SDK API Reference Manual v2.0.0

NXP Semiconductors

Document Number: KSDK20LPC5460XAPIRM
Rev. 0
Oct 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

Chapter 1
Introduction

The Software Development Kit (KSDK) v2.0 is a collection of software enablement, for NXP Kinetis Microcontrollers, that includes peripheral drivers, multicore support, USB stack, 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 provided by KSDK v2.0. The KEx Web UI is available to provide access to all SDK v2.0 packages. See the *SDK v.2.0.0 Release Notes* (document KSDK200RN) and the supported Devices section at www.nxp.com/ksdk for details.

The SDK v2.0 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.
- Peripheral drivers that provide stateless, high-performance, ease-of-use APIs. Communication drivers provide higher-level transactional APIs for a higher-performance option.
- 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.
  - The SDK v2.0 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:
  - IAR Embedded Workbench
  - Keil MDK
  - LPCXpresso IDE

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.

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.

Deliverable	Location
Examples	<install_dir>/examples/
Demo Applications	<install_dir>/examples/<board_name>/demo_apps/
Driver Examples	<install_dir>/examples/<board_name>/driver_examples/
Documentation	<install_dir>/doc/
USB Documentation	<install_dir>/doc/usb/
Middleware	<install_dir>/middleware/
USB Stack	<install_dir>/middleware/usb_<version>
Drivers	<install_dir>/<device_name>/drivers/
CMSIS Standard ARM Cortex-M Headers, math and DSP Libraries	<install_dir>/<device_name>/CMSIS/
Device Startup and Linker	<install_dir>/<device_name>/<toolchain>/
SDK Utilities	<install_dir>/<device_name>/utilities/
RTOS Kernels	<install_dir>/rtos/

Table 2: KSDK Folder Structure

Chapter 2
Driver errors status

- kStatus_DMA_Busy = 5000
- kStatus_ENET_RxFrameError = 4000
- kStatus_ENET_RxFrameFail = 4001
- kStatus_ENET_RxFrameEmpty = 4002
- kStatus_ENET_TxFrameBusy = 4003
- kStatus_ENET_TxFrameFail = 4004
- kStatus_ENET_TxFrameOverLen = 4005
- #kStatus_ENET_PtpTsRingFull = 4006
- #kStatus_ENET_PtpTsRingEmpty = 4007
- kStatus_SPI_Busy = 1400
- kStatus_SPI_Idle = 1401
- kStatus_SPI_Error = 1402
- kStatus_SPIFI_Busy = 5900
- kStatus_SPIFI_Idle = 5901
- kStatus_SPIFI_Error = 5902

Chapter 3
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 4
Architectural Overview

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

Overview

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

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

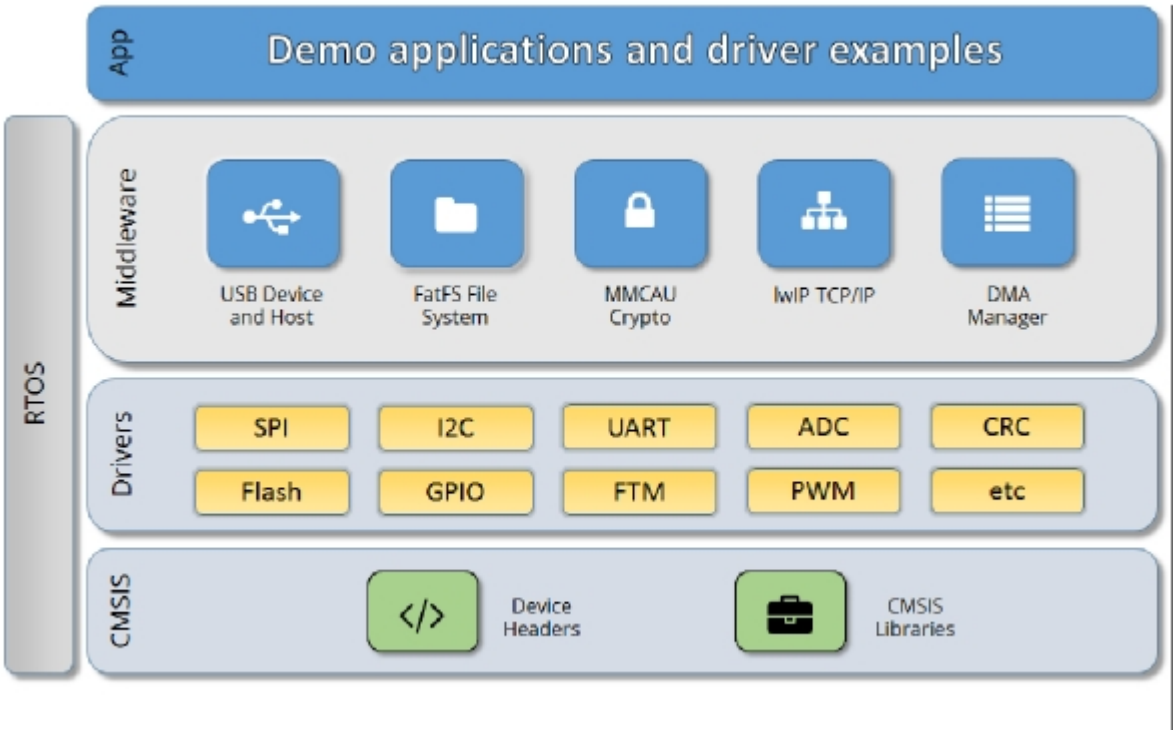

Figure 1: KSDK Block Diagram

Kinetis MCU header files

Each supported Kinetis MCU device in the KSDK has an overall System-on Chip (SoC) memory-mapped

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

CMSIS Support

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

KSDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
        PUBWEAK SPI0_IRQHandler
        PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

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

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

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

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

Feature Header Files

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

Application

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

Chapter 5
ADC: 12-bit SAR Analog-to-Digital Converter Driver

5.1 Overview

The SDK provides a Peripheral driver for the 12-bit SAR Analog-to-Digital Converter (ADC) module of LPC devices.

5.2 Typical use case

5.2.1 Polling Configuration

```
void main(void)
{
    adc_config_t adcConfigStruct;
    adc_conv_seq_config_t adcConvSeqConfigStruct;

    /* Enable the power and clock firstly. */
    ...

    /* Calibration. */
    if (ADC_DoSelfCalibration(DEMO_ADC_BASE))
    {
        PRINTF("ADC_DoSelfCalibration() Done.\r\n");
    }
    else
    {
        PRINTF("ADC_DoSelfCalibration() Failed.\r\n");
    }

    /* Configure the converter. */
    adcConfigStruct.clockMode = kADC_ClockSynchronousMode;
    adcConfigStruct.clockDividerNumber = 0;
    adcConfigStruct.resolution = kADC_Resolution12bit;
    adcConfigStruct.enableBypassCalibration = false;
    adcConfigStruct.sampleTimeNumber = 0U;
    ADC_Init(DEMO_ADC_BASE, &adcConfigStruct);

    /* Use the temperature sensor input to channel 0. */
    ADC_EnableTemperatureSensor(DEMO_ADC_BASE, true);

    /* Enable channel 0's conversion in Sequence A. */
    adcConvSeqConfigStruct.channelMask = (1U << 0); /* Includes channel 0. */
    adcConvSeqConfigStruct.triggerMask = 0U;
    adcConvSeqConfigStruct.triggerPolarity =
      kADC_TriggerPolarityNegativeEdge;
    adcConvSeqConfigStruct.enableSingleStep = false;
    adcConvSeqConfigStruct.enableSyncBypass = false;
    adcConvSeqConfigStruct.interruptMode =
      kADC_InterruptForEachSequence;
    ADC_SetConvSeqAConfig(DEMO_ADC_BASE, &adcConvSeqConfigStruct);
    ADC_EnableConvSeqA(DEMO_ADC_BASE, true); /* Enable the conversion sequence A. */

    PRINTF("Configuration Done.\r\n");

    while (1)
    {
        /* Get the input from terminal and trigger the converter by software. */
```

```
        GETCHAR();
        ADC_DoSoftwareTriggerConvSeqA(DEMO_ADC_BASE);

        /* Wait for the converter to be done. */
        while (!ADC_GetChannelConversionResult(DEMO_ADC_BASE,
      DEMO_ADC_SAMPLE_CHANNEL_NUMBER, &adcResultInfoStruct))
        {
        }
        PRINTF("adcResultInfoStruct.result        = %d\r\n", adcResultInfoStruct.result);
        PRINTF("adcResultInfoStruct.channelNumber = %d\r\n", adcResultInfoStruct.channelNumber);
        PRINTF("adcResultInfoStruct.overrunFlag   = %d\r\n", adcResultInfoStruct.overrunFlag ? 1U : 0U);
        PRINTF("\r\n");
    }
}
```

5.2.2 Interrupt Configuration

```
/* Global variables. */
static adc_result_info_t gAdcResultInfoStruct;
adc_result_info_t *volatile gAdcResultInfoPtr = &gAdcResultInfoStruct;
volatile bool gAdcConvSeqAIntFlag;

void main(void)
{
    adc_config_t adcConfigStruct;
    adc_conv_seq_config_t adcConvSeqConfigStruct;

    /* Enable the power and clock firstly. */
    ...

    /* Calibration. */
    if (ADC_DoSelfCalibration(DEMO_ADC_BASE))
    {
        PRINTF("ADC_DoSelfCalibration() Done.\r\n");
    }
    else
    {
        PRINTF("ADC_DoSelfCalibration() Failed.\r\n");
    }

    /* Configure the ADC as basic polling mode. */
    /* Configure the converter. */
    adcConfigStruct.clockMode = kADC_ClockSynchronousMode;
    adcConfigStruct.clockDividerNumber = 0;
    adcConfigStruct.resolution = kADC_Resolution12bit;
    adcConfigStruct.enableBypassCalibration = false;
    adcConfigStruct.sampleTimeNumber = 0U;
    ADC_Init(DEMO_ADC_BASE, &adcConfigStruct);

    /* Use the sensor input to channel 0. */
    ADC_EnableTemperatureSensor(DEMO_ADC_BASE, true);

    /* Enable channel 0's conversion in Sequence A. */
    adcConvSeqConfigStruct.channelMask = (1U << 0); /* Includes channel 0. */
    adcConvSeqConfigStruct.triggerMask = 0U;
    adcConvSeqConfigStruct.triggerPolarity =
      kADC_TriggerPolarityNegativeEdge;
    adcConvSeqConfigStruct.enableSingleStep = false;
    adcConvSeqConfigStruct.enableSyncBypass = false;
    adcConvSeqConfigStruct.interruptMode =
      kADC_InterruptForEachSequence;
    ADC_SetConvSeqAConfig(DEMO_ADC_BASE, &adcConvSeqConfigStruct);
    ADC_EnableConvSeqA(DEMO_ADC_BASE, true); /* Enable the conversion sequence A. */

    /* Enable the interrupt. */
```

```
    ADC_EnableInterrupts(DEMO_ADC_BASE,
                         kADC_ConvSeqAInterruptEnable); /* Enable the interrupt
       the for sequence A done. */
    NVIC_EnableIRQ(DEMO_ADC_IRQ_ID);

    PRINTF("Configuration Done.\r\n");

    while (1)
    {
        GETCHAR();
        gAdcConvSeqAIntFlag = false;
        ADC_DoSoftwareTriggerConvSeqA(DEMO_ADC_BASE);

        while (!gAdcConvSeqAIntFlag)
        {
        }
        PRINTF("gAdcResultInfoStruct.result        = %d\r\n", gAdcResultInfoStruct.
      result);
        PRINTF("gAdcResultInfoStruct.channelNumber = %d\r\n", gAdcResultInfoStruct.
      channelNumber);
        PRINTF("gAdcResultInfoStruct.overrunFlag   = %d\r\n", gAdcResultInfoStruct.
      overrunFlag ? 1U : 0U);
        PRINTF("\r\n");
    }
}

/*
   ISR for ADC conversion sequence A done.
 */
void DEMO_ADC_IRQ_HANDLER_FUNC(void)
{
    if (kADC_ConvSeqAInterruptFlag == (
      kADC_ConvSeqAInterruptFlag & ADC_GetStatusFlags(DEMO_ADC_BASE))
      )
    {
        ADC_GetChannelConversionResult(DEMO_ADC_BASE,
      DEMO_ADC_SAMPLE_CHANNEL_NUMBER, gAdcResultInfoPtr);
        ADC_ClearStatusFlags(DEMO_ADC_BASE,
      kADC_ConvSeqAInterruptFlag);
        gAdcConvSeqAIntFlag = true;
    }
}
```

Files

- file fsl_adc.h

Data Structures

- struct adc_config_t
  Define structure for configuring the block. More...
- struct adc_conv_seq_config_t
  Define structure for configuring conversion sequence. More...
- struct adc_result_info_t
  Define structure of keeping conversion result information. More...

Typical use case

Enumerations

- enum _adc_status_flags {
  kADC_ThresholdCompareFlagOnChn0 = 1U << 0U,
  kADC_ThresholdCompareFlagOnChn1 = 1U << 1U,
  kADC_ThresholdCompareFlagOnChn2 = 1U << 2U,
  kADC_ThresholdCompareFlagOnChn3 = 1U << 3U,
  kADC_ThresholdCompareFlagOnChn4 = 1U << 4U,
  kADC_ThresholdCompareFlagOnChn5 = 1U << 5U,
  kADC_ThresholdCompareFlagOnChn6 = 1U << 6U,
  kADC_ThresholdCompareFlagOnChn7 = 1U << 7U,
  kADC_ThresholdCompareFlagOnChn8 = 1U << 8U,
  kADC_ThresholdCompareFlagOnChn9 = 1U << 9U,
  kADC_ThresholdCompareFlagOnChn10 = 1U << 10U,
  kADC_ThresholdCompareFlagOnChn11 = 1U << 11U,
  kADC_OverrunFlagForChn0,
  kADC_OverrunFlagForChn1,
  kADC_OverrunFlagForChn2,
  kADC_OverrunFlagForChn3,
  kADC_OverrunFlagForChn4,
  kADC_OverrunFlagForChn5,
  kADC_OverrunFlagForChn6,
  kADC_OverrunFlagForChn7,
  kADC_OverrunFlagForChn8,
  kADC_OverrunFlagForChn9,
  kADC_OverrunFlagForChn10,
  kADC_OverrunFlagForChn11,
  kADC_GlobalOverrunFlagForSeqA = 1U << 24U,
  kADC_GlobalOverrunFlagForSeqB = 1U << 25U,
  kADC_ConvSeqAInterruptFlag = 1U << 28U,
  kADC_ConvSeqBInterruptFlag = 1U << 29U,
  kADC_ThresholdCompareInterruptFlag = 1U << 30U,
  kADC_OverrunInterruptFlag = 1U << 31U }
  Flags.
- enum _adc_interrupt_enable {
  kADC_ConvSeqAInterruptEnable = ADC_INTEN_SEQA_INTEN_MASK,
  kADC_ConvSeqBInterruptEnable = ADC_INTEN_SEQB_INTEN_MASK,
  kADC_OverrunInterruptEnable = ADC_INTEN_OVR_INTEN_MASK }
  Interrupts.
- enum adc_clock_mode_t {
  kADC_ClockSynchronousMode,
  kADC_ClockAsynchronousMode = 1U }
  Define selection of clock mode.
- enum adc_resolution_t {

kADC_Resolution6bit = 0U,
kADC_Resolution8bit = 1U,
kADC_Resolution10bit = 2U,
kADC_Resolution12bit = 3U }
Define selection of resolution.

- enum adc_trigger_polarity_t {
kADC_TriggerPolarityNegativeEdge = 0U,
kADC_TriggerPolarityPositiveEdge = 1U }
Define selection of polarity of selected input trigger for conversion sequence.
- enum adc_priority_t {
kADC_PriorityLow = 0U,
kADC_PriorityHigh = 1U }
Define selection of conversion sequence's priority.
- enum adc_seq_interrupt_mode_t {
kADC_InterruptForEachConversion = 0U,
kADC_InterruptForEachSequence = 1U }
Define selection of conversion sequence's interrupt.
- enum adc_threshold_compare_status_t {
kADC_ThresholdCompareInRange = 0U,
kADC_ThresholdCompareBelowRange = 1U,
kADC_ThresholdCompareAboveRange = 2U }
Define status of threshold compare result.
- enum adc_threshold_crossing_status_t {
kADC_ThresholdCrossingNoDetected = 0U,
kADC_ThresholdCrossingDownward = 2U,
kADC_ThresholdCrossingUpward = 3U }
Define status of threshold crossing detection result.
- enum adc_threshold_interrupt_mode_t {
kADC_ThresholdInterruptDisabled = 0U,
kADC_ThresholdInterruptOnOutside = 1U,
kADC_ThresholdInterruptOnCrossing = 2U }
Define interrupt mode for threshold compare event.

Driver version

- #define LPC_ADC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
ADC driver version 2.0.0.

Initialization and Deinitialization

- void ADC_Init (ADC_Type ∗base, const adc_config_t ∗config)
Initialize the ADC module.
- void ADC_Deinit (ADC_Type ∗base)
Deinitialize the ADC module.
- void ADC_GetDefaultConfig (adc_config_t ∗config)
Gets an available pre-defined settings for initial configuration.
- bool ADC_DoSelfCalibration (ADC_Type ∗base)
Do the self hardware calibration.

- static void ADC_EnableTemperatureSensor (ADC_Type *base, bool enable)
  Enable the internal temperature sensor measurement.

Control conversion sequence A.

- static void ADC_EnableConvSeqA (ADC_Type *base, bool enable)
  Enable the conversion sequence A.
- void ADC_SetConvSeqAConfig (ADC_Type *base, const adc_conv_seq_config_t *config)
  Configure the conversion sequence A.
- static void ADC_DoSoftwareTriggerConvSeqA (ADC_Type *base)
  Do trigger the sequence's conversion by software.
- static void ADC_EnableConvSeqABurstMode (ADC_Type *base, bool enable)
  Enable the burst conversion of sequence A.
- static void ADC_SetConvSeqAHighPriority (ADC_Type *base)
  Set the high priority for conversion sequence A.

Control conversion sequence B.

- static void ADC_EnableConvSeqB (ADC_Type *base, bool enable)
  Enable the conversion sequence B.
- void ADC_SetConvSeqBConfig (ADC_Type *base, const adc_conv_seq_config_t *config)
  Configure the conversion sequence B.
- static void ADC_DoSoftwareTriggerConvSeqB (ADC_Type *base)
  Do trigger the sequence's conversion by software.
- static void ADC_EnableConvSeqBBurstMode (ADC_Type *base, bool enable)
  Enable the burst conversion of sequence B.
- static void ADC_SetConvSeqBHighPriority (ADC_Type *base)
  Set the high priority for conversion sequence B.

Data result.

- bool ADC_GetConvSeqAGlobalConversionResult (ADC_Type *base, adc_result_info_t *info)
  Get the global ADC conversion infomation of sequence A.
- bool ADC_GetConvSeqBGlobalConversionResult (ADC_Type *base, adc_result_info_t *info)
  Get the global ADC conversion infomation of sequence B.
- bool ADC_GetChannelConversionResult (ADC_Type *base, uint32_t channel, adc_result_info_t *info)
  Get the channel's ADC conversion completed under each conversion sequence.

Threshold function.

- static void ADC_SetThresholdPair0 (ADC_Type *base, uint32_t lowValue, uint32_t highValue)
  Set the threshhold pair 0 with low and high value.
- static void ADC_SetThresholdPair1 (ADC_Type *base, uint32_t lowValue, uint32_t highValue)
  Set the threshhold pair 1 with low and high value.
- static void ADC_SetChannelWithThresholdPair0 (ADC_Type *base, uint32_t channelMask)
  Set given channels to apply the threshold pare 0.
- static void ADC_SetChannelWithThresholdPair1 (ADC_Type *base, uint32_t channelMask)
  Set given channels to apply the threshold pare 1.

Interrupts.

- static void ADC_EnableInterrupts (ADC_Type *base, uint32_t mask)
  Enable interrupts for conversion sequences.
- static void ADC_DisableInterrupts (ADC_Type *base, uint32_t mask)
  Disable interrupts for conversion sequence.
- static void ADC_EnableShresholdCompareInterrupt (ADC_Type *base, uint32_t channel, adc_threshold_interrupt_mode_t mode)
  Enable the interrupt of shreshold compare event for each channel.

Status.

- static uint32_t ADC_GetStatusFlags (ADC_Type *base)
  Get status flags of ADC module.
- static void ADC_ClearStatusFlags (ADC_Type *base, uint32_t mask)
  Clear status flags of ADC module.

5.3 Data Structure Documentation

5.3.1 struct adc_config_t

Data Fields

- adc_clock_mode_t clockMode
  Select the clock mode for ADC converter.
- uint32_t clockDividerNumber
  This field is only available when using kADC_ClockSynchronousMode for "clockMode" field.
- adc_resolution_t resolution
  Select the conversion bits.
- bool enableBypassCalibration
  By default, a calibration cycle must be performed each time the chip is powered-up.
- uint32_t sampleTimeNumber
  By default, with value as "0U", the sample period would be 2.5 ADC clocks.

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 adc_clock_mode_t adc_config_t::clockMode

5.3.1.0.0.1.2 uint32_t adc_config_t::clockDividerNumber

The divider would be plused by 1 based on the value in this field. The available range is in 8 bits.

5.3.1.0.0.1.3 adc_resolution_t adc_config_t::resolution

5.3.1.0.0.1.4 bool adc_config_t::enableBypassCalibration

Re-calibration may be warranted periodically - especially if operating conditions have changed. To enable this option would avoid the need to calibrate if offset error is not a concern in the application.

5.3.1.0.0.1.5 uint32_t adc_config_t::sampleTimeNumber

Then, to plus the "sampleTimeNumber" value here. The available value range is in 3 bits.

5.3.2 struct adc_conv_seq_config_t

Data Fields

- uint32_t channelMask
  Selects which one or more of the ADC channels will be sampled and converted when this sequence is launched.
- uint32_t triggerMask
  Selects which one or more of the available hardware trigger sources will cause this conversion sequence to be initiated.
- adc_trigger_polarity_t triggerPolarity
  Select the trigger to lauch conversion sequence.
- bool enableSyncBypass
  To enable this feature allows the hardware trigger input to bypass synchronization flip-flop stages and therefore shorten the time between the trigger input signal and the start of a conversion.
- bool enableSingleStep
  When enabling this feature, a trigger will launch a single conversion on the next channel in the sequence instead of the default response of launching an entire sequence of conversions.
- adc_seq_interrupt_mode_t interruptMode
  Select the interrpt/DMA trigger mode.

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 uint32_t adc_conv_seq_config_t::channelMask

The masked channels would be involved in current conversion sequence, beginning with the lowest-order. The available range is in 12-bit.

5.3.2.0.0.2.2 uint32_t adc_conv_seq_config_t::triggerMask

The available range is 6-bit.

5.3.2.0.0.2.3 adc_trigger_polarity_t adc_conv_seq_config_t::triggerPolarity

5.3.2.0.0.2.4 bool adc_conv_seq_config_t::enableSyncBypass

5.3.2.0.0.2.5 bool adc_conv_seq_config_t::enableSingleStep

5.3.2.0.0.2.6 adc_seq_interrupt_mode_t adc_conv_seq_config_t::interruptMode

5.3.3 struct adc_result_info_t

Data Fields

- uint32_t result
  Keey the conversion data value.
- adc_threshold_compare_status_t thresholdCompareStatus
  Keep the threshold compare status.
- adc_threshold_crossing_status_t thresholdCorssingStatus
  Keep the threshold crossing status.
- uint32_t channelNumber
  Keep the channel number for this conversion.
- bool overrunFlag
  Keep the status whether the conversion is overrun or not.

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 uint32_t adc_result_info_t::result

5.3.3.0.0.3.2 adc_threshold_compare_status_t adc_result_info_t::thresholdCompareStatus

5.3.3.0.0.3.3 adc_threshold_crossing_status_t adc_result_info_t::thresholdCorssingStatus

5.3.3.0.0.3.4 uint32_t adc_result_info_t::channelNumber

5.3.3.0.0.3.5 bool adc_result_info_t::overrunFlag

5.4 Macro Definition Documentation

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

5.5 Enumeration Type Documentation

5.5.1 enum _adc_status_flags

Enumerator

kADC_ThresholdCompareFlagOnChn0	Threshold comparison event on Channel 0.
kADC_ThresholdCompareFlagOnChn1	Threshold comparison event on Channel 1.
kADC_ThresholdCompareFlagOnChn2	Threshold comparison event on Channel 2.
kADC_ThresholdCompareFlagOnChn3	Threshold comparison event on Channel 3.
kADC_ThresholdCompareFlagOnChn4	Threshold comparison event on Channel 4.

kADC_ThresholdCompareFlagOnChn5 Threshold comparison event on Channel 5.
kADC_ThresholdCompareFlagOnChn6 Threshold comparison event on Channel 6.
kADC_ThresholdCompareFlagOnChn7 Threshold comparison event on Channel 7.
kADC_ThresholdCompareFlagOnChn8 Threshold comparison event on Channel 8.
kADC_ThresholdCompareFlagOnChn9 Threshold comparison event on Channel 9.
kADC_ThresholdCompareFlagOnChn10 Threshold comparison event on Channel 10.
kADC_ThresholdCompareFlagOnChn11 Threshold comparison event on Channel 11.
kADC_OverrunFlagForChn0 Mirror the OVERRUN status flag from the result register for ADC channel 0.
kADC_OverrunFlagForChn1 Mirror the OVERRUN status flag from the result register for ADC channel 1.
kADC_OverrunFlagForChn2 Mirror the OVERRUN status flag from the result register for ADC channel 2.
kADC_OverrunFlagForChn3 Mirror the OVERRUN status flag from the result register for ADC channel 3.
kADC_OverrunFlagForChn4 Mirror the OVERRUN status flag from the result register for ADC channel 4.
kADC_OverrunFlagForChn5 Mirror the OVERRUN status flag from the result register for ADC channel 5.
kADC_OverrunFlagForChn6 Mirror the OVERRUN status flag from the result register for ADC channel 6.
kADC_OverrunFlagForChn7 Mirror the OVERRUN status flag from the result register for ADC channel 7.
kADC_OverrunFlagForChn8 Mirror the OVERRUN status flag from the result register for ADC channel 8.
kADC_OverrunFlagForChn9 Mirror the OVERRUN status flag from the result register for ADC channel 9.
kADC_OverrunFlagForChn10 Mirror the OVERRUN status flag from the result register for ADC channel 10.
kADC_OverrunFlagForChn11 Mirror the OVERRUN status flag from the result register for ADC channel 11.
kADC_GlobalOverrunFlagForSeqA Mirror the glabal OVERRUN status flag for conversion sequence A.
kADC_GlobalOverrunFlagForSeqB Mirror the global OVERRUN status flag for conversion sequence B.
kADC_ConvSeqAInterruptFlag Sequence A interrupt/DMA trigger.
kADC_ConvSeqBInterruptFlag Sequence B interrupt/DMA trigger.
kADC_ThresholdCompareInterruptFlag Threshold comparision interrupt flag.
kADC_OverrunInterruptFlag Overrun interrupt flag.

5.5.2 enum _adc_interrupt_enable

Note

Not all the interrupt options are listed here

Enumerator

kADC_ConvSeqAInterruptEnable Enable interrupt upon completion of each individual conversion in sequence A, or entire sequence.

kADC_ConvSeqBInterruptEnable Enable interrupt upon completion of each individual conversion in sequence B, or entire sequence.

kADC_OverrunInterruptEnable Enable the detection of an overrun condition on any of the channel data registers will cause an overrun interrupt/DMA trigger.

5.5.3 enum adc_clock_mode_t

Enumerator

kADC_ClockSynchronousMode The ADC clock would be derived from the system clock based on "clockDividerNumber".

kADC_ClockAsynchronousMode The ADC clock would be based on the SYSCON block's divider.

5.5.4 enum adc_resolution_t

Enumerator

kADC_Resolution6bit 6-bit resolution.

kADC_Resolution8bit 8-bit resolution.

kADC_Resolution10bit 10-bit resolution.

kADC_Resolution12bit 12-bit resolution.

5.5.5 enum adc_trigger_polarity_t

Enumerator

kADC_TriggerPolarityNegativeEdge A negative edge launches the conversion sequence on the trigger(s).

kADC_TriggerPolarityPositiveEdge A positive edge launches the conversion sequence on the trigger(s).

5.5.6 enum adc_priority_t

Enumerator

kADC_PriorityLow This sequence would be preempted when another sequence is started.
kADC_PriorityHigh This sequence would preempt other sequence even when is is started.

5.5.7 enum adc_seq_interrupt_mode_t

Enumerator

kADC_InterruptForEachConversion The sequence interrupt/DMA trigger will be set at the end of each individual ADC conversion inside this conversion sequence.
kADC_InterruptForEachSequence The sequence interrupt/DMA trigger will be set when the entire set of this sequence conversions completes.

5.5.8 enum adc_threshold_compare_status_t

Enumerator

kADC_ThresholdCompareInRange LOW threshold <= conversion value <= HIGH threshold.
kADC_ThresholdCompareBelowRange conversion value < LOW threshold.
kADC_ThresholdCompareAboveRange conversion value > HIGH threshold.

5.5.9 enum adc_threshold_crossing_status_t

Enumerator

kADC_ThresholdCrossingNoDetected No threshold Crossing detected.
kADC_ThresholdCrossingDownward Downward Threshold Crossing detected.
kADC_ThresholdCrossingUpward Upward Threshold Crossing Detected.

5.5.10 enum adc_threshold_interrupt_mode_t

Enumerator

kADC_ThresholdInterruptDisabled Threshold comparison interrupt is disabled.
kADC_ThresholdInterruptOnOutside Threshold comparison interrupt is enabled on outside threshold.
kADC_ThresholdInterruptOnCrossing Threshold comparison interrupt is enabled on crossing threshold.

5.6 Function Documentation

5.6.1 void ADC_Init (ADC_Type * *base,* const adc_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_config_t.

5.6.2 void ADC_Deinit (ADC_Type * *base*)

Parameters

base	ADC peripheral base address.

5.6.3 void ADC_GetDefaultConfig (adc_config_t * *config*)

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

```
*   config->clockMode = kADC_ClockSynchronousMode;
*   config->clockDividerNumber = 0U;
*   config->resolution = kADC_Resolution12bit;
*   config->enableBypassCalibration = false;
*   config->sampleTimeNumber = 0U;
*
```

Parameters

config	Pointer to configuration structure.

5.6.4 bool ADC_DoSelfCalibration (ADC_Type * *base*)

Parameters

base	ADC peripheral base address.

Return values

true	Calibration succeed.
false	Calibration failed.

5.6.5 static void ADC_EnableTemperatureSensor (ADC_Type * *base,* bool *enable*) [inline], [static]

When enabling the internal temperature sensor measurement, the channel 0 would be connected to internal sensor instead of external pin.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable the feature or not.

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

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable the feature or not.

5.6.7 void ADC_SetConvSeqAConfig (ADC_Type * *base,* const adc_conv_seq_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_conv_seq_config_t.

5.6.8 static void ADC_DoSoftwareTriggerConvSeqA (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

5.6.9 static void ADC_EnableConvSeqABurstMode (ADC_Type * *base*, bool *enable*) [inline], [static]

Enable the burst mode would cause the conversion sequence to be cntinuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before cnversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable this feature.

5.6.10 static void ADC_SetConvSeqAHighPriority (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral bass address.

5.6.11 static void ADC_EnableConvSeqB (ADC_Type * *base*, bool *enable*) [inline], [static]

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

base	ADC peripheral base address.

enable	Switcher to enable the feature or not.

5.6.12 void ADC_SetConvSeqBConfig (ADC_Type * *base*, const adc_conv_seq_config_t * *config*)

Parameters

base	ADC peripheral base address.
config	Pointer to configuration structure, see to adc_conv_seq_config_t.

5.6.13 static void ADC_DoSoftwareTriggerConvSeqB (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

5.6.14 static void ADC_EnableConvSeqBBurstMode (ADC_Type * *base*, bool *enable*) [inline], [static]

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before cnversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

base	ADC peripheral base address.
enable	Switcher to enable this feature.

5.6.15 static void ADC_SetConvSeqBHighPriority (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral bass address.

5.6.16 bool ADC_GetConvSeqAGlobalConversionResult (ADC_Type * *base*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

5.6.17 bool ADC_GetConvSeqBGlobalConversionResult (ADC_Type * *base*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

5.6.18 bool ADC_GetChannelConversionResult (ADC_Type * *base*, uint32_t *channel*, adc_result_info_t * *info*)

Parameters

base	ADC peripheral base address.
channel	The indicated channel number.
info	Pointer to information structure, see to adc_result_info_t;

Return values

true	The conversion result is ready.
false	The conversion result is not ready yet.

5.6.19 static void ADC_SetThresholdPair0 (ADC_Type * *base*, uint32_t *lowValue*, uint32_t *highValue*) [inline], [static]

Parameters

base	ADC peripheral base address.
lowValue	LOW threshold value.
highValue	HIGH threshold value.

5.6.20 static void ADC_SetThresholdPair1 (ADC_Type * *base*, uint32_t *lowValue*, uint32_t *highValue*) [inline], [static]

Parameters

base	ADC peripheral base address.
lowValue	LOW threshold value. The available value is with 12-bit.
highValue	HIGH threshold value. The available value is with 12-bit.

