNumber_21time 21 times consecutive scan
kTSI_ConsecutiveScansNumber_22time 22 times consecutive scan

kTSI_ConsecutiveScansNumber_23time 23 times consecutive scan
kTSI_ConsecutiveScansNumber_24time 24 times consecutive scan
kTSI_ConsecutiveScansNumber_25time 25 times consecutive scan
kTSI_ConsecutiveScansNumber_26time 26 times consecutive scan
kTSI_ConsecutiveScansNumber_27time 27 times consecutive scan
kTSI_ConsecutiveScansNumber_28time 28 times consecutive scan
kTSI_ConsecutiveScansNumber_29time 29 times consecutive scan
kTSI_ConsecutiveScansNumber_30time 30 times consecutive scan
kTSI_ConsecutiveScansNumber_31time 31 times consecutive scan
kTSI_ConsecutiveScansNumber_32time 32 times consecutive scan

52.2.4.2 enum tsi_electrode_osc_prescaler_t

These constants define the TSI electrode oscillator prescaler in a TSI instance.

Enumerator

kTSI_ElecOscPrescaler_1div Electrode oscillator frequency divided by 1.
kTSI_ElecOscPrescaler_2div Electrode oscillator frequency divided by 2.
kTSI_ElecOscPrescaler_4div Electrode oscillator frequency divided by 4.
kTSI_ElecOscPrescaler_8div Electrode oscillator frequency divided by 8.
kTSI_ElecOscPrescaler_16div Electrode oscillator frequency divided by 16.
kTSI_ElecOscPrescaler_32div Electrode oscillator frequency divided by 32.
kTSI_ElecOscPrescaler_64div Electrode oscillator frequency divided by 64.
kTSI_ElecOscPrescaler_128div Electrode oscillator frequency divided by 128.
kTSI_ElecOscPrescaler_1div Electrode oscillator frequency divided by 1.
kTSI_ElecOscPrescaler_2div Electrode oscillator frequency divided by 2.
kTSI_ElecOscPrescaler_4div Electrode oscillator frequency divided by 4.
kTSI_ElecOscPrescaler_8div Electrode oscillator frequency divided by 8.
kTSI_ElecOscPrescaler_16div Electrode oscillator frequency divided by 16.
kTSI_ElecOscPrescaler_32div Electrode oscillator frequency divided by 32.
kTSI_ElecOscPrescaler_64div Electrode oscillator frequency divided by 64.
kTSI_ElecOscPrescaler_128div Electrode oscillator frequency divided by 128.

52.2.4.3 enum tsi_low_power_clock_source_t

Enumerator

kTSI_LowPowerClockSource_LPOCLK LPOCLK is selected.
kTSI_LowPowerClockSource_VLPOSCCLK VLPOSCCLK is selected.

52.2.4.4 enum tsi_low_power_scan_interval_t

These constants define the TSI low power scan intervals in a TSI instance.

Enumerator

kTSI_LowPowerInterval_1ms 1 ms scan interval
kTSI_LowPowerInterval_5ms 5 ms scan interval
kTSI_LowPowerInterval_10ms 10 ms scan interval
kTSI_LowPowerInterval_15ms 15 ms scan interval
kTSI_LowPowerInterval_20ms 20 ms scan interval
kTSI_LowPowerInterval_30ms 30 ms scan interval
kTSI_LowPowerInterval_40ms 40 ms scan interval
kTSI_LowPowerInterval_50ms 50 ms scan interval
kTSI_LowPowerInterval_75ms 75 ms scan interval
kTSI_LowPowerInterval_100ms 100 ms scan interval
kTSI_LowPowerInterval_125ms 125 ms scan interval
kTSI_LowPowerInterval_150ms 150 ms scan interval
kTSI_LowPowerInterval_200ms 200 ms scan interval
kTSI_LowPowerInterval_300ms 300 ms scan interval
kTSI_LowPowerInterval_400ms 400 ms scan interval
kTSI_LowPowerInterval_500ms 500 ms scan interval

52.2.4.5 enum tsi_reference_osc_charge_current_t

These constants define the TSI Reference oscillator charge current select in a TSI instance.

Enumerator

kTSI_RefOscChargeCurrent_2uA Reference oscillator charge current is 2 µA.
kTSI_RefOscChargeCurrent_4uA Reference oscillator charge current is 4 µA.
kTSI_RefOscChargeCurrent_6uA Reference oscillator charge current is 6 µA.
kTSI_RefOscChargeCurrent_8uA Reference oscillator charge current is 8 µA.
kTSI_RefOscChargeCurrent_10uA Reference oscillator charge current is 10 µA.
kTSI_RefOscChargeCurrent_12uA Reference oscillator charge current is 12 µA.
kTSI_RefOscChargeCurrent_14uA Reference oscillator charge current is 14 µA.
kTSI_RefOscChargeCurrent_16uA Reference oscillator charge current is 16 µA.
kTSI_RefOscChargeCurrent_18uA Reference oscillator charge current is 18 µA.
kTSI_RefOscChargeCurrent_20uA Reference oscillator charge current is 20 µA.
kTSI_RefOscChargeCurrent_22uA Reference oscillator charge current is 22 µA.
kTSI_RefOscChargeCurrent_24uA Reference oscillator charge current is 24 µA.
kTSI_RefOscChargeCurrent_26uA Reference oscillator charge current is 26 µA.
kTSI_RefOscChargeCurrent_28uA Reference oscillator charge current is 28 µA.
kTSI_RefOscChargeCurrent_30uA Reference oscillator charge current is 30 µA.
kTSI_RefOscChargeCurrent_32uA Reference oscillator charge current is 32 µA.

kTSI_RefOscChargeCurrent_500nA Reference oscillator charge current is 500 µA.
kTSI_RefOscChargeCurrent_1uA Reference oscillator charge current is 1 µA.
kTSI_RefOscChargeCurrent_2uA Reference oscillator charge current is 2 µA.
kTSI_RefOscChargeCurrent_4uA Reference oscillator charge current is 4 µA.
kTSI_RefOscChargeCurrent_8uA Reference oscillator charge current is 8 µA.
kTSI_RefOscChargeCurrent_16uA Reference oscillator charge current is 16 µA.
kTSI_RefOscChargeCurrent_32uA Reference oscillator charge current is 32 µA.
kTSI_RefOscChargeCurrent_64uA Reference oscillator charge current is 64 µA.

52.2.4.6 enum tsi_external_osc_charge_current_t

These constants define the TSI External oscillator charge current select in a TSI instance.

Enumerator

kTSI_ExtOscChargeCurrent_2uA External oscillator charge current is 2 µA.
kTSI_ExtOscChargeCurrent_4uA External oscillator charge current is 4 µA.
kTSI_ExtOscChargeCurrent_6uA External oscillator charge current is 6 µA.
kTSI_ExtOscChargeCurrent_8uA External oscillator charge current is 8 µA.
kTSI_ExtOscChargeCurrent_10uA External oscillator charge current is 10 µA.
kTSI_ExtOscChargeCurrent_12uA External oscillator charge current is 12 µA.
kTSI_ExtOscChargeCurrent_14uA External oscillator charge current is 14 µA.
kTSI_ExtOscChargeCurrent_16uA External oscillator charge current is 16 µA.
kTSI_ExtOscChargeCurrent_18uA External oscillator charge current is 18 µA.
kTSI_ExtOscChargeCurrent_20uA External oscillator charge current is 20 µA.
kTSI_ExtOscChargeCurrent_22uA External oscillator charge current is 22 µA.
kTSI_ExtOscChargeCurrent_24uA External oscillator charge current is 24 µA.
kTSI_ExtOscChargeCurrent_26uA External oscillator charge current is 26 µA.
kTSI_ExtOscChargeCurrent_28uA External oscillator charge current is 28 µA.
kTSI_ExtOscChargeCurrent_30uA External oscillator charge current is 30 µA.
kTSI_ExtOscChargeCurrent_32uA External oscillator charge current is 32 µA.
kTSI_ExtOscChargeCurrent_500nA External oscillator charge current is 500 µA.
kTSI_ExtOscChargeCurrent_1uA External oscillator charge current is 1 µA.
kTSI_ExtOscChargeCurrent_2uA External oscillator charge current is 2 µA.
kTSI_ExtOscChargeCurrent_4uA External oscillator charge current is 4 µA.
kTSI_ExtOscChargeCurrent_8uA External oscillator charge current is 8 µA.
kTSI_ExtOscChargeCurrent_16uA External oscillator charge current is 16 µA.
kTSI_ExtOscChargeCurrent_32uA External oscillator charge current is 32 µA.
kTSI_ExtOscChargeCurrent_64uA External oscillator charge current is 64 µA.

52.2.4.7 enum tsi_active_mode_clock_source_t

These constants define the active mode clock source in a TSI instance.

Enumerator

kTSI_ActiveClkSource_LPOSCCLK Active mode clock source is set to LPOOSC Clock.
kTSI_ActiveClkSource_MCGIRCLK Active mode clock source is set to MCG Internal reference clock.
kTSI_ActiveClkSource_OSCERCLK Active mode clock source is set to System oscillator output.

52.2.4.8 enum tsi_active_mode_prescaler_t

These constants define the TSI active mode prescaler in a TSI instance.

Enumerator

kTSI_ActiveModePrescaler_1div Input clock source divided by 1.
kTSI_ActiveModePrescaler_2div Input clock source divided by 2.
kTSI_ActiveModePrescaler_4div Input clock source divided by 4.
kTSI_ActiveModePrescaler_8div Input clock source divided by 8.
kTSI_ActiveModePrescaler_16div Input clock source divided by 16.
kTSI_ActiveModePrescaler_32div Input clock source divided by 32.
kTSI_ActiveModePrescaler_64div Input clock source divided by 64.
kTSI_ActiveModePrescaler_128div Input clock source divided by 128.

52.2.4.9 enum tsi_status_flags_t

Enumerator

kTSI_EndOfScanFlag End-Of-Scan flag.
kTSI_OutOfRangeFlag Out-Of-Range flag.
kTSI_ExternalElectrodeErrorFlag External electrode error flag.
kTSI_OverrunErrorFlag Overrun error flag.
kTSI_EndOfScanFlag End-Of-Scan flag.
kTSI_OutOfRangeFlag Out-Of-Range flag.

52.2.4.10 enum tsi_interrupt_enable_t

Enumerator

kTSI_GlobalInterruptEnable TSI module global interrupt.
kTSI_OutOfRangeInterruptEnable Out-Of-Range interrupt.
kTSI_EndOfScanInterruptEnable End-Of-Scan interrupt.
kTSI_ErrorInterrruptEnable Error interrupt.
kTSI_GlobalInterruptEnable TSI module global interrupt.
kTSI_OutOfRangeInterruptEnable Out-Of-Range interrupt.
kTSI_EndOfScanInterruptEnable End-Of-Scan interrupt.

52.2.5 Function Documentation

52.2.5.1 void TSI_Init (TSI_Type * *base*, const tsi_config_t * *config*)

Initializes the peripheral to the targeted state specified by the parameter configuration, such as initialize and set prescalers, number of scans, clocks, delta voltage capacitance trimmer, reference, and electrode charge current and threshold.

Parameters

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

Returns

none

52.2.5.2 void TSI_Deinit (TSI_Type * *base*)

De-initializes the peripheral to default state.

Parameters

base	TSI peripheral base address.

Returns

none

52.2.5.3 void TSI_GetNormalModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets the userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig.lpclks = kTSI_LowPowerClockSource_LPOCLK;
userConfig.lpscnitv = kTSI_LowPowerInterval_100ms;
userConfig.amclks = kTSI_ActiveClkSource_LPOSCCLK;
userConfig.ampsc = kTSI_ActiveModePrescaler_8div;
userConfig.ps = kTSI_ElecOscPrescaler_2div;
userConfig.extchrg = kTSI_ExtOscChargeCurrent_10uA;
userConfig.refchrg = kTSI_RefOscChargeCurrent_10uA;
userConfig.nscn = kTSI_ConsecutiveScansNumber_8time;
userConfig.thresh = 0U;
userConfig.thresl = 0U;
```

Parameters

userConfig	Pointer to the TSI user configuration structure.

52.2.5.4 void TSI_GetLowPowerModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig.lpclks = kTSI_LowPowerClockSource_LPOCLK;
userConfig.lpscnitv = kTSI_LowPowerInterval_100ms;
userConfig.amclks = kTSI_ActiveClkSource_LPOSCCLK;
userConfig.ampsc = kTSI_ActiveModePrescaler_64div;
userConfig.ps = kTSI_ElecOscPrescaler_1div;
userConfig.extchrg = kTSI_ExtOscChargeCurrent_2uA;
userConfig.refchrg = kTSI_RefOscChargeCurrent_32uA;
userConfig.nscn = kTSI_ConsecutiveScansNumber_26time;
userConfig.thresh = 15000U;
userConfig.thresl = 1000U;
```

Parameters

userConfig	Pointer to the TSI user configuration structure.

52.2.5.5 void TSI_Calibrate (TSI_Type * *base*, tsi_calibration_data_t * *calBuff*)

Calibrates the peripheral to fetch the initial counter value of the enabled electrodes. This API is mostly used at the initial application setup. Call this function after the TSI_Init API and use the calibrated counter values to set up applications (such as to determine under which counter value a touch event occurs).

Note

This API is called in normal power modes.
For K60 series, the calibrated baseline counter value CANNOT be used in low power modes. To obtain the calibrated counter values in low power modes, see K60 Mask Set Errata for Mask 5N22D.

Parameters

base	TSI peripheral base address.

calBuff	Data buffer that store the calibrated counter value.