5.6.21 static void ADC_SetChannelWithThresholdPair0 (ADC_Type * *base*, uint32_t *channelMask*) [inline], [static]

Parameters

base	ADC peripheral base address.
channelMask	Indicated channels' mask.

5.6.22 static void ADC_SetChannelWithThresholdPair1 (ADC_Type * *base*, uint32_t *channelMask*) [inline], [static]

Parameters

base	ADC peripheral base address.
channelMask	Indicated channels' mask.

5.6.23 static void ADC_EnableInterrupts (ADC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of interrupt mask value for global block except each channal, see to _adc_interrupt_enable.

5.6.24 static void ADC_DisableInterrupts (ADC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of interrupt mask value for global block except each channel, see to _adc_interrupt_enable.

5.6.25 static void ADC_EnableShresholdCompareInterrupt (ADC_Type * *base*, uint32_t *channel*, adc_threshold_interrupt_mode_t *mode*) [inline], [static]

Parameters

base	ADC peripheral base address.
channel	Channel number.
mode	Interrupt mode for threshold compare event, see to adc_threshold_interrupt_mode_t.

5.6.26 static uint32_t ADC_GetStatusFlags (ADC_Type * *base*) [inline], [static]

Parameters

base	ADC peripheral base address.

Returns

Mask of status flags of module, see to _adc_status_flags.

5.6.27 static void ADC_ClearStatusFlags (ADC_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	ADC peripheral base address.
mask	Mask of status flags of module, see to _adc_status_flags.

Chapter 6
CRC: Cyclic Redundancy Check Driver

6.1 Overview

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

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

6.2 CRC Driver Initialization and Configuration

CRC_Init() function enables the clock for the CRC module in the LPC SYSCON block and fully (re-)configures the CRC module according to configuration structure. It also starts checksum computation by writing the seed.

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_GetConfig() 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. 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/CCITT-FALSE protocol.

CRC_Deinit() function disables clock to the CRC module.

CRC_Reset() performs hardware reset of the CRC module.

6.3 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 of CRC configuration function CRC_Init() fully specifies the CRC module configuration for CRC_WriteData() call.

6.4 CRC Get Checksum

The CRC_Get16bitResult() or CRC_Get32bitResult() function is used to read the CRC module checksum register. The bit reverse and 1's complement operations are already applied to the result if previously configured. Use CRC_GetConfig() function to get the actual checksum without bit reverse and 1's complement applied so it can be used as seed when resuming calculation later.

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_GetConfig() to get intermediate checksum to be used as seed value in future.

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

6.5 Comments about API usage in RTOS

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

The triplets

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

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;
```

Alternatively, the context switch handler could read original configuration and restore it when switching back to original task/thread:

```
CRC_GetConfig(base, &originalConfig);
/* ... other task using CRC engine... */
CRC_Init(base, &originalConfig);
```

6.6 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.

6.7 CRC Driver Examples

6.7.1 Simple examples

Simple example with default CRC-16/CCITT-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/CCITT-FALSE */
CRC_Init(base, &config);
CRC_WriteData(base, data, sizeof(data));
checksum = CRC_Get16bitResult(base);
CRC_Deinit(base);
```

Simple example with CRC-32 protocol configuration

```
crc_config_t config;
uint32_t checksum;

config.polynomial = kCRC_Polynomial_CRC_32;
config.reverseIn = true;
config.complementIn = false;
config.reverseOut = true;
config.complementOut = true;
config.seed = 0xFFFFFFFFu;

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);
CRC_Deinit(base);
```

6.7.2 Advanced examples

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

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

	CRC-16/CCITT-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 functions to get configurations:

```
void GetConfigCrc16Ccitt(CRC_Type *base, crc_config_t *config)
{
    config->polynomial = kCRC_Polynomial_CRC_CCITT;
    config->reverseIn = false;
    config->complementIn = false;
    config->reverseOut = false;
    config->complementOut = false;
    config->seed = 0xFFFFU;
}
```

```
void GetConfigCrc16(CRC_Type *base, crc_config_t *config)
{
    config->polynomial = kCRC_Polynomial_CRC_16;
    config->reverseIn = true;
    config->complementIn = false;
    config->reverseOut = true;
    config->complementOut = false;
    config->seed = 0x0U;
}

void GetConfigCrc32(CRC_Type *base, crc_config_t *config)
{
    config->polynomial = kCRC_Polynomial_CRC_32;
    config->reverseIn = true;
    config->complementIn = false;
    config->reverseOut = true;
    config->complementOut = true;
    config->seed = 0xFFFFFFFFU;
}
```

The following context switches show possible API usage:

```
uint16_t checksumCrc16;
uint32_t checksumCrc32;
uint16_t checksumCrc16Ccitt;

crc_config_t configCrc16;
crc_config_t configCrc32;
crc_config_t configCrc16Ccitt;

GetConfigCrc16(base, &configCrc16);
GetConfigCrc32(base, &configCrc32);
GetConfigCrc16Ccitt(base, &configCrc16Ccitt);

/* Task A bytes[0-3] */
CRC_Init(base, &configCrc16);
CRC_WriteData(base, &data[0], 4);
CRC_GetConfig(base, &configCrc16);

/* Task B bytes[0-3] */
CRC_Init(base, &configCrc16Ccitt);
CRC_WriteData(base, &data[0], 4);
CRC_GetConfig(base, &configCrc16Ccitt);

/* Task C 4 bytes[0-3] */
CRC_Init(base, &configCrc32);
CRC_WriteData(base, &data[0], 4);
CRC_GetConfig(base, &configCrc32);

/* Task B add final 5 bytes[4-8] */
CRC_Init(base, &configCrc16Ccitt);
CRC_WriteData(base, &data[4], 5);
checksumCrc16Ccitt = CRC_Get16bitResult(base);

/* Task C 3 bytes[4-6] */
CRC_Init(base, &configCrc32);
CRC_WriteData(base, &data[4], 3);
CRC_GetConfig(base, &configCrc32);

/* Task A 3 bytes[4-6] */
CRC_Init(base, &configCrc16);
CRC_WriteData(base, &data[4], 3);
CRC_GetConfig(base, &configCrc16);

/* Task C add final 2 bytes[7-8] */
```

```
CRC_Init(base, &configCrc32);
CRC_WriteData(base, &data[7], 2);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task A add final 2 bytes[7-8] */
CRC_Init(base, &configCrc16);
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_CCITT_FALSE_AS_DEFAULT 1
  Default configuration structure filled by CRC_GetDefaultConfig().

Enumerations

- enum crc_polynomial_t {
  kCRC_Polynomial_CRC_CCITT = 0U,
  kCRC_Polynomial_CRC_16 = 1U,
  kCRC_Polynomial_CRC_32 = 2U }
  CRC polynomials to use.

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_Reset (CRC_Type *base)
  resets CRC peripheral module.
- void CRC_GetDefaultConfig (crc_config_t *config)
  Loads default values to CRC protocol configuration structure.
- void CRC_GetConfig (CRC_Type *base, crc_config_t *config)
  Loads actual values configured in CRC peripheral 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.
- static 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.

6.8 Data Structure Documentation

6.8.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

- crc_polynomial_t polynomial
  CRC polynomial.
- bool reverseIn
  Reverse bits on input.
- bool complementIn
  Perform 1's complement on input.
- bool reverseOut
  Reverse bits on output.
- bool complementOut
  Perform 1's complement on output.
- uint32_t seed
  Starting checksum value.

6.8.1.0.0.4 Field Documentation

6.8.1.0.0.4.1 crc_polynomial_t crc_config_t::polynomial

6.8.1.0.0.4.2 bool crc_config_t::reverseIn

6.8.1.0.0.4.3 bool crc_config_t::complementIn

6.8.1.0.0.4.4 bool crc_config_t::reverseOut

6.8.1.0.0.4.5 bool crc_config_t::complementOut

6.8.1.0.0.4.6 uint32_t crc_config_t::seed

6.9 Macro Definition Documentation

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

Version 2.0.0.

Current version: 2.0.0

Change log:

- Version 2.0.0

– initial version

6.9.2 #define CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT 1

Uses CRC-16/CCITT-FALSE as default.

6.10 Enumeration Type Documentation

6.10.1 enum crc_polynomial_t

Enumerator

kCRC_Polynomial_CRC_CCITT x^16+x^12+x^5+1
kCRC_Polynomial_CRC_16 x^16+x^15+x^2+1
kCRC_Polynomial_CRC_32 x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x+

6.11 Function Documentation

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

This functions enables the CRC peripheral clock in the LPC SYSCON block. It also configures the CRC engine and starts checksum computation by writing the seed.

Parameters

base	CRC peripheral address.
config	CRC module configuration structure.

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

This functions disables the CRC peripheral clock in the LPC SYSCON block.

Parameters

base	CRC peripheral address.

6.11.3 void CRC_Reset (CRC_Type * *base*)

Parameters

base	CRC peripheral address, currently not used.

6.11.4 void CRC_GetDefaultConfig (crc_config_t * *config*)

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

```
*   config->polynomial = kCRC_Polynomial_CRC_CCITT;
*   config->reverseIn = false;
*   config->complementIn = false;
*   config->reverseOut = false;
*   config->complementOut = false;
*   config->seed = 0xFFFFU;
*
```

Parameters

config	CRC protocol configuration structure

6.11.5 void CRC_GetConfig (CRC_Type * *base*, crc_config_t * *config*)

The values, including seed, can be used to resume CRC calculation later.

Parameters

base	CRC peripheral address.
config	CRC protocol configuration structure

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

Writes input data buffer bytes to CRC data register.

Parameters

base	CRC peripheral address.

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

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

Reads CRC data register.

Parameters

base	CRC peripheral address.

Returns

final 32-bit checksum, after configured bit reverse and complement operations.

6.11.8 static uint16_t CRC_Get16bitResult (CRC_Type * *base*) [inline], [static]

Reads CRC data register.

Parameters

base	CRC peripheral address.

Returns

final 16-bit checksum, after configured bit reverse and complement operations.

Chapter 7
CTIMER: Standard counter/timers

7.1 Overview

The SDK provides a driver for the ctimer module of LPC devices.

7.2 Function groups

The ctimer driver supports the generation of PWM signals, input capture and setting up the timer match conditions.

7.2.1 Initialization and deinitialization

The function CTIMER_Init() initializes the ctimer with specified configurations. The function CTIMER_GetDefaultConfig() gets the default configurations. The initialization function configures the counter/timer mode and input selection when running in counter mode.

The function CTIMER_Deinit() stops the timer and turns off the module clock.

7.2.2 PWM Operations

The function CTIMER_SetupPwm() sets up channels for PWM output. Each channel has its own duty cycle, however the same PWM period 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 CTIMER_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular channel.

7.2.3 Match Operation

The function CTIMER_SetupMatch() sets up channels for match operation. Each channel is configured with a match value, if the counter should stop on match, if counter should reset on match and output pin action. The output signal can be cleared, set or toggled on match.

7.2.4 Input capture operations

The function CTIMER_SetupCapture() sets up an channel for input capture. The user can specify the capture edge and if a interrupt should be generated when processing the input signal.

7.3 Typical use case

7.3.1 Match example

Set up a match channel to toggle output when a match occurs.

```
int main(void)
{
    ctimer_config_t config;
    ctimer_match_config_t matchConfig;

    /* Init hardware*/
    BOARD_InitHardware();

    PRINTF("CTimer match example to toggle the output on a match\r\n");

    CTIMER_GetDefaultConfig(&config);

    CTIMER_Init(CTIMER, &config);

    matchConfig.enableCounterReset = true;
    matchConfig.enableCounterStop = false;
    matchConfig.matchValue = CLOCK_GetFreq(kCLOCK_BusClk) / 2;
    matchConfig.outControl = kCTIMER_Output_Toggle;
    matchConfig.outPinInitState = true;
    matchConfig.enableInterrupt = false;
    matchConfig.cb_func = NULL;
    CTIMER_SetupMatch(CTIMER, CTIMER_MAT_OUT, &matchConfig);
    CTIMER_StartTimer(CTIMER);

    while (1)
    {
    }
}
```

7.3.2 PWM output example

Set up a channel for PWM output.

```
int main(void)
{
    ctimer_config_t config;
    uint32_t srcClock_Hz;

    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc clock */
    srcClock_Hz = CLOCK_GetFreq(kCLOCK_BusClk);

    PRINTF("CTimer example to generate a PWM signal\r\n");

    CTIMER_GetDefaultConfig(&config);

    CTIMER_Init(CTIMER, &config);
    CTIMER_SetupPwm(CTIMER, CTIMER_MAT_OUT, 20, 20000, srcClock_Hz, NULL);
    CTIMER_StartTimer(CTIMER);

    while (1)
    {
    }
}
```

Files

- file fsl_ctimer.h

Data Structures

- struct ctimer_match_config_t
  Match configuration. More...
- struct ctimer_config_t
  Timer configuration structure. More...

Enumerations

- enum ctimer_capture_channel_t {
  kCTIMER_Capture_0 = 0U,
  kCTIMER_Capture_1,
  kCTIMER_Capture_2,
  kCTIMER_Capture_3 }
  List of Timer capture channels.
- enum ctimer_capture_edge_t {
  kCTIMER_Capture_RiseEdge = 1U,
  kCTIMER_Capture_FallEdge = 2U,
  kCTIMER_Capture_BothEdge = 3U }
  List of capture edge options.
- enum ctimer_match_t {
  kCTIMER_Match_0 = 0U,
  kCTIMER_Match_1,
  kCTIMER_Match_2,
  kCTIMER_Match_3 }
  List of Timer match registers.
- enum ctimer_match_output_control_t {
  kCTIMER_Output_NoAction = 0U,
  kCTIMER_Output_Clear,
  kCTIMER_Output_Set,
  kCTIMER_Output_Toggle }
  List of output control options.
- enum ctimer_timer_mode_t
  List of Timer modes.
- enum ctimer_interrupt_enable_t {
  kCTIMER_Match0InterruptEnable = CTIMER_MCR_MR0I_MASK,
  kCTIMER_Match1InterruptEnable = CTIMER_MCR_MR1I_MASK,
  kCTIMER_Match2InterruptEnable = CTIMER_MCR_MR2I_MASK,
  kCTIMER_Match3InterruptEnable = CTIMER_MCR_MR3I_MASK,
  kCTIMER_Capture0InterruptEnable = CTIMER_CCR_CAP0I_MASK,
  kCTIMER_Capture1InterruptEnable = CTIMER_CCR_CAP1I_MASK,
  kCTIMER_Capture2InterruptEnable = CTIMER_CCR_CAP2I_MASK,
  kCTIMER_Capture3InterruptEnable = CTIMER_CCR_CAP3I_MASK }
  List of Timer interrupts.

Typical use case

- enum ctimer_status_flags_t {
  kCTIMER_Match0Flag = CTIMER_IR_MR0INT_MASK,
  kCTIMER_Match1Flag = CTIMER_IR_MR1INT_MASK,
  kCTIMER_Match2Flag = CTIMER_IR_MR2INT_MASK,
  kCTIMER_Match3Flag = CTIMER_IR_MR3INT_MASK,
  kCTIMER_Capture0Flag = CTIMER_IR_CR0INT_MASK,
  kCTIMER_Capture1Flag = CTIMER_IR_CR1INT_MASK,
  kCTIMER_Capture2Flag = CTIMER_IR_CR2INT_MASK,
  kCTIMER_Capture3Flag = CTIMER_IR_CR3INT_MASK }
  List of Timer flags.
- enum ctimer_callback_type_t {
  kCTIMER_SingleCallback,
  kCTIMER_MultipleCallback }
  Callback type when registering for a callback.

Functions

- void CTIMER_SetupMatch (CTIMER_Type *base, ctimer_match_t matchChannel, const ctimer_match_config_t *config)
  Setup the match register.
- void CTIMER_SetupCapture (CTIMER_Type *base, ctimer_capture_channel_t capture, ctimer_capture_edge_t edge, bool enableInt)
  Setup the capture.
- void CTIMER_RegisterCallBack (CTIMER_Type *base, ctimer_callback_t *cb_func, ctimer_callback_type_t cb_type)
  Register callback.
- static void CTIMER_Reset (CTIMER_Type *base)
  Reset the counter.

Driver version

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

Initialization and deinitialization

- void CTIMER_Init (CTIMER_Type *base, const ctimer_config_t *config)
  Ungates the clock and configures the peripheral for basic operation.
- void CTIMER_Deinit (CTIMER_Type *base)
  Gates the timer clock.
- void CTIMER_GetDefaultConfig (ctimer_config_t *config)
  Fills in the timers configuration structure with the default settings.

PWM setup operations

- status_t CTIMER_SetupPwm (CTIMER_Type *base, ctimer_match_t matchChannel, uint8_t dutyCyclePercent, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, bool enableInt)
  Configures the PWM signal parameters.

- void CTIMER_UpdatePwmDutycycle (CTIMER_Type *base, ctimer_match_t matchChannel, uint8_t dutyCyclePercent)
  Updates the duty cycle of an active PWM signal.

Interrupt Interface

- static void CTIMER_EnableInterrupts (CTIMER_Type *base, uint32_t mask)
  Enables the selected Timer interrupts.
- static void CTIMER_DisableInterrupts (CTIMER_Type *base, uint32_t mask)
  Disables the selected Timer interrupts.
- static uint32_t CTIMER_GetEnabledInterrupts (CTIMER_Type *base)
  Gets the enabled Timer interrupts.

Status Interface

- static uint32_t CTIMER_GetStatusFlags (CTIMER_Type *base)
  Gets the Timer status flags.
- static void CTIMER_ClearStatusFlags (CTIMER_Type *base, uint32_t mask)
  Clears the Timer status flags.

Counter Start and Stop

- static void CTIMER_StartTimer (CTIMER_Type *base)
  Starts the Timer counter.
- static void CTIMER_StopTimer (CTIMER_Type *base)
  Stops the Timer counter.

7.4 Data Structure Documentation

7.4.1 struct ctimer_match_config_t

This structure holds the configuration settings for each match register.

Data Fields

- uint32_t matchValue
  This is stored in the match register.
- bool enableCounterReset
  true: Match will reset the counter false: Match will not reser the counter
- bool enableCounterStop
  true: Match will stop the counter false: Match will not stop the counter
- ctimer_match_output_control_t outControl
  Action to be taken on a match on the EM bit/output.
- bool outPinInitState
  Initial value of the EM bit/output.
- bool enableInterrupt
  true: Generate interrupt upon match false: Do not generate interrupt on match

7.4.2 struct ctimer_config_t

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the CTIMER_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

- ctimer_timer_mode_t mode
  Timer mode.
- ctimer_capture_channel_t input
  Input channel to increment the timer, used only in timer modes that rely on this input signal to increment TC.
- uint32_t prescale
  Prescale value.

7.5 Enumeration Type Documentation

7.5.1 enum ctimer_capture_channel_t

Enumerator

kCTIMER_Capture_0 Timer capture channel 0.
kCTIMER_Capture_1 Timer capture channel 1.
kCTIMER_Capture_2 Timer capture channel 2.
kCTIMER_Capture_3 Timer capture channel 3.

7.5.2 enum ctimer_capture_edge_t

Enumerator

kCTIMER_Capture_RiseEdge Capture on rising edge.
kCTIMER_Capture_FallEdge Capture on falling edge.
kCTIMER_Capture_BothEdge Capture on rising and falling edge.

7.5.3 enum ctimer_match_t

Enumerator

kCTIMER_Match_0 Timer match register 0.
kCTIMER_Match_1 Timer match register 1.
kCTIMER_Match_2 Timer match register 2.

kCTIMER_Match_3 Timer match register 3.

7.5.4 enum ctimer_match_output_control_t

Enumerator

kCTIMER_Output_NoAction No action is taken.
kCTIMER_Output_Clear Clear the EM bit/output to 0.
kCTIMER_Output_Set Set the EM bit/output to 1.
kCTIMER_Output_Toggle Toggle the EM bit/output.

7.5.5 enum ctimer_interrupt_enable_t

Enumerator

kCTIMER_Match0InterruptEnable Match 0 interrupt.
kCTIMER_Match1InterruptEnable Match 1 interrupt.
kCTIMER_Match2InterruptEnable Match 2 interrupt.
kCTIMER_Match3InterruptEnable Match 3 interrupt.
kCTIMER_Capture0InterruptEnable Capture 0 interrupt.
kCTIMER_Capture1InterruptEnable Capture 1 interrupt.
kCTIMER_Capture2InterruptEnable Capture 2 interrupt.
kCTIMER_Capture3InterruptEnable Capture 3 interrupt.

7.5.6 enum ctimer_status_flags_t

Enumerator

kCTIMER_Match0Flag Match 0 interrupt flag.
kCTIMER_Match1Flag Match 1 interrupt flag.
kCTIMER_Match2Flag Match 2 interrupt flag.
kCTIMER_Match3Flag Match 3 interrupt flag.
kCTIMER_Capture0Flag Capture 0 interrupt flag.
kCTIMER_Capture1Flag Capture 1 interrupt flag.
kCTIMER_Capture2Flag Capture 2 interrupt flag.
kCTIMER_Capture3Flag Capture 3 interrupt flag.

7.5.7 enum ctimer_callback_type_t

When registering a callback an array of function pointers is passed the size could be 1 or 8, the callback type will tell that.

Enumerator

kCTIMER_SingleCallback Single Callback type where there is only one callback for the timer. based on the status flags different channels needs to be handled differently

kCTIMER_MultipleCallback Multiple Callback type where there can be 8 valid callbacks, one per channel. for both match/capture

7.6 Function Documentation

7.6.1 void CTIMER_Init (CTIMER_Type * *base*, const ctimer_config_t * *config*)

Note

This API should be called at the beginning of the application before using the driver.

Parameters

base	Ctimer peripheral base address
config	Pointer to the user configuration structure.

7.6.2 void CTIMER_Deinit (CTIMER_Type * *base*)

Parameters

base	Ctimer peripheral base address

7.6.3 void CTIMER_GetDefaultConfig (ctimer_config_t * *config*)

The default values are:

```
*   config->mode = kCTIMER_TimerMode;
*   config->input = kCTIMER_Capture_0;
*   config->prescale = 0;
*
```

Parameters

config	Pointer to the user configuration structure.

7.6.4 status_t CTIMER_SetupPwm (CTIMER_Type * *base*, ctimer_match_t *matchChannel*, uint8_t *dutyCyclePercent*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*, bool *enableInt*)

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function will assign match channel 3 to set the PWM cycle.

Note

When setting PWM output from multiple output pins, all should use the same PWM frequency

Parameters

base	Ctimer peripheral base address
matchChannel	Match pin to be used to output the PWM signal
dutyCycle-Percent	PWM pulse width; the value should be between 0 to 100
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	Timer counter clock in Hz
enableInt	Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt is generated

Returns

kStatus_Success on success kStatus_Fail If matchChannel passed in is 3; this channel is reserved to set the PWM cycle

7.6.5 void CTIMER_UpdatePwmDutycycle (CTIMER_Type * *base*, ctimer_match_t *matchChannel*, uint8_t *dutyCyclePercent*)

Parameters

base	Ctimer peripheral base address
matchChannel	Match pin to be used to output the PWM signal
dutyCycle-Percent	New PWM pulse width; the value should be between 0 to 100

7.6.6 void CTIMER_SetupMatch (CTIMER_Type * *base*, ctimer_match_t *matchChannel*, const ctimer_match_config_t * *config*)

User configuration is used to setup the match value and action to be taken when a match occurs.

Parameters

base	Ctimer peripheral base address
matchChannel	Match register to configure
config	Pointer to the match configuration structure

7.6.7 void CTIMER_SetupCapture (CTIMER_Type * *base*, ctimer_capture-_channel_t *capture*, ctimer_capture_edge_t *edge*, bool *enableInt*)

Parameters

base	Ctimer peripheral base address
capture	Capture channel to configure
edge	Edge on the channel that will trigger a capture
enableInt	Flag to enable channel interrupts, if enabled then the registered call back is called upon capture

7.6.8 void CTIMER_RegisterCallBack (CTIMER_Type * *base*, ctimer_callback_t * *cb_func*, ctimer_callback_type_t *cb_type*)

Parameters

base	Ctimer peripheral base address
cb_func	callback function
cb_type	callback function type, singular or multiple

7.6.9 static void CTIMER_EnableInterrupts (CTIMER_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Ctimer peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

7.6.10 static void CTIMER_DisableInterrupts (CTIMER_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Ctimer peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

7.6.11 static uint32_t CTIMER_GetEnabledInterrupts (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration ctimer_interrupt_enable_t

7.6.12 static uint32_t CTIMER_GetStatusFlags (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration ctimer_status_flags_t

7.6.13 static void CTIMER_ClearStatusFlags (CTIMER_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	Ctimer peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration ctimer_status_flags_t

7.6.14 static void CTIMER_StartTimer (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

7.6.15 static void CTIMER_StopTimer (CTIMER_Type * *base*) [inline], [static]

Parameters

base	Ctimer peripheral base address

7.6.16 static void CTIMER_Reset (CTIMER_Type * *base*) [inline], [static]

The timer counter and prescale counter are reset on the next positive edge of the APB clock.

Parameters

base	Ctimer peripheral base address

Chapter 8
Common Driver

8.1 Overview

The KSDK provides a driver for the common module of Kinetis and LPC devices.

Macros

- #define MAKE_STATUS(group, code) ((((group)∗100) + (code)))
  Construct a status code value from a group and code number.
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
  Construct the version number for drivers.
- #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U
  No debug console.
- #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U
  Debug console base on UART.
- #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U
  Debug console base on LPUART.
- #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U
  Debug console base on LPSCI.
- #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U
  Debug console base on USBCDC.
- #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U
  Debug console base on USBCDC.
- #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
  Computes the number of elements in an array.
- #define ADC_RSTS

Typedefs

- typedef int32_t status_t
  Type used for all status and error return values.

Enumerations

- enum _status_groups {
  kStatusGroup_Generic = 0,
  kStatusGroup_FLASH = 1,
  kStatusGroup_LPSPI = 4,
  kStatusGroup_FLEXIO_SPI = 5,
  kStatusGroup_DSPI = 6,
  kStatusGroup_FLEXIO_UART = 7,
  kStatusGroup_FLEXIO_I2C = 8,
  kStatusGroup_LPI2C = 9,
  kStatusGroup_UART = 10,
  kStatusGroup_I2C = 11,
  kStatusGroup_LPSCI = 12,
  kStatusGroup_LPUART = 13,
  kStatusGroup_SPI = 14,
  kStatusGroup_XRDC = 15,
  kStatusGroup_SEMA42 = 16,
  kStatusGroup_SDHC = 17,
  kStatusGroup_SDMMC = 18,
  kStatusGroup_SAI = 19,
  kStatusGroup_MCG = 20,
  kStatusGroup_SCG = 21,
  kStatusGroup_SDSPI = 22,
  kStatusGroup_FLEXIO_I2S = 23,
  kStatusGroup_FLASHIAP = 25,
  kStatusGroup_FLEXCOMM_I2C = 26,
  kStatusGroup_I2S = 27,
  kStatusGroup_SDRAMC = 35,
  kStatusGroup_POWER = 39,
  kStatusGroup_ENET = 40,
  kStatusGroup_PHY = 41,
  kStatusGroup_TRGMUX = 42,
  kStatusGroup_SMARTCARD = 43,
  kStatusGroup_LMEM = 44,
  kStatusGroup_QSPI = 45,
  kStatusGroup_DMA = 50,
  kStatusGroup_EDMA = 51,
  kStatusGroup_DMAMGR = 52,
  kStatusGroup_FLEXCAN = 53,
  kStatusGroup_LTC = 54,
  kStatusGroup_FLEXIO_CAMERA = 55,
  kStatusGroup_LPC_SPI = 56,
  kStatusGroup_LPC_USART = 57,
  kStatusGroup_DMIC = 58,
  kStatusGroup_SDIF = 59,
  kStatusGroup_SPIFI = 60,
  kStatusGroup_OTP = 61,
  kStatusGroup_MCAN = 62,
  kStatusGroup_NOTIFIER = 98,

kStatusGroup_ApplicationRangeStart = 100 }
Status group numbers.
- enum _generic_status
Generic status return codes.
- enum SYSCON_RSTn_t {

kFLASH_RST_SHIFT_RSTn = 0 | 7U,
kFMC_RST_SHIFT_RSTn = 0 | 8U,
kEEPROM_RST_SHIFT_RSTn = 0 | 9U,
kSPIFI_RST_SHIFT_RSTn = 0 | 10U,
kMUX_RST_SHIFT_RSTn = 0 | 11U,
kIOCON_RST_SHIFT_RSTn = 0 | 13U,
kGPIO0_RST_SHIFT_RSTn = 0 | 14U,
kGPIO1_RST_SHIFT_RSTn = 0 | 15U,
kGPIO2_RST_SHIFT_RSTn = 0 | 16U,
kGPIO3_RST_SHIFT_RSTn = 0 | 17U,
kPINT_RST_SHIFT_RSTn = 0 | 18U,
kGINT_RST_SHIFT_RSTn = 0 | 19U,
kDMA_RST_SHIFT_RSTn = 0 | 20U,
kCRC_RST_SHIFT_RSTn = 0 | 21U,
kWWDT_RST_SHIFT_RSTn = 0 | 22U,
kADC0_RST_SHIFT_RSTn = 0 | 27U,
kMRT_RST_SHIFT_RSTn = 65536 | 0U,
kSCT0_RST_SHIFT_RSTn = 65536 | 2U,
kMCAN0_RST_SHIFT_RSTn = 65536 | 7U,
kMCAN1_RST_SHIFT_RSTn = 65536 | 8U,
kUTICK_RST_SHIFT_RSTn = 65536 | 10U,
kFC0_RST_SHIFT_RSTn = 65536 | 11U,
kFC1_RST_SHIFT_RSTn = 65536 | 12U,
kFC2_RST_SHIFT_RSTn = 65536 | 13U,
kFC3_RST_SHIFT_RSTn = 65536 | 14U,
kFC4_RST_SHIFT_RSTn = 65536 | 15U,
kFC5_RST_SHIFT_RSTn = 65536 | 16U,
kFC6_RST_SHIFT_RSTn = 65536 | 17U,
kFC7_RST_SHIFT_RSTn = 65536 | 18U,
kDMIC_RST_SHIFT_RSTn = 65536 | 19U,
kCT32B2_RST_SHIFT_RSTn = 65536 | 22U,
kUSB0D_RST_SHIFT_RSTn = 65536 | 25U,
kCT32B0_RST_SHIFT_RSTn = 65536 | 26U,
kCT32B1_RST_SHIFT_RSTn = 65536 | 27U,
kLCD_RST_SHIFT_RSTn = 131072 | 2U,
kSDIO_RST_SHIFT_RSTn = 131072 | 3U,
kUSB1H_RST_SHIFT_RSTn = 131072 | 4U,
kUSB1D_RST_SHIFT_RSTn = 131072 | 5U,
kUSB1RAM_RST_SHIFT_RSTn = 131072 | 6U,
kEMC_RST_SHIFT_RSTn = 131072 | 7U,
kETH_RST_SHIFT_RSTn = 131072 | 8U,
kGPIO4_RST_SHIFT_RSTn = 131072 | 9U,
kGPIO5_RST_SHIFT_RSTn = 131072 | 10U,
kAES_RST_SHIFT_RSTn = 131072 | 11U,
kOTP_RST_SHIFT_RSTn = 131072 | 12U,
kRNG_RST_SHIFT_RSTn = 131072 | 13U,
kFC8_RST_SHIFT_RSTn = 131072 | 14U,
kFC9_RST_SHIFT_RSTn = 131072 | 15U,
kUSB0HMR_RST_SHIFT_RSTn = 131072 | 16U,

kCT32B4_RST_SHIFT_RSTn = 67108864 | 14U }
Enumeration for peripheral reset control bits.

Functions

- static void EnableIRQ (IRQn_Type interrupt)
  Enable specific interrupt.
- static void DisableIRQ (IRQn_Type interrupt)
  Disable specific interrupt.
- static uint32_t DisableGlobalIRQ (void)
  Disable the global IRQ.
- static void EnableGlobalIRQ (uint32_t primask)
  Enaable the global IRQ.
- uint32_t InstallIRQHandler (IRQn_Type irq, uint32_t irqHandler)
  install IRQ handler
- void EnableDeepSleepIRQ (IRQn_Type interrupt)
  Enable specific interrupt for wake-up from deep-sleep mode.
- void DisableDeepSleepIRQ (IRQn_Type interrupt)
  Disable specific interrupt for wake-up from deep-sleep mode.
- void RESET_SetPeripheralReset (reset_ip_name_t peripheral)
  Assert reset to peripheral.
- void RESET_ClearPeripheralReset (reset_ip_name_t peripheral)
  Clear reset to peripheral.
- void RESET_PeripheralReset (reset_ip_name_t peripheral)
  Reset peripheral module.

Min/max macros

- #define **MIN**(a, b) ((a) < (b) ? (a) : (b))
- #define **MAX**(a, b) ((a) > (b) ? (a) : (b))

UINT16_MAX/UINT32_MAX value

- #define **UINT16_MAX** ((uint16_t)-1)
- #define **UINT32_MAX** ((uint32_t)-1)

Timer utilities

- #define USEC_TO_COUNT(us, clockFreqInHz) (uint64_t)((uint64_t)us * clockFreqInHz / 1000000U)
  Macro to convert a microsecond period to raw count value.
- #define COUNT_TO_USEC(count, clockFreqInHz) (uint64_t)((uint64_t)count * 1000000U / clockFreqInHz)
  Macro to convert a raw count value to microsecond.
- #define MSEC_TO_COUNT(ms, clockFreqInHz) (uint64_t)((uint64_t)ms * clockFreqInHz / 1000-U)
  Macro to convert a millisecond period to raw count value.
- #define COUNT_TO_MSEC(count, clockFreqInHz) (uint64_t)((uint64_t)count * 1000U / clock-FreqInHz)
  Macro to convert a raw count value to millisecond.

8.2 Macro Definition Documentation

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

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

8.2.3 #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U

8.2.4 #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U

8.2.5 #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U

8.2.6 #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U

8.2.7 #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U

8.2.8 #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U

8.2.9 #define ARRAY_SIZE(*x*) (sizeof(x) / sizeof((x)[0]))

8.2.10 #define ADC_RSTS

Value:

```
{                       \
        kADC0_RST_SHIFT_RSTn \
    } /* Reset bits for ADC peripheral */
```

Array initializers with peripheral reset bits

8.3 Typedef Documentation

8.3.1 typedef int32_t status_t

8.4 Enumeration Type Documentation

8.4.1 enum _status_groups

Enumerator

kStatusGroup_Generic Group number for generic status codes.
kStatusGroup_FLASH Group number for FLASH status codes.

kStatusGroup_LPSPI Group number for LPSPI status codes.
kStatusGroup_FLEXIO_SPI Group number for FLEXIO SPI status codes.
kStatusGroup_DSPI Group number for DSPI status codes.
kStatusGroup_FLEXIO_UART Group number for FLEXIO UART status codes.
kStatusGroup_FLEXIO_I2C Group number for FLEXIO I2C status codes.
kStatusGroup_LPI2C Group number for LPI2C status codes.
kStatusGroup_UART Group number for UART status codes.
kStatusGroup_I2C Group number for UART status codes.
kStatusGroup_LPSCI Group number for LPSCI status codes.
kStatusGroup_LPUART Group number for LPUART status codes.
kStatusGroup_SPI Group number for SPI status code.
kStatusGroup_XRDC Group number for XRDC status code.
kStatusGroup_SEMA42 Group number for SEMA42 status code.
kStatusGroup_SDHC Group number for SDHC status code.
kStatusGroup_SDMMC Group number for SDMMC status code.
kStatusGroup_SAI Group number for SAI status code.
kStatusGroup_MCG Group number for MCG status codes.
kStatusGroup_SCG Group number for SCG status codes.
kStatusGroup_SDSPI Group number for SDSPI status codes.
kStatusGroup_FLEXIO_I2S Group number for FLEXIO I2S status codes.
kStatusGroup_FLASHIAP Group number for FLASHIAP status codes.
kStatusGroup_FLEXCOMM_I2C Group number for FLEXCOMM I2C status codes.
kStatusGroup_I2S Group number for I2S status codes.
kStatusGroup_SDRAMC Group number for SDRAMC status codes.
kStatusGroup_POWER Group number for POWER status codes.
kStatusGroup_ENET Group number for ENET status codes.
kStatusGroup_PHY Group number for PHY status codes.
kStatusGroup_TRGMUX Group number for TRGMUX status codes.
kStatusGroup_SMARTCARD Group number for SMARTCARD status codes.
kStatusGroup_LMEM Group number for LMEM status codes.
kStatusGroup_QSPI Group number for QSPI status codes.
kStatusGroup_DMA Group number for DMA status codes.
kStatusGroup_EDMA Group number for EDMA status codes.
kStatusGroup_DMAMGR Group number for DMAMGR status codes.
kStatusGroup_FLEXCAN Group number for FlexCAN status codes.
kStatusGroup_LTC Group number for LTC status codes.
kStatusGroup_FLEXIO_CAMERA Group number for FLEXIO CAMERA status codes.
kStatusGroup_LPC_SPI Group number for LPC_SPI status codes.
kStatusGroup_LPC_USART Group number for LPC_USART status codes.
kStatusGroup_DMIC Group number for DMIC status codes.
kStatusGroup_SDIF Group number for SDIF status codes.
kStatusGroup_SPIFI Group number for SPIFI status codes.
kStatusGroup_OTP Group number for OTP status codes.
kStatusGroup_MCAN Group number for MCAN status codes.
kStatusGroup_NOTIFIER Group number for NOTIFIER status codes.

kStatusGroup_DebugConsole Group number for debug console status codes.
kStatusGroup_ApplicationRangeStart Starting number for application groups.

8.4.2 enum _generic_status

8.4.3 enum SYSCON_RSTn_t

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

Enumerator

kFLASH_RST_SHIFT_RSTn Flash controller reset control
kFMC_RST_SHIFT_RSTn Flash accelerator reset control
kEEPROM_RST_SHIFT_RSTn EEPROM reset control
kSPIFI_RST_SHIFT_RSTn SPIFI reset control
kMUX_RST_SHIFT_RSTn Input mux reset control
kIOCON_RST_SHIFT_RSTn IOCON reset control
kGPIO0_RST_SHIFT_RSTn GPIO0 reset control
kGPIO1_RST_SHIFT_RSTn GPIO1 reset control
kGPIO2_RST_SHIFT_RSTn GPIO2 reset control
kGPIO3_RST_SHIFT_RSTn GPIO3 reset control
kPINT_RST_SHIFT_RSTn Pin interrupt (PINT) reset control
kGINT_RST_SHIFT_RSTn Grouped interrupt (PINT) reset control.
kDMA_RST_SHIFT_RSTn DMA reset control
kCRC_RST_SHIFT_RSTn CRC reset control
kWWDT_RST_SHIFT_RSTn Watchdog timer reset control
kADC0_RST_SHIFT_RSTn ADC0 reset control
kMRT_RST_SHIFT_RSTn Multi-rate timer (MRT) reset control
kSCT0_RST_SHIFT_RSTn SCTimer/PWM 0 (SCT0) reset control
kMCAN0_RST_SHIFT_RSTn MCAN0 reset control
kMCAN1_RST_SHIFT_RSTn MCAN1 reset control
kUTICK_RST_SHIFT_RSTn Micro-tick timer reset control
kFC0_RST_SHIFT_RSTn Flexcomm Interface 0 reset control
kFC1_RST_SHIFT_RSTn Flexcomm Interface 1 reset control
kFC2_RST_SHIFT_RSTn Flexcomm Interface 2 reset control
kFC3_RST_SHIFT_RSTn Flexcomm Interface 3 reset control
kFC4_RST_SHIFT_RSTn Flexcomm Interface 4 reset control
kFC5_RST_SHIFT_RSTn Flexcomm Interface 5 reset control
kFC6_RST_SHIFT_RSTn Flexcomm Interface 6 reset control
kFC7_RST_SHIFT_RSTn Flexcomm Interface 7 reset control
kDMIC_RST_SHIFT_RSTn Digital microphone interface reset control
kCT32B2_RST_SHIFT_RSTn CT32B2 reset control

kUSB0D_RST_SHIFT_RSTn USB0D reset control
kCT32B0_RST_SHIFT_RSTn CT32B0 reset control
kCT32B1_RST_SHIFT_RSTn CT32B1 reset control
kLCD_RST_SHIFT_RSTn LCD reset control
kSDIO_RST_SHIFT_RSTn SDIO reset control
kUSB1H_RST_SHIFT_RSTn USB1H reset control
kUSB1D_RST_SHIFT_RSTn USB1D reset control
kUSB1RAM_RST_SHIFT_RSTn USB1RAM reset control
kEMC_RST_SHIFT_RSTn EMC reset control
kETH_RST_SHIFT_RSTn ETH reset control
kGPIO4_RST_SHIFT_RSTn GPIO4 reset control
kGPIO5_RST_SHIFT_RSTn GPIO5 reset control
kAES_RST_SHIFT_RSTn AES reset control
kOTP_RST_SHIFT_RSTn OTP reset control
kRNG_RST_SHIFT_RSTn RNG reset control
kFC8_RST_SHIFT_RSTn Flexcomm Interface 8 reset control
kFC9_RST_SHIFT_RSTn Flexcomm Interface 9 reset control
kUSB0HMR_RST_SHIFT_RSTn USB0HMR reset control
kUSB0HSL_RST_SHIFT_RSTn USB0HSL reset control
kSHA_RST_SHIFT_RSTn SHA reset control
kSC0_RST_SHIFT_RSTn SC0 reset control
kSC1_RST_SHIFT_RSTn SC1 reset control
kCT32B3_RST_SHIFT_RSTn CT32B3 reset control
kCT32B4_RST_SHIFT_RSTn CT32B4 reset control

8.5 Function Documentation

8.5.1 static void EnableIRQ (IRQn_Type *interrupt*) [inline], [static]

Enable the interrupt not routed from intmux.

Parameters

interrupt	The IRQ number.

8.5.2 static void DisableIRQ (IRQn_Type *interrupt*) [inline], [static]

Disable the interrupt not routed from intmux.

Parameters

interrupt	The IRQ number.

8.5.3 static uint32_t DisableGlobalIRQ (void) [inline], [static]

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

8.5.4 static void EnableGlobalIRQ (uint32_t *primask*) [inline], [static]

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convinience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

primask	value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

8.5.5 uint32_t InstallIRQHandler (IRQn_Type *irq*, uint32_t *irqHandler*)

Parameters

irq	IRQ number
irqHandler	IRQ handler address

Returns

The old IRQ handler address

8.5.6 void EnableDeepSleepIRQ (IRQn_Type *interrupt*)

Enable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is

possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

Note

This function also enables the interrupt in the NVIC (EnableIRQ() is called internally).

Parameters

interrupt	The IRQ number.

8.5.7 void DisableDeepSleepIRQ (IRQn_Type *interrupt*)

Disable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

Note

This function also disables the interrupt in the NVIC (DisableIRQ() is called internally).

Parameters

interrupt	The IRQ number.

8.5.8 void RESET_SetPeripheralReset (reset_ip_name_t *peripheral*)

Asserts reset signal to specified peripheral module.

Parameters

peripheral	Assert reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

8.5.9 void RESET_ClearPeripheralReset (reset_ip_name_t *peripheral*)

Clears reset signal to specified peripheral module, allows it to operate.

Parameters

peripheral	Clear reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

8.5.10 void RESET_PeripheralReset (reset_ip_name_t *peripheral*)

Reset peripheral module.

Parameters

peripheral	Peripheral to reset. The enum argument contains encoding of reset register and reset bit position in the reset register.

Chapter 9
Debug Console

9.1 Overview

This part describes the programming interface of the debug console driver. The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

9.2 Function groups

9.2.1 Initialization

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

```
/*
 * @brief Initializes the the peripheral used to debug messages.
 *
 * @param baseAddr      Indicates which address of the peripheral is used to send debug messages.
 * @param baudRate      The desired baud rate in bits per second.
 * @param device        Low level device type for the debug console, can be one of:
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_UART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPSCI,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_USBCDC.
 * @param clkSrcFreq    Frequency of peripheral source clock.
 *
 * @return              Whether initialization was successful or not.
 */
status_t DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device, uint32_t clkSrcFreq)
```

Selects the supported debug console hardware device type, such as

```
DEBUG_CONSOLE_DEVICE_TYPE_NONE
DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
DEBUG_CONSOLE_DEVICE_TYPE_UART
DEBUG_CONSOLE_DEVICE_TYPE_LPUART
DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral. The debug console state is stored in the debug_console_state_t structure, such as shown here.

```
typedef struct DebugConsoleState
{
    uint8_t                    type;
    void*                      base;
    debug_console_ops_t        ops;
} debug_console_state_t;
```

This example shows how to call the DbgConsole_Init() given the user configuration structure.

```
uint32_t uartClkSrcFreq = CLOCK_GetFreq(BOARD_DEBUG_UART_CLKSRC);

DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE,
      DEBUG_CONSOLE_DEVICE_TYPE_UART, uartClkSrcFreq);
```

9.2.2 Advanced Feature

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

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

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

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

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

length	Description
Do not support	

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

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

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

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

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

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

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

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

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the KSDK printf/scanf.

```
#if SDK_DEBUGCONSOLE    /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF                DbgConsole_Printf
#define SCANF                 DbgConsole_Scanf
#define PUTCHAR               DbgConsole_Putchar
#define GETCHAR               DbgConsole_Getchar
#else                   /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF                printf
#define SCANF                 scanf
#define PUTCHAR               putchar
#define GETCHAR               getchar
#endif /* SDK_DEBUGCONSOLE */
```

9.3 Typical use case

Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

Typical use case

Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

Print out failure messages using KSDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \" %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file ,
      line, func);
    for (;;)
    {}
}
```

Note:

To use 'printf' and 'scanf' for GNUC Base, add file **'fsl_sbrk.c'** in path: **..\{package}\devices\{subset}\utilities\fsl_sbrk.c** to your project.

Modules

- Semihosting

9.4 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism can be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

9.4.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

9.4.2 Guide Semihosting for Keil µVision

NOTE: Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

Step 1: Prepare code

Remove function fputc and fgetc is used to support KEIL in "fsl_debug_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;       /* used for Debug Input */
```

```
struct __FILE
{
    int handle;
};
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{ /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}
```

Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select “Debug” tab, select “J-Link/J-Trace Cortex” and click “Setting button”.
4. Select “Debug” tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

Step 4: Building the project

1. Choose “Debug” on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

9.4.3 Guide Semihosting for KDS

NOTE: After the setting use "printf" for debugging.

Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select “Libraries” on “Cross ARM C Linker” and delete “nosys”.
3. Select “Miscellaneous” on "Cross ARM C Linker”, add “-specs=rdimon.specs” to “Other link flages” and tick “Use newlib-nano”, and click OK.

Step 2: Building the project

1. In menu bar, choose Project>Build Project.

Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick “Enable semihosting and Telnet”. Press “Apply” and “Debug”.
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

9.4.4 Guide Semihosting for ATL

NOTE: J-Link has to be used to enable semihosting.

Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
  /* Implement your write code here. This is used by puts and printf. */
  int i=0;
  for(i=0 ; i<len ; i++)
    ITM_SendChar((*ptr++));
  return len;
}
```

Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Aplication" choose "-Semihosting_ATL_xxx debug J-Link".
2. In tab "Debugger" set up as follows.
   - JTAG mode must be selected

- SWV tracing must be enabled
- Enter the Core Clock frequency, which is hardware board-specific.
- Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.

3. Click "Apply" and "Debug".

Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view “SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high-speed). Save the SWV configuration by clicking the OK button. The configuration is saved with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board to enable SWV trace recoding. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

9.4.5 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
   - "Host Name (or IP address)" : localhost
   - "Port" :2333
   - "Connection type" : Telet.
   - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --

defsym=__heap_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__heap_size__=0x2000")

Step 2: Building the project

1. Change "CMakeLists.txt":
   Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=nano.specs")"
   to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} –specs=rdimon.specs")"
   Replace paragraph
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
   To
   SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-

G} --specs=rdimon.specs ")
Remove
target_link_libraries(semihosting_ARMGCC.elf debug nosys)

2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

(a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

(b) After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 10
DMA: Direct Memory Access Controller Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMA Operation

```
dma_transfer_config_t transferConfig;
uint32_t transferDone = false;

DMA_Init(DMA0);
DMA_EnableChannel(DMA0, channel);
DMA_SetChannelPriority(DMA0, channel,
      kDMA_ChannelPriority0);
DMA_CreateHandle(&g_DMA_Handle, DMA0, channel);
DMA_SetCallback(&g_DMA_Handle, DMA_Callback, &transferDone);
DMA_PrepareTransfer(&transferConfig, srcAddr, destAddr, transferByteWidth, transferBytes
      ,
                    kDMA_MemoryToMemory, NULL);
DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig);
DMA_StartTransfer(&g_DMA_Handle);
/* Wait for DMA transfer finish */
while (transferDone != true);
```

Files

- file fsl_dma.h

Data Structures

- struct dma_descriptor_t
  DMA descriptor structure. More...
- struct dma_xfercfg_t
  DMA transfer configuration. More...
- struct dma_channel_trigger_t
  DMA channel trigger. More...
- struct dma_transfer_config_t
  DMA transfer configuration. More...
- struct dma_handle_t
  DMA transfer handle structure. More...

Typedefs

- typedef void(* dma_callback)(struct _dma_handle *handle, void *userData, bool transferDone, uint32_t intmode)
  Define Callback function for DMA.

Typical use case

Enumerations

- enum dma_priority_t {
  kDMA_ChannelPriority0 = 0,
  kDMA_ChannelPriority1,
  kDMA_ChannelPriority2,
  kDMA_ChannelPriority3,
  kDMA_ChannelPriority4,
  kDMA_ChannelPriority5,
  kDMA_ChannelPriority6,
  kDMA_ChannelPriority7 }
  DMA channel priority.
- enum dma_irq_t {
  kDMA_IntA,
  kDMA_IntB }
  DMA interrupt flags.
- enum dma_trigger_type_t {
  kDMA_NoTrigger = 0,
  kDMA_LowLevelTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1) | DMA_CHANNEL_CFG_TRIGTYPE(1),
  kDMA_HighLevelTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1) | DMA_CHANNEL_CFG_TRIGTYPE(1) | DMA_CHANNEL_CFG_TRIGPOL(1),
  kDMA_FallingEdgeTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1),
  kDMA_RisingEdgeTrigger = DMA_CHANNEL_CFG_HWTRIGEN(1) | DMA_CHANNEL_CFG_TRIGPOL(1) }
  DMA trigger type.
- enum dma_trigger_burst_t {
  kDMA_SingleTransfer = 0,
  kDMA_LevelBurstTransfer = DMA_CHANNEL_CFG_TRIGBURST(1),
  kDMA_EdgeBurstTransfer1 = DMA_CHANNEL_CFG_TRIGBURST(1),
  kDMA_EdgeBurstTransfer2 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(1),
  kDMA_EdgeBurstTransfer4 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(2),
  kDMA_EdgeBurstTransfer8 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(3),
  kDMA_EdgeBurstTransfer16 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(4),
  kDMA_EdgeBurstTransfer32 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(5),
  kDMA_EdgeBurstTransfer64 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(6),
  kDMA_EdgeBurstTransfer128 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL_CFG_BURSTPOWER(7),
  kDMA_EdgeBurstTransfer256 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL-

_CFG_BURSTPOWER(8),
kDMA_EdgeBurstTransfer512 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNEL-_CFG_BURSTPOWER(9),
kDMA_EdgeBurstTransfer1024 = DMA_CHANNEL_CFG_TRIGBURST(1) | DMA_CHANNE-L_CFG_BURSTPOWER(10) }
DMA trigger burst.

- enum dma_burst_wrap_t {
kDMA_NoWrap = 0,
kDMA_SrcWrap = DMA_CHANNEL_CFG_SRCBURSTWRAP(1),
kDMA_DstWrap = DMA_CHANNEL_CFG_DSTBURSTWRAP(1),
kDMA_SrcAndDstWrap = DMA_CHANNEL_CFG_SRCBURSTWRAP(1) | DMA_CHANNEL-_CFG_DSTBURSTWRAP(1) }
DMA burst wrapping.
- enum dma_transfer_type_t {
kDMA_MemoryToMemory = 0x0U,
kDMA_PeripheralToMemory,
kDMA_MemoryToPeripheral,
kDMA_StaticToStatic }
DMA transfer type.
- enum _dma_transfer_status { kStatus_DMA_Busy = MAKE_STATUS(kStatusGroup_DMA, 0) }
DMA transfer status.

Driver version

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

DMA initialization and De-initialization

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

DMA Channel Operation

- static bool DMA_ChannelIsActive (DMA_Type *base, uint32_t channel)
Return whether DMA channel is processing transfer.
- static void DMA_EnableChannelInterrupts (DMA_Type *base, uint32_t channel)
Enables the interrupt source for the DMA transfer.
- static void DMA_DisableChannelInterrupts (DMA_Type *base, uint32_t channel)
Disables the interrupt source for the DMA transfer.
- static void DMA_EnableChannel (DMA_Type *base, uint32_t channel)
Enable DMA channel.
- static void DMA_DisableChannel (DMA_Type *base, uint32_t channel)
Disable DMA channel.
- static void DMA_EnableChannelPeriphRq (DMA_Type *base, uint32_t channel)
Set PERIPHREQEN of channel configuration register.
- static void DMA_DisableChannelPeriphRq (DMA_Type *base, uint32_t channel)

Get PERIPHREQEN value of channel configuration register.

- void DMA_ConfigureChannelTrigger (DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger)
  Set trigger settings of DMA channel.
- uint32_t DMA_GetRemainingBytes (DMA_Type *base, uint32_t channel)
  Gets the remaining bytes of the current DMA descriptor transfer.
- static void DMA_SetChannelPriority (DMA_Type *base, uint32_t channel, dma_priority_t priority)
  Set priority of channel configuration register.
- static dma_priority_t DMA_GetChannelPriority (DMA_Type *base, uint32_t channel)
  Get priority of channel configuration register.
- void DMA_CreateDescriptor (dma_descriptor_t *desc, dma_xfercfg_t *xfercfg, void *srcAddr, void *dstAddr, void *nextDesc)
  Create application specific DMA descriptor to be used in a chain in transfer.

DMA Transactional Operation

- void DMA_AbortTransfer (dma_handle_t *handle)
  Abort running transfer by handle.
- 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)
  Installs a callback function for the DMA transfer.
- void DMA_PrepareTransfer (dma_transfer_config_t *config, void *srcAddr, void *dstAddr, uint32_t byteWidth, uint32_t transferBytes, dma_transfer_type_t type, void *nextDesc)
  Prepares the DMA transfer structure.
- status_t DMA_SubmitTransfer (dma_handle_t *handle, dma_transfer_config_t *config)
  Submits the DMA transfer request.
- void DMA_StartTransfer (dma_handle_t *handle)
  DMA start transfer.
- void DMA_HandleIRQ (void)
  DMA IRQ handler for descriptor transfer complete.

10.3 Data Structure Documentation

10.3.1 struct dma_descriptor_t

Data Fields

- uint32_t xfercfg
  Transfer configuration.
- void * srcEndAddr
  Last source address of DMA transfer.
- void * dstEndAddr
  Last destination address of DMA transfer.
- void * linkToNextDesc
  Address of next DMA descriptor in chain.

10.3.2 struct dma_xfercfg_t

Data Fields

- bool valid
  Descriptor is ready to transfer.
- bool reload
  Reload channel configuration register after current descriptor is exhausted.
- bool swtrig
  Perform software trigger.
- bool clrtrig
  Clear trigger.
- bool intA
  Raises IRQ when transfer is done and set IRQA status register flag.
- bool intB
  Raises IRQ when transfer is done and set IRQB status register flag.
- uint8_t byteWidth
  Byte width of data to transfer.
- uint8_t srcInc
  Increment source address by 'srcInc' x 'byteWidth'.
- uint8_t dstInc
  Increment destination address by 'dstInc' x 'byteWidth'.
- uint16_t transferCount
  Number of transfers.

10.3.2.0.0.5 Field Documentation

10.3.2.0.0.5.1 bool dma_xfercfg_t::swtrig

Transfer if fired when 'valid' is set

10.3.3 struct dma_channel_trigger_t

10.3.4 struct dma_transfer_config_t

Data Fields

- uint8_t * srcAddr
  Source data address.
- uint8_t * dstAddr
  Destination data address.
- uint8_t * nextDesc
  Chain custom descriptor.
- dma_xfercfg_t xfercfg
  Transfer options.
- bool isPeriph
  DMA transfer is driven by peripheral.

10.3.5 struct dma_handle_t

Data Fields

- dma_callback callback
  Callback function.
- void * userData
  Callback function parameter.
- DMA_Type * base
  DMA peripheral base address.
- uint8_t channel
  DMA channel number.

10.3.5.0.0.6 Field Documentation

10.3.5.0.0.6.1 dma_callback dma_handle_t::callback

Invoked when transfer of descriptor with interrupt flag finishes

10.4 Macro Definition Documentation

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

Version 2.0.0.

10.5 Typedef Documentation

10.5.1 typedef void(* dma_callback)(struct _dma_handle *handle, void *userData, bool transferDone, uint32_t intmode)

10.6 Enumeration Type Documentation

10.6.1 enum dma_priority_t

Enumerator

kDMA_ChannelPriority0	Highest channel priority - priority 0.
kDMA_ChannelPriority1	Channel priority 1.
kDMA_ChannelPriority2	Channel priority 2.
kDMA_ChannelPriority3	Channel priority 3.
kDMA_ChannelPriority4	Channel priority 4.
kDMA_ChannelPriority5	Channel priority 5.
kDMA_ChannelPriority6	Channel priority 6.
kDMA_ChannelPriority7	Lowest channel priority - priority 7.

10.6.2 enum dma_irq_t

Enumerator

kDMA_IntA DMA interrupt flag A.
kDMA_IntB DMA interrupt flag B.

10.6.3 enum dma_trigger_type_t

Enumerator

kDMA_NoTrigger Trigger is disabled.
kDMA_LowLevelTrigger Low level active trigger.
kDMA_HighLevelTrigger High level active trigger.
kDMA_FallingEdgeTrigger Falling edge active trigger.
kDMA_RisingEdgeTrigger Rising edge active trigger.

10.6.4 enum dma_trigger_burst_t

Enumerator

kDMA_SingleTransfer Single transfer.
kDMA_LevelBurstTransfer Burst transfer driven by level trigger.
kDMA_EdgeBurstTransfer1 Perform 1 transfer by edge trigger.
kDMA_EdgeBurstTransfer2 Perform 2 transfers by edge trigger.
kDMA_EdgeBurstTransfer4 Perform 4 transfers by edge trigger.
kDMA_EdgeBurstTransfer8 Perform 8 transfers by edge trigger.
kDMA_EdgeBurstTransfer16 Perform 16 transfers by edge trigger.
kDMA_EdgeBurstTransfer32 Perform 32 transfers by edge trigger.
kDMA_EdgeBurstTransfer64 Perform 64 transfers by edge trigger.
kDMA_EdgeBurstTransfer128 Perform 128 transfers by edge trigger.
kDMA_EdgeBurstTransfer256 Perform 256 transfers by edge trigger.
kDMA_EdgeBurstTransfer512 Perform 512 transfers by edge trigger.
kDMA_EdgeBurstTransfer1024 Perform 1024 transfers by edge trigger.

10.6.5 enum dma_burst_wrap_t

Enumerator

kDMA_NoWrap Wrapping is disabled.
kDMA_SrcWrap Wrapping is enabled for source.
kDMA_DstWrap Wrapping is enabled for destination.
kDMA_SrcAndDstWrap Wrapping is enabled for source and destination.

10.6.6 enum dma_transfer_type_t

Enumerator

kDMA_MemoryToMemory Transfer from memory to memory (increment source and destination)
kDMA_PeripheralToMemory Transfer from peripheral to memory (increment only destination)
kDMA_MemoryToPeripheral Transfer from memory to peripheral (increment only source)
kDMA_StaticToStatic Peripheral to static memory (do not increment source or destination)

10.6.7 enum _dma_transfer_status

Enumerator

kStatus_DMA_Busy Channel is busy and can't handle the transfer request.

10.7 Function Documentation

10.7.1 void DMA_Init (DMA_Type * *base*)

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

base	DMA peripheral base address.

10.7.2 void DMA_Deinit (DMA_Type * *base*)

This function gates the DMA clock.

Parameters

base	DMA peripheral base address.

10.7.3 static bool DMA_ChannelIsActive (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

True for active state, false otherwise.

10.7.4 static void DMA_EnableChannelInterrupts (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

10.7.5 static void DMA_DisableChannelInterrupts (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

10.7.6 static void DMA_EnableChannel (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

10.7.7 static void DMA_DisableChannel (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

10.7.8 static void DMA_EnableChannelPeriphRq (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

10.7.9 static void DMA_DisableChannelPeriphRq (DMA_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

True for enabled PeriphRq, false for disabled.

10.7.10 void DMA_ConfigureChannelTrigger (DMA_Type * *base,* uint32_t *channel,* dma_channel_trigger_t * *trigger*)

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

trigger	trigger configuration.

10.7.11 uint32_t DMA_GetRemainingBytes (DMA_Type * *base*, uint32_t *channel*)

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

The number of bytes which have not been transferred yet.

10.7.12 static void DMA_SetChannelPriority (DMA_Type * *base*, uint32_t *channel*, dma_priority_t *priority*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.
priority	Channel priority value.

10.7.13 static dma_priority_t DMA_GetChannelPriority (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

base	DMA peripheral base address.
channel	DMA channel number.

Returns

Channel priority value.

10.7.14 void DMA_CreateDescriptor (dma_descriptor_t * *desc*, dma_xfercfg_t * *xfercfg*, void * *srcAddr*, void * *dstAddr*, void * *nextDesc*)

Parameters

desc	DMA descriptor address.
xfercfg	Transfer configuration for DMA descriptor.
srcAddr	Address of last item to transmit
dstAddr	Address of last item to receive.
nextDesc	Address of next descriptor in chain.

10.7.15 void DMA_AbortTransfer (dma_handle_t * *handle*)

This function aborts DMA transfer specified by handle.

Parameters

handle	DMA handle pointer.

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

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

Parameters

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

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

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

Parameters

handle	DMA handle pointer.
callback	DMA callback function pointer.
userData	Parameter for callback function.

10.7.18 void DMA_PrepareTransfer (dma_transfer_config_t * *config*, void * *srcAddr*, void * *dstAddr*, uint32_t *byteWidth*, uint32_t *transferBytes*, dma_transfer_type_t *type*, void * *nextDesc*)

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

Parameters

config	The user configuration structure of type dma_transfer_t.
srcAddr	DMA transfer source address.
dstAddr	DMA transfer destination address.
byteWidth	DMA transfer destination address width(bytes).
transferBytes	DMA transfer bytes to be transferred.
type	DMA transfer type.
nextDesc	Chain custom descriptor to transfer.

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).

10.7.19 status_t DMA_SubmitTransfer (dma_handle_t * *handle*, dma_transfer_config_t * *config*)

This function submits the DMA 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	DMA handle pointer.
config	Pointer to DMA transfer configuration structure.

Return values

kStatus_DMA_Success	It means submit transfer request succeed.
kStatus_DMA_QueueFull	It means TCD queue is full. Submit transfer request is not allowed.
kStatus_DMA_Busy	It means the given channel is busy, need to submit request later.

10.7.20 void DMA_StartTransfer (dma_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	DMA handle pointer.

10.7.21 void DMA_HandleIRQ (void)

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

Chapter 11
DMIC: Digital Microphone

11.1 Overview

The SDK provides Peripheral driver for the Digital Microphone (DMIC) module.

DMIC driver is created to help user to operate the DMIC module better. This driver can be used to performed basic and advance DMIC operations. Driver can be used to transfer data from DMIC to memory using DMA as well as in interrupt mode. DMIC, DMA transfer in pingpong mode is preferred as DMIC is a streaming device.

11.2 Function groups

11.2.1 Initialization and deinitialization

This function group implements DMIC initialization and deinitialization API. DMIC_Init() function Enables the clock to the DMIC register interface. DMIC_Dinit() function Disables the clock to the DMIC register interface.

11.2.2 Configuration

This function group implements DMIC configration API. DMIC_ConfigIO()function configures the use of PDM(Pulse Density moulation) pins. DMIC_SetOperationMode()function configures the mode of operation either in DMA or in interrupt. DMIC_ConfigChannel() function configures the various property of a DMIC channel. DMIC_Use2fs()function configures the Clock scaling used for PCM data output. DMIC_EnableChannnel() function enables a particualr DMIC channel. DMIC_FifoChannel() function configures FIFO settings for a DMIC channel.

11.2.3 DMIC Data and status

This function group implements the API to get data and status of DMIC FIFO. DMIC_FifoGetStatus() function gives the status of a DMIC FIFO. DMIC_ClearStatus() function clears the status of a DMIC FIFO. DMIC_FifoGetData() function gets data from a DMIC FIFO.

11.2.4 DMIC Interrupt Functions

DMIC_EnablebleIntCallback() enables the interrupt for the selected DMIC peripheral. DMIC_DisableIntCallback() disables the interrupt for the selected DMIC peripheral.

11.2.5 DMIC HWVAD Functions

This function group implements the API for HWVAD DMIC_SetGainNoiseEstHwvad() Sets the gain value for the noise estimator. DMIC_SetGainSignalEstHwvad() Sets the gain value for the signal estimator. DMIC_SetFilterCtrlHwvad() Sets the hwvad filter cutoff frequency parameter. DMIC_SetInputGainHwvad() Sets the input gain of hwvad. DMIC_CtrlClrIntrHwvad() Clears hwvad internal interrupt flag. DMIC_FilterResetHwvad() Resets hwvad filters. DMIC_GetNoiseEnvlpEst() Gets the value from output of the filter z7.

11.2.6 DMIC HWVAD Interrupt Functions

DMIC_HwvadEnableIntCallback() enables the hwvad interrupt for the selected DMIC peripheral. DMIC_HwvadDisableIntCallback() disables the hwvad interrupt for the selected DMIC peripheral.

11.3 Typical use case

11.3.1 DMIC DMA Configuration

```
dmic_channel_config_t dmic_channel_cfg;
dma_transfer_config_t transferConfig;

BOARD_InitHardware();