Returns

none

52.2.5.6 void TSI_EnableInterrupts (TSI_Type * *base,* uint32_t *mask*)

Parameters

base	TSI peripheral base address.
mask	interrupt source The parameter can be a combination of the following source if defined: • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable • kTSI_ErrorInterrruptEnable

52.2.5.7 void TSI_DisableInterrupts (TSI_Type * *base,* uint32_t *mask*)

Parameters

base	TSI peripheral base address.
mask	interrupt source The parameter can be a combination of the following source if defined: • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable • kTSI_ErrorInterrruptEnable

52.2.5.8 static uint32_t TSI_GetStatusFlags (TSI_Type * *base*) [inline], [static]

This function get TSI interrupt flags.

Parameters

base	TSI peripheral base address.

Returns

The mask of these status flag bits.

52.2.5.9 void TSI_ClearStatusFlags (TSI_Type * *base,* uint32_t *mask*)

This function clears the TSI interrupt flags.

Note

The automatically cleared flags can't be cleared by this function.

Parameters

base	TSI peripheral base address.
mask	the status flags to clear.

52.2.5.10 static uint32_t TSI_GetScanTriggerMode (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Scan trigger mode.

52.2.5.11 static bool TSI_IsScanInProgress (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

True - if scan is in progress. False - if scan is not in progress.

52.2.5.12 static void TSI_SetElectrodeOSCPrescaler (TSI_Type * *base,* tsi_electrode_osc_prescaler_t *prescaler*) [inline], [static]

Parameters

base	TSI peripheral base address.
prescaler	Prescaler value.

Returns

none.

52.2.5.13 static void TSI_SetNumberOfScans (TSI_Type * *base,* tsi_n_consecutive_scans_t *number*) [inline], [static]

Parameters

base	TSI peripheral base address.
number	Number of scans.

Returns

none.

52.2.5.14 static void TSI_EnableModule (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose whether to enable TSI module. • true Enable module; • false Disable module;

Returns

none.

52.2.5.15 static void TSI_EnableLowPower (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose whether to enable TSI module in low power modes. • true Enable module in low power modes; • false Disable module in low power modes;

Returns

none.

52.2.5.16 static void TSI_EnablePeriodicalScan (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose whether to enable periodical trigger scan. • true Enable periodical trigger scan; • false Enable software trigger scan;

Returns

none.

52.2.5.17 static void TSI_StartSoftwareTrigger (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

none.

52.2.5.18 static void TSI_SetLowPowerScanInterval (TSI_Type * *base*, tsi_low_power_scan_interval_t *interval*) [inline], [static]

Parameters

base	TSI peripheral base address.
interval	Interval for low power scan.

Returns

none.

52.2.5.19 static void TSI_SetLowPowerClock (TSI_Type * *base*, uint32_t *clock*) [inline], [static]

Parameters

base	TSI peripheral base address.
clock	Low power clock selection.

52.2.5.20 static void TSI_SetReferenceChargeCurrent (TSI_Type * *base*, tsi_reference_osc_charge_current_t *current*) [inline], [static]

Parameters

base	TSI peripheral base address.
current	The reference oscillator charge current.

Returns

none.

52.2.5.21 static void TSI_SetElectrodeChargeCurrent (TSI_Type * *base,* tsi_external_osc_charge_current_t *current*) [inline], [static]

Parameters

base	TSI peripheral base address.
current	The external electrode charge current.

Returns

none.

52.2.5.22 static void TSI_SetScanModulo (TSI_Type * *base,* uint32_t *modulo*) [inline], [static]

Parameters

base	TSI peripheral base address.
modulo	Scan modulo value.

Returns

none.

52.2.5.23 static void TSI_SetActiveModeSource (TSI_Type * *base,* uint32_t *source*) [inline], [static]

Parameters

base	TSI peripheral base address.
source	Active mode clock source (LPOSCCLK, MCGIRCLK, OSCERCLK).

Returns

none.

52.2.5.24 static void TSI_SetActiveModePrescaler (TSI_Type * *base,* tsi_active_mode_prescaler_t *prescaler*) [inline], [static]

Parameters

base	TSI peripheral base address.
prescaler	Prescaler value.

Returns

none.

52.2.5.25 static void TSI_SetLowPowerChannel (TSI_Type * *base,* uint16_t *channel*) [inline], [static]

Only one channel can be enabled in low power mode.

Parameters

base	TSI peripheral base address.
channel	Channel number.

Returns

none.

52.2.5.26 static uint32_t TSI_GetLowPowerChannel (TSI_Type * *base*) [inline], [static]

Note

Only one channel can be enabled in low power mode.

Parameters

base	TSI peripheral base address.

Returns

Channel number.

52.2.5.27 static void TSI_EnableChannel (TSI_Type * *base,* uint16_t *channel,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
channel	Channel to be enabled.
enable	Choose whether to enable specific channel. • true Enable the specific channel; • false Disable the specific channel;

Returns

none.

52.2.5.28 static void TSI_EnableChannels (TSI_Type * *base,* uint16_t *channelsMask,* bool *enable*) [inline], [static]

The function enables or disables channels by mask. It can enable/disable all channels at once.

Parameters

base	TSI peripheral base address.
channelsMask	Channels mask that indicate channels that are to be enabled/disabled.
enable	Choose to enable or disable the specified channels. • true Enable the specified channels; • false Disable the specified channels;

Returns

none.

52.2.5.29 static bool TSI_IsChannelEnabled (TSI_Type * *base,* uint16_t *channel*) [inline], [static]

Parameters

base	TSI peripheral base address.
channel	Channel to be checked.

Returns

true - if the channel is enabled; false - if the channel is disabled;

52.2.5.30 static uint16_t TSI_GetEnabledChannels (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Channels mask that indicates currently enabled channels.

52.2.5.31 static uint16_t TSI_GetWakeUpChannelCounter (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Wake up counter value.

52.2.5.32 static uint16_t TSI_GetNormalModeCounter (TSI_Type * *base,* uint16_t *channel*) [inline], [static]

Note

This API can only be used in normal active modes.

Parameters

base	TSI peripheral base address.
channel	Index of the specific TSI channel.

Returns

The counter value of the specific channel.

52.2.5.33 static void TSI_SetLowThreshold (TSI_Type * *base,* uint16_t *low_threshold*) [inline], [static]

Parameters

base	TSI peripheral base address.
low_threshold	Low counter threshold.

Returns

none.

52.2.5.34 static void TSI_SetHighThreshold (TSI_Type * *base,* uint16_t *high_threshold*) [inline], [static]

Parameters

base	TSI peripheral base address.
high_threshold	High counter threshold.

Returns

none.

52.3 TSIv4 Driver

52.3.1 Overview

The KSDK provides driver for the Touch Sensing Input (TSI) module of Kinetis devices.

Typical use case

```
TSI_Init(TSI0);
TSI_Configure(TSI0, &user_config);
TSI_SetMeasuredChannelNumber(TSI0, channelMask);
TSI_EnableInterrupts(TSI0, kTSI_GlobalInterruptEnable |
      kTSI_EndOfScanInterruptEnable);

TSI_EnableSoftwareTriggerScan(TSI0);
TSI_EnableModule(TSI0);
while(1)
{
   TSI_StartSoftwareTrigger(TSI0);
   TSI_GetCounter(TSI0);
}
```

Files

- file fsl_tsi_v4.h

Data Structures

- struct tsi_calibration_data_t
  TSI calibration data storage. *More...*
- struct tsi_config_t
  TSI configuration structure. *More...*

Macros

- #define TSI_V4_EXTCHRG_RESISTOR_BIT_SHIFT TSI_GENCS_EXTCHRG_SHIFT
  resistor bit shift in EXTCHRG bit-field
- #define TSI_V4_EXTCHRG_FILTER_BITS_SHIFT (1U + TSI_GENCS_EXTCHRG_SHIFT)
  filter bits shift in EXTCHRG bit-field
- #define TSI_V4_EXTCHRG_RESISTOR_BIT_CLEAR ((uint32_t)((∼TSI_GENCS_EXTCHRG-_MASK) | (3U << TSI_V4_EXTCHRG_FILTER_BITS_SHIFT)))
  macro of clearing the resistor bit in EXTCHRG bit-field
- #define TSI_V4_EXTCHRG_FILTER_BITS_CLEAR ((uint32_t)((∼TSI_GENCS_EXTCHRG_-MASK) | (1U << TSI_V4_EXTCHRG_RESISTOR_BIT_SHIFT)))
  macro of clearing the filter bits in EXTCHRG bit-field

Enumerations

- enum tsi_n_consecutive_scans_t {
  kTSI_ConsecutiveScansNumber_1time = 0U,
  kTSI_ConsecutiveScansNumber_2time = 1U,
  kTSI_ConsecutiveScansNumber_3time = 2U,
  kTSI_ConsecutiveScansNumber_4time = 3U,
  kTSI_ConsecutiveScansNumber_5time = 4U,
  kTSI_ConsecutiveScansNumber_6time = 5U,
  kTSI_ConsecutiveScansNumber_7time = 6U,
  kTSI_ConsecutiveScansNumber_8time = 7U,
  kTSI_ConsecutiveScansNumber_9time = 8U,
  kTSI_ConsecutiveScansNumber_10time = 9U,
  kTSI_ConsecutiveScansNumber_11time = 10U,
  kTSI_ConsecutiveScansNumber_12time = 11U,
  kTSI_ConsecutiveScansNumber_13time = 12U,
  kTSI_ConsecutiveScansNumber_14time = 13U,
  kTSI_ConsecutiveScansNumber_15time = 14U,
  kTSI_ConsecutiveScansNumber_16time = 15U,
  kTSI_ConsecutiveScansNumber_17time = 16U,
  kTSI_ConsecutiveScansNumber_18time = 17U,
  kTSI_ConsecutiveScansNumber_19time = 18U,
  kTSI_ConsecutiveScansNumber_20time = 19U,
  kTSI_ConsecutiveScansNumber_21time = 20U,
  kTSI_ConsecutiveScansNumber_22time = 21U,
  kTSI_ConsecutiveScansNumber_23time = 22U,
  kTSI_ConsecutiveScansNumber_24time = 23U,
  kTSI_ConsecutiveScansNumber_25time = 24U,
  kTSI_ConsecutiveScansNumber_26time = 25U,
  kTSI_ConsecutiveScansNumber_27time = 26U,
  kTSI_ConsecutiveScansNumber_28time = 27U,
  kTSI_ConsecutiveScansNumber_29time = 28U,
  kTSI_ConsecutiveScansNumber_30time = 29U,
  kTSI_ConsecutiveScansNumber_31time = 30U,
  kTSI_ConsecutiveScansNumber_32time = 31U,
  kTSI_ConsecutiveScansNumber_1time = 0U,
  kTSI_ConsecutiveScansNumber_2time = 1U,
  kTSI_ConsecutiveScansNumber_3time = 2U,
  kTSI_ConsecutiveScansNumber_4time = 3U,
  kTSI_ConsecutiveScansNumber_5time = 4U,
  kTSI_ConsecutiveScansNumber_6time = 5U,
  kTSI_ConsecutiveScansNumber_7time = 6U,
  kTSI_ConsecutiveScansNumber_8time = 7U,
  kTSI_ConsecutiveScansNumber_9time = 8U,
  kTSI_ConsecutiveScansNumber_10time = 9U,
  kTSI_ConsecutiveScansNumber_11time = 10U,
  kTSI_ConsecutiveScansNumber_12time = 11U,
  kTSI_ConsecutiveScansNumber_13time = 12U,
  kTSI_ConsecutiveScansNumber_14time = 13U,

kTSI_ConsecutiveScansNumber_32time = 31U }
TSI number of scan intervals for each electrode.

- enum tsi_electrode_osc_prescaler_t {
  kTSI_ElecOscPrescaler_1div = 0U,
  kTSI_ElecOscPrescaler_2div = 1U,
  kTSI_ElecOscPrescaler_4div = 2U,
  kTSI_ElecOscPrescaler_8div = 3U,
  kTSI_ElecOscPrescaler_16div = 4U,
  kTSI_ElecOscPrescaler_32div = 5U,
  kTSI_ElecOscPrescaler_64div = 6U,
  kTSI_ElecOscPrescaler_128div = 7U,
  kTSI_ElecOscPrescaler_1div = 0U,
  kTSI_ElecOscPrescaler_2div = 1U,
  kTSI_ElecOscPrescaler_4div = 2U,
  kTSI_ElecOscPrescaler_8div = 3U,
  kTSI_ElecOscPrescaler_16div = 4U,
  kTSI_ElecOscPrescaler_32div = 5U,
  kTSI_ElecOscPrescaler_64div = 6U,
  kTSI_ElecOscPrescaler_128div = 7U }
  TSI electrode oscillator prescaler.
- enum tsi_analog_mode_t {
  kTSI_AnalogModeSel_Capacitive = 0U,
  kTSI_AnalogModeSel_NoiseNoFreqLim = 4U,
  kTSI_AnalogModeSel_NoiseFreqLim = 8U,
  kTSI_AnalogModeSel_AutoNoise = 12U }
  TSI analog mode select.
- enum tsi_reference_osc_charge_current_t {

kTSI_RefOscChargeCurrent_2uA = 0U,
kTSI_RefOscChargeCurrent_4uA = 1U,
kTSI_RefOscChargeCurrent_6uA = 2U,
kTSI_RefOscChargeCurrent_8uA = 3U,
kTSI_RefOscChargeCurrent_10uA = 4U,
kTSI_RefOscChargeCurrent_12uA = 5U,
kTSI_RefOscChargeCurrent_14uA = 6U,
kTSI_RefOscChargeCurrent_16uA = 7U,
kTSI_RefOscChargeCurrent_18uA = 8U,
kTSI_RefOscChargeCurrent_20uA = 9U,
kTSI_RefOscChargeCurrent_22uA = 10U,
kTSI_RefOscChargeCurrent_24uA = 11U,
kTSI_RefOscChargeCurrent_26uA = 12U,
kTSI_RefOscChargeCurrent_28uA = 13U,
kTSI_RefOscChargeCurrent_30uA = 14U,
kTSI_RefOscChargeCurrent_32uA = 15U,
kTSI_RefOscChargeCurrent_500nA = 0U,
kTSI_RefOscChargeCurrent_1uA = 1U,
kTSI_RefOscChargeCurrent_2uA = 2U,
kTSI_RefOscChargeCurrent_4uA = 3U,
kTSI_RefOscChargeCurrent_8uA = 4U,
kTSI_RefOscChargeCurrent_16uA = 5U,
kTSI_RefOscChargeCurrent_32uA = 6U,
kTSI_RefOscChargeCurrent_64uA = 7U }

TSI Reference oscillator charge and discharge current select.

- enum tsi_osc_voltage_rails_t {
  kTSI_OscVolRailsOption_0 = 0U,
  kTSI_OscVolRailsOption_1 = 1U,
  kTSI_OscVolRailsOption_2 = 2U,
  kTSI_OscVolRailsOption_3 = 3U }

  TSI oscilator's voltage rails.
- enum tsi_external_osc_charge_current_t {

kTSI_ExtOscChargeCurrent_2uA = 0U,
kTSI_ExtOscChargeCurrent_4uA = 1U,
kTSI_ExtOscChargeCurrent_6uA = 2U,
kTSI_ExtOscChargeCurrent_8uA = 3U,
kTSI_ExtOscChargeCurrent_10uA = 4U,
kTSI_ExtOscChargeCurrent_12uA = 5U,
kTSI_ExtOscChargeCurrent_14uA = 6U,
kTSI_ExtOscChargeCurrent_16uA = 7U,
kTSI_ExtOscChargeCurrent_18uA = 8U,
kTSI_ExtOscChargeCurrent_20uA = 9U,
kTSI_ExtOscChargeCurrent_22uA = 10U,
kTSI_ExtOscChargeCurrent_24uA = 11U,
kTSI_ExtOscChargeCurrent_26uA = 12U,
kTSI_ExtOscChargeCurrent_28uA = 13U,
kTSI_ExtOscChargeCurrent_30uA = 14U,
kTSI_ExtOscChargeCurrent_32uA = 15U,
kTSI_ExtOscChargeCurrent_500nA = 0U,
kTSI_ExtOscChargeCurrent_1uA = 1U,
kTSI_ExtOscChargeCurrent_2uA = 2U,
kTSI_ExtOscChargeCurrent_4uA = 3U,
kTSI_ExtOscChargeCurrent_8uA = 4U,
kTSI_ExtOscChargeCurrent_16uA = 5U,
kTSI_ExtOscChargeCurrent_32uA = 6U,
kTSI_ExtOscChargeCurrent_64uA = 7U }

TSI External oscillator charge and discharge current select.

- enum tsi_series_resistor_t {
  kTSI_SeriesResistance_32k = 0U,
  kTSI_SeriesResistance_187k = 1U }

  TSI series resistance RS value select.
- enum tsi_filter_bits_t {
  kTSI_FilterBits_3 = 0U,
  kTSI_FilterBits_2 = 1U,
  kTSI_FilterBits_1 = 2U,
  kTSI_FilterBits_0 = 3U }

  TSI series filter bits select.
- enum tsi_status_flags_t {
  kTSI_EndOfScanFlag = TSI_GENCS_EOSF_MASK,
  kTSI_OutOfRangeFlag = TSI_GENCS_OUTRGF_MASK,
  kTSI_ExternalElectrodeErrorFlag = TSI_GENCS_EXTERF_MASK,
  kTSI_OverrunErrorFlag = TSI_GENCS_OVRF_MASK,
  kTSI_EndOfScanFlag = TSI_GENCS_EOSF_MASK,
  kTSI_OutOfRangeFlag = TSI_GENCS_OUTRGF_MASK }

  TSI status flags.
- enum tsi_interrupt_enable_t {

kTSI_GlobalInterruptEnable = 1U,
kTSI_OutOfRangeInterruptEnable = 2U,
kTSI_EndOfScanInterruptEnable = 4U,
kTSI_ErrorInterrruptEnable = 8U,
kTSI_GlobalInterruptEnable = 1U,
kTSI_OutOfRangeInterruptEnable = 2U,
kTSI_EndOfScanInterruptEnable = 4U }
TSI feature interrupt source.

Functions

- void TSI_Init (TSI_Type *base, const tsi_config_t *config)
  Initializes hardware.
- void TSI_Deinit (TSI_Type *base)
  De-initializes hardware.
- void TSI_GetNormalModeDefaultConfig (tsi_config_t *userConfig)
  Gets the TSI normal mode user configuration structure.
- void TSI_GetLowPowerModeDefaultConfig (tsi_config_t *userConfig)
  Gets the TSI low power mode default user configuration structure.
- void TSI_Calibrate (TSI_Type *base, tsi_calibration_data_t *calBuff)
  Hardware calibration.
- void TSI_EnableInterrupts (TSI_Type *base, uint32_t mask)
  Enables the TSI interrupt requests.
- void TSI_DisableInterrupts (TSI_Type *base, uint32_t mask)
  Disables the TSI interrupt requests.
- static uint32_t TSI_GetStatusFlags (TSI_Type *base)
  Gets an interrupt flag.
- void TSI_ClearStatusFlags (TSI_Type *base, uint32_t mask)
  Clears the interrupt flag.
- static uint32_t TSI_GetScanTriggerMode (TSI_Type *base)
  Gets the TSI scan trigger mode.
- static bool TSI_IsScanInProgress (TSI_Type *base)
  Gets the scan in progress flag.
- static void TSI_SetElectrodeOSCPrescaler (TSI_Type *base, tsi_electrode_osc_prescaler_t prescaler)
  Sets the prescaler.
- static void TSI_SetNumberOfScans (TSI_Type *base, tsi_n_consecutive_scans_t number)
  Sets the number of scans (NSCN).
- static void TSI_EnableModule (TSI_Type *base, bool enable)
  Enables/disables the TSI module.
- static void TSI_EnableLowPower (TSI_Type *base, bool enable)
  Sets the TSI low power STOP mode as enabled or disabled.
- static void TSI_EnableHardwareTriggerScan (TSI_Type *base, bool enable)
  Enables/disables the hardware trigger scan.
- static void TSI_StartSoftwareTrigger (TSI_Type *base)
  Starts a software trigger measurement (triggers a new measurement).
- static void TSI_SetMeasuredChannelNumber (TSI_Type *base, uint8_t channel)
  Sets the the measured channel number.
- static uint8_t TSI_GetMeasuredChannelNumber (TSI_Type *base)

Gets the current measured channel number.
- static void TSI_EnableDmaTransfer (TSI_Type *base, bool enable)
  Enables/disables the DMA transfer.
- static uint16_t TSI_GetCounter (TSI_Type *base)
  Gets the conversion counter value.
- static void TSI_SetLowThreshold (TSI_Type *base, uint16_t low_threshold)
  Sets the TSI wake-up channel low threshold.
- static void TSI_SetHighThreshold (TSI_Type *base, uint16_t high_threshold)
  Sets the TSI wake-up channel high threshold.
- static void TSI_SetAnalogMode (TSI_Type *base, tsi_analog_mode_t mode)
  Sets the analog mode of the TSI module.
- static uint8_t TSI_GetNoiseModeResult (TSI_Type *base)
  Gets the noise mode result of the TSI module.
- static void TSI_SetReferenceChargeCurrent (TSI_Type *base, tsi_reference_osc_charge_current_t current)
  Sets the reference oscillator charge current.
- static void TSI_SetElectrodeChargeCurrent (TSI_Type *base, tsi_external_osc_charge_current_t current)
  Sets the external electrode charge current.
- static void TSI_SetOscVoltageRails (TSI_Type *base, tsi_osc_voltage_rails_t dvolt)
  Sets the oscillator's voltage rails.
- static void TSI_SetElectrodeSeriesResistor (TSI_Type *base, tsi_series_resistor_t resistor)
  Sets the electrode series resistance value in EXTCHRG[0] bit.
- static void TSI_SetFilterBits (TSI_Type *base, tsi_filter_bits_t filter)
  Sets the electrode filter bits value in EXTCHRG[2:1] bits.

Driver version

- #define FSL_TSI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  TSI driver version 2.0.0.

52.3.2 Data Structure Documentation

52.3.2.1 struct tsi_calibration_data_t

Data Fields

- uint16_t calibratedData [FSL_FEATURE_TSI_CHANNEL_COUNT]
  TSI calibration data storage buffer.

52.3.2.2 struct tsi_config_t

This structure contains the settings for the most common TSI configurations including the TSI module charge currents, number of scans, thresholds, and so on.

Data Fields

- uint16_t thresh
  High threshold.
- uint16_t thresl
  Low threshold.
- tsi_low_power_clock_source_t lpclks
  Low power clock.
- tsi_low_power_scan_interval_t lpscnitv
  Low power scan interval.
- tsi_active_mode_clock_source_t amclks
  Active mode clock source.
- tsi_active_mode_prescaler_t ampsc
  Active mode prescaler.
- tsi_electrode_osc_prescaler_t ps
  Electrode Oscillator Prescaler.
- tsi_external_osc_charge_current_t extchrg
  External Oscillator charge current.
- tsi_reference_osc_charge_current_t refchrg
  Reference Oscillator charge current.
- tsi_n_consecutive_scans_t nscn
  Number of scans.
- tsi_electrode_osc_prescaler_t prescaler
  Prescaler.
- tsi_analog_mode_t mode
  TSI mode of operation.
- tsi_osc_voltage_rails_t dvolt
  Oscillator's voltage rails.
- tsi_series_resistor_t resistor
  Series resistance value.
- tsi_filter_bits_t filter
  Noise mode filter bits.

52.3.2.2.0.11 Field Documentation

52.3.2.2.0.11.1 uint16_t tsi_config_t::thresh

52.3.2.2.0.11.2 uint16_t tsi_config_t::thresl

52.3.2.2.0.11.3 tsi_low_power_clock_source_t tsi_config_t::lpclks

52.3.2.2.0.11.4 tsi_low_power_scan_interval_t tsi_config_t::lpscnitv

52.3.2.2.0.11.5 tsi_active_mode_clock_source_t tsi_config_t::amclks

52.3.2.2.0.11.6 tsi_active_mode_prescaler_t tsi_config_t::ampsc

52.3.2.2.0.11.7 tsi_external_osc_charge_current_t tsi_config_t::extchrg

Electrode charge current.

52.3.2.2.0.11.8 tsi_reference_osc_charge_current_t tsi_config_t::refchrg

Reference charge current.

52.3.2.2.0.11.9 tsi_n_consecutive_scans_t tsi_config_t::nscn

52.3.2.2.0.11.10 tsi_analog_mode_t tsi_config_t::mode

52.3.2.2.0.11.11 tsi_osc_voltage_rails_t tsi_config_t::dvolt

52.3.3 Macro Definition Documentation

52.3.3.1 #define FSL_TSI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

52.3.4 Enumeration Type Documentation

52.3.4.1 enum tsi_n_consecutive_scans_t

These constants define the tsi number of consecutive scans in a TSI instance for each electrode.

Enumerator

kTSI_ConsecutiveScansNumber_1time	Once per electrode.
kTSI_ConsecutiveScansNumber_2time	Twice per electrode.
kTSI_ConsecutiveScansNumber_3time	3 times consecutive scan
kTSI_ConsecutiveScansNumber_4time	4 times consecutive scan
kTSI_ConsecutiveScansNumber_5time	5 times consecutive scan
kTSI_ConsecutiveScansNumber_6time	6 times consecutive scan
kTSI_ConsecutiveScansNumber_7time	7 times consecutive scan
kTSI_ConsecutiveScansNumber_8time	8 times consecutive scan
kTSI_ConsecutiveScansNumber_9time	9 times consecutive scan
kTSI_ConsecutiveScansNumber_10time	10 times consecutive scan
kTSI_ConsecutiveScansNumber_11time	11 times consecutive scan
kTSI_ConsecutiveScansNumber_12time	12 times consecutive scan
kTSI_ConsecutiveScansNumber_13time	13 times consecutive scan
kTSI_ConsecutiveScansNumber_14time	14 times consecutive scan
kTSI_ConsecutiveScansNumber_15time	15 times consecutive scan
kTSI_ConsecutiveScansNumber_16time	16 times consecutive scan
kTSI_ConsecutiveScansNumber_17time	17 times consecutive scan
kTSI_ConsecutiveScansNumber_18time	18 times consecutive scan
kTSI_ConsecutiveScansNumber_19time	19 times consecutive scan
kTSI_ConsecutiveScansNumber_20time	20 times consecutive scan
kTSI_ConsecutiveScansNumber_21time	21 times consecutive scan
kTSI_ConsecutiveScansNumber_22time	22 times consecutive scan
kTSI_ConsecutiveScansNumber_23time	23 times consecutive scan
kTSI_ConsecutiveScansNumber_24time	24 times consecutive scan