APPInit();
dmic_channel_cfg.divhfclk = kDMIC_Pdm_Div1;
dmic_channel_cfg.osr = 25U;
dmic_channel_cfg.gainshft = 1U;
dmic_channel_cfg.preac2coef = kDMIC_Comp0_0;
dmic_channel_cfg.preac4coef = kDMIC_Comp0_0;
dmic_channel_cfg.dc_cut_level = kDMIC_Dc_Cut155;
dmic_channel_cfg.post_dc_gain_reduce = 0U;
dmic_channel_cfg.saturate16bit = 1U;
dmic_channel_cfg.sample_rate = kDMIC_Phy_Full_Speed;
DMIC_Init(DMIC0);

DMIC_CfgIO(DMIC0, kPDM_Dual);
DMIC_Use2fs(DMIC0, true);
DMIC_SetOpMode(DMIC0, kDMIC_Op_Dma);
DMIC_CfgChannel(DMIC0, kDMIC_Ch0, kDMIC_Left, &dmic_channel_cfg);

DMIC_FifoChannel(DMIC0, kDMIC_Ch0, FIFO_DEPTH, true, true);

DMIC_EnableChannnel(DMIC0, DMIC_CHANEN_EN_CH0(1));

PRINTF("Configure DMA\r\n");

DMA_Init(DMA0);

DMA_EnableChannel(DMA0, DMAREQ_DMIC0);

/* Request dma channels from DMA manager. */
DMA_CreateHandle(&g_DMA_Handle, DMA0, DMAREQ_DMIC0);

DMA_SetCallback(&g_DMA_Handle, DMA_Callback, NULL);
DMA_PrepareTransfer(&transferConfig, (void *)&DMIC0->CHANNEL[kDMIC_Ch0].FIFO_DATA,
      g_data_buffer, 2, BUFFER_LENGTH,
                  kDMA_PeripheralToMemory, &g_pingpong_desc[1]);
```

```
DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig);
transferConfig.xfercfg.intA = false;
transferConfig.xfercfg.intB = true;
DMA_CreateDescriptor(&g_pingpong_desc[1], &transferConfig.
      xfercfg, (void *)&DMIC0->CHANNEL[kDMIC_Ch0].FIFO_DATA,
                       &g_data_buffer[BUFFER_LENGTH / 2], &g_pingpong_desc[0]);
transferConfig.xfercfg.intA = true;
transferConfig.xfercfg.intB = false;
DMA_CreateDescriptor(&g_pingpong_desc[0], &transferConfig.
      xfercfg, (void *)&DMIC0->CHANNEL[kDMIC_Ch0].FIFO_DATA,
                       &g_data_buffer[0], &g_pingpong_desc[1]);
DMA_StartTransfer(&g_DMA_Handle);
```

11.3.2 DMIC use case

```
void DMA_Callback(dma_handle_t *handle, void *param, bool transferDone, uint32_t tcds)
{
    if (tcds == kDMA_IntB)
    {
    }
    if (tcds == kDMA_IntA)
    {
    }
    if (first_int == 0U)
    {
        audioPosition = 0U;
        first_int = 1U;
    }
}
```

Modules

- DMIC DMA Driver
- DMIC Driver

11.4 DMIC Driver

11.4.1 Overview

Files

- file fsl_dmic.h

Data Structures

- struct dmic_channel_config_t
  DMIC Channel configuration structure. *More...*

Typedefs

- typedef void(∗ dmic_callback_t)(void)
  DMIC Callback function.
- typedef void(∗ dmic_hwvad_callback_t)(void)
  HWVAD Callback function.

Enumerations

- enum operation_mode_t {
  kDMIC_OperationModePoll = 0U,
  kDMIC_OperationModeInterrupt = 1U,
  kDMIC_OperationModeDma = 2U }
  DMIC different operation modes.
- enum stereo_side_t {
  kDMIC_Left = 0U,
  kDMIC_Right = 1U }
  DMIC left/right values.
- enum pdm_div_t {

kDMIC_PdmDiv1 = 0U,
kDMIC_PdmDiv2 = 1U,
kDMIC_PdmDiv3 = 2U,
kDMIC_PdmDiv4 = 3U,
kDMIC_PdmDiv6 = 4U,
kDMIC_PdmDiv8 = 5U,
kDMIC_PdmDiv12 = 6U,
kDMIC_PdmDiv16 = 7U,
kDMIC_PdmDiv24 = 8U,
kDMIC_PdmDiv32 = 9U,
kDMIC_PdmDiv48 = 10U,
kDMIC_PdmDiv64 = 11U,
kDMIC_PdmDiv96 = 12U,
kDMIC_PdmDiv128 = 13U }
DMIC Clock pre-divider values.

- enum compensation_t {
  kDMIC_CompValueZero = 0U,
  kDMIC_CompValueNegativePoint16 = 1U,
  kDMIC_CompValueNegativePoint15 = 2U,
  kDMIC_CompValueNegativePoint13 = 3U }
  Pre-emphasis Filter coefficient value for 2FS and 4FS modes.
- enum dc_removal_t {
  kDMIC_DcNoRemove = 0U,
  kDMIC_DcCut155 = 1U,
  kDMIC_DcCut78 = 2U,
  kDMIC_DcCut39 = 3U }
  DMIC DC filter control values.
- enum dmic_io_t {
  kDMIC_PdmDual = 0U,
  kDMIC_PdmStereo = 4U,
  kDMIC_PdmBypass = 3U,
  kDMIC_PdmBypassClk0 = 1U,
  kDMIC_PdmBypassClk1 = 2U }
  DMIC IO configiration.
- enum dmic_channel_t {
  kDMIC_Channel0 = 0U,
  kDMIC_Channel1 = 1U }
  DMIC Channel number.
- enum dmic_phy_sample_rate_t {
  kDMIC_PhyFullSpeed = 0U,
  kDMIC_PhyHalfSpeed = 1U }
  DMIC and decimator sample rates.
- enum _dmic_status {
  kStatus_DMIC_Busy = MAKE_STATUS(kStatusGroup_DMIC, 0),
  kStatus_DMIC_Idle = MAKE_STATUS(kStatusGroup_DMIC, 1),
  kStatus_DMIC_OverRunError = MAKE_STATUS(kStatusGroup_DMIC, 2),

kStatus_DMIC_UnderRunError = MAKE_STATUS(kStatusGroup_DMIC, 3) }
DMIC transfer status.

Functions

- uint32_t DMIC_GetInstance (DMIC_Type *base)
  Get the DMIC instance from peripheral base address.
- void DMIC_Init (DMIC_Type *base)
  Turns DMIC Clock on.
- void DMIC_DeInit (DMIC_Type *base)
  Turns DMIC Clock off.
- void DMIC_ConfigIO (DMIC_Type *base, dmic_io_t config)
  Configure DMIC io.
- void DMIC_SetOperationMode (DMIC_Type *base, operation_mode_t mode)
  Set DMIC operating mode.
- void DMIC_ConfigChannel (DMIC_Type *base, dmic_channel_t channel, stereo_side_t side, dmic_channel_config_t *channel_config)
  Configure DMIC channel.
- void DMIC_Use2fs (DMIC_Type *base, bool use2fs)
  Configure Clock scaling.
- void DMIC_EnableChannnel (DMIC_Type *base, uint32_t channelmask)
  Enable a particualr channel.
- void DMIC_FifoChannel (DMIC_Type *base, uint32_t channel, uint32_t trig_level, uint32_t enable, uint32_t resetn)
  Configure fifo settings for DMIC channel.
- static uint32_t DMIC_FifoGetStatus (DMIC_Type *base, uint32_t channel)
  Get FIFO status.
- static void DMIC_FifoClearStatus (DMIC_Type *base, uint32_t channel, uint32_t mask)
  Clear FIFO status.
- static uint32_t DMIC_FifoGetData (DMIC_Type *base, uint32_t channel)
  Get FIFO data.
- void DMIC_EnableIntCallback (DMIC_Type *base, dmic_callback_t cb)
  Enable callback.
- void DMIC_DisableIntCallback (DMIC_Type *base, dmic_callback_t cb)
  Disable callback.

DMIC version

- #define FSL_DMIC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  DMIC driver version 2.0.0.

11.4.2 Data Structure Documentation

11.4.2.1 struct dmic_channel_config_t

Data Fields

- pdm_div_t divhfclk
  DMIC Clock pre-divider values.
- uint32_t osr
  oversampling rate(CIC decimation rate) for PCM
- int32_t gainshft
  4FS PCM data gain control
- compensation_t preac2coef
  Pre-emphasis Filter coefficient value for 2FS.
- compensation_t preac4coef
  Pre-emphasis Filter coefficient value for 4FS.
- dc_removal_t dc_cut_level
  DMIC DC filter control values.
- uint32_t post_dc_gain_reduce
  Fine gain adjustment in the form of a number of bits to downshift.
- dmic_phy_sample_rate_t sample_rate
  DMIC and decimator sample rates.
- bool saturate16bit
  Selects 16-bit saturation.

11.4.2.1.0.7 Field Documentation

11.4.2.1.0.7.1 dc_removal_t dmic_channel_config_t::dc_cut_level

11.4.2.1.0.7.2 bool dmic_channel_config_t::saturate16bit

0 means results roll over if out range and do not saturate. 1 means if the result overflows, it saturates at 0xFFFF for positive overflow and 0x8000 for negative overflow.

11.4.3 Macro Definition Documentation

11.4.3.1 #define FSL_DMIC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

11.4.4 Typedef Documentation

11.4.4.1 typedef void(∗ dmic_callback_t)(void)

11.4.4.2 typedef void(∗ dmic_hwvad_callback_t)(void)

11.4.5 Enumeration Type Documentation

11.4.5.1 enum operation_mode_t

Enumerator

kDMIC_OperationModePoll Polling mode.
kDMIC_OperationModeInterrupt Interrupt mode.
kDMIC_OperationModeDma DMA mode.

11.4.5.2 enum stereo_side_t

Enumerator

kDMIC_Left Left Stereo channel.
kDMIC_Right Right Stereo channel.

11.4.5.3 enum pdm_div_t

Enumerator

kDMIC_PdmDiv1 DMIC pre-divider set in divide by 1.
kDMIC_PdmDiv2 DMIC pre-divider set in divide by 2.
kDMIC_PdmDiv3 DMIC pre-divider set in divide by 3.
kDMIC_PdmDiv4 DMIC pre-divider set in divide by 4.
kDMIC_PdmDiv6 DMIC pre-divider set in divide by 6.
kDMIC_PdmDiv8 DMIC pre-divider set in divide by 8.
kDMIC_PdmDiv12 DMIC pre-divider set in divide by 12.
kDMIC_PdmDiv16 DMIC pre-divider set in divide by 16.
kDMIC_PdmDiv24 DMIC pre-divider set in divide by 24.
kDMIC_PdmDiv32 DMIC pre-divider set in divide by 32.
kDMIC_PdmDiv48 DMIC pre-divider set in divide by 48.
kDMIC_PdmDiv64 DMIC pre-divider set in divide by 64.

kDMIC_PdmDiv96 DMIC pre-divider set in divide by 96.
kDMIC_PdmDiv128 DMIC pre-divider set in divide by 128.

11.4.5.4 enum compensation_t

Enumerator

kDMIC_CompValueZero Compensation 0.
kDMIC_CompValueNegativePoint16 Compensation -0.16.
kDMIC_CompValueNegativePoint15 Compensation -0.15.
kDMIC_CompValueNegativePoint13 Compensation -0.13.

11.4.5.5 enum dc_removal_t

Enumerator

kDMIC_DcNoRemove Flat response no filter.
kDMIC_DcCut155 Cut off Frequency is 155 Hz.
kDMIC_DcCut78 Cut off Frequency is 78 Hz.
kDMIC_DcCut39 Cut off Frequency is 39 Hz.

11.4.5.6 enum dmic_io_t

Enumerator

kDMIC_PdmDual Two separate pairs of PDM wires.
kDMIC_PdmStereo Stereo Mic.
kDMIC_PdmBypass Clk Bypass clocks both channels.
kDMIC_PdmBypassClk0 Clk Bypass clocks only channel0.
kDMIC_PdmBypassClk1 Clk Bypas clocks only channel1.

11.4.5.7 enum dmic_channel_t

Enumerator

kDMIC_Channel0 DMIC channel 0.
kDMIC_Channel1 DMIC channel 1.

11.4.5.8 enum dmic_phy_sample_rate_t

Enumerator

kDMIC_PhyFullSpeed Decimator gets one sample per each chosen clock edge of PDM interface.
kDMIC_PhyHalfSpeed PDM clock to Microphone is halved, decimator receives each sample twice.

11.4.5.9 enum _dmic_status

Enumerator

kStatus_DMIC_Busy DMIC is busy.
kStatus_DMIC_Idle DMIC is idle.
kStatus_DMIC_OverRunError DMIC over run Error.
kStatus_DMIC_UnderRunError DMIC under run Error.

11.4.6 Function Documentation

11.4.6.1 uint32_t DMIC_GetInstance (DMIC_Type * *base*)

Parameters

base	DMIC peripheral base address.

Returns

DMIC instance.

11.4.6.2 void DMIC_Init (DMIC_Type * *base*)

Parameters

base	: DMIC base

Returns

Nothing

11.4.6.3 void DMIC_DeInit (DMIC_Type * *base*)

Parameters

base	: DMIC base

Returns

Nothing

11.4.6.4 void DMIC_ConfigIO (DMIC_Type * *base*, dmic_io_t *config*)

Parameters

base	: The base address of DMIC interface
config	: DMIC io configuration

Returns

Nothing

11.4.6.5 void DMIC_SetOperationMode (DMIC_Type * *base*, operation_mode_t *mode*)

Parameters

base	: The base address of DMIC interface
mode	: DMIC mode

Returns

Nothing

11.4.6.6 void DMIC_ConfigChannel (DMIC_Type * *base*, dmic_channel_t *channel*, stereo_side_t *side*, dmic_channel_config_t * *channel_config*)

Parameters

base	: The base address of DMIC interface
channel	: DMIC channel
side	: stereo_side_t, choice of left or right
channel_config	: Channel configuration

Returns

Nothing

11.4.6.7 void DMIC_Use2fs (DMIC_Type * *base,* bool *use2fs*)

Parameters

base	: The base address of DMIC interface
use2fs	: clock scaling

Returns

Nothing

11.4.6.8 void DMIC_EnableChannnel (DMIC_Type * *base,* uint32_t *channelmask*)

Parameters

base	: The base address of DMIC interface
channelmask	: Channel selection

Returns

Nothing

11.4.6.9 void DMIC_FifoChannel (DMIC_Type * *base,* uint32_t *channel,* uint32_t *trig_level,* uint32_t *enable,* uint32_t *resetn*)

Parameters

base	: The base address of DMIC interface
channel	: DMIC channel
trig_level	: FIFO trigger level
enable	: FIFO level
resetn	: FIFO reset

Returns

Nothing

11.4.6.10 static uint32_t DMIC_FifoGetStatus (DMIC_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	: The base address of DMIC interface
channel	: DMIC channel

Returns

FIFO status

11.4.6.11 static void DMIC_FifoClearStatus (DMIC_Type * *base,* uint32_t *channel,* uint32_t *mask*) [inline], [static]

Parameters

base	: The base address of DMIC interface
channel	: DMIC channel
mask	: Bits to be cleared

Returns

FIFO status

11.4.6.12 static uint32_t DMIC_FifoGetData (DMIC_Type * *base,* uint32_t *channel*) [inline], [static]

Parameters

base	: The base address of DMIC interface
channel	: DMIC channel

Returns

FIFO data

11.4.6.13 void DMIC_EnableIntCallback (DMIC_Type * *base*, dmic_callback_t *cb*)

This function enables the interrupt for the selected DMIC peripheral. The callback function is not enabled until this function is called.

Parameters

base	Base address of the DMIC peripheral.
cb	callback Pointer to store callback function.

Return values

None.	

11.4.6.14 void DMIC_DisableIntCallback (DMIC_Type * *base*, dmic_callback_t *cb*)

This function disables the interrupt for the selected DMIC peripheral.

Parameters

base	Base address of the DMIC peripheral.
cb	callback Pointer to store callback function..

Return values

None.	

11.5 DMIC DMA Driver

11.5.1 Overview

Files

- file fsl_dmic_dma.h

Data Structures

- struct dmic_transfer_t
  DMIC transfer structure. More...
- struct dmic_dma_handle_t
  DMIC DMA handle. More...

Typedefs

- typedef void(∗ dmic_dma_transfer_callback_t)(DMIC_Type ∗base, dmic_dma_handle_t ∗handle, status_t status, void ∗userData)
  DMIC transfer callback function.

DMA transactional

- status_t DMIC_TransferCreateHandleDMA (DMIC_Type ∗base, dmic_dma_handle_t ∗handle, dmic_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗rxDmaHandle)
  Initializes the DMIC handle which is used in transactional functions.
- status_t DMIC_TransferReceiveDMA (DMIC_Type ∗base, dmic_dma_handle_t ∗handle, dmic_transfer_t ∗xfer, uint32_t dmic_channel)
  Receives data using DMA.
- void DMIC_TransferAbortReceiveDMA (DMIC_Type ∗base, dmic_dma_handle_t ∗handle)
  Aborts the received data using DMA.
- status_t DMIC_TransferGetReceiveCountDMA (DMIC_Type ∗base, dmic_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

11.5.2 Data Structure Documentation

11.5.2.1 struct dmic_transfer_t

Data Fields

- uint16_t ∗ data
  The buffer of data to be transfer.
- size_t dataSize

The byte count to be transfer.

11.5.2.1.0.8 Field Documentation

11.5.2.1.0.8.1 uint16_t* dmic_transfer_t::data

11.5.2.1.0.8.2 size_t dmic_transfer_t::dataSize

11.5.2.2 struct _dmic_dma_handle

Data Fields

- DMIC_Type * base
  DMIC peripheral base address.
- dma_handle_t * rxDmaHandle
  The DMA RX channel used.
- dmic_dma_transfer_callback_t callback
  Callback function.
- void * userData
  DMIC callback function parameter.
- size_t transferSize
  Size of the data to receive.
- volatile uint8_t state
  Internal state of DMIC DMA transfer.

11.5.2.2.0.9 Field Documentation

11.5.2.2.0.9.1 DMIC_Type* dmic_dma_handle_t::base

11.5.2.2.0.9.2 dma_handle_t* dmic_dma_handle_t::rxDmaHandle

11.5.2.2.0.9.3 dmic_dma_transfer_callback_t dmic_dma_handle_t::callback

11.5.2.2.0.9.4 void* dmic_dma_handle_t::userData

11.5.2.2.0.9.5 size_t dmic_dma_handle_t::transferSize

11.5.3 Typedef Documentation

11.5.3.1 typedef void(* dmic_dma_transfer_callback_t)(DMIC_Type *base, dmic_dma_handle_t *handle, status_t status, void *userData)

11.5.4 Function Documentation

11.5.4.1 status_t DMIC_TransferCreateHandleDMA (DMIC_Type * *base*, dmic_dma_handle_t * *handle*, dmic_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *rxDmaHandle*)

Parameters

base	DMIC peripheral base address.
handle	Pointer to dmic_dma_handle_t structure.
callback	Callback function.
userData	User data.
rxDmaHandle	User-requested DMA handle for RX DMA transfer.

11.5.4.2 status_t DMIC_TransferReceiveDMA (DMIC_Type * *base*, dmic_dma_handle_t * *handle*, dmic_transfer_t * *xfer*, uint32_t *dmic_channel*)

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	USART peripheral base address.
handle	Pointer to usart_dma_handle_t structure.
xfer	DMIC DMA transfer structure. See dmic_transfer_t.
dmic_channel	DMIC channel

Return values

kStatus_Success	

11.5.4.3 void DMIC_TransferAbortReceiveDMA (DMIC_Type * *base*, dmic_dma_handle_t * *handle*)

This function aborts the received data using DMA.

Parameters

base	DMIC peripheral base address
handle	Pointer to dmic_dma_handle_t structure

11.5.4.4 status_t DMIC_TransferGetReceiveCountDMA (DMIC_Type * *base*, dmic_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	DMIC peripheral base address.
handle	DMIC 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;

Chapter 12
EEPROM: EEPROM memory driver

12.1 Overview

The KSDK provides a peripheral driver for the eeprom module of Lpc devices.

12.2 Typical use case

```
eeprom_config_t config;
eeprom_GetDefaultConfig(&config);
eeprom_Init(base, &config);
eeprom_WritePage(base, pageNum, data);
```

Data Structures

- struct eeprom_config_t
  EEPROM region configuration structure. More...

Enumerations

- enum eeprom_auto_program_t {
  kEEPROM_AutoProgramDisable = 0x0,
  kEEPROM_AutoProgramWriteWord = 0x1,
  kEEPROM_AutoProgramLastWord = 0x2 }
  EEPROM automatic program option.
- enum eeprom_interrupt_enable_t { kEEPROM_ProgramFinishInterruptEnable = EEPROM_INTENSET_PROG_SET_EN_MASK }
  EEPROM interrupt source.

Driver version

- #define FSL_EEPROM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  EEPROM driver version 2.0.0.

Initialization and deinitialization

- void EEPROM_Init (EEPROM_Type *base, const eeprom_config_t *config, uint32_t sourceClock_Hz)
  Initializes the EEPROM with the user configuration structure.
- void EEPROM_GetDefaultConfig (eeprom_config_t *config)
  Get EEPROM default configure settings.
- void EEPROM_Deinit (EEPROM_Type *base)
  Deinitializes the EEPROM regions.

Basic Control Operations

- static void EEPROM_SetAutoProgram (EEPROM_Type *base, eeprom_auto_program_t autoProgram)
  Set EEPROM automatic program feature.
- static void EEPROM_SetPowerDownMode (EEPROM_Type *base, bool enable)
  Set EEPROM to in/out power down mode.
- static void EEPROM_EnableInterrupt (EEPROM_Type *base, uint32_t mask)
  Enable EEPROM interrupt.
- static void EEPROM_DisableInterrupt (EEPROM_Type *base, uint32_t mask)
  Disable EEPROM interrupt.
- static uint32_t EEPROM_GetInterruptStatus (EEPROM_Type *base)
  Get the status of all interrupt flags for ERPROM.
- static uint32_t EEPROM_GetEnabledInterruptStatus (EEPROM_Type *base)
  Get the status of enabled interrupt flags for ERPROM.
- static void EEPROM_SetInterruptFlag (EEPROM_Type *base, uint32_t mask)
  Set interrupt flags manually.
- static void EEPROM_ClearInterruptFlag (EEPROM_Type *base, uint32_t mask)
  Clear interrupt flags manually.
- status_t EEPROM_WriteWord (EEPROM_Type *base, uint32_t offset, uint32_t data)
  Write a word data in address of EEPROM.
- status_t EEPROM_WritePage (EEPROM_Type *base, uint32_t pageNum, uint32_t *data)
  Write a page data into EEPROM.

12.3 Data Structure Documentation

12.3.1 struct eeprom_config_t

Data Fields

- eeprom_auto_program_t autoProgram
  Automatic program feature.
- uint8_t readWaitPhase1
  EEPROM read waiting phase 1.
- uint8_t readWaitPhase2
  EEPROM read waiting phase 2.
- uint8_t writeWaitPhase1
  EEPROM write waiting phase 1.
- uint8_t writeWaitPhase2
  EEPROM write waiting phase 2.
- uint8_t writeWaitPhase3
  EEPROM write waiting phase 3.
- bool lockTimingParam
  If lock the read and write wait phase settings.

12.3.1.0.0.10 Field Documentation

12.3.1.0.0.10.1 eeprom_auto_program_t eeprom_config_t::autoProgram

12.4 Macro Definition Documentation

12.4.1 #define FSL_EEPROM_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

12.5 Enumeration Type Documentation

12.5.1 enum eeprom_auto_program_t

Enumerator

kEEPROM_AutoProgramDisable Disable auto program.
kEEPROM_AutoProgramWriteWord Auto program triggered after 1 word is written.
kEEPROM_AutoProgramLastWord Auto program triggered after last word of a page written.

12.5.2 enum eeprom_interrupt_enable_t

Enumerator

kEEPROM_ProgramFinishInterruptEnable Interrupt while program finished.

12.6 Function Documentation

12.6.1 void EEPROM_Init (EEPROM_Type * *base,* const eeprom_config_t * *config,* uint32_t *sourceClock_Hz*)

This function configures the EEPROM module with the user-defined configuration. This function also sets the internal clock frequency to about 155kHz according to the source clock frequency.

Parameters

base	EEPROM peripheral base address.
config	The pointer to the configuration structure.
sourceClock_-Hz	EEPROM source clock frequency in Hz.

12.6.2 void EEPROM_GetDefaultConfig (eeprom_config_t * *config*)

Parameters

config	EEPROM config structure pointer.

12.6.3 void EEPROM_Deinit (EEPROM_Type * *base*)

Parameters

base	EEPROM peripheral base address.

12.6.4 static void EEPROM_SetAutoProgram (EEPROM_Type * *base*, eeprom_auto_program_t *autoProgram*) [inline], [static]

EEPROM write always needs a program and erase cycle to write the data into EEPROM. This program and erase cycle can be finished automaticlly or manually. If users want to use or disable auto program feature, users can call this API.

Parameters

base	EEPROM peripheral base address.
autoProgram	EEPROM auto program feature need to set.

12.6.5 static void EEPROM_SetPowerDownMode (EEPROM_Type * *base*, bool *enable*) [inline], [static]

This function make EEPROM eneter or out of power mode. Notice that, users shall not put EEPROM into power down mode while there is still any pending EEPROM operation. While EEPROM is wakes up from power down mode, any EEPROM operation has to be suspended for 100 us.

Parameters

base	EEPROM peripheral base address.
enable	True means enter to power down mode, false means wake up.

12.6.6 static void EEPROM_EnableInterrupt (EEPROM_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	EEPROM peripheral base address.
mask	EEPROM interrupt enable mask. It is a logic OR of members the enumeration :: eeprom_interrupt_enable_t

12.6.7 static void EEPROM_DisableInterrupt (EEPROM_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	EEPROM peripheral base address.
mask	EEPROM interrupt enable mask. It is a logic OR of members the enumeration :: eeprom_interrupt_enable_t

12.6.8 static uint32_t EEPROM_GetInterruptStatus (EEPROM_Type * *base*) [inline], [static]

Parameters

base	EEPROM peripheral base address.

Returns

EEPROM interrupt flag status

12.6.9 static uint32_t EEPROM_GetEnabledInterruptStatus (EEPROM_Type * *base*) [inline], [static]

Parameters

base	EEPROM peripheral base address.

Returns

EEPROM enabled interrupt flag status

12.6.10 static void EEPROM_SetInterruptFlag (EEPROM_Type * *base,* uint32_t *mask*) [inline], [static]

This API trigger a interrupt manually, users can no need to wait for hardware trigger interrupt. Call this API will set the corresponding bit in INSTAT register.

Parameters

base	EEPROM peripheral base address.
mask	EEPROM interrupt flag need to be set. It is a logic OR of members of enumeration:: eeprom_interrupt_enable_t

12.6.11 static void EEPROM_ClearInterruptFlag (EEPROM_Type * *base*, uint32_t *mask*) [inline], [static]

This API clears interrupt flags manually. Call this API will clear the corresponding bit in INSTAT register.

Parameters

base	EEPROM peripheral base address.
mask	EEPROM interrupt flag need to be cleared. It is a logic OR of members of enumeration:: eeprom_interrupt_enable_t

12.6.12 status_t EEPROM_WriteWord (EEPROM_Type * *base*, uint32_t *offset*, uint32_t *data*)

Users can write a page or at least a word data into EEPROM address.

Parameters

base	EEPROM peripheral base address.
offset	Offset from the begining address of EEPROM. This value shall be 4-byte aligned.
data	Data need be write.

12.6.13 status_t EEPROM_WritePage (EEPROM_Type * *base*, uint32_t *pageNum*, uint32_t * *data*)

Users can write a page or at least a word data into EEPROM address.

Parameters

base	EEPROM peripheral base address.
pageNum	Page number to be written.
data	Data need be write. This array data size shall equals to the page size.

Chapter 13
EMC: External Memory Controller Driver

13.1 Overview

The KSDK provides a peripheral driver for the External Memory Controller block of Kinetis devices.

The EMC driver is provides support for synchronous static memory devices such as RAM, rom and flash, in addition to dynamic memories such as single data rate SDRAM with an SDRAM clock of up to 100Mhz. From software control, three mainly function blocks are related:

1. Baisc controller a.Timing control with programmable delay elements. b.Module enable.
2. Dynamic memory controller
3. Static memory controller

When using EMC, call EMC_Init() function firstly to do module basic initialize. Note that this function enables the module clock, configure the module system level clock/delay and enable the module. It is the initialization of the Basic controller. To initialize the external dynamic memory. The EMC_DynamicMemInit() function shall be called before you can access any dynamic memory. The EMC_DynamicMemInit() function is provided to initialize the Static memory controller and it shall be called when you want to access any exterenal static memory. The function EMC_Deinit() deinitializes the EMC module.

EMC Provides many basic opeartion APIs for application to do flexible control. The function EMC_Enable() is provided to enable/disable the EMC module. The function EMC_EnableDynamicMemControl() is provided to enable/disble the EMC dynamic memory controller. The function EMC_SendDynamicMemCommand() is provided to send the NOP/PRECHARGE/MODE/SELF-REFRESH commands. Call EMC_EnterLowPowerMode() to enter or exit the low-power mode. There is a calibration function EMC_DelayCalibrate() which do calibaration of the EMC programmable delays by providing a real-time representation of the values of those deays.

13.2 Typical use case

This example shows how to use the EMC to initialize the external 64M 16-bit bus width SDRAM chip (4 banks nd 9 columns). The SDRAM is on the CS0.

First, initialize the EMC Controller.

```
// Basic configuration
emc_basic_config_t basiConfig =
{
    kEMC_LittleEndian,
    kEMC_IntloopbackEmcclk,
    1,    // EMC CLOCK DIV is 2
    7,    // cmd delay is 7
    7,    // feeback clock delay is 7
};

// Dynamic timing configuration.
emc_dynamic_timing_config_t timing =
{
```

```
    (64000000/4096),   // refresh period in unit of nanosecond
    kEMC_Cmddelay,
    18,   //tRP
    42,   //tRAS
    70,  // TSREX
    2,  //TAPR
    5,  //TDAL
    6,  //TWR
    60,  //TRC
    64,  //TREF
    70,  //TXSR
    12,  //TRRD
    12,  //TMRD
};

// EMC dynamic memory chip independent configuration.
emc_dynamic_chip_config_t dynConfig =
{
    0,    // dynamic memory chip 0
    kEMC_Sdram,
    2,    // RAS delay is 2 EMC clock cycles
    0x23, // Burst length is 8, CAS latency is 2.
    0x09, // 16-bit high performance, 4 banks, 9 columns
}

// EMC controller initialization.
uint8_t totalDynchips = 1;
EMC_Init(base, &basiConfig);
EMC_DynamicMemInit(base, &timing, &dynConfig, totalDynchips);

// Add Delay Calibration
APP_DelayCal()
// R/W access to SDRAM Memory
```

For the APP_DelayCal(): The system configure provided the command and feedback clock delay calibration for EMC EMCDYCTRL and EMCCAL. Application may require the change for these two system configure register. please use the recommded work flow to choose the best delay.

```
volatile uint32_t ringosccount[2] = {0,0};

// function calibration
uint32_t calibration(uint16_t times)
{
  uint32_t i;
  uint32_t value;
  uint32_t count = 0;

  if (!times)
  {
      return 0;
  }

  for (i = 0; i < times; i++)
  {
      value = SYSCON->EMCDLYCAL & ~0x4000;
      SYSCON->EMCDLYCAL = value | 0x4000;

      while ((SYSCON->EMCDLYCAL & 0x8000) == 0x0000)
      {
        value = SYSCON->EMCDLYCAL;
      }
      count += (value & 0xFF);
  }
  return (count / times);
}
```

```
// sdram read and write test function
uint32_t sdram_rwtest( void )
{
  volatile uint32_t *wr_ptr;
  volatile uint16_t *short_wr_ptr;
  uint32_t data;
  uint32_t i, j;

  wr_ptr = (uint32_t *)SDRAM_BASE;
  short_wr_ptr = (uint16_t *)wr_ptr;
  /* Clear content before 16 bit access test */
  memset(wr_ptr, 0, SDRAM_SIZE/4);

  /* 16 bit write */
  for (i = 0; i < SDRAM_SIZE/0x40000; i++)
  {
    for (j = 0; j < 0x10000; j++)
    {
      *short_wr_ptr++ = (i + j);
      *short_wr_ptr++ = (i + j) + 1;
    }
  }

  /* Verifying */
  wr_ptr = (uint32_t *)SDRAM_BASE;
  for (i = 0; i < SDRAM_SIZE/0x40000; i++)
  {
    for (j = 0; j < 0x10000; j++)
    {
      data = *wr_ptr;
      if (data != (((((i + j) + 1) & 0xFFFF) << 16) | ((i + j) & 0xFFFF)))
      {
      return 0x0;
      }
      wr_ptr++;
    }
  }
  return 0x1;
}

// find the best cmd delay
uint32_t find_cmddly(void)
{
  uint32_t cmddly, cmddlystart, cmddlyend, dwtemp;
  uint32_t ppass = 0x0, pass = 0x0;

  cmddly = 0x0;
  cmddlystart = cmddlyend = 0xFF;

  while (cmddly < 32)
  {
  dwtemp = SYSCON->EMCDLYCTRL & ~0x1F;
  SYSCON->EMCDLYCTRL = dwtemp | cmddly;

  if (sdram_rwtest() == 0x1)
  {
    /* Test passed */
    if (cmddlystart == 0xFF)
    {
    cmddlystart = cmddly;
    }
    ppass = 0x1;
  }
  else
  {
    /* Test failed */
    if (ppass == 1)
```

Typical use case

```
    {
    cmddlyend = cmddly;
    pass = 0x1;
    ppass = 0x0;
    }
  }

  /* Try next value */
  cmddly++;
  }

  /* If the test passed, the we can use the average of the min and max values to get an optimal DQSIN delay
      */
  if (pass == 0x1)
  {
  cmddly = (cmddlystart + cmddlyend) / 2;
  }
  else if (ppass == 0x1)
  {
  cmddly = (cmddlystart + 0x1F) / 2;
  }
  else
  {
  /* A working value couldn't be found, just pick something safe so the system doesn't become unstable */
  cmddly = 0x10;
  }

  dwtemp = SYSCON->EMCDLYCTRL & ~0x1F;
  SYSCON->EMCDLYCTRL = dwtemp | cmddly;

  return (pass | ppass);
}

// found the best feedback delay
uint32_t find_fbclkdly(void)
{
  uint32_t fbclkdly, fbclkdlystart, fbclkdlyend, dwtemp;
  uint32_t ppass = 0x0, pass = 0x0;

  fbclkdly = 0x0;
  fbclkdlystart = fbclkdlyend = 0xFF;

  while (fbclkdly < 32)
  {
  dwtemp = SYSCON->EMCDLYCTRL & ~0x1F00;
  SYSCON->EMCDLYCTRL = dwtemp | (fbclkdly << 8);

  if (sdram_rwtest() == 0x1)
  {
    /* Test passed */
    if (fbclkdlystart == 0xFF)
    {
    fbclkdlystart = fbclkdly;
    }
    ppass = 0x1;
  }
  else
  {
    /* Test failed */
    if (ppass == 1)
    {
    fbclkdlyend = fbclkdly;
    pass = 0x1;
    ppass = 0x0;
    }
  }

  /* Try next value */
```

```
  fbclkdly++;
  }

  /* If the test passed, the we can use the average of the min and max values to get an optimal DQSIN delay
       */
  if (pass == 0x1)
  {
  fbclkdly = (fbclkdlystart + fbclkdlyend) / 2;
  }
  else if (ppass == 0x1)
  {
  fbclkdly = (fbclkdlystart + 0x1F) / 2;
  }
  else
  {
  /* A working value couldn't be found, just pick something safe so the system doesn't become unstable */
  fbclkdly = 0x10;
  }

  dwtemp = SYSCON->EMCDLYCTRL & ~0x1F00;
  SYSCON->EMCDLYCTRL = dwtemp | (fbclkdly << 8);

  return (pass | ppass);
}

// adjust the found the delay to the system configuration delay control register
void adjust_timing( void )
{
  uint32_t dwtemp, cmddly, fbclkdly;

  /* Current value */
  ringosccount[1] = calibration();

  dwtemp = SYSCON->EMCDLYCTRL;
  cmddly = ((dwtemp & 0x1F) * ringosccount[0] / ringosccount[1]) & 0x1F;

  fbclkdly = ((dwtemp & 0x1F00) * ringosccount[0] / ringosccount[1]) & 0x1F00;

  SYSCON->EMCDLYCTRL = (dwtemp & ~0x1F1F) | fbclkdly | cmddly;
}


APP_DelayCal()
{
   ringosccount[0] = calibration();

  if (find_cmddly() == 0x0)
  {
    while (1);  /* fatal error */
  }

  if (find_fbclkdly() == 0x0)
  {
    while (1);  /* fatal error */
  }
   adjust_timing();
}
```

Data Structures

- struct emc_dynamic_timing_config_t
  EMC dynamic timing/delay configure structure. *More...*
- struct emc_dynamic_chip_config_t
  EMC dynamic memory controller independent chip configuration structure. *More...*
- struct emc_static_chip_config_t

EMC static memory controller independent chip configuration structure. More...

- struct emc_basic_config_t
  EMC module basic configuration structure. More...

Macros

- #define EMC_STATIC_MEMDEV_NUM (4U)
  Define the chip numbers for dynamic and static memory devices.

Enumerations

- enum emc_static_memwidth_t {
  kEMC_8BitWidth = 0x0U,
  kEMC_16BitWidth,
  kEMC_32BitWidth }
  Define EMC memory width for static memory device.
- enum emc_static_special_config_t {
  kEMC_AsynchronosPageEnable = 0x0008U,
  kEMC_ActiveHighChipSelect = 0x0040U,
  kEMC_ByteLaneStateAllLow = 0x0080U,
  kEMC_ExtWaitEnable = 0x0100U,
  kEMC_BufferEnable = 0x80000U }
  Define EMC static configuration.
- enum emc_dynamic_device_t {
  kEMC_Sdram = 0x0U,
  kEMC_Lpsdram }
  EMC dynamic memory device.
- enum emc_dynamic_read_t {
  kEMC_NoDelay = 0x0U,
  kEMC_Cmddelay,
  kEMC_CmdDelayPulseOneclk,
  kEMC_CmddelayPulsetwoclk }
  EMC dynamic read strategy.
- enum emc_endian_mode_t {
  kEMC_LittleEndian = 0x0U,
  kEMC_BigEndian }
  EMC endian mode.
- enum emc_fbclk_src_t {
  kEMC_IntloopbackEmcclk = 0U,
  kEMC_EMCFbclkInput }
  EMC Feedback clock input source select.

Driver version

- #define FSL_EMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  EMC driver version 2.0.0.

EMC Initialize and de-initialize opeartion

- void EMC_Init (EMC_Type *base, emc_basic_config_t *config)
  Initializes the basic for EMC.
- void EMC_DynamicMemInit (EMC_Type *base, emc_dynamic_timing_config_t *timing, emc_dynamic_chip_config_t *config, uint32_t totalChips)
  Initializes the dynamic memory controller.
- void EMC_StaticMemInit (EMC_Type *base, uint32_t *extWait_Ns, emc_static_chip_config_t *config, uint32_t totalChips)
  Initializes the static memory controller.
- void EMC_Deinit (EMC_Type *base)
  Deinitializes the EMC module and gates the clock.

EMC Basic Operation

- static void EMC_Enable (EMC_Type *base, bool enable)
  Enables/disables the EMC module.
- static void EMC_EnableDynamicMemControl (EMC_Type *base, bool enable)
  Enables/disables the EMC Dynaimc memory controller.
- static void EMC_MirrorChipAddr (EMC_Type *base, bool enable)
  Enables/disables the EMC address mirror.
- static void EMC_EnterSelfRefreshCommand (EMC_Type *base, bool enable)
  Enter the self-refresh mode for dynamic memory controller.
- static bool EMC_IsInSelfrefreshMode (EMC_Type *base)
  Get the operating mode of the EMC.
- static void EMC_EnterLowPowerMode (EMC_Type *base, bool enable)
  Enter/exit the low-power mode.

13.3 Data Structure Documentation

13.3.1 struct emc_dynamic_timing_config_t

Data Fields

- uint32_t refreshPeriod_Nanosec
  The refresh period in unit of nanosecond.
- uint32_t tRp_Ns
  Precharge command period in unit of nanosecond.
- uint32_t tRas_Ns
  Active to precharge command period in unit of nanosecond.
- uint32_t tSrex_Ns
  Self-refresh exit time in unit of nanosecond.
- uint32_t tApr_Ns
  Last data out to active command time in unit of nanosecond.
- uint32_t tDal_Ns
  Data-in to active command in unit of nanosecond.
- uint32_t tWr_Ns
  Write recovery time in unit of nanosecond.
- uint32_t tRc_Ns
  Active to active command period in unit of nanosecond.

- uint32_t tRfc_Ns
  Auto-refresh period and auto-refresh to active command period in unit of nanosecond.
- uint32_t tXsr_Ns
  Exit self-refresh to active command time in unit of nanosecond.
- uint32_t tRrd_Ns
  Active bank A to active bank B latency in unit of nanosecond.
- uint8_t tMrd_Nclk
  Load mode register to active command time in unit of EMCCLK cycles.

13.3.1.0.0.11 Field Documentation

13.3.1.0.0.11.1 uint32_t emc_dynamic_timing_config_t::refreshPeriod_Nanosec

13.3.1.0.0.11.2 uint32_t emc_dynamic_timing_config_t::tRp_Ns

13.3.1.0.0.11.3 uint32_t emc_dynamic_timing_config_t::tRas_Ns

13.3.1.0.0.11.4 uint32_t emc_dynamic_timing_config_t::tSrex_Ns

13.3.1.0.0.11.5 uint32_t emc_dynamic_timing_config_t::tApr_Ns

13.3.1.0.0.11.6 uint32_t emc_dynamic_timing_config_t::tDal_Ns

13.3.1.0.0.11.7 uint32_t emc_dynamic_timing_config_t::tWr_Ns

13.3.1.0.0.11.8 uint32_t emc_dynamic_timing_config_t::tRc_Ns

13.3.1.0.0.11.9 uint32_t emc_dynamic_timing_config_t::tRfc_Ns

13.3.1.0.0.11.10 uint32_t emc_dynamic_timing_config_t::tXsr_Ns

13.3.1.0.0.11.11 uint32_t emc_dynamic_timing_config_t::tRrd_Ns

13.3.1.0.0.11.12 uint8_t emc_dynamic_timing_config_t::tMrd_Nclk

13.3.2 struct emc_dynamic_chip_config_t

Please take refer to the address mapping table in the RM in EMC chapter when you set the "devAddrMap". Choose the right Bit 14 Bit12 ~ Bit 7 group in the table according to the bus width/banks/row/colum length for you device. Set devAddrMap with the value make up with the seven bits (bit14 bit12 ~ bit 7) and inset the bit 13 with 0. for example, if the bit 14 and bit12 ~ bit7 is 1000001 is choosen according to the 32bit high-performance bus width with 2 banks, 11 row lwngth, 8 column length. Set devAddrMap with 0x81.

Data Fields

- uint8_t chipIndex
  Chip Index, range from 0 ~ EMC_DYNAMIC_MEMDEV_NUM - 1.
- emc_dynamic_device_t dynamicDevice
  All chips shall use the same device setting.

- uint8_t rAS_Nclk
  Active to read/write delay tRCD.
- uint16_t sdramModeReg
  Sdram mode register setting.
- uint16_t sdramExtModeReg
  Used for low-power sdram device.
- uint8_t devAddrMap
  dynamic device address mapping, choose the address mapping for your specific device.

13.3.2.0.0.12 Field Documentation

13.3.2.0.0.12.1 uint8_t emc_dynamic_chip_config_t::chipIndex

13.3.2.0.0.12.2 emc_dynamic_device_t emc_dynamic_chip_config_t::dynamicDevice

mixed use are not supported.

13.3.2.0.0.12.3 uint8_t emc_dynamic_chip_config_t::rAS_Nclk

13.3.2.0.0.12.4 uint16_t emc_dynamic_chip_config_t::sdramModeReg

13.3.2.0.0.12.5 uint16_t emc_dynamic_chip_config_t::sdramExtModeReg

The extended mode register.

13.3.2.0.0.12.6 uint8_t emc_dynamic_chip_config_t::devAddrMap

13.3.3 struct emc_static_chip_config_t

Data Fields

- emc_static_memwidth_t memWidth
  Memory width.
- uint32_t specailConfig
  Static configuration,a logical OR of "emc_static_special_config_t".
- uint32_t tWaitWriteEn_Ns
  The delay form chip select to write enable in unit of nanosecond.
- uint32_t tWaitOutEn_Ns
  The delay from chip selcet to output enable in unit of nanosecond.
- uint32_t tWaitReadNoPage_Ns
  In No-page mode, the delay from chip select to read access in unit of nanosecond.
- uint32_t tWaitReadPage_Ns
  In page mode, the read after the first read wait states in unit of nanosecond.
- uint32_t tWaitWrite_Ns
  The delay from chip select to write access in unit of nanosecond.
- uint32_t tWaitTurn_Ns
  The Bus turn-around time in unit of nanosecond.

13.3.3.0.0.13 Field Documentation

13.3.3.0.0.13.1 emc_static_memwidth_t emc_static_chip_config_t::memWidth

13.3.3.0.0.13.2 uint32_t emc_static_chip_config_t::specailConfig

13.3.3.0.0.13.3 uint32_t emc_static_chip_config_t::tWaitWriteEn_Ns

13.3.3.0.0.13.4 uint32_t emc_static_chip_config_t::tWaitOutEn_Ns

13.3.3.0.0.13.5 uint32_t emc_static_chip_config_t::tWaitReadNoPage_Ns

13.3.3.0.0.13.6 uint32_t emc_static_chip_config_t::tWaitReadPage_Ns

13.3.3.0.0.13.7 uint32_t emc_static_chip_config_t::tWaitWrite_Ns

13.3.3.0.0.13.8 uint32_t emc_static_chip_config_t::tWaitTurn_Ns

13.3.4 struct emc_basic_config_t

Defines the static memory controller configure structure and uses the EMC_Init() function to make necessary initializations.

Data Fields

- emc_endian_mode_t endian
  Endian mode .
- emc_fbclk_src_t fbClkSrc
  The feedback clock source.
- uint8_t emcClkDiv
  EMC_CLK = AHB_CLK / (emc_clkDiv + 1).

13.3.4.0.0.14 Field Documentation

13.3.4.0.0.14.1 emc_endian_mode_t emc_basic_config_t::endian

13.3.4.0.0.14.2 emc_fbclk_src_t emc_basic_config_t::fbClkSrc

13.3.4.0.0.14.3 uint8_t emc_basic_config_t::emcClkDiv

13.4 Macro Definition Documentation

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

13.4.2 #define EMC_STATIC_MEMDEV_NUM (4U)

13.5 Enumeration Type Documentation

13.5.1 enum emc_static_memwidth_t

Enumerator

kEMC_8BitWidth 8 bit memory width.
kEMC_16BitWidth 16 bit memory width.
kEMC_32BitWidth 32 bit memory width.

13.5.2 enum emc_static_special_config_t

Enumerator

kEMC_AsynchronosPageEnable Enable the asynchronous page mode. page length four.
kEMC_ActiveHighChipSelect Chip select active high.
kEMC_ByteLaneStateAllLow Reads/writes the respective valuie bits in BLS3:0 are low.
kEMC_ExtWaitEnable Extended wait enable.
kEMC_BufferEnable Buffer enable.

13.5.3 enum emc_dynamic_device_t

Enumerator

kEMC_Sdram Dynamic memory device: SDRAM.
kEMC_Lpsdram Dynamic memory device: Low-power SDRAM.

13.5.4 enum emc_dynamic_read_t

Enumerator

kEMC_NoDelay No delay.
kEMC_Cmddelay Command delayed strategy, using EMCCLKDELAY.
kEMC_CmdDelayPulseOneclk Command delayed strategy pluse one clock cycle using EMCCLKDELAY.
kEMC_CmddelayPulsetwoclk Command delayed strategy pulse two clock cycle using EMCCLKDELAY.

13.5.5 enum emc_endian_mode_t

Enumerator

kEMC_LittleEndian Little endian mode.
kEMC_BigEndian Big endian mode.

13.5.6 enum emc_fbclk_src_t

Enumerator

kEMC_IntloopbackEmcclk Use the internal loop back from EMC_CLK output.
kEMC_EMCFbclkInput Use the external EMC_FBCLK input.

13.6 Function Documentation

13.6.1 void EMC_Init (EMC_Type * *base,* emc_basic_config_t * *config*)

This function ungates the EMC clock, initializes the emc system configure and enable the EMC module. This function must be called in the first step to initialize the external memory.

Parameters

base	EMC peripheral base address.
config	The EMC basic configuration.

13.6.2 void EMC_DynamicMemInit (EMC_Type * *base,* emc_dynamic_timing_config_t * *timing,* emc_dynamic_chip_config_t * *config,* uint32_t *totalChips*)

This function initializes the dynamic memory controller in external memory controller. This function must be called after EMC_Init and before accessing the external dynamic memory.

Parameters

base	EMC peripheral base address.
timing	The timing and latency for dynamica memory controller setting. It shall be used for all dynamica memory chips, threfore the worst timing value for all used chips must be given.

configure	The EMC dynamic memory controller chip independent configuration pointer. This configuration pointer is actually pointer to a configration array. the array number depends on the "totalChips".
totalChips	The total dynamic memory chip numbers been used or the length of the "emc_dynamic_chip_config_t" type memory.

13.6.3 void EMC_StaticMemInit (EMC_Type * *base*, uint32_t * *extWait_Ns*, emc_static_chip_config_t * *config*, uint32_t *totalChips*)

This function initializes the static memory controller in external memory controller. This function must be called after EMC_Init and before accessing the external static memory.

Parameters

base	EMC peripheral base address.
extWait_Ns	The extended wait timeout or the read/write transfer time. This is common for all static memory chips and set with NULL if not required.
configure	The EMC static memory controller chip independent configuration pointer. This configuration pointer is actually pointer to a configration array. the array number depends on the "totalChips".
totalChips	The total static memory chip numbers been used or the length of the "emc_static_chip_config_t" type memory.

13.6.4 void EMC_Deinit (EMC_Type * *base*)

This function gates the EMC controller clock. As a result, the EMC module doesn't work after calling this function.

Parameters

base	EMC peripheral base address.

13.6.5 static void EMC_Enable (EMC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	EMC peripheral base address.
enable	True enable EMC module, false disable.

13.6.6 static void EMC_EnableDynamicMemControl (EMC_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	EMC peripheral base address.
enable	True enable EMC dynamic memory controller, false disable.

13.6.7 static void EMC_MirrorChipAddr (EMC_Type * *base*, bool *enable*) [static]

Enable the address mirror the EMC_CS1is mirrored to both EMC_CS0 and EMC_DYCS0 memory areas. Disable the address mirror enables EMC_cS0 and EMC_DYCS0 memory to be accessed.

Parameters

base	EMC peripheral base address.
enable	True enable the address mirror, false disable the address mirror.

13.6.8 static void EMC_EnterSelfRefreshCommand (EMC_Type * *base*, bool *enable*) [inline], [static]

This function provided self-refresh mode enter or exit for application.

Parameters

base	EMC peripheral base address.
enable	True enter the self-refresh mode, false to exit self-refresh and enter the normal mode.

13.6.9 static bool EMC_IsInSelfrefreshMode (EMC_Type * *base*) [inline], [static]

This function can be used to get the operating mode of the EMC.

Parameters

base	EMC peripheral base address.

Returns

The EMC in self-refresh mode if true, else in normal mode.

13.6.10 static void EMC_EnterLowPowerMode (EMC_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	EMC peripheral base address.
enable	True Enter the low-power mode, false exit low-power mode and return to normal mode.

Chapter 14
ENET: Ethernet Driver

14.1 Overview

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

Use the ENET_GetDefaultConfig() to get the default basic configuration, Use the default configuration unchanged or changed as the input to the ENET_Init() to do basic configuration for ENET module. Call ENET_DescriptorInit() to intialization the descriptors and Call ENET_StartRxTx() to start the ENET engine after all initialization. ENET_Deinit() is used to to ENET Deinitialization.

The MII interface is the interface connected with MAC and PHY. the Serial management interface - MII management interface should be set before any access to the external PHY chip register. Call ENET_SetSMI() to initialize MII management interface. Use ENET_StartSMIRead(), ENET_StartSMIWrite(), and ENET_ReadSMIData() to read/write to PHY registers, ENET_IsSMIBusy() to check the SMI busy status. 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 getting data from the external PHY.

This group provides the ENET mac address set/get operation with ENET_SetMacAddr() and ENET_GetMacAddr(). The ENET_EnterPowerDown() and ENET_ExitPowerDown() can be used to do power management.

This group provide the DMA interrupt get and clear APIs. This can be used by application to create new IRQ handler.

This group functions are low level tx/rx descriptor operations. It is convenient to use these tx/rx APIs to do application specific rx/tx. For TX: Use ENET_IsTxDescriptorDmaOwn(), ENET_SetupTxDescriptor() to build your packet for transfer and ENET_UpdateTxDescriptorTail to update the tx tail pointer. For RX: Use ENET_GetRxDescriptor() to get the received data/length and use the ENET_UpdateRxDescriptor() to update the buffers/status.

When use the Transactional APIs, please make sure to call the ENET_CreateHandler to create the handler which are used to maintain all datas related to tx/tx process.

For ENET receive, the ENET_GetRxFrameSize() function must be called to get the received data size. Then, call the ENET_ReadFrame() function to get the received data.

For ENET transmit, call the ENET_SendFrame() function to send the data out. To save memory and avoid the memory copy in the TX process. The ENET_SendFrame() here is a zero-copy API, so make sure the input data buffers are not requeued or freed before the data are really sent out. To makesure the data buffers reclaim is rightly done. the transmit interrupt must be used. so For transactional APIs here we enabled the tx interrupt in ENET_CreateHandler(). That means the tx interrupt is automatically enabled in transctional APIs. is recommended to be called on the transmit interrupt handler.ENET_ReclaimTxDescriptor() is a transactional API to get the information from the finished transmit data buffers and reclaim the tx index. it is called by the transmit interrupt IRQ handler.

Typical use case

All PTP 1588 fatures are enabled by define "ENET_PTP1588FEATURE_REQUIRED" This function group configures the PTP IEEE 1588 feature, starts/stops/gets/sets/corrects the PTP IEEE 1588 timer, gets the receive/transmit frame timestamp

The ENET_GetRxFrameTime() and ENET_GetTxFrameTime() functions are called by the PTP stack to get the timestamp captured by the ENET driver.

14.2 Typical use case

14.2.1 ENET Initialization, receive, and transmit operations

For use the transactional APIs, receive polling

```
enet_config_t config;
uint8_t index;
void *buff;
uint32_t refClock = 50000000;
phy_speed_t speed;
phy_duplex_t duplex;
uint32_t length = 0;
uint8_t *buffer;
uint32_t timedelay;
status_t status;

enet_buffer_config_t buffConfig = {
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    &g_txBuffDescrip[0],
    &g_txBuffDescrip[0],
    &g_rxBuffDescrip[0],
    &g_rxBuffDescrip[ENET_RXBD_NUM],
    &rxbuffer[0],
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
};

PHY_Init(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR);

ENET_GetDefaultConfig(&config);

PHY_GetLinkSpeedDuplex(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR, &speed, &duplex);

config.miiSpeed = (enet_mii_speed_t)speed;
config.miiDuplex = (enet_mii_duplex_t)duplex;
config.interrupt = kENET_DmaTx;
ENET_Init(EXAMPLE_ENET_BASE, &config, &g_macAddr[0], refClock);
ENET_CreateHandler(EXAMPLE_ENET_BASE, &g_handle, &config, &buffConfig, ENET_IntCallback,
      NULL);
ENET_DescriptorInit(EXAMPLE_ENET_BASE, &config, &buffConfig);
ENET_StartRxTx(EXAMPLE_ENET_BASE, 1, 1);

ENET_BuildBroadCastFrame();

while (1)
{
    status = ENET_GetRxFrameSize(EXAMPLE_ENET_BASE, &g_handle, &length, 0);
    if ((status == kStatus_Success) && (length != 0))
    {
        uint8_t *data = (uint8_t *)malloc(length);
        if (data)
        {
            status = ENET_ReadFrame(EXAMPLE_ENET_BASE, &g_handle, data, length, 0);
            if (status == kStatus_Success)
```

```
            {
                PRINTF(" One frame received. the length %d \r\n", length);
                PRINTF(" Dest Address %02x:%02x:%02x:%02x:%02x:%02x Src Address
     %02x:%02x:%02x:%02x:%02x:%02x \r\n",
                       data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7], data[8],
    data[9],
                       data[10], data[11]);
            }
            free(data);
        }
    }
    else if (status == kStatus_ENET_RxFrameError)
    {
        ENET_ReadFrame(EXAMPLE_ENET_BASE, &g_handle, NULL, 0, 0);
    }
    if (g_testIdx < ENET_EXAMPLE_SEND_COUNT)
    {
        if (PHY_GetLinkStatus(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR))
        {
            buffer = (uint8_t *)malloc(ENET_EXAMPLE_FRAME_SIZE);
            if (buffer)
            {
                memcpy(buffer, &g_frame[g_txIdx], ENET_EXAMPLE_FRAME_SIZE);
                g_txIdx = (g_txIdx + 1) % ENET_EXAMPLE_PACKAGETYPE;
                g_txbuff[g_txbuffIdx] = buffer;
                g_txbuffIdx = (g_txbuffIdx + 1) % ENET_TXBD_NUM;

                if (kStatus_Success ==
                    ENET_SendFrame(EXAMPLE_ENET_BASE, &g_handle, buffer,
    ENET_EXAMPLE_FRAME_SIZE))
                {
                    g_testIdx++;
                }
            }
        }
    }
}
```

For the functional API, rx polling

```
    static const IRQn_Type s_enetIrqId[] = ENET_IRQS;

   enet_config_t config;
    uint8_t index;
    void *buff;
    uint32_t refClock = 50000000;
    phy_speed_t speed;
    phy_duplex_t duplex;
    uint32_t length = 0;
    uint8_t *buffer;
    uint32_t data1, data2;
    uint32_t timedelay;

    enet_buffer_config_t buffConfig = {
        ENET_RXBD_NUM,
        ENET_TXBD_NUM,
        &g_txBuffDescrip[0],
        &g_txBuffDescrip[0],
        &g_rxBuffDescrip[0],
        &g_rxBuffDescrip[ENET_RXBD_NUM],
        &rxbuffer[0],
        ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    };

    PHY_Init(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR);
```

Typical use case

```
    ENET_GetDefaultConfig(&config);

    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR, &speed, &duplex);
    config.miiSpeed = (enet_mii_speed_t)speed;
    config.miiDuplex = (enet_mii_duplex_t)duplex;
    ENET_Init(EXAMPLE_ENET_BASE, &config, &g_macAddr[0], refClock);

    ENET_EnableInterrupts(ENET,  kENET_DmaTx);
    EnableIRQ(ENET_EXAMPLE_IRQ);

    ENET_DescriptorInit(EXAMPLE_ENET_BASE, &config, &buffConfig);

    ENET_StartRxTx(EXAMPLE_ENET_BASE, 1, 1);

    ENET_BuildBroadCastFrame();

    while (1)
    {
        ENET_GetRxDescriptor(&g_rxBuffDescrip[g_rxGenIdx], &data1, &data2, &length);
        if (length > 0)
        {
            g_rxGenIdx = (g_rxGenIdx + 1) % ENET_RXBD_NUM;

            void *buffer1;
            buffer1 = malloc(ENET_RXBUFF_SIZE);
            if (buffer1)
            {
                ENET_UpdateRxDescriptor(&g_rxBuffDescrip[g_rxCosumIdx], buffer1,
      NULL, false, false);
                g_rxCosumIdx = (g_rxCosumIdx + 1) % ENET_RXBD_NUM;
            }

            uint8_t *data = (uint8_t *)data1;
            PRINTF(" One frame received. the length %d \r\n", length);
            PRINTF(" Dest Address %02x:%02x:%02x:%02x:%02x:%02x Src Address %02x:%02x:%02x:%02x:%02x:%02x
      \r\n",
                   data[0], data[1], data[2], data[3], data[4], data[5], data[6], data[7], data[8], data[9]
      , data[10],
                   data[11]);

            free((void *)data1);
        }

        if (g_testIdx < ENET_EXAMPLE_SEND_COUNT)
        {
            if (PHY_GetLinkStatus(EXAMPLE_ENET_BASE, EXAMPLE_PHY_ADDR))
            {
                buffer = (uint8_t *)malloc(ENET_EXAMPLE_FRAME_SIZE);
                if (buffer)
                {
                    memcpy(buffer, &g_frame[g_txIdx], ENET_EXAMPLE_FRAME_SIZE);
                    g_txIdx = (g_txIdx + 1) % ENET_EXAMPLE_PACKAGETYPE;

                    g_txbuff[g_txbuffIdx] = buffer;
                    g_txbuffIdx = (g_txbuffIdx + 1) % ENET_TXBD_NUM;
                    while (ENET_TXQueue(buffer, ENET_EXAMPLE_FRAME_SIZE) != kStatus_Success)
                        ;
                    g_testIdx++;
                }
            }
        }
    }

static status_t ENET_TXQueue(uint8_t *data, uint16_t length)
{
    uint32_t txdescTailAddr;

    if (ENET_IsTxDescriptorDmaOwn(&g_txBuffDescrip[g_txGenIdx]))
```

```
    {
        return kStatus_Fail;
    }
    ENET_SetupTxDescriptor(&g_txBuffDescrip[g_txGenIdx], data, length, NULL, 0,
      length, true, false, kENET_FirstLastFlag, 0);

    g_txGenIdx = (g_txGenIdx + 1) % ENET_TXBD_NUM;
    g_txUsed++;