kTSI_ConsecutiveScansNumber_25time 25 times consecutive scan
kTSI_ConsecutiveScansNumber_26time 26 times consecutive scan
kTSI_ConsecutiveScansNumber_27time 27 times consecutive scan
kTSI_ConsecutiveScansNumber_28time 28 times consecutive scan
kTSI_ConsecutiveScansNumber_29time 29 times consecutive scan
kTSI_ConsecutiveScansNumber_30time 30 times consecutive scan
kTSI_ConsecutiveScansNumber_31time 31 times consecutive scan
kTSI_ConsecutiveScansNumber_32time 32 times consecutive scan
kTSI_ConsecutiveScansNumber_1time Once per electrode.
kTSI_ConsecutiveScansNumber_2time Twice per electrode.
kTSI_ConsecutiveScansNumber_3time 3 times consecutive scan
kTSI_ConsecutiveScansNumber_4time 4 times consecutive scan
kTSI_ConsecutiveScansNumber_5time 5 times consecutive scan
kTSI_ConsecutiveScansNumber_6time 6 times consecutive scan
kTSI_ConsecutiveScansNumber_7time 7 times consecutive scan
kTSI_ConsecutiveScansNumber_8time 8 times consecutive scan
kTSI_ConsecutiveScansNumber_9time 9 times consecutive scan
kTSI_ConsecutiveScansNumber_10time 10 times consecutive scan
kTSI_ConsecutiveScansNumber_11time 11 times consecutive scan
kTSI_ConsecutiveScansNumber_12time 12 times consecutive scan
kTSI_ConsecutiveScansNumber_13time 13 times consecutive scan
kTSI_ConsecutiveScansNumber_14time 14 times consecutive scan
kTSI_ConsecutiveScansNumber_15time 15 times consecutive scan
kTSI_ConsecutiveScansNumber_16time 16 times consecutive scan
kTSI_ConsecutiveScansNumber_17time 17 times consecutive scan
kTSI_ConsecutiveScansNumber_18time 18 times consecutive scan
kTSI_ConsecutiveScansNumber_19time 19 times consecutive scan
kTSI_ConsecutiveScansNumber_20time 20 times consecutive scan
kTSI_ConsecutiveScansNumber_21time 21 times consecutive scan
kTSI_ConsecutiveScansNumber_22time 22 times consecutive scan
kTSI_ConsecutiveScansNumber_23time 23 times consecutive scan
kTSI_ConsecutiveScansNumber_24time 24 times consecutive scan
kTSI_ConsecutiveScansNumber_25time 25 times consecutive scan
kTSI_ConsecutiveScansNumber_26time 26 times consecutive scan
kTSI_ConsecutiveScansNumber_27time 27 times consecutive scan
kTSI_ConsecutiveScansNumber_28time 28 times consecutive scan
kTSI_ConsecutiveScansNumber_29time 29 times consecutive scan
kTSI_ConsecutiveScansNumber_30time 30 times consecutive scan
kTSI_ConsecutiveScansNumber_31time 31 times consecutive scan
kTSI_ConsecutiveScansNumber_32time 32 times consecutive scan

52.3.4.2 enum tsi_electrode_osc_prescaler_t

These constants define the TSI electrode oscillator prescaler in a TSI instance.

Enumerator

kTSI_ElecOscPrescaler_1div Electrode oscillator frequency divided by 1.
kTSI_ElecOscPrescaler_2div Electrode oscillator frequency divided by 2.
kTSI_ElecOscPrescaler_4div Electrode oscillator frequency divided by 4.
kTSI_ElecOscPrescaler_8div Electrode oscillator frequency divided by 8.
kTSI_ElecOscPrescaler_16div Electrode oscillator frequency divided by 16.
kTSI_ElecOscPrescaler_32div Electrode oscillator frequency divided by 32.
kTSI_ElecOscPrescaler_64div Electrode oscillator frequency divided by 64.
kTSI_ElecOscPrescaler_128div Electrode oscillator frequency divided by 128.
kTSI_ElecOscPrescaler_1div Electrode oscillator frequency divided by 1.
kTSI_ElecOscPrescaler_2div Electrode oscillator frequency divided by 2.
kTSI_ElecOscPrescaler_4div Electrode oscillator frequency divided by 4.
kTSI_ElecOscPrescaler_8div Electrode oscillator frequency divided by 8.
kTSI_ElecOscPrescaler_16div Electrode oscillator frequency divided by 16.
kTSI_ElecOscPrescaler_32div Electrode oscillator frequency divided by 32.
kTSI_ElecOscPrescaler_64div Electrode oscillator frequency divided by 64.
kTSI_ElecOscPrescaler_128div Electrode oscillator frequency divided by 128.

52.3.4.3 enum tsi_analog_mode_t

Set up TSI analog modes in a TSI instance.

Enumerator

kTSI_AnalogModeSel_Capacitive Active TSI capacitive sensing mode.
kTSI_AnalogModeSel_NoiseNoFreqLim Single threshold noise detection mode with no freq. limitation.
kTSI_AnalogModeSel_NoiseFreqLim Single threshold noise detection mode with freq. limitation.

kTSI_AnalogModeSel_AutoNoise Active TSI analog in automatic noise detection mode.

52.3.4.4 enum tsi_reference_osc_charge_current_t

These constants define the TSI Reference oscillator charge current select in a TSI (REFCHRG) instance.

Enumerator

kTSI_RefOscChargeCurrent_2uA Reference oscillator charge current is 2 µA.
kTSI_RefOscChargeCurrent_4uA Reference oscillator charge current is 4 µA.
kTSI_RefOscChargeCurrent_6uA Reference oscillator charge current is 6 µA.
kTSI_RefOscChargeCurrent_8uA Reference oscillator charge current is 8 µA.
kTSI_RefOscChargeCurrent_10uA Reference oscillator charge current is 10 µA.
kTSI_RefOscChargeCurrent_12uA Reference oscillator charge current is 12 µA.
kTSI_RefOscChargeCurrent_14uA Reference oscillator charge current is 14 µA.

kTSI_RefOscChargeCurrent_16uA Reference oscillator charge current is 16 µA.
kTSI_RefOscChargeCurrent_18uA Reference oscillator charge current is 18 µA.
kTSI_RefOscChargeCurrent_20uA Reference oscillator charge current is 20 µA.
kTSI_RefOscChargeCurrent_22uA Reference oscillator charge current is 22 µA.
kTSI_RefOscChargeCurrent_24uA Reference oscillator charge current is 24 µA.
kTSI_RefOscChargeCurrent_26uA Reference oscillator charge current is 26 µA.
kTSI_RefOscChargeCurrent_28uA Reference oscillator charge current is 28 µA.
kTSI_RefOscChargeCurrent_30uA Reference oscillator charge current is 30 µA.
kTSI_RefOscChargeCurrent_32uA Reference oscillator charge current is 32 µA.
kTSI_RefOscChargeCurrent_500nA Reference oscillator charge current is 500 µA.
kTSI_RefOscChargeCurrent_1uA Reference oscillator charge current is 1 µA.
kTSI_RefOscChargeCurrent_2uA Reference oscillator charge current is 2 µA.
kTSI_RefOscChargeCurrent_4uA Reference oscillator charge current is 4 µA.
kTSI_RefOscChargeCurrent_8uA Reference oscillator charge current is 8 µA.
kTSI_RefOscChargeCurrent_16uA Reference oscillator charge current is 16 µA.
kTSI_RefOscChargeCurrent_32uA Reference oscillator charge current is 32 µA.
kTSI_RefOscChargeCurrent_64uA Reference oscillator charge current is 64 µA.

52.3.4.5 enum tsi_osc_voltage_rails_t

These bits indicate the oscillator's voltage rails.

Enumerator

kTSI_OscVolRailsOption_0 DVOLT value option 0, the value may differ on different platforms.
kTSI_OscVolRailsOption_1 DVOLT value option 1, the value may differ on different platforms.
kTSI_OscVolRailsOption_2 DVOLT value option 2, the value may differ on different platforms.
kTSI_OscVolRailsOption_3 DVOLT value option 3, the value may differ on different platforms.

52.3.4.6 enum tsi_external_osc_charge_current_t

These bits indicate the electrode oscillator charge and discharge current value in TSI (EXTCHRG) instance.

Enumerator

kTSI_ExtOscChargeCurrent_2uA External oscillator charge current is 2 µA.
kTSI_ExtOscChargeCurrent_4uA External oscillator charge current is 4 µA.
kTSI_ExtOscChargeCurrent_6uA External oscillator charge current is 6 µA.
kTSI_ExtOscChargeCurrent_8uA External oscillator charge current is 8 µA.
kTSI_ExtOscChargeCurrent_10uA External oscillator charge current is 10 µA.
kTSI_ExtOscChargeCurrent_12uA External oscillator charge current is 12 µA.
kTSI_ExtOscChargeCurrent_14uA External oscillator charge current is 14 µA.
kTSI_ExtOscChargeCurrent_16uA External oscillator charge current is 16 µA.

kTSI_ExtOscChargeCurrent_18uA External oscillator charge current is 18 µA.
kTSI_ExtOscChargeCurrent_20uA External oscillator charge current is 20 µA.
kTSI_ExtOscChargeCurrent_22uA External oscillator charge current is 22 µA.
kTSI_ExtOscChargeCurrent_24uA External oscillator charge current is 24 µA.
kTSI_ExtOscChargeCurrent_26uA External oscillator charge current is 26 µA.
kTSI_ExtOscChargeCurrent_28uA External oscillator charge current is 28 µA.
kTSI_ExtOscChargeCurrent_30uA External oscillator charge current is 30 µA.
kTSI_ExtOscChargeCurrent_32uA External oscillator charge current is 32 µA.
kTSI_ExtOscChargeCurrent_500nA External oscillator charge current is 500 µA.
kTSI_ExtOscChargeCurrent_1uA External oscillator charge current is 1 µA.
kTSI_ExtOscChargeCurrent_2uA External oscillator charge current is 2 µA.
kTSI_ExtOscChargeCurrent_4uA External oscillator charge current is 4 µA.
kTSI_ExtOscChargeCurrent_8uA External oscillator charge current is 8 µA.
kTSI_ExtOscChargeCurrent_16uA External oscillator charge current is 16 µA.
kTSI_ExtOscChargeCurrent_32uA External oscillator charge current is 32 µA.
kTSI_ExtOscChargeCurrent_64uA External oscillator charge current is 64 µA.

52.3.4.7 enum tsi_series_resistor_t

These bits indicate the electrode RS series resistance for the noise mode in TSI (EXTCHRG) instance.

Enumerator

kTSI_SeriesResistance_32k Series Resistance is 32 kilo ohms.
kTSI_SeriesResistance_187k Series Resistance is 18 7 kilo ohms.

52.3.4.8 enum tsi_filter_bits_t

These bits indicate the count of the filter bits in TSI noise mode EXTCHRG[2:1] bits

Enumerator

kTSI_FilterBits_3 3 filter bits, 8 peaks increments the cnt+1
kTSI_FilterBits_2 2 filter bits, 4 peaks increments the cnt+1
kTSI_FilterBits_1 1 filter bits, 2 peaks increments the cnt+1
kTSI_FilterBits_0 no filter bits,1 peak increments the cnt+1

52.3.4.9 enum tsi_status_flags_t

Enumerator

kTSI_EndOfScanFlag End-Of-Scan flag.
kTSI_OutOfRangeFlag Out-Of-Range flag.

kTSI_ExternalElectrodeErrorFlag External electrode error flag.
kTSI_OverrunErrorFlag Overrun error flag.
kTSI_EndOfScanFlag End-Of-Scan flag.
kTSI_OutOfRangeFlag Out-Of-Range flag.

52.3.4.10 enum tsi_interrupt_enable_t

Enumerator

kTSI_GlobalInterruptEnable TSI module global interrupt.
kTSI_OutOfRangeInterruptEnable Out-Of-Range interrupt.
kTSI_EndOfScanInterruptEnable End-Of-Scan interrupt.
kTSI_ErrorInterrruptEnable Error interrupt.
kTSI_GlobalInterruptEnable TSI module global interrupt.
kTSI_OutOfRangeInterruptEnable Out-Of-Range interrupt.
kTSI_EndOfScanInterruptEnable End-Of-Scan interrupt.

52.3.5 Function Documentation

52.3.5.1 void TSI_Init (TSI_Type * *base,* const tsi_config_t * *config*)

Initializes the peripheral to the targeted state specified by parameter configuration, such as sets prescalers, number of scans, clocks, delta voltage series resistor, filter bits, reference, and electrode charge current and threshold.

Parameters

base	TSI peripheral base address.
config	Pointer to TSI module configuration structure.

Returns

none

52.3.5.2 void TSI_Deinit (TSI_Type * *base*)

De-initializes the peripheral to default state.

Parameters

base	TSI peripheral base address.

Returns

none

52.3.5.3 void TSI_GetNormalModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig->prescaler = kTSI_ElecOscPrescaler_2div;
userConfig->extchrg = kTSI_ExtOscChargeCurrent_4uA;
userConfig->refchrg = kTSI_RefOscChargeCurrent_4uA;
userConfig->nscn = kTSI_ConsecutiveScansNumber_10time;
userConfig->mode = kTSI_AnalogModeSel_Capacitive;
userConfig->dvolt = kTSI_OscVolRailsOption_0;
userConfig->resistor = kTSI_SeriesResistance_32k;
userConfig->filter = kTSI_FilterBits_1;
userConfig->thresh = 0U;
userConfig->thresl = 0U;
```

Parameters

userConfig	Pointer to the TSI user configuration structure.

52.3.5.4 void TSI_GetLowPowerModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig->prescaler = kTSI_ElecOscPrescaler_2div;
userConfig->extchrg = kTSI_ExtOscChargeCurrent_4uA;
userConfig->refchrg = kTSI_RefOscChargeCurrent_4uA;
userConfig->nscn = kTSI_ConsecutiveScansNumber_10time;
userConfig->mode = kTSI_AnalogModeSel_Capacitive;
userConfig->dvolt = kTSI_OscVolRailsOption_0;
userConfig->resistor = kTSI_SeriesResistance_32k;
userConfig->filter = kTSI_FilterBits_1;
userConfig->thresh = 400U;
userConfig->thresl = 0U;
```

Parameters

userConfig	Pointer to the TSI user configuration structure.