    txdescTailAddr = (uint32_t)&g_txBuffDescrip[g_txGenIdx];
    if (!g_txGenIdx)
    {
        txdescTailAddr = (uint32_t)&g_txBuffDescrip[ENET_TXBD_NUM];
    }
    ENET_UpdateTxDescriptorTail(EXAMPLE_ENET_BASE, 0, txdescTailAddr);
    return kStatus_Success;
}
```

Data Structures

- struct enet_rx_bd_struct_t
  Defines the receive descriptor structure has the read-format and write-back format structure. More...
- struct enet_tx_bd_struct_t
  Defines the transmit descriptor structure has the read-format and write-back format structure. More...
- struct enet_buffer_config_t
  Defines the buffer descriptor configure structure. More...
- struct enet_multiqueue_config_t
  Defines the configuration when multi-queue is used. More...
- struct enet_config_t
  Defines the basic configuration structure for the ENET device. More...
- struct enet_tx_bd_ring_t
  Defines the ENET transmit buffer descriptor ring/queue structure. More...
- struct enet_rx_bd_ring_t
  Defines the ENET receive buffer descriptor ring/queue structure. More...
- struct enet_handle_t
  Defines the ENET handler structure. More...

Typedefs

- typedef void(∗ enet_callback_t)(ENET_Type ∗base, enet_handle_t ∗handle, enet_event_t event, uint8_t channel, 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 = MAKE_STATUS(kStatusGroup_ENET, 3U),
  kStatus_ENET_TxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 4U),
  kStatus_ENET_TxFrameOverLen = MAKE_STATUS(kStatusGroup_ENET, 5U) }
  Defines the status return codes for transaction.

Typical use case

- enum enet_mii_mode_t {
  kENET_MiiMode = 0U,
  kENET_RmiiMode = 1U }
  Defines the MII/RMII mode for data interface between the MAC and the PHY.
- enum enet_mii_speed_t {
  kENET_MiiSpeed10M = 0U,
  kENET_MiiSpeed100M = 1U }
  Defines the 10/100 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_normal_opcode {
  kENET_MiiWriteFrame = 1U,
  kENET_MiiReadFrame = 3U }
  Define the MII opcode for normal MDIO_CLAUSES_22 Frame.
- enum enet_dma_burstlen {
  kENET_BurstLen1 = 0x00001U,
  kENET_BurstLen2 = 0x00002U,
  kENET_BurstLen4 = 0x00004U,
  kENET_BurstLen8 = 0x00008U,
  kENET_BurstLen16 = 0x00010U,
  kENET_BurstLen32 = 0x00020U,
  kENET_BurstLen64 = 0x10008U,
  kENET_BurstLen128 = 0x10010U,
  kENET_BurstLen256 = 0x10020U }
  Define the DMA maximum transmit burst length.
- enum enet_desc_flag {
  kENET_MiddleFlag = 0,
  kENET_FirstFlagOnly,
  kENET_LastFlagOnly,
  kENET_FirstLastFlag }
  Define the flag for the descriptor.
- enum enet_systime_op {
  kENET_SystimeAdd = 0U,
  kENET_SystimeSubtract = 1U }
  Define the system time adjust operation control.
- enum enet_ts_rollover_type {
  kENET_BinaryRollover = 0,
  kENET_DigitalRollover = 1 }
  Define the system time rollover control.
- enum enet_special_config_t {

kENET_DescDoubleBuffer = 0x0001U,
kENET_StoreAndForward = 0x0002U,
kENET_PromiscuousEnable = 0x0004U,
kENET_FlowControlEnable = 0x0008U,
kENET_BroadCastRxDisable = 0x0010U,
kENET_MulticastAllEnable = 0x0020U,
kENET_8023AS2KPacket = 0x0040U }
Defines some special configuration for ENET.

- enum enet_dma_interrupt_enable_t {
kENET_DmaTx = ENET_DMA_CH_DMA_CHX_INT_EN_TIE_MASK,
kENET_DmaTxStop = ENET_DMA_CH_DMA_CHX_INT_EN_TSE_MASK,
kENET_DmaTxBuffUnavail = ENET_DMA_CH_DMA_CHX_INT_EN_TBUE_MASK,
kENET_DmaRx = ENET_DMA_CH_DMA_CHX_INT_EN_RIE_MASK,
kENET_DmaRxBuffUnavail = ENET_DMA_CH_DMA_CHX_INT_EN_RBUE_MASK,
kENET_DmaRxStop = ENET_DMA_CH_DMA_CHX_INT_EN_RSE_MASK,
kENET_DmaRxWatchdogTimeout = ENET_DMA_CH_DMA_CHX_INT_EN_RWTE_MASK,
kENET_DmaEarlyTx = ENET_DMA_CH_DMA_CHX_INT_EN_ETIE_MASK,
kENET_DmaEarlyRx = ENET_DMA_CH_DMA_CHX_INT_EN_ERIE_MASK,
kENET_DmaBusErr = ENET_DMA_CH_DMA_CHX_INT_EN_FBEE_MASK }
List of DMA interrupts supported by the ENET interrupt.
- enum enet_mac_interrupt_enable_t
List of mac interrupts supported by the ENET interrupt.
- enum enet_event_t {
kENET_RxIntEvent,
kENET_TxIntEvent,
kENET_WakeUpIntEvent,
kENET_TimeStampIntEvent }
Defines the common interrupt event for callback use.
- enum enet_dma_tx_sche {
kENET_FixPri = 0,
kENET_WeightStrPri,
kENET_WeightRoundRobin }
Define the DMA transmit arbitration for multi-queue.
- enum enet_mtl_multiqueue_txsche {
kENET_txWeightRR = 0U,
kENET_txStrPrio = 3U }
Define the MTL tx scheduling algorithm for multiple queues/rings.
- enum enet_mtl_multiqueue_rxsche {
kENET_rxStrPrio = 0U,
kENET_rxWeightStrPrio }
Define the MTL rx scheduling algorithm for multiple queues/rings.
- enum enet_mtl_rxqueuemap {
kENET_StaticDirctMap = 0x100U,
kENET_DynamicMap }
Define the MTL rx queue and DMA channel mapping.
- enum enet_ptp_event_type_t {

kENET_PtpEventMsgType = 3U,
kENET_PtpSrcPortIdLen = 10U,
kENET_PtpEventPort = 319U,
kENET_PtpGnrlPort = 320U }
Defines the ENET PTP message related constant.

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_RXDESCRIP_RD_BUFF1VALID_MASK (1U << 24)
  Defines for read format.
  Buffer2 address valid.
- #define ENET_RXDESCRIP_RD_BUFF2VALID_MASK (1U << 25)
  Interrupt enable on complete.
- #define ENET_RXDESCRIP_RD_IOC_MASK (1U << 30)
  Own bit.
- #define ENET_RXDESCRIP_RD_OWN_MASK (1U << 31)
  Defines for write back format.
- #define ENET_RXDESCRIP_WR_ERR_MASK ((1U << 3) | (1U << 7))
- #define **ENET_RXDESCRIP_WR_PYLOAD_MASK** (0x7U)
- #define **ENET_RXDESCRIP_WR_PTPMSGTYPE_MASK** (0xF00U)
- #define **ENET_RXDESCRIP_WR_PTPTYPE_MASK** (1U << 12)
- #define **ENET_RXDESCRIP_WR_PTPVERSION_MASK** (1U << 13)
- #define **ENET_RXDESCRIP_WR_PTPTSA_MASK** (1U << 14)
- #define **ENET_RXDESCRIP_WR_PACKETLEN_MASK** (0x7FFFU)
- #define **ENET_RXDESCRIP_WR_ERRSUM_MASK** (1U << 15)
- #define **ENET_RXDESCRIP_WR_TYPE_MASK** (0x30000U)
- #define **ENET_RXDESCRIP_WR_DE_MASK** (1U << 19)
- #define **ENET_RXDESCRIP_WR_RE_MASK** (1U << 20)
- #define **ENET_RXDESCRIP_WR_OE_MASK** (1U << 21)
- #define **ENET_RXDESCRIP_WR_RS0V_MASK** (1U << 25)
- #define **ENET_RXDESCRIP_WR_RS1V_MASK** (1U << 26)
- #define **ENET_RXDESCRIP_WR_RS2V_MASK** (1U << 27)
- #define **ENET_RXDESCRIP_WR_LD_MASK** (1U << 28)
- #define **ENET_RXDESCRIP_WR_FD_MASK** (1U << 29)
- #define **ENET_RXDESCRIP_WR_CTXT_MASK** (1U << 30)
- #define **ENET_RXDESCRIP_WR_OWN_MASK** (1U << 31)

Control and status bit masks of the transmit buffer descriptor.

- #define ENET_TXDESCRIP_RD_BL1_MASK (0x3fffU)
  Defines for read format.
- #define **ENET_TXDESCRIP_RD_BL2_MASK** (ENET_TXDESCRIP_RD_BL1_MASK << 16)
- #define **ENET_TXDESCRIP_RD_BL1**(n) ((uint32_t)(n) & ENET_TXDESCRIP_RD_BL1_MASK)
- #define **ENET_TXDESCRIP_RD_BL2**(n) (((uint32_t)(n) & ENET_TXDESCRIP_RD_BL1_MASK) << 16)

- #define **ENET_TXDESCRIP_RD_TTSE_MASK** (1U << 30)
- #define **ENET_TXDESCRIP_RD_IOC_MASK** (1U << 31)
- #define **ENET_TXDESCRIP_RD_FL_MASK** (0x7FFFU)
- #define **ENET_TXDESCRIP_RD_FL**(n) ((uint32_t)(n) & ENET_TXDESCRIP_RD_FL_MASK)
- #define **ENET_TXDESCRIP_RD_CIC**(n) (((uint32_t)(n) & 0x3) << 16)
- #define **ENET_TXDESCRIP_RD_TSE_MASK** (1U << 18)
- #define **ENET_TXDESCRIP_RD_SLOT**(n) (((uint32_t)(n) & 0x0f) << 19)
- #define **ENET_TXDESCRIP_RD_SAIC**(n) (((uint32_t)(n) & 0x07) << 23)
- #define **ENET_TXDESCRIP_RD_CPC**(n) (((uint32_t)(n) & 0x03) << 26)
- #define **ENET_TXDESCRIP_RD_LDFD**(n) (((uint32_t)(n) & 0x03) << 28)
- #define **ENET_TXDESCRIP_RD_LD_MASK** (1U << 28)
- #define **ENET_TXDESCRIP_RD_FD_MASK** (1U << 29)
- #define **ENET_TXDESCRIP_RD_CTXT_MASK** (1U << 30)
- #define **ENET_TXDESCRIP_RD_OWN_MASK** (1UL << 31)
- #define ENET_TXDESCRIP_WB_TTSS_MASK (1UL << 17)

  Defines for write back format.

Bit mask for interrupt enable type.

- #define **ENET_ABNORM_INT_MASK**
- #define **ENET_NORM_INT_MASK**

Defines some Ethernet parameters.

- #define ENET_FRAME_MAX_FRAMELEN (1522U)

  Maximum Ethernet frame size (normal vlan is supportedw).
- #define ENET_ADDR_ALIGNMENT (0x3U)

  Recommended ethernet buffer alignment.
- #define ENET_BUFF_ALIGNMENT (4U)

  Receive buffer alignment shall be 4bytes-aligned.
- #define ENET_RING_NUM_MAX (2U)

  The Maximum number of tx/rx descriptor rings.
- #define ENET_MTL_RXFIFOSIZE (2048U)

  The rx fifo size.
- #define ENET_MTL_TXFIFOSIZE (2048U)

  The tx fifo size.
- #define ENET_MACINT_ENUM_OFFSET (16U)

  The offest for mac interrupt in enum type.

Initialization and De-initialization

- void ENET_GetDefaultConfig (enet_config_t *config)

  Gets the ENET default configuration structure.
- void ENET_Init (ENET_Type *base, const enet_config_t *config, uint8_t *macAddr, uint32_t refclkSrc_Hz)

  Initializes the ENET module.
- void ENET_Deinit (ENET_Type *base)

  Deinitializes the ENET module.
- status_t ENET_DescriptorInit (ENET_Type *base, enet_config_t *config, enet_buffer_config_t *bufferConfig)

  Initialize for all ENET descriptors.
- void ENET_StartRxTx (ENET_Type *base, uint8_t txRingNum, uint8_t rxRingNum)

Starts the ENET rx/tx.

MII interface operation

- static 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)
  Sets the ENET SMI(serial management interface)- MII management interface.
- static bool ENET_IsSMIBusy (ENET_Type ∗base)
  Checks if the SMI is busy.
- static uint16_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)
  Starts an SMI read command.
- void ENET_StartSMIWrite (ENET_Type ∗base, uint32_t phyAddr, uint32_t phyReg, uint32_t data)
  Starts a SMI write command.

Other basic operation

- static 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_EnterPowerDown (ENET_Type ∗base, uint32_t ∗wakeFilter)
  Set the MAC to enter into power down mode.
- static void ENET_ExitPowerDown (ENET_Type ∗base)
  Set the MAC to exit power down mode.

Interrupts.

- void ENET_EnableInterrupts (ENET_Type ∗base, uint32_t mask)
  Enables the ENET DMA and MAC interrupts.
- void ENET_DisableInterrupts (ENET_Type ∗base, uint32_t mask)
  Disables the ENET DMA and MAC interrupts.
- static uint32_t ENET_GetDmaInterruptStatus (ENET_Type ∗base, uint8_t channel)
  Gets the ENET DMA interrupt status flag.
- static void ENET_ClearDmaInterruptStatus (ENET_Type ∗base, uint8_t channel, uint32_t mask)
  Clear the ENET DMA interrupt status flag.
- static uint32_t ENET_GetMacInterruptStatus (ENET_Type ∗base)
  Gets the ENET MAC interrupt status flag.
- void ENET_ClearMacInterruptStatus (ENET_Type ∗base, uint32_t mask)
  Clears the ENET mac interrupt events status flag.

Functional operation.

- static bool ENET_IsTxDescriptorDmaOwn (enet_tx_bd_struct_t ∗txDesc)
  Get the tx descriptor DMA Own flag.
- void ENET_SetupTxDescriptor (enet_tx_bd_struct_t ∗txDesc, void ∗buffer1, uint32_t bytes1, void ∗buffer2, uint32_t bytes2, uint32_t framelen, bool intEnable, bool tsEnable, enet_desc_flag flag, uint8_t slotNum)

Setup a given tx descriptor.
- static void ENET_UpdateTxDescriptorTail (ENET_Type *base, uint8_t channel, uint32_t txDescTailAddrAlign)
  Update the tx descriptor tail pointer.
- static void ENET_UpdateRxDescriptorTail (ENET_Type *base, uint8_t channel, uint32_t rxDescTailAddrAlign)
  Update the rx descriptor tail pointer.
- static uint32_t ENET_GetRxDescriptor (enet_rx_bd_struct_t *rxDesc)
  Gets the context in the ENET rx descriptor.
- void ENET_UpdateRxDescriptor (enet_rx_bd_struct_t *rxDesc, void *buffer1, void *buffer2, bool intEnable, bool doubleBuffEnable)
  Updates the buffers and the own status for a given rx descriptor.

Transactional operation

- void ENET_CreateHandler (ENET_Type *base, enet_handle_t *handle, enet_config_t *config, enet_buffer_config_t *bufferConfig, enet_callback_t callback, void *userData)
  Create ENET Handler.
- status_t ENET_GetRxFrameSize (ENET_Type *base, enet_handle_t *handle, uint32_t *length, uint8_t channel)
  Gets the size of the read frame.
- status_t ENET_ReadFrame (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length, uint8_t channel)
  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_ReclaimTxDescriptor (ENET_Type *base, enet_handle_t *handle, uint8_t channel)
  Reclaim tx descriptors.
- void ENET_PMTIRQHandler (ENET_Type *base, enet_handle_t *handle)
  The ENET PMT IRQ handler.
- void ENET_IRQHandler (ENET_Type *base, enet_handle_t *handle)
  The ENET IRQ handler.

14.3 Data Structure Documentation

14.3.1 struct enet_rx_bd_struct_t

They both has the same size with different region definition. so we define the read-format region as the recive descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the receive data process.

Data Fields

- __IO uint32_t buff1Addr
  Buffer 1 address.
- __IO uint32_t reserved
  Reserved.

- __IO uint32_t buff2Addr
  Buffer 2 or next descriptor address.
- __IO uint32_t control
  Buffer 1/2 byte counts and control.

14.3.2 struct enet_tx_bd_struct_t

They both has the same size with different region definition. so we define the read-format region as the transmit descriptor structure Use the read-format region mask bits in the descriptor initialization Use the write-back format region mask bits in the transmit data process.

Data Fields

- __IO uint32_t buff1Addr
  Buffer 1 address.
- __IO uint32_t buff2Addr
  Buffer 2 address.
- __IO uint32_t buffLen
  Buffer 1/2 byte counts.
- __IO uint32_t controlStat
  TDES control and status word.

14.3.3 struct enet_buffer_config_t

Notes:

1. The receive and transmit descriptor start address pointer and tail pointer must be word-aligned.
2. The recommended minimum tx/rx ring length is 4.
3. The tx/rx descriptor tail address shall be the address pointer to the address just after the end of the last last descriptor. because only the descriptors between the start address and the tail address will be used by DMA.
4. The decriptor address is the start address of all used contiguous memory. for example, the rxDescStartAddrAlign is the start address of rxRingLen contiguous descriptor memorise for rx descriptor ring 0.
5. The "∗rxBufferstartAddr" is the first element of rxRingLen (2∗rxRingLen for double buffers) rx buffers. It means the ∗rxBufferStartAddr is the rx buffer for the first descriptor the ∗rxBufferStartAddr + 1 is the rx buffer for the second descriptor or the rx buffer for the second buffer in the first descriptor. so please make sure the rxBufferStartAddr is the address of a rxRingLen or 2∗rxRingLen array.

Data Fields

- uint8_t rxRingLen

The length of receive buffer descriptor ring.
- uint8_t txRingLen
  The length of transmit buffer descriptor ring.
- enet_tx_bd_struct_t * txDescStartAddrAlign
  Aligned transmit descriptor start address.
- enet_tx_bd_struct_t * txDescTailAddrAlign
  Aligned transmit descriptor tail address.
- enet_rx_bd_struct_t * rxDescStartAddrAlign
  Aligned receive descriptor start address.
- enet_rx_bd_struct_t * rxDescTailAddrAlign
  Aligned receive descriptor tail address.
- uint32_t * rxBufferStartAddr
  Start address of the rx buffers.
- uint32_t rxBuffSizeAlign
  Aligned receive data buffer size.

14.3.3.0.0.15 Field Documentation

14.3.3.0.0.15.1 uint8_t enet_buffer_config_t::rxRingLen

14.3.3.0.0.15.2 uint8_t enet_buffer_config_t::txRingLen

14.3.3.0.0.15.3 enet_tx_bd_struct_t* enet_buffer_config_t::txDescStartAddrAlign

14.3.3.0.0.15.4 enet_tx_bd_struct_t* enet_buffer_config_t::txDescTailAddrAlign

14.3.3.0.0.15.5 enet_rx_bd_struct_t* enet_buffer_config_t::rxDescStartAddrAlign

14.3.3.0.0.15.6 enet_rx_bd_struct_t* enet_buffer_config_t::rxDescTailAddrAlign

14.3.3.0.0.15.7 uint32_t* enet_buffer_config_t::rxBufferStartAddr

14.3.3.0.0.15.8 uint32_t enet_buffer_config_t::rxBuffSizeAlign

14.3.4 struct enet_multiqueue_config_t

Data Fields

- enet_dma_tx_sche dmaTxSche
  Transmit arbitation.
- enet_dma_burstlen burstLen
  Burset len for the queue 1.
- uint8_t txdmaChnWeight [ENET_RING_NUM_MAX]
  Transmit channel weight.
- enet_mtl_multiqueue_txsche mtltxSche
  Transmit schedule for multi-queue.
- enet_mtl_multiqueue_rxsche mtlrxSche
  Receive schedule for multi-queue.
- uint8_t rxqueweight [ENET_RING_NUM_MAX]
  Refer to the MTL RxQ Control register.

- uint32_t txqueweight [ENET_RING_NUM_MAX]
  Refer to the MTL TxQ Quantum Weight register.
- uint8_t rxqueuePrio [ENET_RING_NUM_MAX]
  Receive queue priority.
- uint8_t txqueuePrio [ENET_RING_NUM_MAX]
  Refer to Transmit Queue Priority Mapping register.
- enet_mtl_rxqueuemap mtlrxQuemap
  Rx queue DMA Channel mapping.

14.3.4.0.0.16 Field Documentation

14.3.4.0.0.16.1 enet_dma_tx_sche enet_multiqueue_config_t::dmaTxSche

14.3.4.0.0.16.2 enet_dma_burstlen enet_multiqueue_config_t::burstLen

14.3.4.0.0.16.3 uint8_t enet_multiqueue_config_t::txdmaChnWeight[ENET_RING_NUM_MAX]

14.3.4.0.0.16.4 enet_mtl_multiqueue_txsche enet_multiqueue_config_t::mtltxSche

14.3.4.0.0.16.5 enet_mtl_multiqueue_rxsche enet_multiqueue_config_t::mtlrxSche

14.3.4.0.0.16.6 uint8_t enet_multiqueue_config_t::rxqueweight[ENET_RING_NUM_MAX]

14.3.4.0.0.16.7 uint32_t enet_multiqueue_config_t::txqueweight[ENET_RING_NUM_MAX]

14.3.4.0.0.16.8 uint8_t enet_multiqueue_config_t::rxqueuePrio[ENET_RING_NUM_MAX]

14.3.4.0.0.16.9 uint8_t enet_multiqueue_config_t::txqueuePrio[ENET_RING_NUM_MAX]

14.3.4.0.0.16.10 enet_mtl_rxqueuemap enet_multiqueue_config_t::mtlrxQuemap

14.3.5 struct enet_config_t

Note:

1. Default the signal queue is used so the "∗multiqueueCfg" is set default with NULL. Set the pointer with a valid configration pointer if the multiple queues are required. If multiple queue is enabled, please make sure the buffer configuration for all are prepared also.

Data Fields

- uint16_t specialControl
  The logicl or of enet_special_config_t.
- enet_multiqueue_config_t ∗ multiqueueCfg
  Use both tx/rx queue(dma channel) 0 and 1.
- enet_mii_mode_t miiMode
  MII mode.
- enet_mii_speed_t miiSpeed
  MII Speed.

- enet_mii_duplex_t miiDuplex
  MII duplex.
- uint16_t pauseDuration
  Used in the tx flow control frame, only valid when kENET_FlowControlEnable is set.

14.3.5.0.0.17 Field Documentation

14.3.5.0.0.17.1 enet_multiqueue_config_t* enet_config_t::multiqueueCfg

14.3.5.0.0.17.2 enet_mii_mode_t enet_config_t::miiMode

14.3.5.0.0.17.3 enet_mii_speed_t enet_config_t::miiSpeed

14.3.5.0.0.17.4 enet_mii_duplex_t enet_config_t::miiDuplex

14.3.5.0.0.17.5 uint16_t enet_config_t::pauseDuration

14.3.6 struct enet_tx_bd_ring_t

Data Fields

- enet_tx_bd_struct_t * txBdBase
  Buffer descriptor base address pointer.
- uint16_t txGenIdx
  tx generate index.
- uint16_t txConsumIdx
  tx consum index.
- volatile uint16_t txDescUsed
  tx descriptor used number.
- uint16_t txRingLen
  tx ring length.

14.3.6.0.0.18 Field Documentation

14.3.6.0.0.18.1 enet_tx_bd_struct_t* enet_tx_bd_ring_t::txBdBase

14.3.6.0.0.18.2 uint16_t enet_tx_bd_ring_t::txGenIdx

14.3.6.0.0.18.3 uint16_t enet_tx_bd_ring_t::txConsumIdx

14.3.6.0.0.18.4 volatile uint16_t enet_tx_bd_ring_t::txDescUsed

14.3.6.0.0.18.5 uint16_t enet_tx_bd_ring_t::txRingLen

14.3.7 struct enet_rx_bd_ring_t

Data Fields

- enet_rx_bd_struct_t * rxBdBase

Buffer descriptor base address pointer.
- uint16_t rxGenIdx
  The current available receive buffer descriptor pointer.
- uint16_t rxRingLen
  Receive ring length.
- uint32_t rxBuffSizeAlign
  Receive buffer size.

14.3.7.0.0.19 Field Documentation

14.3.7.0.0.19.1 enet_rx_bd_struct_t* enet_rx_bd_ring_t::rxBdBase

14.3.7.0.0.19.2 uint16_t enet_rx_bd_ring_t::rxGenIdx

14.3.7.0.0.19.3 uint16_t enet_rx_bd_ring_t::rxRingLen

14.3.7.0.0.19.4 uint32_t enet_rx_bd_ring_t::rxBuffSizeAlign

14.3.8 struct _enet_handle

Data Fields

- bool multiQueEnable
  Enable multi-queue.
- bool doubleBuffEnable
  The double buffer is used in the descriptor.
- bool rxintEnable
  Rx interrup enabled.
- enet_rx_bd_ring_t rxBdRing [ENET_RING_NUM_MAX]
  Receive buffer descriptor.
- enet_tx_bd_ring_t txBdRing [ENET_RING_NUM_MAX]
  Transmit buffer descriptor.
- enet_callback_t callback
  Callback function.
- void * userData
  Callback function parameter.

14.3.8.0.0.20 Field Documentation

14.3.8.0.0.20.1 bool enet_handle_t::multiQueEnable

14.3.8.0.0.20.2 bool enet_handle_t::doubleBuffEnable

14.3.8.0.0.20.3 bool enet_handle_t::rxintEnable

14.3.8.0.0.20.4 enet_rx_bd_ring_t enet_handle_t::rxBdRing[ENET_RING_NUM_MAX]

14.3.8.0.0.20.5 enet_tx_bd_ring_t enet_handle_t::txBdRing[ENET_RING_NUM_MAX]

14.3.8.0.0.20.6 enet_callback_t enet_handle_t::callback

14.3.8.0.0.20.7 void* enet_handle_t::userData

14.4 Macro Definition Documentation

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

Version 2.0.0.

14.4.2 #define ENET_RXDESCRIP_RD_BUFF1VALID_MASK (1U << 24)

Buffer1 address valid.

14.4.3 #define ENET_RXDESCRIP_RD_BUFF2VALID_MASK (1U << 25)

14.4.4 #define ENET_RXDESCRIP_RD_IOC_MASK (1U << 30)

14.4.5 #define ENET_RXDESCRIP_RD_OWN_MASK (1U << 31)

14.4.6 #define ENET_RXDESCRIP_WR_ERR_MASK ((1U << 3) | (1U << 7))

14.4.7 #define ENET_TXDESCRIP_RD_BL1_MASK (0x3fffU)

14.4.8 #define ENET_TXDESCRIP_WB_TTSS_MASK (1UL << 17)

14.4.9 #define ENET_FRAME_MAX_FRAMELEN (1522U)

14.4.10 #define ENET_ADDR_ALIGNMENT (0x3U)

14.4.11 #define ENET_BUFF_ALIGNMENT (4U)

14.4.12 #define ENET_RING_NUM_MAX (2U)

14.4.13 #define ENET_MTL_RXFIFOSIZE (2048U)

14.4.14 #define ENET_MTL_TXFIFOSIZE (2048U)

14.4.15 #define ENET_MACINT_ENUM_OFFSET (16U)

14.5 Typedef Documentation

14.5.1 typedef void(* enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event, uint8_t channel, void *userData)

14.6 Enumeration Type Documentation

14.6.1 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 descriptors are under process.

kStatus_ENET_TxFrameFail Transmit frame fail.
kStatus_ENET_TxFrameOverLen Transmit oversize.

14.6.2 enum enet_mii_mode_t

Enumerator

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

14.6.3 enum enet_mii_speed_t

Enumerator

kENET_MiiSpeed10M Speed 10 Mbps.
kENET_MiiSpeed100M Speed 100 Mbps.

14.6.4 enum enet_mii_duplex_t

Enumerator

kENET_MiiHalfDuplex Half duplex mode.
kENET_MiiFullDuplex Full duplex mode.

14.6.5 enum enet_mii_normal_opcode

Enumerator

kENET_MiiWriteFrame Write frame operation for a valid MII management frame.
kENET_MiiReadFrame Read frame operation for a valid MII management frame.

14.6.6 enum enet_dma_burstlen

Enumerator

kENET_BurstLen1 DMA burst length 1.
kENET_BurstLen2 DMA burst length 2.
kENET_BurstLen4 DMA burst length 4.
kENET_BurstLen8 DMA burst length 8.

kENET_BurstLen16 DMA burst length 16.
kENET_BurstLen32 DMA burst length 32.
kENET_BurstLen64 DMA burst length 64. eight times enabled.
kENET_BurstLen128 DMA burst length 128. eight times enabled.
kENET_BurstLen256 DMA burst length 256. eight times enabled.

14.6.7 enum enet_desc_flag

Enumerator

kENET_MiddleFlag It's a middle descriptor of the frame.
kENET_FirstFlagOnly It's the first descriptor of the frame.
kENET_LastFlagOnly It's the last descriptor of the frame.
kENET_FirstLastFlag It's the first and last descriptor of the frame.

14.6.8 enum enet_systime_op

Enumerator

kENET_SystimeAdd System time add to.
kENET_SystimeSubtract System time subtract.

14.6.9 enum enet_ts_rollover_type

Enumerator

kENET_BinaryRollover System time binary rollover.
kENET_DigitalRollover System time digital rollover.

14.6.10 enum enet_special_config_t

These control flags are provided for special user requirements. Normally, these is no need to set this control flags for ENET initialization. But if you have some special requirements, set the flags to specialControl in the enet_config_t.

Note

"kENET_StoreAndForward" is recommended to be set when the ENET_PTP1588FEATURE_REQUIRED is defined or else the timestamp will be mess-up when the overflow happens.

Enumerator

kENET_DescDoubleBuffer The double buffer is used in the tx/rx descriptor.
kENET_StoreAndForward The rx/tx store and forward enable.
kENET_PromiscuousEnable The promiscuous enabled.
kENET_FlowControlEnable The flow control enabled.
kENET_BroadCastRxDisable The broadcast disabled.
kENET_MulticastAllEnable All multicast are passed.
kENET_8023AS2KPacket 8023as support for 2K packets.

14.6.11 enum enet_dma_interrupt_enable_t

This enumeration uses one-bot encoding to allow a logical OR of multiple members.

Enumerator

kENET_DmaTx Tx interrupt.
kENET_DmaTxStop Tx stop interrupt.
kENET_DmaTxBuffUnavail Tx buffer unavailable.
kENET_DmaRx Rx interrupt.
kENET_DmaRxBuffUnavail Rx buffer unavailable.
kENET_DmaRxStop Rx stop.
kENET_DmaRxWatchdogTimeout Rx watchdog timeout.
kENET_DmaEarlyTx Early transmit.
kENET_DmaEarlyRx Early receive.
kENET_DmaBusErr Fatal bus error.

14.6.12 enum enet_mac_interrupt_enable_t

This enumeration uses one-bot encoding to allow a logical OR of multiple members.

14.6.13 enum enet_event_t

Enumerator

kENET_RxIntEvent Receive interrupt event.
kENET_TxIntEvent Transmit interrupt event.
kENET_WakeUpIntEvent Wake up interrupt event.
kENET_TimeStampIntEvent Time stamp interrupt event.

14.6.14 enum enet_dma_tx_sche

Enumerator

kENET_FixPri Fixed priority. channel 0 has lower priority than channel 1.
kENET_WeightStrPri Weighted(burst length) strict priority.
kENET_WeightRoundRobin Weighted (weight factor) round robin.

14.6.15 enum enet_mtl_multiqueue_txsche

Enumerator

kENET_txWeightRR Tx weight round-robin.
kENET_txStrPrio Tx strict priority.

14.6.16 enum enet_mtl_multiqueue_rxsche

Enumerator

kENET_rxStrPrio Tx weight round-robin, rx strict priority.
kENET_rxWeightStrPrio Tx strict priority, rx weight strict priority.

14.6.17 enum enet_mtl_rxqueuemap

Enumerator

kENET_StaticDirctMap The received fame in rx Qn(n = 0,1) direclty map to dma channel n.
kENET_DynamicMap The received frame in rx Qn(n = 0,1) map to the dma channel m(m = 0,1) related with the same Mac.

14.6.18 enum enet_ptp_event_type_t

Enumerator

kENET_PtpEventMsgType PTP event message type.
kENET_PtpSrcPortIdLen PTP message sequence id length.
kENET_PtpEventPort PTP event port number.
kENET_PtpGnrlPort PTP general port number.

14.7 Function Documentation

14.7.1 void ENET_GetDefaultConfig (enet_config_t * *config*)

The purpose of this API is to get the default ENET 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.

14.7.2 void ENET_Init (ENET_Type * *base*, const enet_config_t * *config*, uint8_t * *macAddr*, uint32_t *refclkSrc_Hz*)

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

Parameters

base	ENET peripheral base address.
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.
macAddr	ENET mac address of Ethernet device. This MAC address should be provided.
refclkSrc_Hz	ENET input reference clock.

14.7.3 void ENET_Deinit (ENET_Type * *base*)

This function gates the module clock and disables the ENET module.

Parameters

base	ENET peripheral base address.

14.7.4 status_t ENET_DescriptorInit (ENET_Type * *base*, enet_config_t * *config*, enet_buffer_config_t * *bufferConfig*)

Note

This function is do all tx/rx descriptors initialization. Because this API read all interrupt registers first and then set the interrupt flag for all descriptos, if the interrupt register is set. so the descriptor initialization should be called after ENET_Init(), ENET_EnableInterrupts() and ENET_CreateHandle()(if transactional APIs are used).

Parameters

base	ENET peripheral base address.
config	The configuration for ENET.
bufferConfig	All buffers configuration.

14.7.5 void ENET_StartRxTx (ENET_Type * *base*, uint8_t *txRingNum*, uint8_t *rxRingNum*)

This function enable the tx/rx and starts the rx/tx DMA. This shall be set after ENET initialization and before starting to receive the data.

Parameters

base	ENET peripheral base address.
rxRingNum	The number of the used rx rings. It shall not be larger than the ENET_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.
txRingNum	The number of the used tx rings. It shall not be larger than the ENET_RING_NUM_MAX(2). If the ringNum is set with 1, the ring 0 will be used.

Note

This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

14.7.6 static void ENET_SetMII (ENET_Type * *base*, enet_mii_speed_t *speed*, enet_mii_duplex_t *duplex*) [static]

This API is provided to dynamically change the speed and dulpex for MAC.

Parameters

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

14.7.7 void ENET_SetSMI (ENET_Type * *base*)

Parameters

base	ENET peripheral base address.

14.7.8 static bool ENET_IsSMIBusy (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The status of MII Busy status.

14.7.9 static uint16_t ENET_ReadSMIData (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The data read from PHY

14.7.10 void ENET_StartSMIRead (ENET_Type * *base,* uint32_t *phyAddr,* uint32_t *phyReg*)

support both MDIO IEEE802.3 Clause 22 and clause 45.

Parameters

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

14.7.11 void ENET_StartSMIWrite (ENET_Type * *base*, uint32_t *phyAddr*, uint32_t *phyReg*, uint32_t *data*)

support both MDIO IEEE802.3 Clause 22 and clause 45.

Parameters

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

14.7.12 static void ENET_SetMacAddr (ENET_Type * *base*, uint8_t * *macAddr*) [inline], [static]

Parameters

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

14.7.13 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.

14.7.14 void ENET_EnterPowerDown (ENET_Type * *base*, uint32_t * *wakeFilter*)

the remote power wake up frame and magic frame can wake up the ENET from the power down mode.

Parameters

base	ENET peripheral base address.
wakeFilter	The wakeFilter provided to configure the wake up frame fitlter. Set the wakeFilter to NULL is not required. But if you have the filter requirement, please make sure the wakeFilter pointer shall be eight continous 32-bits configuration.

14.7.15 static void ENET_ExitPowerDown (ENET_Type * *base*) [inline], [static]

Eixt from the power down mode and recover to noraml work mode.

Parameters

base	ENET peripheral base address.

14.7.16 void ENET_EnableInterrupts (ENET_Type * *base*, uint32_t *mask*)

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t. For example, to enable the dma and mac interrupt, do the following.

```
*     ENET_EnableInterrupts(ENET, kENET_DmaRx |
      kENET_DmaTx | kENET_MacPmt);
*
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to enable. This is a logical OR of both enumeration :: enet_dma_-interrupt_enable_t and enet_mac_interrupt_enable_t.

14.7.17 void ENET_DisableInterrupts (ENET_Type * *base*, uint32_t *mask*)

This function disables the ENET interrupt according to the provided mask. The mask is a logical OR of enet_dma_interrupt_enable_t and enet_mac_interrupt_enable_t. For example, to disable the dma and mac interrupt, do the following.

```
*     ENET_DisableInterrupts(ENET, kENET_DmaRx |
      kENET_DmaTx | kENET_MacPmt);
*
```

Parameters

base	ENET peripheral base address.
mask	ENET interrupts to disables. This is a logical OR of both enumeration :: enet_dma_-interrupt_enable_t and enet_mac_interrupt_enable_t.

14.7.18 static uint32_t ENET_GetDmaInterruptStatus (ENET_Type * *base*, uint8_t *channel*) [inline], [static]

Parameters

base	ENET peripheral base address.
channel	The DMA Channel. Shall not be larger than ENET_RING_NUM_MAX.

Returns

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

14.7.19 static void ENET_ClearDmaInterruptStatus (ENET_Type * *base*, uint8_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	ENET peripheral base address.
channel	The DMA Channel. Shall not be larger than ENET_RING_NUM_MAX.

Returns

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

14.7.20 static uint32_t ENET_GetMacInterruptStatus (ENET_Type * *base*) [inline], [static]

Parameters

base	ENET peripheral base address.

Returns

The event status of the interrupt source. Use the enum in enet_mac_interrupt_enable_t and right shift ENET_MACINT_ENUM_OFFSET to mask the returned value to get the exact interrupt status.