52.3.5.5 void TSI_Calibrate (TSI_Type * *base*, tsi_calibration_data_t * *calBuff*)

Calibrates the peripheral to fetch the initial counter value of the enabled electrodes. This API is mostly used at initial application setup. Call this function after the TSI_Init API and use the calibrated counter values to set up applications (such as to determine under which counter value we can confirm a touch event occurs).

Parameters

base	TSI peripheral base address.
calBuff	Data buffer that store the calibrated counter value.

Returns

none

52.3.5.6 void TSI_EnableInterrupts (TSI_Type * *base*, uint32_t *mask*)

Parameters

base	TSI peripheral base address.
mask	interrupt source The parameter can be combination of the following source if defined: • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable

52.3.5.7 void TSI_DisableInterrupts (TSI_Type * *base*, uint32_t *mask*)

Parameters

base	TSI peripheral base address.
mask	interrupt source The parameter can be combination of the following source if defined: • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable

52.3.5.8 static uint32_t TSI_GetStatusFlags (TSI_Type * *base*) [inline], [static]

This function gets the TSI interrupt flags.

Parameters

base	TSI peripheral base address.

Returns

The mask of these status flags combination.

52.3.5.9 void TSI_ClearStatusFlags (TSI_Type * *base,* uint32_t *mask*)

This function clears the TSI interrupt flag, automatically cleared flags can't be cleared by this function.

Parameters

base	TSI peripheral base address.
mask	The status flags to clear.

52.3.5.10 static uint32_t TSI_GetScanTriggerMode (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Scan trigger mode.

52.3.5.11 static bool TSI_IsScanInProgress (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

True - scan is in progress. False - scan is not in progress.

52.3.5.12 static void TSI_SetElectrodeOSCPrescaler (TSI_Type * *base,* tsi_electrode_osc_prescaler_t *prescaler*) [inline], [static]

Parameters

base	TSI peripheral base address.
prescaler	Prescaler value.

Returns

none.

52.3.5.13 static void TSI_SetNumberOfScans (TSI_Type * *base,* tsi_n_consecutive_scans_t *number*) [inline], [static]

Parameters

base	TSI peripheral base address.
number	Number of scans.

Returns

none.

52.3.5.14 static void TSI_EnableModule (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose whether to enable or disable module; • true Enable TSI module; • false Disable TSI module;

Returns

none.

52.3.5.15 static void TSI_EnableLowPower (TSI_Type * *base,* bool *enable*) [inline], [static]

This enables the TSI module function in low power modes.

Parameters

base	TSI peripheral base address.
enable	Choose to enable or disable STOP mode. • true Enable module in STOP mode; • false Disable module in STOP mode;

Returns

none.

52.3.5.16 static void TSI_EnableHardwareTriggerScan (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose to enable hardware trigger or software trigger scan. • true Enable hardware trigger scan; • false Enable software trigger scan;

Returns

none.

52.3.5.17 static void TSI_StartSoftwareTrigger (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

none.

52.3.5.18 static void TSI_SetMeasuredChannelNumber (TSI_Type * *base,* uint8_t *channel*) [inline], [static]

Parameters

base	TSI peripheral base address.
channel	Channel number 0 ... 15.

Returns

none.

52.3.5.19 static uint8_t TSI_GetMeasuredChannelNumber (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

uint8_t Channel number 0 ... 15.

52.3.5.20 static void TSI_EnableDmaTransfer (TSI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	TSI peripheral base address.
enable	Choose to enable DMA transfer or not. • true Enable DMA transfer; • false Disable DMA transfer;

Returns

none.

52.3.5.21 static uint16_t TSI_GetCounter (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Accumulated scan counter value ticked by the reference clock.

52.3.5.22 static void TSI_SetLowThreshold (TSI_Type * *base,* uint16_t *low_threshold*) [inline], [static]

Parameters

base	TSI peripheral base address.
low_threshold	Low counter threshold.

Returns

none.

52.3.5.23 static void TSI_SetHighThreshold (TSI_Type * *base,* uint16_t *high_threshold*) [inline], [static]

Parameters

base	TSI peripheral base address.
high_threshold	High counter threshold.

Returns

none.

52.3.5.24 static void TSI_SetAnalogMode (TSI_Type * *base,* tsi_analog_mode_t *mode*) [inline], [static]

Parameters

base	TSI peripheral base address.
mode	Mode value.

Returns

none.

52.3.5.25 static uint8_t TSI_GetNoiseModeResult (TSI_Type * *base*) [inline], [static]

Parameters

base	TSI peripheral base address.

Returns

Value of the GENCS[MODE] bit-fields.

52.3.5.26 static void TSI_SetReferenceChargeCurrent (TSI_Type * *base,* tsi_reference_osc_charge_current_t *current*) [inline], [static]

Parameters

base	TSI peripheral base address.
current	The reference oscillator charge current.

Returns

none.

52.3.5.27 static void TSI_SetElectrodeChargeCurrent (TSI_Type * *base*, tsi_external_osc_charge_current_t *current*) [inline], [static]

Parameters

base	TSI peripheral base address.
current	External electrode charge current.

Returns

none.

52.3.5.28 static void TSI_SetOscVoltageRails (TSI_Type * *base*, tsi_osc_voltage_rails_t *dvolt*) [inline], [static]

Parameters

base	TSI peripheral base address.
dvolt	The voltage rails.

Returns

none.

52.3.5.29 static void TSI_SetElectrodeSeriesResistor (TSI_Type * *base*, tsi_series_resistor_t *resistor*) [inline], [static]

Parameters

base	TSI peripheral base address.
resistor	Series resistance.

Returns

none.

52.3.5.30 static void TSI_SetFilterBits (TSI_Type * *base,* tsi_filter_bits_t *filter*) [inline], [static]

Parameters

base	TSI peripheral base address.
filter	Series resistance.

Returns

none.

Chapter 53
UART: Universal Asynchronous Receiver/Transmitter Driver

53.1 Overview

Modules

- UART Driver
- UART FreeRTOS Driver
- UART µCOS/II Driver
- UART µCOS/III Driver

53.2 UART Driver

53.2.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of Kinetis devices.

The UART driver includes two parts: functional APIs and transactional APIs.

Functional APIs are used for UART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the UART peripheral and know how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. UART functional operation groups provide the functional APIs set.

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the uart_handle_t as the first parameter. Initialize the handle by calling the UART_Create-Handle() API.

Transactional APIs support asynchronous transfer, which means that the functions UART_SendNon-Blocking() and UART_ReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_UART_TxIdle and kStatus_UART_RxIdle.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the UART_CreateHandle(). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The UART_ReceiveNonBlocking() function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the kStatus_UART_RxIdle.

If the receive ring buffer is full, the upper layer is informed through a callback with the kStatus_UART-_RxRingBufferOverrun. In the callback function, the upper layer reads data out from the ring buffer. If not, the oldest data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code:

```
UART_CreateHandle(&handle, UART0, &ringBuffer, 32);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

53.2.2 Typical use case

53.2.2.1 UART Send/receive using a polling method

```
uint8_t ch;
```

```
UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

UART_Init(UART1,&user_config,120000000U);

while(1)
{
    UART_TransferReceiveBlocking(UART1, &ch, 1);
    UART_TransferSendBlocking(UART1, &ch, 1);
}
```

53.2.2.2 UART Send/receive using an interrupt method

```
uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_CreateHandle(&g_uartHandle, UART1, NULL, 0);
    UART_SetTransferCallback(&g_uartHandle, UART_UserCallback, NULL);

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

    // Send out.
    UART_SendNonBlocking(&g_uartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }
```

```
    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receive.
    UART_ReceiveNonBlocking(&g_uartHandle, &receiveXfer, NULL);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

53.2.2.3 UART Receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE     32

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_CreateHandle(&g_uartHandle, UART1, &ringBuffer, RING_BUFFER_SIZE);
    UART_SetTransferCallback(&g_uartHandle, UART_UserCallback, NULL);

    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = RX_DATA_SIZE;
    rxFinished = false;

    // Receive.
    UART_ReceiveNonBlocking(&g_uartHandle, &receiveXfer, &bytesRead);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data.
```

```
    {
        ;
    }
    else
    {
        if (bytesRead) /* Received some data, process first.
        {
            ;
        }

        // Wait receive finished.
        while (!rxFinished)
        {
        }
    }

    // ...
}
```

53.2.2.4 UART Send/Receive using the DMA method

```
uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = 'H', 'e', 'l', 'l', 'o';
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);

    // Set up the DMA
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, UART_TX_DMA_CHANNEL, UART_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, UART_RX_DMA_CHANNEL, UART_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_RX_DMA_CHANNEL);
```

```
    DMA_Init(DMA0);

    /* Create DMA handle.
    DMA_CreateHandle(&g_uartTxDmaHandle, DMA0, UART_TX_DMA_CHANNEL);
    DMA_CreateHandle(&g_uartRxDmaHandle, DMA0, UART_RX_DMA_CHANNEL);

    UART_CreateHandleDMA(&g_uartHandle, UART1, &g_uartTxDmaHandle, &g_uartRxDmaHandle);
    UART_SetTransferCallbackDMA(&g_uartDmaHandle, UART_UserCallback, NULL);

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

    // Send out.
    UART_SendDMA(&g_uartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
    rxFinished = false;

    // Receive.
    UART_ReceiveDMA(&g_uartHandle, &receiveXfer, NULL);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

Modules

- UART DMA Driver
- UART eDMA Driver

Files

- file fsl_uart.h

Data Structures

- struct uart_config_t
  UART configuration structure. *More...*
- struct uart_transfer_t
  UART transfer structure. *More...*
- struct uart_handle_t
  UART handle structure. *More...*

Typedefs

- typedef void(∗ uart_transfer_callback_t)(UART_Type ∗base, uart_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

Enumerations

- enum _uart_status {
  kStatus_UART_TxBusy = MAKE_STATUS(kStatusGroup_UART, 0),
  kStatus_UART_RxBusy = MAKE_STATUS(kStatusGroup_UART, 1),
  kStatus_UART_TxIdle = MAKE_STATUS(kStatusGroup_UART, 2),
  kStatus_UART_RxIdle = MAKE_STATUS(kStatusGroup_UART, 3),
  kStatus_UART_TxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 4),
  kStatus_UART_RxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_UART, 5),
  kStatus_UART_FlagCannotClearManually,
  kStatus_UART_Error = MAKE_STATUS(kStatusGroup_UART, 7),
  kStatus_UART_RxRingBufferOverrun = MAKE_STATUS(kStatusGroup_UART, 8),
  kStatus_UART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_UART, 9),
  kStatus_UART_NoiseError = MAKE_STATUS(kStatusGroup_UART, 10),
  kStatus_UART_FramingError = MAKE_STATUS(kStatusGroup_UART, 11),
  kStatus_UART_ParityError = MAKE_STATUS(kStatusGroup_UART, 12) }
  Error codes for the UART driver.
- enum uart_parity_mode_t {
  kUART_ParityDisabled = 0x0U,
  kUART_ParityEven = 0x2U,
  kUART_ParityOdd = 0x3U }
  UART parity mode.
- enum uart_stop_bit_count_t {
  kUART_OneStopBit = 0U,
  kUART_TwoStopBit = 1U }
  UART stop bit count.
- enum _uart_interrupt_enable {
  kUART_RxActiveEdgeInterruptEnable = (UART_BDH_RXEDGIE_MASK),
  kUART_TxDataRegEmptyInterruptEnable = (UART_C2_TIE_MASK << 8),
  kUART_TransmissionCompleteInterruptEnable = (UART_C2_TCIE_MASK << 8),
  kUART_RxDataRegFullInterruptEnable = (UART_C2_RIE_MASK << 8),
  kUART_IdleLineInterruptEnable = (UART_C2_ILIE_MASK << 8),
  kUART_RxOverrunInterruptEnable = (UART_C3_ORIE_MASK << 16),
  kUART_NoiseErrorInterruptEnable = (UART_C3_NEIE_MASK << 16),
  kUART_FramingErrorInterruptEnable = (UART_C3_FEIE_MASK << 16),
  kUART_ParityErrorInterruptEnable = (UART_C3_PEIE_MASK << 16) }
  UART interrupt configuration structure, default settings all disabled.
- enum _uart_flags {

kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),
kUART_RxOverrunFlag = (UART_S1_OR_MASK),
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),
kUART_FramingErrorFlag = (UART_S1_FE_MASK),
kUART_ParityErrorFlag = (UART_S1_PF_MASK),
kUART_RxActiveEdgeFlag = (UART_S2_RXEDGIF_MASK << 8),
kUART_RxActiveFlag = (UART_S2_RAF_MASK << 8) }
UART status flags.

Driver version

- #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  UART driver version 2.1.0.

Initialization and deinitialization

- void UART_Init (UART_Type *base, const uart_config_t *config, uint32_t srcClock_Hz)
  Initializes a UART instance with user configuration structure and peripheral clock.
- void UART_Deinit (UART_Type *base)
  Deinitializes a UART instance.
- void UART_GetDefaultConfig (uart_config_t *config)
  Gets the default configuration structure.
- void UART_SetBaudRate (UART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the UART instance baud rate.

Status

- uint32_t UART_GetStatusFlags (UART_Type *base)
  Get UART status flags.
- status_t UART_ClearStatusFlags (UART_Type *base, uint32_t mask)
  Clears status flags with the provided mask.

Interrupts

- void UART_EnableInterrupts (UART_Type *base, uint32_t mask)
  Enables UART interrupts according to the provided mask.
- void UART_DisableInterrupts (UART_Type *base, uint32_t mask)
  Disables the UART interrupts according to the provided mask.
- uint32_t UART_GetEnabledInterrupts (UART_Type *base)
  Gets the enabled UART interrupts.

Bus Operations

- static void UART_EnableTx (UART_Type *base, bool enable)
  Enables or disables the UART transmitter.
- static void UART_EnableRx (UART_Type *base, bool enable)
  Enables or disables the UART receiver.
- static void UART_WriteByte (UART_Type *base, uint8_t data)
  Writes to the TX register.
- static uint8_t UART_ReadByte (UART_Type *base)
  Reads the RX register directly.
- void UART_WriteBlocking (UART_Type *base, const uint8_t *data, size_t length)
  Writes to the TX register using a blocking method.
- status_t UART_ReadBlocking (UART_Type *base, uint8_t *data, size_t length)
  Read RX data register using a blocking method.

Transactional

- void UART_TransferCreateHandle (UART_Type *base, uart_handle_t *handle, uart_transfer_callback_t callback, void *userData)
  Initializes the UART handle.
- void UART_TransferStartRingBuffer (UART_Type *base, uart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void UART_TransferStopRingBuffer (UART_Type *base, uart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- status_t UART_TransferSendNonBlocking (UART_Type *base, uart_handle_t *handle, uart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void UART_TransferAbortSend (UART_Type *base, uart_handle_t *handle)
  Aborts the interrupt driven data transmit.
- status_t UART_TransferGetSendCount (UART_Type *base, uart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been written to UART TX register.
- status_t UART_TransferReceiveNonBlocking (UART_Type *base, uart_handle_t *handle, uart_transfer_t *xfer, size_t *receivedBytes)
  Receives a buffer of data using an interrupt method.
- void UART_TransferAbortReceive (UART_Type *base, uart_handle_t *handle)
  Aborts the interrupt-driven data receiving.
- status_t UART_TransferGetReceiveCount (UART_Type *base, uart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been received.
- void UART_TransferHandleIRQ (UART_Type *base, uart_handle_t *handle)
  UART IRQ handle function.
- void UART_TransferHandleErrorIRQ (UART_Type *base, uart_handle_t *handle)
  UART Error IRQ handle function.

53.2.3 Data Structure Documentation

53.2.3.1 struct uart_config_t

Data Fields

- uint32_t baudRate_Bps
  UART baud rate.
- uart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- bool enableTx
  Enable TX.
- bool enableRx
  Enable RX.

53.2.3.2 struct uart_transfer_t

Data Fields

- uint8_t * data
  The buffer of data to be transfer.
- size_t dataSize
  The byte count to be transfer.

53.2.3.2.0.12 Field Documentation

53.2.3.2.0.12.1 uint8_t* uart_transfer_t::data

53.2.3.2.0.12.2 size_t uart_transfer_t::dataSize

53.2.3.3 struct _uart_handle

Data Fields

- uint8_t *volatile txData
  Address of remaining data to send.
- volatile size_t txDataSize
  Size of the remaining data to send.
- size_t txDataSizeAll
  Size of the data to send out.
- uint8_t *volatile rxData
  Address of remaining data to receive.
- volatile size_t rxDataSize
  Size of the remaining data to receive.
- size_t rxDataSizeAll
  Size of the data to receive.
- uint8_t * rxRingBuffer
  Start address of the receiver ring buffer.
- size_t rxRingBufferSize

Size of the ring buffer.

- volatile uint16_t rxRingBufferHead
  Index for the driver to store received data into ring buffer.
- volatile uint16_t rxRingBufferTail
  Index for the user to get data from the ring buffer.
- uart_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

53.2.3.3.0.13 Field Documentation

53.2.3.3.0.13.1 uint8_t* volatile uart_handle_t::txData

53.2.3.3.0.13.2 volatile size_t uart_handle_t::txDataSize

53.2.3.3.0.13.3 size_t uart_handle_t::txDataSizeAll

53.2.3.3.0.13.4 uint8_t* volatile uart_handle_t::rxData

53.2.3.3.0.13.5 volatile size_t uart_handle_t::rxDataSize

53.2.3.3.0.13.6 size_t uart_handle_t::rxDataSizeAll

53.2.3.3.0.13.7 uint8_t* uart_handle_t::rxRingBuffer

53.2.3.3.0.13.8 size_t uart_handle_t::rxRingBufferSize

53.2.3.3.0.13.9 volatile uint16_t uart_handle_t::rxRingBufferHead

53.2.3.3.0.13.10 volatile uint16_t uart_handle_t::rxRingBufferTail

53.2.3.3.0.13.11 uart_transfer_callback_t uart_handle_t::callback

53.2.3.3.0.13.12 void* uart_handle_t::userData

53.2.3.3.0.13.13 volatile uint8_t uart_handle_t::txState

53.2.4 Macro Definition Documentation

53.2.4.1 #define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

53.2.5 Typedef Documentation

53.2.5.1 typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)

53.2.6 Enumeration Type Documentation

53.2.6.1 enum _uart_status

Enumerator

kStatus_UART_TxBusy Transmitter is busy.
kStatus_UART_RxBusy Receiver is busy.
kStatus_UART_TxIdle UART transmitter is idle.
kStatus_UART_RxIdle UART receiver is idle.
kStatus_UART_TxWatermarkTooLarge TX FIFO watermark too large.

kStatus_UART_RxWatermarkTooLarge RX FIFO watermark too large.
kStatus_UART_FlagCannotClearManually UART flag can't be manually cleared.
kStatus_UART_Error Error happens on UART.
kStatus_UART_RxRingBufferOverrun UART RX software ring buffer overrun.
kStatus_UART_RxHardwareOverrun UART RX receiver overrun.
kStatus_UART_NoiseError UART noise error.
kStatus_UART_FramingError UART framing error.
kStatus_UART_ParityError UART parity error.

53.2.6.2 enum uart_parity_mode_t

Enumerator

kUART_ParityDisabled Parity disabled.
kUART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kUART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

53.2.6.3 enum uart_stop_bit_count_t

Enumerator

kUART_OneStopBit One stop bit.
kUART_TwoStopBit Two stop bits.

53.2.6.4 enum _uart_interrupt_enable

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

Enumerator

kUART_RxActiveEdgeInterruptEnable RX active edge interrupt.
kUART_TxDataRegEmptyInterruptEnable Transmit data register empty interrupt.
kUART_TransmissionCompleteInterruptEnable Transmission complete interrupt.
kUART_RxDataRegFullInterruptEnable Receiver data register full interrupt.
kUART_IdleLineInterruptEnable Idle line interrupt.
kUART_RxOverrunInterruptEnable Receiver overrun interrupt.
kUART_NoiseErrorInterruptEnable Noise error flag interrupt.
kUART_FramingErrorInterruptEnable Framing error flag interrupt.
kUART_ParityErrorInterruptEnable Parity error flag interrupt.

53.2.6.5 enum _uart_flags

This provides constants for the UART status flags for use in the UART functions.

Enumerator

kUART_TxDataRegEmptyFlag TX data register empty flag.
kUART_TransmissionCompleteFlag Transmission complete flag.
kUART_RxDataRegFullFlag RX data register full flag.
kUART_IdleLineFlag Idle line detect flag.
kUART_RxOverrunFlag RX overrun flag.
kUART_NoiseErrorFlag RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kUART_ParityErrorFlag If parity enabled, sets upon parity error detection.
kUART_RxActiveEdgeFlag RX pin active edge interrupt flag, sets when active edge detected.
kUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.

53.2.7 Function Documentation

53.2.7.1 void UART_Init (UART_Type * *base*, const uart_config_t * *config*, uint32_t *srcClock_Hz*)

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the UART_GetDefaultConfig() function. Example below shows how to use this API to configure UART.

```
uart_config_t uartConfig;
uartConfig.baudRate_Bps = 115200U;
uartConfig.parityMode = kUART_ParityDisabled;
uartConfig.stopBitCount = kUART_OneStopBit;
uartConfig.txFifoWatermark = 0;
uartConfig.rxFifoWatermark = 1;
UART_Init(UART1, &uartConfig, 20000000U);
```

Parameters

base	UART peripheral base address.
config	Pointer to user-defined configuration structure.
srcClock_Hz	UART clock source frequency in HZ.

53.2.7.2 void UART_Deinit (UART_Type * *base*)

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

base	UART peripheral base address.

53.2.7.3 void UART_GetDefaultConfig (uart_config_t * *config*)

This function initializes the UART configuration structure to a default value. The default values are: uartConfig->baudRate_Bps = 115200U; uartConfig->bitCountPerChar = kUART_8BitsPerChar; uartConfig->parityMode = kUART_ParityDisabled; uartConfig->stopBitCount = kUART_OneStopBit; uartConfig->txFifoWatermark = 0; uartConfig->rxFifoWatermark = 1; uartConfig->enableTx = false; uartConfig->enableRx = false;

Parameters

config	Pointer to configuration structure.

53.2.7.4 void UART_SetBaudRate (UART_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the UART_Init.

```
UART_SetBaudRate(UART1, 115200U, 20000000U);
```

Parameters

base	UART peripheral base address.
baudRate_Bps	UART baudrate to be set.
srcClock_Hz	UART clock source freqency in HZ.

53.2.7.5 uint32_t UART_GetStatusFlags (UART_Type * *base*)

This function get all UART status flags, the flags are returned as the logical OR value of the enumerators _uart_flags. To check specific status, compare the return value with enumerators in _uart_flags. For example, to check whether the TX is empty:

```
if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
{
    ...
}
```

Parameters

base	UART peripheral base address.

Returns

UART status flags which are ORed by the enumerators in the _uart_flags.

53.2.7.6 status_t UART_ClearStatusFlags (UART_Type * *base*, uint32_t *mask*)

This function clears UART status flags with a provided mask. Automatically cleared flag can't be cleared by this function. Some flags can only be cleared or set by hardware itself. These flags are: kUART_TxDataRegEmptyFlag, kUART_TransmissionCompleteFlag, kUART_RxDataRegFullFlag, kUART_RxActiveFlag, kUART_NoiseErrorInRxDataRegFlag, kUART_ParityErrorInRxDataRegFlag, kUART_TxFifoEmptyFlag,kUART_RxFifoEmptyFlag Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

base	UART peripheral base address.
mask	The status flags to be cleared, it is logical OR value of _uart_flags.

Return values

kStatus_UART_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.

53.2.7.7 void UART_EnableInterrupts (UART_Type * *base*, uint32_t *mask*)

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to enable TX empty interrupt and RX full interrupt:

```
UART_EnableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
```

Parameters

base	UART peripheral base address.
mask	The interrupts to enable. Logical OR of _uart_interrupt_enable.

53.2.7.8 void UART_DisableInterrupts (UART_Type * *base*, uint32_t *mask*)

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _uart_interrupt_enable. For example, to disable TX empty interrupt and RX full interrupt:

```
UART_DisableInterrupts(UART1,
      kUART_TxDataRegEmptyInterruptEnable |
      kUART_RxDataRegFullInterruptEnable);
```

Parameters

base	UART peripheral base address.
mask	The interrupts to disable. Logical OR of _uart_interrupt_enable.

53.2.7.9 uint32_t UART_GetEnabledInterrupts (UART_Type * *base*)

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators _uart_interrupt_enable. To check specific interrupts enable status, compare the return value with enumerators in _uart_interrupt_enable. For example, to check whether TX empty interrupt is enabled:

```
uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);

if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
{
    ...
}
```

Parameters

base	UART peripheral base address.

Returns

UART interrupt flags which are logical OR of the enumerators in _uart_interrupt_enable.

53.2.7.10 static void UART_EnableTx (UART_Type * *base,* bool *enable*) [inline], [static]

This function enables or disables the UART transmitter.

Parameters

base	UART peripheral base address.
enable	True to enable, false to disable.

53.2.7.11 static void UART_EnableRx (UART_Type * *base,* bool *enable*) [inline], [static]

This function enables or disables the UART receiver.

Parameters

base	UART peripheral base address.
enable	True to enable, false to disable.

53.2.7.12 static void UART_WriteByte (UART_Type * *base,* uint8_t *data*) [inline], [static]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

base	UART peripheral base address.
data	The byte to write.

53.2.7.13 static uint8_t UART_ReadByte (UART_Type * *base*) [inline], [static]

This function reads data from the TX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

base	UART peripheral base address.

Returns

The byte read from UART data register.

53.2.7.14 void UART_WriteBlocking (UART_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Note

This function does not check whether all the data has been sent out to the bus. Before disabling the TX, check kUART_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	UART peripheral base address.
data	Start address of the data to write.
length	Size of the data to write.

53.2.7.15 status_t UART_ReadBlocking (UART_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data and read data from the TX register.

Parameters

base	UART peripheral base address.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_UART_RxHardwareOverrun	Receiver overrun happened while receiving data.
kStatus_UART_NoiseError	Noise error happened while receiving data.

kStatus_UART_Framing-Error	Framing error happened while receiving data.
kStatus_UART_Parity-Error	Parity error happened while receiving data.
kStatus_Success	Successfully received all data.

53.2.7.16 void UART_TransferCreateHandle (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_callback_t *callback*, void * *userData*)

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

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

53.2.7.17 void UART_TransferStartRingBuffer (UART_Type * *base*, uart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the UART_TransferReceiveNonBlocking() API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
ringBuffer	Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
ringBufferSize	size of the ring buffer.

53.2.7.18 void UART_TransferStopRingBuffer (UART_Type * *base,* uart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

53.2.7.19 status_t UART_TransferSendNonBlocking (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_t * *xfer*)

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the kStatus_UART_TxIdle as status parameter.

Note

The kStatus_UART_TxIdle is passed to the upper layer when all data is written to the TX register. However it does not ensure that all data are sent out. Before disabling the TX, check the kUART_-TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART transfer structure. See uart_transfer_t.