14.7.21 void ENET_ClearMacInterruptStatus (ENET_Type * *base*, uint32_t *mask*)

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

```
*     ENET_ClearMacInterruptStatus(ENET, kENET_MacPmt);
*
```

Parameters

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

14.7.22 static bool ENET_IsTxDescriptorDmaOwn (enet_tx_bd_struct_t * *txDesc*) [inline], [static]

Parameters

txDesc	The given tx descriptor.

Return values

True	the dma own tx descriptor, false application own tx descriptor.

14.7.23 void ENET_SetupTxDescriptor (enet_tx_bd_struct_t * *txDesc,* void * *buffer1,* uint32_t *bytes1,* void * *buffer2,* uint32_t *bytes2,* uint32_t *framelen,* bool *intEnable,* bool *tsEnable,* enet_desc_flag *flag,* uint8_t *slotNum*)

This function is a low level functional API to setup or prepare a given tx descriptor.

Parameters

txDesc	The given tx descriptor.
buffer1	The first buffer address in the descriptor.
bytes1	The bytes in the fist buffer.
buffer2	The second buffer address in the descriptor.
bytes1	The bytes in the second buffer.
framelen	The length of the frame to be transmitted.
intEnable	Interrupt enable flag.
tsEnable	The timestamp enable.
flag	The flag of this tx desciriptor, see "enet_desc_flag" .
slotNum	The slot num used for AV only.

Note

This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required. Transmit buffers are 'zero-copy' buffers, so the buffer must remain in memory until the packet has been fully transmitted. The buffers should be free or requeued in the transmit interrupt irq handler.

14.7.24 static void ENET_UpdateTxDescriptorTail (ENET_Type * *base,* uint8_t *channel,* uint32_t *txDescTailAddrAlign*) [inline], [static]

This function is a low level functional API to update the the tx descriptor tail. This is called after you setup a new tx descriptor to update the tail pointer to make the new descritor accessable by DMA.

Parameters

base	ENET peripheral base address.
channel	The tx DMA channel.
txDescTail-AddrAlign	The new tx tail pointer address.

14.7.25 static void ENET_UpdateRxDescriptorTail (ENET_Type * *base,* uint8_t *channel,* uint32_t *rxDescTailAddrAlign*) [inline], [static]

This function is a low level functional API to update the the rx descriptor tail. This is called after you setup a new rx descriptor to update the tail pointer to make the new descritor accessable by DMA and to anouse the rx poll command for DMA.

Parameters

base	ENET peripheral base address.
channel	The rx DMA channel.
rxDescTail-AddrAlign	The new rx tail pointer address.

14.7.26 static uint32_t ENET_GetRxDescriptor (enet_rx_bd_struct_t * *rxDesc*) [inline], [static]

This function is a low level functional API to get the the status flag from a given rx descriptor.

Parameters

rxDesc	The given rx descriptor.

Return values

The	RDES3 regions for write-back format rx buffer descriptor.

Note

This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

14.7.27 void ENET_UpdateRxDescriptor (enet_rx_bd_struct_t * *rxDesc,* void * *buffer1,* void * *buffer2,* bool *intEnable,* bool *doubleBuffEnable*)

This function is a low level functional API to Updates the buffers and the own status for a given rx descriptor.

Parameters

rxDesc	The given rx descriptor.
buffer1	The first buffer address in the descriptor.
buffer2	The second buffer address in the descriptor.
intEnable	Interrupt enable flag.
doubleBuffEnable	The double buffer enable flag.

Note

This must be called after all the ENET initilization. And should be called when the ENET receive/transmit is required.

14.7.28 void ENET_CreateHandler (ENET_Type * *base*, enet_handle_t * *handle*, enet_config_t * *config*, enet_buffer_config_t * *bufferConfig*, enet_callback_t *callback*, void * *userData*)

This is a transactional API and it's provided to store all datas which are needed during the whole transactional process. This API should not be used when you use functional APIs to do data tx/rx. This is funtion will store many data/flag for transactional use, so all configure API such as ENET_Init(), ENET_DescriptorInit(), ENET_EnableInterrupts() etc.

Note

as our transactional transmit API use the zero-copy transmit buffer. so there are two thing we emphasize here:

1. tx buffer free/requeue for application should be done in the tx interrupt handler. Please set callback: kENET_TxIntEvent with tx buffer free/requeue process APIs.
2. the tx interrupt is forced to open.

Parameters

base	ENET peripheral base address.
handle	ENET handler.

config	ENET configuration.
bufferConfig	ENET buffer configuration.
callback	The callback function.
userData	The application data.

14.7.29 status_t ENET_GetRxFrameSize (ENET_Type * *base*, enet_handle_t * *handle*, uint32_t * *length*, uint8_t *channel*)

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

Note

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

Parameters

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

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.

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

This function reads a frame from the ENET DMA descriptors. The ENET_GetRxFrameSize should be used to get the size of the prepared data buffer. For example use rx dma channel 0:

```
*       uint32_t length;
*       enet_handle_t g_handle;
```

```
*       //Get the received frame size firstly.
*       status = ENET_GetRxFrameSize(&g_handle, &length, 0);
*       if (length != 0)
*       {
*           //Allocate memory here with the size of "length"
*           uint8_t *data = memory allocate interface;
*           if (!data)
*           {
*               ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0);
*               //Add the console warning log.
*           }
*           else
*           {
*              status = ENET_ReadFrame(ENET, &g_handle, data, length, 0);
*              //Call stack input API to deliver the data to stack
*           }
*       }
*       else if (status == kStatus_ENET_RxFrameError)
*       {
*          //Update the received buffer when a error frame is received.
*           ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0);
*       }
*
```

Parameters

base	ENET peripheral base address.
handle	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to store the frame which memory size should be at least "length".
length	The size of the data buffer which is still the length of the received frame.
channel	The rx DMA channel. shall not be larger than 2.

Returns

The execute status, successful or failure.

14.7.31 status_t ENET_SendFrame (ENET_Type * *base*, enet_handle_t * *handle*, uint8_t * *data*, uint32_t *length*)

Note

The CRC is automatically appended to the data. Input the data to send without the CRC.

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
data	The data buffer provided by user to be send.
length	The length of the data to be send.

Return values

kStatus_Success	Send frame succeed.
kStatus_ENET_TxFrame-Busy	Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added after each call return with kStatus-_ENET_TxFrameBusy.

14.7.32 void ENET_ReclaimTxDescriptor (ENET_Type * *base,* enet_handle_t * *handle,* uint8_t *channel*)

This function is used to update the tx descriptor status and store the tx timestamp when the 1588 feature is enabled. This is called by the transmit interupt IRQ handler after the complete of a frame transmission.

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
channel	The tx DMA channnel.

14.7.33 void ENET_PMTIRQHandler (ENET_Type * *base,* enet_handle_t * *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

14.7.34 void ENET_IRQHandler (ENET_Type * *base,* enet_handle_t * *handle*)

Parameters

base	ENET peripheral base address.
handle	The ENET handler pointer.

Chapter 15
FLASHIAP: Flash In Application Programming Driver

15.1 Overview

The SDK provides a driver for the Flash In Application Programming (FLASHIAP).

It provides a set of functions to call the on chip in application flash programming interface. User code executing from on chip flash or ram can call these function to erase and write the flash memory.

15.2 GFlash In Application Programming operation

FLASHIAP_PrepareSectorForWrite() prepares a sector for write or erase operation.

FLASHIAP_CopyRamToFlash() function programs the flash memory.

FLASHIAP_EraseSector() function erase a flash sector. A sector must be erased before write operation.

15.3 Typical use case

```
/* Prepare sector before erase operation */
FLASHIAP_PrepareSectorForWrite(1, 1);

/* Erase sector 1 */
FLASHIAP_EraseSector(1, 1, SystemCoreClock);

/* Prepare sector before write operation */
FLASHIAP_PrepareSectorForWrite(1, 1);

/* Write sector 1  */
FLASHIAP_CopyRamToFlash(SECTOR_1_ADRESS, DATA_BUFFER_ADDRESS, NUM_OF_BYTES_TO_WRITE,
       SystemCoreClock);
```

Files

- file fsl_flashiap.h

Typedefs

- typedef void(∗ IAP_ENTRY_T)(uint32_t cmd[5], uint32_t stat[4])
  IAP_ENTRY API function type.

Typical use case

Enumerations

- enum _flashiap_status {
  kStatus_FLASHIAP_Success = kStatus_Success,
  kStatus_FLASHIAP_InvalidCommand = MAKE_STATUS(kStatusGroup_FLASHIAP, 1U),
  kStatus_FLASHIAP_SrcAddrError,
  kStatus_FLASHIAP_DstAddrError,
  kStatus_FLASHIAP_SrcAddrNotMapped,
  kStatus_FLASHIAP_DstAddrNotMapped,
  kStatus_FLASHIAP_CountError,
  kStatus_FLASHIAP_InvalidSector,
  kStatus_FLASHIAP_SectorNotblank = MAKE_STATUS(kStatusGroup_FLASHIAP, 8U),
  kStatus_FLASHIAP_NotPrepared,
  kStatus_FLASHIAP_CompareError,
  kStatus_FLASHIAP_Busy,
  kStatus_FLASHIAP_ParamError,
  kStatus_FLASHIAP_AddrError = MAKE_STATUS(kStatusGroup_FLASHIAP, 13U),
  kStatus_FLASHIAP_AddrNotMapped,
  kStatus_FLASHIAP_NoPower = MAKE_STATUS(kStatusGroup_FLASHIAP, 24U),
  kStatus_FLASHIAP_NoClock }
  Flashiap status codes.
- enum _flashiap_commands {
  kIapCmd_FLASHIAP_PrepareSectorforWrite = 50U,
  kIapCmd_FLASHIAP_CopyRamToFlash = 51U,
  kIapCmd_FLASHIAP_EraseSector = 52U,
  kIapCmd_FLASHIAP_BlankCheckSector = 53U,
  kIapCmd_FLASHIAP_ReadPartId = 54U,
  kIapCmd_FLASHIAP_Read_BootromVersion = 55U,
  kIapCmd_FLASHIAP_Compare = 56U,
  kIapCmd_FLASHIAP_ReinvokeISP = 57U,
  kIapCmd_FLASHIAP_ReadUid = 58U,
  kIapCmd_FLASHIAP_ErasePage = 59U,
  kIapCmd_FLASHIAP_ReadMisr = 70U,
  kIapCmd_FLASHIAP_ReinvokeI2cSpiISP = 71U }
  Flashiap command codes.

Functions

- static void iap_entry (uint32_t *cmd_param, uint32_t *status_result)
  IAP_ENTRY API function type.
- status_t FLASHIAP_PrepareSectorForWrite (uint32_t startSector, uint32_t endSector)
  Prepare sector for write operation.
- status_t FLASHIAP_CopyRamToFlash (uint32_t dstAddr, uint32_t *srcAddr, uint32_t numOfBytes, uint32_t systemCoreClock)
  Copy RAM to flash.
- status_t FLASHIAP_EraseSector (uint32_t startSector, uint32_t endSector, uint32_t systemCoreClock)

Erase sector.
- status_t FLASHIAP_ErasePage (uint32_t startPage, uint32_t endPage, uint32_t systemCoreClock)
  This function erases page(s).
- status_t FLASHIAP_BlankCheckSector (uint32_t startSector, uint32_t endSector)
  Blank check sector(s)
- status_t FLASHIAP_Compare (uint32_t dstAddr, uint32_t *srcAddr, uint32_t numOfBytes)
  Compare memory contents of flash with ram.

Driver version

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

15.4 Macro Definition Documentation

15.4.1 #define FSL_FLASHIAP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

15.5 Enumeration Type Documentation

15.5.1 enum _flashiap_status

Enumerator

kStatus_FLASHIAP_Success Api is executed successfully.
kStatus_FLASHIAP_InvalidCommand Invalid command.
kStatus_FLASHIAP_SrcAddrError Source address is not on word boundary.
kStatus_FLASHIAP_DstAddrError Destination address is not on a correct boundary.
kStatus_FLASHIAP_SrcAddrNotMapped Source address is not mapped in the memory map.
kStatus_FLASHIAP_DstAddrNotMapped Destination address is not mapped in the memory map.
kStatus_FLASHIAP_CountError Byte count is not multiple of 4 or is not a permitted value.
kStatus_FLASHIAP_InvalidSector Sector number is invalid or end sector number is greater than start sector number.
kStatus_FLASHIAP_SectorNotblank One or more sectors are not blank.
kStatus_FLASHIAP_NotPrepared Command to prepare sector for write operation was not executed.
kStatus_FLASHIAP_CompareError Destination and source memory contents do not match.
kStatus_FLASHIAP_Busy Flash programming hardware interface is busy.
kStatus_FLASHIAP_ParamError Insufficient number of parameters or invalid parameter.
kStatus_FLASHIAP_AddrError Address is not on word boundary.
kStatus_FLASHIAP_AddrNotMapped Address is not mapped in the memory map.
kStatus_FLASHIAP_NoPower Flash memory block is powered down.
kStatus_FLASHIAP_NoClock Flash memory block or controller is not clocked.

15.5.2 enum _flashiap_commands

Enumerator

kIapCmd_FLASHIAP_PrepareSectorforWrite Prepare Sector for write.
kIapCmd_FLASHIAP_CopyRamToFlash Copy RAM to flash.
kIapCmd_FLASHIAP_EraseSector Erase Sector.
kIapCmd_FLASHIAP_BlankCheckSector Blank check sector.
kIapCmd_FLASHIAP_ReadPartId Read part id.
kIapCmd_FLASHIAP_Read_BootromVersion Read bootrom version.
kIapCmd_FLASHIAP_Compare Compare.
kIapCmd_FLASHIAP_ReinvokeISP Reinvoke ISP.
kIapCmd_FLASHIAP_ReadUid Read Uid isp.
kIapCmd_FLASHIAP_ErasePage Erase Page.
kIapCmd_FLASHIAP_ReadMisr Read Misr.
kIapCmd_FLASHIAP_ReinvokeI2cSpiISP Reinvoke I2C/SPI isp.

15.6 Function Documentation

15.6.1 static void iap_entry (uint32_t * *cmd_param*, uint32_t * *status_result*) [inline], [static]

Wrapper for rom iap call

Parameters

cmd_param	IAP command and relevant parameter array.
status_result	IAP status result array.

Return values

None.	Status/Result is returned via status_result array.

15.6.2 status_t FLASHIAP_PrepareSectorForWrite (uint32_t *startSector*, uint32_t *endSector*)

This function prepares sector(s) for write/erase operation. This function must be called before calling the FLASHIAP_CopyRamToFlash() or FLASHIAP_EraseSector() or FLASHIAP_ErasePage() function. The end sector must be greater than or equal to start sector number.

Parameters

startSector	Start sector number.
endSector	End sector number.

Return values

kStatus_FLASHIAP_Success	Api was executed successfully.
kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_InvalidSector	Sector number is invalid or end sector number is greater than start sector number.
kStatus_FLASHIAP_Busy	Flash programming hardware interface is busy.

15.6.3 status_t FLASHIAP_CopyRamToFlash (uint32_t *dstAddr,* uint32_t * *srcAddr,* uint32_t *numOfBytes,* uint32_t *systemCoreClock*)

This function programs the flash memory. Corresponding sectors must be prepared via FLASHIAP_PrepareSectorForWrite before calling calling this function. The addresses should be a 256 byte boundary and the number of bytes should be 256 | 512 | 1024 | 4096.

Parameters

dstAddr	Destination flash address where data bytes are to be written.
srcAddr	Source ram address from where data bytes are to be read.
numOfBytes	Number of bytes to be written.
systemCoreClock	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

kStatus_FLASHIAP_Success	Api was executed successfully.

kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_SrcAddrError	Source address is not on word boundary.
kStatus_FLASHIAP_DstAddrError	Destination address is not on a correct boundary.
kStatus_FLASHIAP_SrcAddrNotMapped	Source address is not mapped in the memory map.
kStatus_FLASHIAP_DstAddrNotMapped	Destination address is not mapped in the memory map.
kStatus_FLASHIAP_CountError	Byte count is not multiple of 4 or is not a permitted value.
kStatus_FLASHIAP_NotPrepared	Command to prepare sector for write operation was not executed.
kStatus_FLASHIAP_Busy	Flash programming hardware interface is busy.

15.6.4 status_t FLASHIAP_EraseSector (uint32_t *startSector,* uint32_t *endSector,* uint32_t *systemCoreClock*)

This function erases sector(s). The end sector must be greater than or equal to start sector number. FLASHIAP_PrepareSectorForWrite must be called before calling this function.

Parameters

startSector	Start sector number.
endSector	End sector number.
systemCoreClock	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

kStatus_FLASHIAP_Success	Api was executed successfully.

kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_InvalidSector	Sector number is invalid or end sector number is greater than start sector number.
kStatus_FLASHIAP_NotPrepared	Command to prepare sector for write operation was not executed.
kStatus_FLASHIAP_Busy	Flash programming hardware interface is busy.

15.6.5 status_t FLASHIAP_ErasePage (uint32_t *startPage*, uint32_t *endPage*, uint32_t *systemCoreClock*)

The end page must be greater than or equal to start page number. Corresponding sectors must be prepared via FLASHIAP_PrepareSectorForWrite before calling calling this function.

Parameters

startPage	Start page number
endPage	End page number
systemCoreClock	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

kStatus_FLASHIAP_Success	Api was executed successfully.
kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_InvalidSector	Page number is invalid or end page number is greater than start page number

kStatus_FLASHIAP_NotPrepared	Command to prepare sector for write operation was not executed.
kStatus_FLASHIAP_Busy	Flash programming hardware interface is busy.

15.6.6 status_t FLASHIAP_BlankCheckSector (uint32_t *startSector*, uint32_t *endSector*)

Blank check single or multiples sectors of flash memory. The end sector must be greater than or equal to start sector number. It can be used to verify the sector eraseure after FLASHIAP_EraseSector call.

Parameters

startSector	: Start sector number. Must be greater than or equal to start sector number
endSector	: End sector number

Return values

kStatus_FLASHIAP_Success	One or more sectors are in erased state.
kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_SectorNotblank	One or more sectors are not blank.

15.6.7 status_t FLASHIAP_Compare (uint32_t *dstAddr*, uint32_t * *srcAddr*, uint32_t *numOfBytes*)

This function compares the contents of flash and ram. It can be used to verify the flash memory contents after FLASHIAP_CopyRamToFlash call.

Parameters

dstAddr	Destination flash address.

srcAddr	Source ram address.
numOfBytes	Number of bytes to be compared.

Return values

kStatus_FLASHIAP_Success	Contents of flash and ram match.
kStatus_FLASHIAP_NoPower	Flash memory block is powered down.
kStatus_FLASHIAP_NoClock	Flash memory block or controller is not clocked.
kStatus_FLASHIAP_AddrError	Address is not on word boundary.
kStatus_FLASHIAP_AddrNotMapped	Address is not mapped in the memory map.
kStatus_FLASHIAP_CountError	Byte count is not multiple of 4 or is not a permitted value.
kStatus_FLASHIAP_CompareError	Destination and source memory contents do not match.

Chapter 16
I2C: Inter-Integrated Circuit Driver

16.1 Overview

The SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of LPC devices.

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.

16.2 Typical use case

16.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* 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);
```

16.2.2 Master Operation in interrupt transactional method

```
i2c_master_handle_t g_m_handle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_handle_t *handle,
      status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* 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;
```

16.2.3 Master Operation in DMA transactional method

```
i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle,
      status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* 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;

DMA_EnableChannel(EXAMPLE_DMA, EXAMPLE_I2C_MASTER_CHANNEL);
DMA_CreateHandle(&dmaHandle, EXAMPLE_DMA, EXAMPLE_I2C_MASTER_CHANNEL);

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;
```

16.2.4 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))
{

}
```

Typical use case

```
/* 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;
```

16.2.5 Slave Operation in interrupt transactional method

```
i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
      userData)
{
    switch (xfer->event)
    {
        /*  Transmit request */
        case kI2C_SlaveTransmitEvent:
            /*  Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /*  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;
```

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C Master Driver
- I2C Slave Driver

16.3 I2C Driver

16.3.1 Overview

Files

- file fsl_i2c.h

Macros

- #define I2C_STAT_MSTCODE_IDLE (0)
  Master Idle State Code.
- #define I2C_STAT_MSTCODE_RXREADY (1)
  Master Receive Ready State Code.
- #define I2C_STAT_MSTCODE_TXREADY (2)
  Master Transmit Ready State Code.
- #define I2C_STAT_MSTCODE_NACKADR (3)
  Master NACK by slave on address State Code.
- #define I2C_STAT_MSTCODE_NACKDAT (4)
  Master NACK by slave on data State Code.

Enumerations

- enum _i2c_status {
  kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 0),
  kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 1),
  kStatus_I2C_Nak,
  kStatus_I2C_InvalidParameter,
  kStatus_I2C_BitError = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 4),
  kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 5),
  kStatus_I2C_NoTransferInProgress,
  kStatus_I2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 7) }
  I2C status return codes.

Driver version

- #define NXP_I2C_DRIVER_VERSION (MAKE_VERSION(1, 0, 0))
  I2C driver version 1.0.0.

16.3.2 Macro Definition Documentation

16.3.2.1 #define NXP_I2C_DRIVER_VERSION (MAKE_VERSION(1, 0, 0))

16.3.3 Enumeration Type Documentation

16.3.3.1 enum _i2c_status

Enumerator

kStatus_I2C_Busy The master is already performing a transfer.
kStatus_I2C_Idle The slave driver is idle.
kStatus_I2C_Nak The slave device sent a NAK in response to a byte.
kStatus_I2C_InvalidParameter Unable to proceed due to invalid parameter.
kStatus_I2C_BitError Transferred bit was not seen on the bus.
kStatus_I2C_ArbitrationLost Arbitration lost error.
kStatus_I2C_NoTransferInProgress Attempt to abort a transfer when one is not in progress.
kStatus_I2C_DmaRequestFail DMA request failed.

16.4 I2C Master Driver

16.4.1 Overview

Data Structures

- struct i2c_master_config_t
  Structure with settings to initialize the I2C master module. More...
- struct i2c_master_transfer_t
  Non-blocking transfer descriptor structure. More...
- struct i2c_master_handle_t
  Driver handle for master non-blocking APIs. More...

Typedefs

- typedef void(∗ i2c_master_transfer_callback_t)(I2C_Type ∗base, i2c_master_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Master completion callback function pointer type.

Enumerations

- enum _i2c_master_flags {
  kI2C_MasterPendingFlag = I2C_STAT_MSTPENDING_MASK,
  kI2C_MasterArbitrationLostFlag = I2C_STAT_MSTARBLOSS_MASK,
  kI2C_MasterStartStopErrorFlag = I2C_STAT_MSTSTSTPERR_MASK }
  I2C master peripheral flags.
- enum i2c_direction_t {
  kI2C_Write = 0U,
  kI2C_Read = 1U }
  Direction of master and slave transfers.
- enum _i2c_master_transfer_flags {
  kI2C_TransferDefaultFlag = 0x00U,
  kI2C_TransferNoStartFlag = 0x01U,
  kI2C_TransferRepeatedStartFlag = 0x02U,
  kI2C_TransferNoStopFlag = 0x04U }
  Transfer option flags.
- enum _i2c_transfer_states
  States for the state machine used by transactional APIs.

Initialization and deinitialization

- void I2C_MasterGetDefaultConfig (i2c_master_config_t ∗masterConfig)
  Provides a default configuration for the I2C master peripheral.
- void I2C_MasterInit (I2C_Type ∗base, const i2c_master_config_t ∗masterConfig, uint32_t srcClock_Hz)

Initializes the I2C master peripheral.
- void I2C_MasterDeinit (I2C_Type ∗base)
  Deinitializes the I2C master peripheral.
- static void I2C_MasterReset (I2C_Type ∗base)
  Performs a software reset.
- static void I2C_MasterEnable (I2C_Type ∗base, bool enable)
  Enables or disables the I2C module as master.

Status

- static uint32_t I2C_GetStatusFlags (I2C_Type ∗base)
  Gets the I2C status flags.
- static void I2C_MasterClearStatusFlags (I2C_Type ∗base, uint32_t statusMask)
  Clears the I2C master status flag state.

Interrupts

- static void I2C_EnableInterrupts (I2C_Type ∗base, uint32_t interruptMask)
  Enables the I2C master interrupt requests.
- static void I2C_DisableInterrupts (I2C_Type ∗base, uint32_t interruptMask)
  Disables the I2C master interrupt requests.
- static uint32_t I2C_GetEnabledInterrupts (I2C_Type ∗base)
  Returns the set of currently enabled I2C master interrupt requests.

Bus operations

- void I2C_MasterSetBaudRate (I2C_Type ∗base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
  Sets the I2C bus frequency for master transactions.
- static bool I2C_MasterGetBusIdleState (I2C_Type ∗base)
  Returns whether the bus is idle.
- 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.
- static 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 void ∗txBuff, size_t txSize, uint32_t flags)
  Performs a polling send transfer on the I2C bus.
- status_t I2C_MasterReadBlocking (I2C_Type ∗base, void ∗rxBuff, size_t rxSize, uint32_t flags)
  Performs a polling receive transfer 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.

Non-blocking

- void I2C_MasterTransferCreateHandle (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_callback_t callback, void *userData)
  Creates a new handle for the I2C master non-blocking APIs.
- status_t I2C_MasterTransferNonBlocking (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_t *xfer)
  Performs a non-blocking transaction on the I2C bus.
- status_t I2C_MasterTransferGetCount (I2C_Type *base, i2c_master_handle_t *handle, size_t *count)
  Returns number of bytes transferred so far.
- void I2C_MasterTransferAbort (I2C_Type *base, i2c_master_handle_t *handle)
  Terminates a non-blocking I2C master transmission early.

IRQ handler

- void I2C_MasterTransferHandleIRQ (I2C_Type *base, i2c_master_handle_t *handle)
  Reusable routine to handle master interrupts.

16.4.2 Data Structure Documentation

16.4.2.1 struct i2c_master_config_t

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_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.
- uint32_t baudRate_Bps
  Desired baud rate in bits per second.
- bool enableTimeout
  Enable internal timeout function.

16.4.2.1.0.21 Field Documentation

16.4.2.1.0.21.1 bool i2c_master_config_t::enableMaster

16.4.2.1.0.21.2 uint32_t i2c_master_config_t::baudRate_Bps

16.4.2.1.0.21.3 bool i2c_master_config_t::enableTimeout

16.4.2.2 struct _i2c_master_transfer

I2C master transfer typedef.

This structure is used to pass transaction parameters to the I2C_MasterTransferNonBlocking() API.

Data Fields

- uint32_t flags
  Bit mask of options for the transfer.
- uint16_t slaveAddress
  The 7-bit slave address.
- i2c_direction_t direction
  Either kI2C_Read or kI2C_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.

16.4.2.2.0.22 Field Documentation

16.4.2.2.0.22.1 uint32_t i2c_master_transfer_t::flags

See enumeration _i2c_master_transfer_flags for available options. Set to 0 or kI2C_TransferDefaultFlag for normal transfers.

16.4.2.2.0.22.2 uint16_t i2c_master_transfer_t::slaveAddress

16.4.2.2.0.22.3 i2c_direction_t i2c_master_transfer_t::direction

16.4.2.2.0.22.4 uint32_t i2c_master_transfer_t::subaddress

Transferred MSB first.

16.4.2.2.0.22.5 size_t i2c_master_transfer_t::subaddressSize

Maximum size is 4 bytes.

16.4.2.2.0.22.6 void* i2c_master_transfer_t::data

16.4.2.2.0.22.7 size_t i2c_master_transfer_t::dataSize

16.4.2.3 struct _i2c_master_handle

I2C master handle typedef.

Note

The contents of this structure are private and subject to change.

Data Fields

- uint8_t state
  Transfer state machine current state.
- uint32_t transferCount
  Indicates progress of the transfer.
- uint32_t remainingBytes
  Remaining byte count in current state.
- uint8_t * buf
  Buffer pointer for current state.
- i2c_master_transfer_t transfer
  Copy of the current transfer info.
- i2c_master_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.

16.4.2.3.0.23 Field Documentation

16.4.2.3.0.23.1 uint8_t i2c_master_handle_t::state

16.4.2.3.0.23.2 uint32_t i2c_master_handle_t::remainingBytes

16.4.2.3.0.23.3 uint8_t* i2c_master_handle_t::buf

16.4.2.3.0.23.4 i2c_master_transfer_t i2c_master_handle_t::transfer

16.4.2.3.0.23.5 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

16.4.2.3.0.23.6 void* i2c_master_handle_t::userData

16.4.3 Typedef Documentation

16.4.3.1 typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_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 I2C_MasterTransferCreateHandle().

Parameters

base	The I2C 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.

16.4.4 Enumeration Type Documentation

16.4.4.1 enum _i2c_master_flags

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kI2C_MasterPendingFlag The I2C module is waiting for software interaction.
kI2C_MasterArbitrationLostFlag The arbitration of the bus was lost. There was collision on the bus
kI2C_MasterStartStopErrorFlag There was an error during start or stop phase of the transaction.

16.4.4.2 enum i2c_direction_t

Enumerator

kI2C_Write Master transmit.
kI2C_Read Master receive.

16.4.4.3 enum _i2c_master_transfer_flags

Note

These enumerations are intended to be OR'd together to form a bit mask of options for the _i2c_-master_transfer::flags field.

Enumerator

kI2C_TransferDefaultFlag Transfer starts with a start signal, stops with a stop signal.
kI2C_TransferNoStartFlag Don't send a start condition, address, and sub address.
kI2C_TransferRepeatedStartFlag Send a repeated start condition.
kI2C_TransferNoStopFlag Don't send a stop condition.