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_UART_TxBusy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	Invalid argument.

53.2.7.20 void UART_TransferAbortSend (UART_Type * *base*, uart_handle_t * *handle*)

This function aborts the interrupt driven data sending. The user can get the remainBytes to find out how many bytes are still not sent out.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

53.2.7.21 status_t UART_TransferGetSendCount (UART_Type * *base*, uart_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to UART TX register by interrupt method.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.2.7.22 status_t UART_TransferReceiveNonBlocking (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_t * *xfer*, size_t * *receivedBytes*)

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter kStatus_UART_RxIdle. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the xfer->data and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the xfer->data[5]. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the xfer->data. When all data is received, the upper layer is notified.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART transfer structure, refer to uart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into transmit queue.
kStatus_UART_RxBusy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

53.2.7.23 void UART_TransferAbortReceive (UART_Type * *base*, uart_handle_t * *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to know how many bytes not received yet.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

53.2.7.24 status_t UART_TransferGetReceiveCount (UART_Type * *base*, uart_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.2.7.25 void UART_TransferHandleIRQ (UART_Type * *base*, uart_handle_t * *handle*)

This function handles the UART transmit and receive IRQ request.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

53.2.7.26 void UART_TransferHandleErrorIRQ (UART_Type * *base*, uart_handle_t * *handle*)

This function handle the UART error IRQ request.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.

53.2.8 UART DMA Driver

53.2.8.1 Overview

Files

- file fsl_uart_dma.h

Data Structures

- struct uart_dma_handle_t
  UART DMA handle. More...

Typedefs

- typedef void(∗ uart_dma_transfer_callback_t)(UART_Type ∗base, uart_dma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

EDMA transactional

- void UART_TransferCreateHandleDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txDmaHandle, dma_handle_t ∗rxDmaHandle)
  Initializes the UART handle which is used in transactional functions and sets the callback.
- status_t UART_TransferSendDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Sends data using DMA.
- status_t UART_TransferReceiveDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Receives data using DMA.
- void UART_TransferAbortSendDMA (UART_Type ∗base, uart_dma_handle_t ∗handle)
  Aborts the send data using DMA.
- void UART_TransferAbortReceiveDMA (UART_Type ∗base, uart_dma_handle_t ∗handle)
  Aborts the received data using DMA.
- status_t UART_TransferGetSendCountDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been written to UART TX register.
- status_t UART_TransferGetReceiveCountDMA (UART_Type ∗base, uart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

53.2.8.2 Data Structure Documentation

53.2.8.2.1 struct _uart_dma_handle

Data Fields

- UART_Type * base
  UART peripheral base address.
- uart_dma_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- size_t rxDataSizeAll
  Size of the data to receive.
- size_t txDataSizeAll
  Size of the data to send out.
- dma_handle_t * txDmaHandle
  The DMA TX channel used.
- dma_handle_t * rxDmaHandle
  The DMA RX channel used.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

53.2.8.2.1.1 Field Documentation

53.2.8.2.1.1.1 UART_Type* uart_dma_handle_t::base

53.2.8.2.1.1.2 uart_dma_transfer_callback_t uart_dma_handle_t::callback

53.2.8.2.1.1.3 void* uart_dma_handle_t::userData

53.2.8.2.1.1.4 size_t uart_dma_handle_t::rxDataSizeAll

53.2.8.2.1.1.5 size_t uart_dma_handle_t::txDataSizeAll

53.2.8.2.1.1.6 dma_handle_t* uart_dma_handle_t::txDmaHandle

53.2.8.2.1.1.7 dma_handle_t* uart_dma_handle_t::rxDmaHandle

53.2.8.2.1.1.8 volatile uint8_t uart_dma_handle_t::txState

53.2.8.3 Typedef Documentation

53.2.8.3.1 typedef void(* uart_dma_transfer_callback_t)(UART_Type *base, uart_dma_handle_t *handle, status_t status, void *userData)

53.2.8.4 Function Documentation

53.2.8.4.1 void UART_TransferCreateHandleDMA (UART_Type * *base,* uart_dma_handle_t * *handle,* uart_dma_transfer_callback_t *callback,* void * *userData,* dma_handle_t * *txDmaHandle,* dma_handle_t * *rxDmaHandle*)

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.
callback	UART callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested DMA handle for RX DMA transfer.
txDmaHandle	User requested DMA handle for TX DMA transfer.

53.2.8.4.2 status_t UART_TransferSendDMA (UART_Type * *base*, uart_dma_handle_t * *handle*, uart_transfer_t * *xfer*)

This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART DMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

53.2.8.4.3 status_t UART_TransferReceiveDMA (UART_Type * *base*, uart_dma_handle_t * *handle*, uart_transfer_t * *xfer*)

This function receives data using DMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.
xfer	UART DMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

53.2.8.4.4 void UART_TransferAbortSendDMA (UART_Type * *base*, uart_dma_handle_t * *handle*)

This function aborts the sent data using DMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.

53.2.8.4.5 void UART_TransferAbortReceiveDMA (UART_Type * *base*, uart_dma_handle_t * *handle*)

This function abort receive data which using DMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_dma_handle_t structure.

53.2.8.4.6 status_t UART_TransferGetSendCountDMA (UART_Type * *base*, uart_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to UART TX register by DMA.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.2.8.4.7 status_t UART_TransferGetReceiveCountDMA (UART_Type * *base*, uart_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.2.9 UART eDMA Driver

53.2.9.1 Overview

Files

- file fsl_uart_edma.h

Data Structures

- struct uart_edma_handle_t
  UART eDMA handle. More...

Typedefs

- typedef void(∗ uart_edma_transfer_callback_t)(UART_Type ∗base, uart_edma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

eDMA transactional

- void UART_TransferCreateHandleEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_edma_transfer_callback_t callback, void ∗userData, edma_handle_t ∗txEdmaHandle, edma_handle_t ∗rxEdmaHandle)
  Initializes the UART handle which is used in transactional functions.
- status_t UART_SendEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Sends data using eDMA.
- status_t UART_ReceiveEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uart_transfer_t ∗xfer)
  Receive data using eDMA.
- void UART_TransferAbortSendEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle)
  Aborts the sent data using eDMA.
- void UART_TransferAbortReceiveEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle)
  Aborts the receive data using eDMA.
- status_t UART_TransferGetSendCountEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been written to UART TX register.
- status_t UART_TransferGetReceiveCountEDMA (UART_Type ∗base, uart_edma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

53.2.9.2 Data Structure Documentation

53.2.9.2.1 struct _uart_edma_handle

Data Fields

- uart_edma_transfer_callback_t callback
  Callback function.
- void * userData
  UART callback function parameter.
- size_t rxDataSizeAll
  Size of the data to receive.
- size_t txDataSizeAll
  Size of the data to send out.
- edma_handle_t * txEdmaHandle
  The eDMA TX channel used.
- edma_handle_t * rxEdmaHandle
  The eDMA RX channel used.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.

53.2.9.2.1.1 Field Documentation

53.2.9.2.1.1.1 uart_edma_transfer_callback_t uart_edma_handle_t::callback

53.2.9.2.1.1.2 void* uart_edma_handle_t::userData

53.2.9.2.1.1.3 size_t uart_edma_handle_t::rxDataSizeAll

53.2.9.2.1.1.4 size_t uart_edma_handle_t::txDataSizeAll

53.2.9.2.1.1.5 edma_handle_t* uart_edma_handle_t::txEdmaHandle

53.2.9.2.1.1.6 edma_handle_t* uart_edma_handle_t::rxEdmaHandle

53.2.9.2.1.1.7 volatile uint8_t uart_edma_handle_t::txState

53.2.9.3 Typedef Documentation

53.2.9.3.1 typedef void(* uart_edma_transfer_callback_t)(UART_Type *base, uart_edma_handle_t *handle, status_t status, void *userData)

53.2.9.4 Function Documentation

53.2.9.4.1 void UART_TransferCreateHandleEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *txEdmaHandle*, edma_handle_t * *rxEdmaHandle*)

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.
callback	UART callback, NULL means no callback.
userData	User callback function data.
rxEdmaHandle	User requested DMA handle for RX DMA transfer.
txEdmaHandle	User requested DMA handle for TX DMA transfer.

53.2.9.4.2 status_t UART_SendEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_transfer_t * *xfer*)

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
xfer	UART eDMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

53.2.9.4.3 status_t UART_ReceiveEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_transfer_t * *xfer*)

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.
xfer	UART eDMA transfer structure. See uart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_UART_RxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

53.2.9.4.4 void UART_TransferAbortSendEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*)

This function aborts sent data using eDMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.

53.2.9.4.5 void UART_TransferAbortReceiveEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*)

This function aborts receive data using eDMA.

Parameters

base	UART peripheral base address.
handle	Pointer to uart_edma_handle_t structure.

53.2.9.4.6 status_t UART_TransferGetSendCountEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to UART TX register by DMA.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Send bytes count.

Return values

kStatus_NoTransferInProgress	No send in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.2.9.4.7 status_t UART_TransferGetReceiveCountEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	UART peripheral base address.
handle	UART handle pointer.
count	Receive bytes count.

Return values

kStatus_NoTransferInProgress	No receive in progress.
kStatus_InvalidArgument	Parameter is invalid.
kStatus_Success	Get successfully through the parameter `count`;

53.3 UART FreeRTOS Driver

53.3.1 Overview

Files

- file fsl_uart_freertos.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

Driver version

- #define FSL_UART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  UART FreeRTOS driver version 2.0.0.

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t *handle, uart_handle_t *t_handle, const struct rtos_uart_config *cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t *handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

53.3.2 Data Structure Documentation

53.3.2.1 struct rtos_uart_config

Data Fields

- UART_Type * base

UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity
  Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

53.3.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

53.3.3 Macro Definition Documentation

53.3.3.1 #define FSL_UART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

53.3.4 Function Documentation

53.3.4.1 int UART_RTOS_Init (uart_rtos_handle_t * *handle*, uart_handle_t * *t_handle*, const struct rtos_uart_config * *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
t_handle	The pointer to allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 succeed, others fail.

53.3.4.2 int UART_RTOS_Deinit (uart_rtos_handle_t * *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

53.3.4.3 int UART_RTOS_Send (uart_rtos_handle_t * *handle*, const uint8_t * *buffer*, uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

53.3.4.4 int UART_RTOS_Receive (uart_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

This function receives data from UART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to a variable of size_t where the number of received data is filled.

53.4 UART µCOS/II Driver

53.4.1 Overview

Files

- file fsl_uart_ucosii.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

Driver version

- #define FSL_UART_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  UART µCOS-II driver version 2.0.0.

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t *handle, uart_handle_t *t_handle, const struct rtos_uart_config *cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t *handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

53.4.2 Data Structure Documentation

53.4.2.1 struct rtos_uart_config

Data Fields

- UART_Type * base

UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity
  Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

53.4.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

53.4.3 Macro Definition Documentation

53.4.3.1 #define FSL_UART_UCOSII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

53.4.4 Function Documentation

53.4.4.1 int UART_RTOS_Init (uart_rtos_handle_t * *handle*, uart_handle_t * *t_handle*, const struct rtos_uart_config * *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
uart_t_handle	The pointer to allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 Succeed, others fail.

53.4.4.2 int UART_RTOS_Deinit (uart_rtos_handle_t * *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

53.4.4.3 int UART_RTOS_Send (uart_rtos_handle_t * *handle,* const uint8_t * *buffer,* uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

53.4.4.4 int UART_RTOS_Receive (uart_rtos_handle_t * *handle,* uint8_t * *buffer,* uint32_t *length,* size_t * *received*)

This function receives data from UART. It is a synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to a variable of size_t where the number of received data is filled.

53.5 UART µCOS/III Driver

53.5.1 Overview

Files

- file fsl_uart_ucosiii.h

Data Structures

- struct rtos_uart_config
  UART configuration structure. More...
- struct uart_rtos_handle_t
  UART FreeRTOS handle. More...

Driver version

- #define FSL_UART_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  UART µCOS-III driver version 2.0.0.

UART RTOS Operation

- int UART_RTOS_Init (uart_rtos_handle_t *handle, uart_handle_t *t_handle, const struct rtos_uart_config *cfg)
  Initializes a UART instance for operation in RTOS.
- int UART_RTOS_Deinit (uart_rtos_handle_t *handle)
  Deinitializes a UART instance for operation.

UART transactional Operation

- int UART_RTOS_Send (uart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
  Sends data in the background.
- int UART_RTOS_Receive (uart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

53.5.2 Data Structure Documentation

53.5.2.1 struct rtos_uart_config

Data Fields

- UART_Type * base

UART base address.
- uint32_t srcclk
  UART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- uart_parity_mode_t parity
  Parity setting.
- uart_stop_bit_count_t stopbits
  Number of stop bits to use.
- uint8_t * buffer
  Buffer for background reception.
- uint32_t buffer_size
  Size of buffer for background reception.

53.5.2.2 struct uart_rtos_handle_t

Data Fields

- UART_Type * base
  UART base address.
- struct _uart_transfer tx_xfer
  TX transfer structure.
- struct _uart_transfer rx_xfer
  RX transfer structure.
- SemaphoreHandle_t rx_sem
  RX semaphore for resource sharing.
- SemaphoreHandle_t tx_sem
  TX semaphore for resource sharing.
- EventGroupHandle_t rx_event
  RX completion event.
- EventGroupHandle_t tx_event
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.
- OS_EVENT * rx_sem
  RX semaphore for resource sharing.
- OS_EVENT * tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP * rx_event
  RX completion event.
- OS_FLAG_GRP * tx_event
  TX completion event.
- OS_SEM rx_sem
  RX semaphore for resource sharing.
- OS_SEM tx_sem
  TX semaphore for resource sharing.
- OS_FLAG_GRP rx_event
  RX completion event.
- OS_FLAG_GRP tx_event
  TX completion event.

53.5.3 Macro Definition Documentation

53.5.3.1 #define FSL_UART_UCOSIII_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

53.5.4 Function Documentation

53.5.4.1 int UART_RTOS_Init (uart_rtos_handle_t * *handle*, uart_handle_t * *t_handle*, const struct rtos_uart_config * *cfg*)

Parameters

handle	The RTOS UART handle, the pointer to allocated space for RTOS context.
uart_t_handle	The pointer to an allocated space where to store transactional layer internal state.
cfg	The pointer to the parameters required to configure the UART after initialization.

Returns

0 Succeed, others fail.

53.5.4.2 int UART_RTOS_Deinit (uart_rtos_handle_t * *handle*)

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS UART handle.

53.5.4.3 int UART_RTOS_Send (uart_rtos_handle_t * *handle*, const uint8_t * *buffer*, uint32_t *length*)

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

53.5.4.4 int UART_RTOS_Receive (uart_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

This function receives data from UART. It is a synchronous API. If any data is immediately available, it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS UART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to variable of a size_t where the number of received data is filled.

Chapter 54
VREF: Voltage Reference Driver

54.1 Overview

The KSDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of Kinetis devices.

54.2 Overview

The Voltage Reference(VREF) is intended to supply 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.

54.2.1 VREF functional Operation

To configure the VREF driver, configure vref_config_t structure in one of two ways.

1. Use the VREF_GetDefaultConfig() function.
2. Sets the parameter in 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.

54.3 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
```

Files

- file fsl_vref.h

Data Structures

- struct vref_config_t
  The description structure for the VREF module. *More...*

Enumerations

- enum vref_buffer_mode_t {
  kVREF_ModeBandgapOnly = 0U,
  kVREF_ModeTightRegulationBuffer = 2U }
  VREF modes.

Driver version

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

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 reference voltage.
- static uint8_t VREF_GetTrimVal (VREF_Type *base)
  Reads the value of the TRIM meaning output voltage.

54.4 Data Structure Documentation

54.4.1 struct vref_config_t

Data Fields

- vref_buffer_mode_t bufferMode
  Buffer mode selection.

54.5 Macro Definition Documentation

54.5.1 #define FSL_VREF_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

54.6 Enumeration Type Documentation

54.6.1 enum vref_buffer_mode_t

Enumerator

kVREF_ModeBandgapOnly Bandgap on only, for stabilization and startup.
kVREF_ModeTightRegulationBuffer Tight regulation buffer enabled.

54.7 Function Documentation

54.7.1 void VREF_Init (VREF_Type * *base,* const vref_config_t * *config*)

This function must be called before calling all the 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: 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.

54.7.2 void VREF_Deinit (VREF_Type * *base*)

This function should be called to shut down the module. Example:

```
vref_config_t vrefUserConfig;
VREF_Init(VREF);
VREF_GetDefaultConfig(&vrefUserConfig);
...
VREF_Deinit(VREF);
```

Parameters

base	VREF peripheral address.

54.7.3 void VREF_GetDefaultConfig (vref_config_t * *config*)

This function initializes the VREF configuration structure to a default value. Example:

```
vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
vrefConfig->enableExternalVoltRef = false;
vrefConfig->enableLowRef = false;
```

Parameters

config	Pointer to the initialization structure.

54.7.4 void VREF_SetTrimVal (VREF_Type * *base,* uint8_t *trimValue*)

This function sets a TRIM value for 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)).

54.7.5 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 55
WDOG: Watchdog Timer Driver

55.1 Overview

The KSDK provides a peripheral driver for the Watchdog module (WDOG) of Kinetis devices.

Typical use case

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG_Init(wdog_base,&config);
```

Files

- file fsl_wdog.h

Data Structures

- struct wdog_work_mode_t
  Defines WDOG work mode. *More...*
- struct wdog_config_t
  Describes WDOG configuration structure. *More...*
- struct wdog_test_config_t
  Describes WDOG test mode configuration structure. *More...*

Enumerations

- enum wdog_clock_source_t {
  kWDOG_LpoClockSource = 0U,
  kWDOG_AlternateClockSource = 1U }
  Describes WDOG clock source.
- enum wdog_clock_prescaler_t {
  kWDOG_ClockPrescalerDivide1 = 0x0U,
  kWDOG_ClockPrescalerDivide2 = 0x1U,
  kWDOG_ClockPrescalerDivide3 = 0x2U,
  kWDOG_ClockPrescalerDivide4 = 0x3U,
  kWDOG_ClockPrescalerDivide5 = 0x4U,
  kWDOG_ClockPrescalerDivide6 = 0x5U,
  kWDOG_ClockPrescalerDivide7 = 0x6U,
  kWDOG_ClockPrescalerDivide8 = 0x7U }
  Describes the selection of the clock prescaler.

- enum wdog_test_mode_t {
  kWDOG_QuickTest = 0U,
  kWDOG_ByteTest = 1U }
  Describes WDOG test mode.
- enum wdog_tested_byte_t {
  kWDOG_TestByte0 = 0U,
  kWDOG_TestByte1 = 1U,
  kWDOG_TestByte2 = 2U,
  kWDOG_TestByte3 = 3U }
  Describes WDOG tested byte selection in byte test mode.
- enum _wdog_interrupt_enable_t { kWDOG_InterruptEnable = WDOG_STCTRLH_IRQRSTEN_-MASK }
  WDOG interrupt configuration structure, default settings all disabled.
- enum _wdog_status_flags_t {
  kWDOG_RunningFlag = WDOG_STCTRLH_WDOGEN_MASK,
  kWDOG_TimeoutFlag = WDOG_STCTRLL_INTFLG_MASK }
  WDOG status flags.

Driver version

- #define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Defines WDOG driver version 2.0.0.

Unlock sequence

- #define WDOG_FIRST_WORD_OF_UNLOCK (0xC520U)
  First word of unlock sequence.
- #define WDOG_SECOND_WORD_OF_UNLOCK (0xD928U)
  Second word of unlock sequence.

Refresh sequence

- #define WDOG_FIRST_WORD_OF_REFRESH (0xA602U)
  First word of refresh sequence.
- #define WDOG_SECOND_WORD_OF_REFRESH (0xB480U)
  Second word of refresh sequence.

WDOG Initialization and De-initialization

- void WDOG_GetDefaultConfig (wdog_config_t *config)
  Initializes WDOG configure sturcture.
- void WDOG_Init (WDOG_Type *base, const wdog_config_t *config)
  Initializes the WDOG.
- void WDOG_Deinit (WDOG_Type *base)
  Shuts down the WDOG.
- void WDOG_SetTestModeConfig (WDOG_Type *base, wdog_test_config_t *config)
  Configures WDOG functional test.

WDOG Functional Operation

- static void WDOG_Enable (WDOG_Type *base)
  Enables the WDOG module.
- static void WDOG_Disable (WDOG_Type *base)
  Disables the WDOG module.
- static void WDOG_EnableInterrupts (WDOG_Type *base, uint32_t mask)
  Enable WDOG interrupt.
- static void WDOG_DisableInterrupts (WDOG_Type *base, uint32_t mask)
  Disable WDOG interrupt.
- uint32_t WDOG_GetStatusFlags (WDOG_Type *base)
  Gets WDOG all status flags.
- void WDOG_ClearStatusFlags (WDOG_Type *base, uint32_t mask)
  Clear WDOG flag.
- static void WDOG_SetTimeoutValue (WDOG_Type *base, uint32_t timeoutCount)
  Set the WDOG timeout value.
- static void WDOG_SetWindowValue (WDOG_Type *base, uint32_t windowValue)
  Sets the WDOG window value.
- static void WDOG_Unlock (WDOG_Type *base)
  Unlocks the WDOG register written.
- void WDOG_Refresh (WDOG_Type *base)
  Refreshes the WDOG timer.
- static uint16_t WDOG_GetResetCount (WDOG_Type *base)
  Gets the WDOG reset count.
- static void WDOG_ClearResetCount (WDOG_Type *base)
  Clears the WDOG reset count.

55.2 Data Structure Documentation

55.2.1 struct wdog_work_mode_t

Data Fields

- bool enableStop
  Enables or disables WDOG in stop mode.
- bool enableDebug
  Enables or disables WDOG in debug mode.

55.2.2 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.

55.2.3 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.

55.3 Macro Definition Documentation

55.3.1 #define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

55.4 Enumeration Type Documentation

55.4.1 enum wdog_clock_source_t

Enumerator

kWDOG_LpoClockSource WDOG clock sourced from LPO.
kWDOG_AlternateClockSource WDOG clock sourced from alternate clock source.

55.4.2 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.

55.4.3 enum wdog_test_mode_t

Enumerator

kWDOG_QuickTest Selects quick test.
kWDOG_ByteTest Selects byte test.

55.4.4 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.

55.4.5 enum _wdog_interrupt_enable_t

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

Enumerator

kWDOG_InterruptEnable WDOG timeout will generate interrupt before reset.

55.4.6 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.

55.5 Function Documentation

55.5.1 void WDOG_GetDefaultConfig (wdog_config_t * *config*)

This function initializes the WDOG configure structure to default value. The default value are:

```
wdogConfig->enableWdog = true;
wdogConfig->clockSource = kWDOG_LpoClockSource;
wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
wdogConfig->workMode.enableWait = true;
wdogConfig->workMode.enableStop = false;
wdogConfig->workMode.enableDebug = false;
wdogConfig->enableUpdate = true;
wdogConfig->enableInterrupt = false;
wdogConfig->enableWindowMode = false;
wdogConfig->windowValue = 0;
wdogConfig->timeoutValue = 0xFFFFU;
```

Parameters

config	Pointer to WDOG config structure.

See Also

wdog_config_t

55.5.2 void WDOG_Init (WDOG_Type * *base*, const wdog_config_t * *config*)

This function initializes the WDOG. When called, the WDOG runs according to the configuration. If user wants to reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in configuration.

Example:

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableUpdate = true;
WDOG_Init(wdog_base,&config);
```

Parameters

base	WDOG peripheral base address

config	The configuration of WDOG

55.5.3 void WDOG_Deinit (WDOG_Type * *base*)

This function shuts down the WDOG. Make sure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

55.5.4 void WDOG_SetTestModeConfig (WDOG_Type * *base*, wdog_test_config_t * *config*)

This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Make sure that the WDOG_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

Example:

```
wdog_test_config_t test_config;
test_config.testMode = kWDOG_QuickTest;
test_config.timeoutValue = 0xffffu;
WDOG_SetTestModeConfig(wdog_base, &test_config);
```

Parameters

base	WDOG peripheral base address
config	The functional test configuration of WDOG

55.5.5 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

55.5.6 static void WDOG_Disable (WDOG_Type * *base*) [inline], [static]

This function write value into WDOG_STCTRLH register to disable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address

55.5.7 static void WDOG_EnableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function write value into WDOG_STCTRLH register to enable WDOG interrupt, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to enable The parameter can be combination of the following source if defined: • kWDOG_InterruptEnable

55.5.8 static void WDOG_DisableInterrupts (WDOG_Type * *base,* uint32_t *mask*) [inline], [static]

This function write value into WDOG_STCTRLH register to disable WDOG interrupt, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
mask	The interrupts to disable The parameter can be combination of the following source if defined: • kWDOG_InterruptEnable

55.5.9 uint32_t WDOG_GetStatusFlags (WDOG_Type * *base*)

This function gets all status flags.

Example for getting Running Flag:

```
uint32_t status;
```

```
status = WDOG_GetStatusFlags(wdog_base) & kWDOG_RunningFlag;
```

Parameters

base	WDOG peripheral base address

Returns

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

See Also

_wdog_status_flags_t
- true: related status flag has been set.
- false: related status flag is not set.

55.5.10 void WDOG_ClearStatusFlags (WDOG_Type * *base,* uint32_t *mask*)

This function clears WDOG status flag.

Example for clearing timeout(interrupt) flag:

```
WDOG_ClearStatusFlags(wdog_base,kWDOG_TimeoutFlag);
```

Parameters

base	WDOG peripheral base address
mask	The status flags to clear. The parameter could be any combination of the following values: kWDOG_TimeoutFlag

55.5.11 static void WDOG_SetTimeoutValue (WDOG_Type * *base,* uint32_t *timeoutCount*) [inline], [static]

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than 2xWCT time + 20 bus clock cycles. This function write value into WDOG_TOVALH and WDOG_TOVALL registers which are wirte-once. Make sure the WCT window is still open and these two registers have not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
timeoutCount	WDOG timeout value, count of WDOG clock tick.

55.5.12 static void WDOG_SetWindowValue (WDOG_Type * *base,* uint32_t *windowValue*) [inline], [static]

This function sets the WDOG window value. This function write value into WDOG_WINH and WDOG_WINL registers which are wirte-once. Make sure the WCT window is still open and these two registers have not been written in this WCT while this function is called.

Parameters

base	WDOG peripheral base address
windowValue	WDOG window value.

55.5.13 static void WDOG_Unlock (WDOG_Type * *base*) [inline], [static]

This function unlocks the WDOG register written. Before starting the unlock sequence and following congfiguration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire, After the configuration finishes, re-enable the global interrupts.

Parameters

base	WDOG peripheral base address

55.5.14 void WDOG_Refresh (WDOG_Type * *base*)

This function feeds the WDOG. This function should be called before WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

base	WDOG peripheral base address

55.5.15 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

55.5.16 static void WDOG_ClearResetCount (WDOG_Type * *base*) [inline], [static]

This function clears the WDOG reset count value.

Parameters

base	WDOG peripheral base address

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: KSDK20APIRM
Rev. 0
Jan 2016