16.4.4.4 enum _i2c_transfer_states

16.4.5 Function Documentation

16.4.5.1 void I2C_MasterGetDefaultConfig (i2c_master_config_t * *masterConfig*)

This function provides the following default configuration for the I2C master peripheral:

```
*  masterConfig->enableMaster            = true;
*  masterConfig->baudRate_Bps            = 100000U;
*  masterConfig->enableTimeout           = false;
*
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with I2C_MasterInit().

Parameters

out	*masterConfig*	User provided configuration structure for default values. Refer to i2c_master_config_t.

16.4.5.2 void I2C_MasterInit (I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

base	The I2C peripheral base address.
masterConfig	User provided peripheral configuration. Use I2C_MasterGetDefaultConfig() to get a set of defaults that you can override.
srcClock_Hz	Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

16.4.5.3 void I2C_MasterDeinit (I2C_Type * *base*)

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

16.4.5.4 static void I2C_MasterReset (I2C_Type * *base*) [inline], [static]

Restores the I2C master peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

16.4.5.5 static void I2C_MasterEnable (I2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	The I2C peripheral base address.
enable	Pass true to enable or false to disable the specified I2C as master.

16.4.5.6 static uint32_t I2C_GetStatusFlags (I2C_Type * *base*) [inline], [static]

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The I2C peripheral base address.

Returns

State of the status flags:
- 1: related status flag is set.
- 0: related status flag is not set.

See Also

_i2c_master_flags

16.4.5.7 static void I2C_MasterClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- kI2C_MasterArbitrationLostFlag
- kI2C_MasterStartStopErrorFlag

Attempts to clear other flags has no effect.

Parameters

base	The I2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _i2c_master_flags enumerators OR'd together. You may pass the result of a previous call to I2C_GetStatusFlags().

See Also

_i2c_master_flags.

16.4.5.8 static void I2C_EnableInterrupts (I2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

Parameters

base	The I2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _i2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

16.4.5.9 static void I2C_DisableInterrupts (I2C_Type * *base*, uint32_t *interruptMask*) [inline], [static]

Parameters

base	The I2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _i2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

16.4.5.10 static uint32_t I2C_GetEnabledInterrupts (I2C_Type * *base*) [inline], [static]

Parameters

base	The I2C peripheral base address.

Returns

A bitmask composed of _i2c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

16.4.5.11 void I2C_MasterSetBaudRate (I2C_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

The I2C 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 I2C peripheral base address.
srcClock_Hz	I2C functional clock frequency in Hertz.
baudRate_Bps	Requested bus frequency in bits per second.

16.4.5.12 static bool I2C_MasterGetBusIdleState (I2C_Type * *base*) [inline], [static]

Requires the master mode to be enabled.

Parameters

base	The I2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

16.4.5.13 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.

16.4.5.14 status_t I2C_MasterStop (I2C_Type * *base*)

Return values

kStatus_Success	Successfully send the stop signal.
kStatus_I2C_Timeout	Send stop signal failed, timeout.

16.4.5.15 static status_t I2C_MasterRepeatedStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*) [inline], [static]

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.

16.4.5.16 status_t I2C_MasterWriteBlocking (I2C_Type * *base*, const void * *txBuff*, size_t *txSize*, uint32_t *flags*)

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_I2C_Nak.

Parameters

base	The I2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.
flags	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

kStatus_Success	Data was sent successfully.
kStatus_I2C_Busy	Another master is currently utilizing the bus.
kStatus_I2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_I2C_Arbitration-Lost	Arbitration lost error.

16.4.5.17 status_t I2C_MasterReadBlocking (I2C_Type * *base*, void * *rxBuff*, size_t *rxSize*, uint32_t *flags*)

Parameters

base	The I2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.
flags	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

kStatus_Success	Data was received successfully.
kStatus_I2C_Busy	Another master is currently utilizing the bus.
kStatus_I2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_I2C_Arbitration-Lost	Arbitration lost error.

16.4.5.18 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.

16.4.5.19 void I2C_MasterTransferCreateHandle (I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_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 I2C_MasterTransferAbort() API shall be called.

Parameters

	base	The I2C peripheral base address.
out	*handle*	Pointer to the I2C master driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

16.4.5.20 status_t I2C_MasterTransferNonBlocking (I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_t * *xfer*)

Parameters

base	The I2C peripheral base address.
handle	Pointer to the I2C master driver handle.
xfer	The pointer to the transfer descriptor.

Return values

kStatus_Success	The transaction was started successfully.
kStatus_I2C_Busy	Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

16.4.5.21 status_t I2C_MasterTransferGetCount (I2C_Type * *base*, i2c_master_handle_t * *handle*, size_t * *count*)

Parameters

	base	The I2C peripheral base address.
	handle	Pointer to the I2C master driver handle.
out	*count*	Number of bytes transferred so far by the non-blocking transaction.

Return values

kStatus_Success	
kStatus_I2C_Busy	

16.4.5.22 void I2C_MasterTransferAbort (I2C_Type * *base*, i2c_master_handle_t * *handle*)

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

Parameters

base	The I2C peripheral base address.
handle	Pointer to the I2C master driver handle.

Return values

kStatus_Success	A transaction was successfully aborted.
kStatus_I2C_Idle	There is not a non-blocking transaction currently in progress.

16.4.5.23 void I2C_MasterTransferHandleIRQ (I2C_Type * *base*, i2c_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 I2C peripheral base address.
handle	Pointer to the I2C master driver handle.

16.5 I2C Slave Driver

16.5.1 Overview

Data Structures

- struct i2c_slave_address_t
  Data structure with 7-bit Slave address and Slave address disable. More...
- struct i2c_slave_config_t
  Structure with settings to initialize the I2C slave module. 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_slave_transfer_callback_t)(I2C_Type ∗base, volatile i2c_slave_transfer_t ∗transfer, void ∗userData)
  Slave event callback function pointer type.

Enumerations

- enum _i2c_slave_flags {
  kI2C_SlavePendingFlag = I2C_STAT_SLVPENDING_MASK,
  kI2C_SlaveNotStretching = I2C_STAT_SLVNOTSTR_MASK,
  kI2C_SlaveSelected = I2C_STAT_SLVSEL_MASK,
  kI2C_SaveDeselected = I2C_STAT_SLVDESEL_MASK }
  I2C slave peripheral flags.
- enum i2c_slave_address_register_t {
  kI2C_SlaveAddressRegister0 = 0U,
  kI2C_SlaveAddressRegister1 = 1U,
  kI2C_SlaveAddressRegister2 = 2U,
  kI2C_SlaveAddressRegister3 = 3U }
  I2C slave address register.
- enum i2c_slave_address_qual_mode_t {
  kI2C_QualModeMask = 0U,
  kI2C_QualModeExtend }
  I2C slave address match options.
- enum i2c_slave_bus_speed_t
  I2C slave bus speed options.
- enum i2c_slave_transfer_event_t {

kI2C_SlaveAddressMatchEvent = 0x01U,
kI2C_SlaveTransmitEvent = 0x02U,
kI2C_SlaveReceiveEvent = 0x04U,
kI2C_SlaveCompletionEvent = 0x20U,
kI2C_SlaveDeselectedEvent,
kI2C_SlaveAllEvents }
Set of events sent to the callback for non blocking slave transfers.

- enum i2c_slave_fsm_t
  I2C slave software finite state machine states.

Slave initialization and deinitialization

- void I2C_SlaveGetDefaultConfig (i2c_slave_config_t *slaveConfig)
  Provides a default configuration for the I2C slave peripheral.
- status_t I2C_SlaveInit (I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t srcClock_Hz)
  Initializes the I2C slave peripheral.
- void I2C_SlaveSetAddress (I2C_Type *base, i2c_slave_address_register_t addressRegister, uint8_t address, bool addressDisable)
  Configures Slave Address n register.
- void I2C_SlaveDeinit (I2C_Type *base)
  Deinitializes the I2C slave peripheral.
- static void I2C_SlaveEnable (I2C_Type *base, bool enable)
  Enables or disables the I2C module as slave.

Slave status

- static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)
  Clears the I2C status flag state.

Slave bus operations

- status_t I2C_SlaveWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
  Performs a polling send transfer on the I2C bus.
- status_t I2C_SlaveReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
  Performs a polling receive transfer on the I2C bus.

Slave non-blocking

- void I2C_SlaveTransferCreateHandle (I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)
  Creates a new handle for the I2C slave non-blocking APIs.
- status_t I2C_SlaveTransferNonBlocking (I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)

Starts accepting slave transfers.

- status_t I2C_SlaveSetSendBuffer (I2C_Type *base, volatile i2c_slave_transfer_t *transfer, const void *txData, size_t txSize, uint32_t eventMask)
  Starts accepting master read from slave requests.
- status_t I2C_SlaveSetReceiveBuffer (I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *rxData, size_t rxSize, uint32_t eventMask)
  Starts accepting master write to slave requests.
- static uint32_t I2C_SlaveGetReceivedAddress (I2C_Type *base, volatile i2c_slave_transfer_t *transfer)
  Returns the slave address sent by the I2C master.
- void I2C_SlaveTransferAbort (I2C_Type *base, i2c_slave_handle_t *handle)
  Aborts the slave non-blocking transfers.
- 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.

Slave IRQ handler

- void I2C_SlaveTransferHandleIRQ (I2C_Type *base, i2c_slave_handle_t *handle)
  Reusable routine to handle slave interrupts.

16.5.2 Data Structure Documentation

16.5.2.1 struct i2c_slave_address_t

Data Fields

- uint8_t address
  7-bit Slave address SLVADR.
- bool addressDisable
  Slave address disable SADISABLE.

16.5.2.1.0.24 Field Documentation

16.5.2.1.0.24.1 uint8_t i2c_slave_address_t::address

16.5.2.1.0.24.2 bool i2c_slave_address_t::addressDisable

16.5.2.2 struct i2c_slave_config_t

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_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

- i2c_slave_address_t address0
  Slave's 7-bit address and disable.
- i2c_slave_address_t address1
  Alternate slave 7-bit address and disable.
- i2c_slave_address_t address2
  Alternate slave 7-bit address and disable.
- i2c_slave_address_t address3
  Alternate slave 7-bit address and disable.
- i2c_slave_address_qual_mode_t qualMode
  Qualify mode for slave address 0.
- uint8_t qualAddress
  Slave address qualifier for address 0.
- i2c_slave_bus_speed_t busSpeed
  Slave bus speed mode.
- bool enableSlave
  Enable slave mode.

16.5.2.2.0.25 Field Documentation

16.5.2.2.0.25.1 i2c_slave_address_t i2c_slave_config_t::address0

16.5.2.2.0.25.2 i2c_slave_address_t i2c_slave_config_t::address1

16.5.2.2.0.25.3 i2c_slave_address_t i2c_slave_config_t::address2

16.5.2.2.0.25.4 i2c_slave_address_t i2c_slave_config_t::address3

16.5.2.2.0.25.5 i2c_slave_address_qual_mode_t i2c_slave_config_t::qualMode

16.5.2.2.0.25.6 uint8_t i2c_slave_config_t::qualAddress

16.5.2.2.0.25.7 i2c_slave_bus_speed_t i2c_slave_config_t::busSpeed

If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The busSpeed value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the busSpeed mode is unknown at compile time, use the longest data setup time kI2C_SlaveStandardMode (250 ns)

16.5.2.2.0.25.8 bool i2c_slave_config_t::enableSlave

16.5.2.3 struct i2c_slave_transfer_t

Data Fields

- i2c_slave_handle_t * handle
  Pointer to handle that contains this transfer.
- i2c_slave_transfer_event_t event

Reason the callback is being invoked.
- uint8_t receivedAddress
  Matching address send by master.
- uint32_t eventMask
  Mask of enabled events.
- uint8_t * rxData
  Transfer buffer for receive data.
- const uint8_t * txData
  Transfer buffer for transmit data.
- size_t txSize
  Transfer size.
- size_t rxSize
  Transfer size.
- size_t transferredCount
  Number of bytes transferred during this transfer.
- status_t completionStatus
  Success or error code describing how the transfer completed.

16.5.2.3.0.26 Field Documentation

16.5.2.3.0.26.1 i2c_slave_handle_t* i2c_slave_transfer_t::handle

16.5.2.3.0.26.2 i2c_slave_transfer_event_t i2c_slave_transfer_t::event

16.5.2.3.0.26.3 uint8_t i2c_slave_transfer_t::receivedAddress

7-bits plus R/nW bit0

16.5.2.3.0.26.4 uint32_t i2c_slave_transfer_t::eventMask

16.5.2.3.0.26.5 size_t i2c_slave_transfer_t::transferredCount

16.5.2.3.0.26.6 status_t i2c_slave_transfer_t::completionStatus

Only applies for kI2C_SlaveCompletionEvent.

16.5.2.4 struct _i2c_slave_handle

I2C slave handle typedef.

Note

The contents of this structure are private and subject to change.

Data Fields

- volatile i2c_slave_transfer_t transfer
  I2C slave transfer.
- volatile bool isBusy

Whether transfer is busy.

- volatile i2c_slave_fsm_t slaveFsm
  slave transfer state machine.
- i2c_slave_transfer_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback.

16.5.2.4.0.27 Field Documentation

16.5.2.4.0.27.1 volatile i2c_slave_transfer_t i2c_slave_handle_t::transfer

16.5.2.4.0.27.2 volatile bool i2c_slave_handle_t::isBusy

16.5.2.4.0.27.3 volatile i2c_slave_fsm_t i2c_slave_handle_t::slaveFsm

16.5.2.4.0.27.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

16.5.2.4.0.27.5 void* i2c_slave_handle_t::userData

16.5.3 Typedef Documentation

16.5.3.1 typedef void(* i2c_slave_transfer_callback_t)(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *userData)

This callback is used only for the slave non-blocking transfer API. To install a callback, use the I2C_SlaveSetCallback() function after you have created a handle.

Parameters

base	Base address for the I2C 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.

16.5.4 Enumeration Type Documentation

16.5.4.1 enum _i2c_slave_flags

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kI2C_SlavePendingFlag The I2C module is waiting for software interaction.
kI2C_SlaveNotStretching Indicates whether the slave is currently stretching clock (0 = yes, 1 = no).

kI2C_SlaveSelected Indicates whether the slave is selected by an address match.
kI2C_SaveDeselected Indicates that slave was previously deselected (deselect event took place, w1c).

16.5.4.2 enum i2c_slave_address_register_t

Enumerator

kI2C_SlaveAddressRegister0 Slave Address 0 register.
kI2C_SlaveAddressRegister1 Slave Address 1 register.
kI2C_SlaveAddressRegister2 Slave Address 2 register.
kI2C_SlaveAddressRegister3 Slave Address 3 register.

16.5.4.3 enum i2c_slave_address_qual_mode_t

Enumerator

kI2C_QualModeMask The SLVQUAL0 field (qualAddress) is used as a logical mask for matching address0.
kI2C_QualModeExtend The SLVQUAL0 (qualAddress) field is used to extend address 0 matching in a range of addresses.

16.5.4.4 enum i2c_slave_bus_speed_t

16.5.4.5 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_SlaveCompletionEvent All data in the active transfer have been consumed.
kI2C_SlaveDeselectedEvent The slave function has become deselected (SLVSEL flag changing from 1 to 0.
kI2C_SlaveAllEvents Bit mask of all available events.

16.5.5 Function Documentation

16.5.5.1 void I2C_SlaveGetDefaultConfig (i2c_slave_config_t * *slaveConfig*)

This function provides the following default configuration for the I2C slave peripheral:

```
*  slaveConfig->enableSlave = true;
*  slaveConfig->address0.disable = false;
*  slaveConfig->address0.address = 0u;
*  slaveConfig->address1.disable = true;
*  slaveConfig->address2.disable = true;
*  slaveConfig->address3.disable = true;
*  slaveConfig->busSpeed = kI2C_SlaveStandardMode;
*
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with I2C_SlaveInit(). Be sure to override at least the *address0.address* 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 i2c_slave_config_t.

16.5.5.2 status_t I2C_SlaveInit (I2C_Type * *base*, const i2c_slave_config_t * *slaveConfig*, uint32_t *srcClock_Hz*)

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

base	The I2C peripheral base address.
slaveConfig	User provided peripheral configuration. Use I2C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
srcClock_Hz	Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

16.5.5.3 void I2C_SlaveSetAddress (I2C_Type * *base*, i2c_slave_address_register_t *addressRegister*, uint8_t *address*, bool *addressDisable*)

This function writes new value to Slave Address register.

Parameters

base	The I2C peripheral base address.
address-Register	The module supports multiple address registers. The parameter determines which one shall be changed.
address	The slave address to be stored to the address register for matching.
addressDisable	Disable matching of the specified address register.

16.5.5.4 void I2C_SlaveDeinit (I2C_Type * *base*)

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The I2C peripheral base address.

16.5.5.5 static void I2C_SlaveEnable (I2C_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	The I2C peripheral base address.
enable	True to enable or flase to disable.

16.5.5.6 static void I2C_SlaveClearStatusFlags (I2C_Type * *base,* uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

Parameters

base	The I2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _i2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to I2C_SlaveGetStatusFlags().

See Also

_i2c_slave_flags.

16.5.5.7 status_t I2C_SlaveWriteBlocking (I2C_Type * *base*, const uint8_t * *txBuff*, size_t *txSize*)

The function executes blocking address phase and blocking data phase.

Parameters

base	The I2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been sent.
kStatus_Fail Unexpected slave state (master data write while master read from slave is expected).

16.5.5.8 status_t I2C_SlaveReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*)

The function executes blocking address phase and blocking data phase.

Parameters

base	The I2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.

Returns

kStatus_Success Data has been received.
kStatus_Fail Unexpected slave state (master data read while master write to slave is expected).

16.5.5.9 void I2C_SlaveTransferCreateHandle (I2C_Type * *base*, i2c_slave_handle_t * *handle*, i2c_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 I2C_SlaveTransferAbort() API shall be called.

Parameters

	base	The I2C peripheral base address.
out	*handle*	Pointer to the I2C slave driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

16.5.5.10 status_t I2C_SlaveTransferNonBlocking (I2C_Type * *base*, i2c_slave_handle_t * *handle*, uint32_t *eventMask*)

Call this API after calling I2C_SlaveInit() and I2C_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 I2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes kI2C_SlaveTransmitEvent callback. If no slave Rx transfer is busy, a master write to slave request invokes kI2C_SlaveReceiveEvent callback.

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 kI2C_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 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.

16.5.5.11 status_t I2C_SlaveSetSendBuffer (I2C_Type * *base*, volatile i2c_slave_transfer_t * *transfer*, const void * *txData*, size_t *txSize*, uint32_t *eventMask*)

The function can be called in response to kI2C_SlaveTransmitEvent callback to start a new slave Tx transfer from within the transfer callback.

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 kI2C_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 kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
transfer	Pointer to i2c_slave_transfer_t structure.
txData	Pointer to data to send to master.
txSize	Size of txData in bytes.
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.

16.5.5.12 status_t I2C_SlaveSetReceiveBuffer (I2C_Type * *base*, volatile i2c_slave_transfer_t * *transfer*, void * *rxData*, size_t *rxSize*, uint32_t *eventMask*)

The function can be called in response to kI2C_SlaveReceiveEvent callback to start a new slave Rx transfer from within the transfer callback.

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 kI2C_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 kI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The I2C peripheral base address.
transfer	Pointer to i2c_slave_transfer_t structure.
rxData	Pointer to data to store data from master.
rxSize	Size of rxData in bytes.
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.

16.5.5.13 static uint32_t I2C_SlaveGetReceivedAddress (I2C_Type * *base*, volatile i2c_slave_transfer_t * *transfer*) [inline], [static]

This function should only be called from the address match event callback kI2C_SlaveAddressMatchEvent.

Parameters

base	The I2C peripheral base address.
transfer	The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

16.5.5.14 void I2C_SlaveTransferAbort (I2C_Type * *base*, i2c_slave_handle_t * *handle*)

Note

This API could be called at any time to stop slave for handling the bus events.

Parameters

base	The I2C peripheral base address.
handle	Pointer to i2c_slave_handle_t structure which stores the transfer state.

Return values

kStatus_Success	
kStatus_I2C_Idle	

16.5.5.15 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.

16.5.5.16 void I2C_SlaveTransferHandleIRQ (I2C_Type * *base*, i2c_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 I2C peripheral base address.
handle	Pointer to i2c_slave_handle_t structure which stores the transfer state.

16.6 I2C DMA Driver

16.6.1 Overview

Data Structures

- struct i2c_master_dma_handle_t
  I2C master dma transfer structure. More...

Macros

- #define I2C_MAX_DMA_TRANSFER_COUNT 1024
  Maximum lenght of single DMA transfer (determined by capability of the DMA engine)

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.

16.6.2 Data Structure Documentation

16.6.2.1 struct _i2c_master_dma_handle

I2C master dma handle typedef.

Data Fields

- uint8_t state

Transfer state machine current state.

- uint32_t transferCount
  Indicates progress of the transfer.
- uint32_t remainingBytesDMA
  Remaining byte count to be transferred using DMA.
- uint8_t * buf
  Buffer pointer for current state.
- dma_handle_t * dmaHandle
  The DMA handler used.
- i2c_master_transfer_t transfer
  Copy of the current transfer info.
- i2c_master_dma_transfer_callback_t completionCallback
  Callback function called after dma transfer finished.
- void * userData
  Callback parameter passed to callback function.

16.6.2.1.0.28 Field Documentation

16.6.2.1.0.28.1 uint8_t i2c_master_dma_handle_t::state

16.6.2.1.0.28.2 uint32_t i2c_master_dma_handle_t::remainingBytesDMA

16.6.2.1.0.28.3 uint8_t* i2c_master_dma_handle_t::buf

16.6.2.1.0.28.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

16.6.2.1.0.28.5 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

16.6.2.1.0.28.6 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

16.6.2.1.0.28.7 void* i2c_master_dma_handle_t::userData

16.6.3 Typedef Documentation

16.6.3.1 typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)

16.6.4 Function Documentation

16.6.4.1 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

16.6.4.2 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.

16.6.4.3 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.

16.6.4.4 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

16.7 I2C FreeRTOS Driver

16.7.1 Overview

Data Structures

- struct i2c_rtos_handle_t
  I2C FreeRTOS handle. More...

I2C RTOS Operation

- status_t I2C_RTOS_Init (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
  Initializes I2C.
- status_t I2C_RTOS_Deinit (i2c_rtos_handle_t *handle)
  Deinitializes the I2C.
- status_t I2C_RTOS_Transfer (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
  Performs I2C transfer.

16.7.2 Data Structure Documentation

16.7.2.1 struct i2c_rtos_handle_t

Data Fields

- I2C_Type * base
  I2C base address.
- i2c_master_handle_t drv_handle
  A handle of the underlying driver, treated as opaque by the RTOS layer.
- status_t async_status
  Transactional state of the underlying driver.
- SemaphoreHandle_t mutex
  A mutex to lock the handle during a transfer.
- SemaphoreHandle_t semaphore
  A semaphore to notify and unblock task when the transfer ends.

16.7.3 Function Documentation

16.7.3.1 status_t I2C_RTOS_Init (i2c_rtos_handle_t * *handle*, I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

This function initializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle, the pointer to an allocated space for RTOS context.
base	The pointer base address of the I2C instance to initialize.
masterConfig	Configuration structure to set-up I2C in master mode.
srcClock_Hz	Frequency of input clock of the I2C module.

Returns

status of the operation.

16.7.3.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and the related RTOS context.

Parameters

handle	The RTOS I2C handle.

16.7.3.3 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.

Chapter 17
I2S: I2S Driver

17.1 Overview

The SDK provides the peripheral driver for the I2S function of FLEXCOMM module of LPC devices.

The I2S module is used to transmit or receive digital audio data. Only transmit or receive is enabled at one time in one module.

Driver currently supports one (primary) channel pair per one I2S enabled FLEXCOMM module only.

17.2 I2S Driver Initialization and Configuration

I2S_TxInit() and I2S_RxInit() functions ungate the clock for the FLEXCOMM module, assign I2S function to FLEXCOMM module and configure audio data format and other I2S operational settings. I2S_TxInit() is used when I2S should transmit data, I2S_RxInit() when it should receive data.

I2S_TxGetDefaultConfig() and I2S_RxGetDefaultConfig() functions can be used to set the module configuration structure with default values for transmit and receive function, respectively.

I2S_Deinit() function resets the FLEXCOMM module.

I2S_TxTransferCreateHandle() function creates transactional handle for transmit in interrupt mode.

I2S_RxTransferCreateHandle() function creates transactional handle for receive in interrupt mode.

I2S_TxTransferCreateHandleDMA() function creates transactional handle for transmit in DMA mode.

I2S_RxTransferCreateHandleDMA() function creates transactional handle for receive in DMA mode.

17.3 I2S Transmit Data

I2S_TxTransferNonBlocking() function is used to add data buffer to transmit in interrupt mode. It also begins transmission if not transmitting yet.

I2S_RxTransferNonBlocking() function is used to add data buffer to receive data into in interrupt mode. It also begins reception if not receiving yet.

I2S_TxTransferSendDMA() function is used to add data buffer to transmit in DMA mode. It also begins transmission if not transmitting yet.

I2S_RxTransferReceiveDMA() function is used to add data buffer to receive data into in DMA mode. It also begins reception if not receiving yet.

The transfer of data will be stopped automatically when all data buffers queued using the above functions will be processed and no new data buffer is enqueued meanwhile. If the above functions are not called frequently enough, I2S stop followed by restart may keep occurring resulting in drops audio stream.

17.4 I2S Interrupt related functions

I2S_EnableInterrupts() function is used to enable interrupts in FIFO interrupt register. Regular use cases do not require this function to be called from application code.

I2S_DisableInterrupts() function is used to disable interrupts in FIFO interrupt register. Regular use cases do not require this function to be called from application code.

I2S_GetEnabledInterrupts() function returns interrupts enabled in FIFO interrupt register. Regular use cases do not require this function to be called from application code.

I2S_TxHandleIRQ() and I2S_RxHandleIRQ() functions are called from ISR which is invoked when actual FIFO level decreases to configured watermark value.

I2S_DMACallback() function is called from ISR which is invoked when DMA transfer (actual descriptor) finishes.

17.5 I2S Other functions

I2S_Enable() function enables I2S function in FLEXCOMM module. Regular use cases do not require this function to be called from application code.

I2S_Disable() function disables I2S function in FLEXCOMM module. Regular use cases do not require this function to be called from application code.

I2S_TransferGetErrorCount() function returns the number of FIFO underruns or overruns in interrupt mode.

I2S_TransferGetCount() function returns the number of bytes transferred in interrupt mode.

I2S_TxTransferAbort() function aborts trasmit operation in interrupt mode.

I2S_RxTransferAbort() function aborts receive operation in interrupt mode.

I2S_TransferAbortDMA() function aborts transmit or receive operation in DMA mode.

17.6 I2S Data formats

17.6.1 DMA mode

Length of buffer for transmit or receive has to be multiply of 4 bytes. Buffer address has to be aligned to 4-bytes. Data are put into or taken from FIFO unaltered in DMA mode so buffer has to be prepared according to following information.

If i2s_config_t.dataLength (channel bit width) is between 4 and 16, every word in buffer should contain data for left and right channels.

MSB																												LSB
R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

Rnn - right channel bit nn

Lnn - left channel bit nn

Note that for example if i2s_config_t.dataLength = 7, bits on positions R07-R15 and L07-L15 are ignored (buffer "wastes space").

If i2s_config_t.dataLength (channel bit width) is between 17 and 24 and i2s_config_t.pack48 = false:

Even words (counting from zero):

MSB																											LSB
							L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

Odd words (counting from zero):

MSB																											LSB
							R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00

If i2s_config_t.dataLength (channel bit width) is between 17 and 24 and i2s_config_t.pack48 = true:

Even words (counting from zero):

MSB																											LSB
R07	R06	R05	R04	R03	R02	R01	R00	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

Odd words (counting from zero):

MSB																											LSB
														R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08

If i2s_config_t.dataLength (channel bit width) is between 25 and 32:

Even words (counting from zero):

MSB																											LSB
L31	L30	L29	L28	L27	L26	L25	L24	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

Odd words (counting from zero):

MSB																											LSB
R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00

17.6.2 Interrupt mode

If i2s_config_t.dataLength (channel bit width) is 4:

Buffer does not need to be aligned (buffer is read / written by single bytes, each byte contain left and right channel):

MSB							LSB
R03	R02	R01	R00	L03	L02	L01	L00

If i2s_config_t.dataLength (channel bit width) is between 5 and 8:

Length of buffer for transmit or receive has to be multiply of 2 bytes. Buffer address has to be aligned to 2-bytes.

MSB															LSB
R07	R06	R05	R04	R03	R02	R01	R00	L07	L06	L05	L04	L03	L02	L01	L00

If i2s_config_t.dataLength (channel bit width) is between 9 and 16:

Length of buffer for transmit or receive has to be multiply of 4 bytes. Buffer address has to be aligned to 4-bytes.

MSB																															LSB
R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

If i2s_config_t.dataLength (channel bit width) is between 17 and 24 and i2s_config_t.pack48 = false:

Length of buffer for transmit or receive has to be multiply of 6 bytes.

MSB																																								
R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07

If i2s_config_t.dataLength (channel bit width) is between 17 and 24 and i2s_config_t.pack48 = true:

Length of buffer for transmit or receive has to be multiply of 6 bytes. Buffer address has to be aligned to 4-bytes.

MSB																																								
R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07

If i2s_config_t.dataLength (channel bit width) is between 25 and 32 and i2s_config_t.oneChannel = false:

Buffer for transmit or receive has to be multiply of 8 bytes. Buffer address has to be aligned to 4-bytes.

Even words (counting from zero):

MSB																												LSB
L31	L30	L29	L28	L27	L26	L25	L24	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

Odd words (counting from zero):

MSB																												LSB
R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17	R16	R15	R14	R13	R12	R11	R10	R09	R08	R07	R06	R05	R04	R03	R02	R01	R00

If i2s_config_t.dataLength (channel bit width) is between 25 and 32 and i2s_config_t.oneChannel = true:

Buffer for transmit or receive has to be multiply of 4 bytes. Buffer address has to be aligned to 4-bytes.

MSB																												LSB
L31	L30	L29	L28	L27	L26	L25	L24	L23	L22	L21	L20	L19	L18	L17	L16	L15	L14	L13	L12	L11	L10	L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

17.7 I2S Driver Examples

17.7.1 Interrupt mode examples

Transmit example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_handle_t handle;

    I2S_TxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_TxInit(I2S0, &config);

    I2S_TxTransferCreateHandle(I2S0, &handle, TxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_TxTransferNonBlocking(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_TxTransferNonBlocking(I2S0, &handle, someTransfer);
}

void TxCallback(I2S_Type *base, i2s_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_tranfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
```

```
        transfer.dataSize = sizeof(buffer);
        I2S_TxTransferNonBlocking(base, handle, transfer);
    }
}
```

Receive example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_handle_t handle;

    I2S_RxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_RxInit(I2S0, &config);

    I2S_RxTransferCreateHandle(I2S0, &handle, RxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_RxTransferNonBlocking(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_RxTransferNonBlocking(I2S0, &handle, someTransfer);
}

void RxCallback(I2S_Type *base, i2s_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_tranfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_RxTransferNonBlocking(base, handle, transfer);
    }
}
```

17.7.2 DMA mode examples

Transmit example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_dma_handle_t handle;

    I2S_TxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_TxInit(I2S0, &config);

    I2S_TxTransferCreateHandleDMA(I2S0, &handle, TxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
```

```
    I2S_TxTransferNonBlockingDMA(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_TxTransferNonBlockingDMA(I2S0, &handle, someTransfer);
}

void TxCallback(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData
      )
{
    i2s_tranfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_TxTransferNonBlockingDMA(base, handle, transfer);
    }
}
```

Receive example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_dma_handle_t handle;

    I2S_RxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_RxInit(I2S0, &config);

    I2S_RxTransferCreateHandleDMA(I2S0, &handle, RxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_RxTransferNonBlockingDMA(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_RxTransferNonBlockingDMA(I2S0, &handle, someTransfer);
}

void RxCallback(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData
      )
{
    i2s_tranfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_RxTransferNonBlockingDMA(base, handle, transfer);
    }
}
```

Modules

- I2S DMA Driver
- I2S Driver

17.8 I2S Driver

17.8.1 Overview

Files

- file fsl_i2s.h

Data Structures

- struct i2s_config_t
  I2S configuration structure. *More...*
- struct i2s_transfer_t
  Buffer to transfer from or receive audio data into. *More...*
- struct i2s_handle_t
  Members not to be accessed / modified outside of the driver. *More...*

Macros

- #define I2S_NUM_BUFFERS (4)
  Number of buffers .

Typedefs

- typedef void(∗ i2s_transfer_callback_t)(I2S_Type ∗base, i2s_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Callback function invoked from transactional API on completion of a single buffer transfer.

Enumerations

- enum _i2s_status {
  kStatus_I2S_BufferComplete,
  kStatus_I2S_Done = MAKE_STATUS(kStatusGroup_I2S, 1),
  kStatus_I2S_Busy }
  I2S status codes.
- enum i2s_flags_t {
  kI2S_TxErrorFlag = I2S_FIFOINTENSET_TXERR_MASK,
  kI2S_TxLevelFlag = I2S_FIFOINTENSET_TXLVL_MASK,
  kI2S_RxErrorFlag = I2S_FIFOINTENSET_RXERR_MASK,
  kI2S_RxLevelFlag = I2S_FIFOINTENSET_RXLVL_MASK }
  I2S flags.

- enum i2s_master_slave_t {
  kI2S_MasterSlaveNormalSlave = 0x0,
  kI2S_MasterSlaveWsSyncMaster = 0x1,
  kI2S_MasterSlaveExtSckMaster = 0x2,
  kI2S_MasterSlaveNormalMaster = 0x3 }
  Master / slave mode.
- enum i2s_mode_t {
  kI2S_ModeI2sClassic = 0x0,
  kI2S_ModeDspWs50 = 0x1,
  kI2S_ModeDspWsShort = 0x2,
  kI2S_ModeDspWsLong = 0x3 }
  I2S mode.

Driver version

- #define FSL_I2S_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  I2S driver version 2.0.0.

Initialization and deinitialization

- void I2S_TxInit (I2S_Type *base, const i2s_config_t *config)
  Initializes the FLEXCOMM peripheral for I2S transmit functionality.
- void I2S_RxInit (I2S_Type *base, const i2s_config_t *config)
  Initializes the FLEXCOMM peripheral for I2S receive functionality.
- void I2S_TxGetDefaultConfig (i2s_config_t *config)
  Sets the I2S Tx configuration structure to default values.
- void I2S_RxGetDefaultConfig (i2s_config_t *config)
  Sets the I2S Rx configuration structure to default values.
- void I2S_Deinit (I2S_Type *base)
  De-initializes the I2S peripheral.

Non-blocking API

- void I2S_TxTransferCreateHandle (I2S_Type *base, i2s_handle_t *handle, i2s_transfer_callback_t callback, void *userData)
  Initializes handle for transfer of audio data.
- status_t I2S_TxTransferNonBlocking (I2S_Type *base, i2s_handle_t *handle, i2s_transfer_t transfer)
  Begins or queue sending of the given data.
- void I2S_TxTransferAbort (I2S_Type *base, i2s_handle_t *handle)
  Aborts sending of data.
- void I2S_RxTransferCreateHandle (I2S_Type *base, i2s_handle_t *handle, i2s_transfer_callback_t callback, void *userData)
  Initializes handle for reception of audio data.

- status_t I2S_RxTransferNonBlocking (I2S_Type *base, i2s_handle_t *handle, i2s_transfer_t transfer)
  Begins or queue reception of data into given buffer.
- void I2S_RxTransferAbort (I2S_Type *base, i2s_handle_t *handle)
  Aborts receiving of data.
- status_t I2S_TransferGetCount (I2S_Type *base, i2s_handle_t *handle, size_t *count)
  Returns number of bytes transferred so far.
- status_t I2S_TransferGetErrorCount (I2S_Type *base, i2s_handle_t *handle, size_t *count)
  Returns number of buffer underruns or overruns.

Enable / disable

- static void I2S_Enable (I2S_Type *base)
  Enables I2S operation.
- static void I2S_Disable (I2S_Type *base)
  Disables I2S operation.

Interrupts

- static void I2S_EnableInterrupts (I2S_Type *base, uint32_t interruptMask)
  Enables I2S FIFO interrupts.
- static void I2S_DisableInterrupts (I2S_Type *base, uint32_t interruptMask)
  Disables I2S FIFO interrupts.
- static uint32_t I2S_GetEnabledInterrupts (I2S_Type *base)
  Returns the set of currently enabled I2S FIFO interrupts.
- void I2S_TxHandleIRQ (I2S_Type *base, i2s_handle_t *handle)
  Invoked from interrupt handler when transmit FIFO level decreases.
- void I2S_RxHandleIRQ (I2S_Type *base, i2s_handle_t *handle)
  Invoked from interrupt handler when receive FIFO level decreases.

17.8.2 Data Structure Documentation

17.8.2.1 struct i2s_config_t

Data Fields

- i2s_master_slave_t masterSlave
  Master / slave configuration.
- i2s_mode_t mode
  I2S mode.
- bool rightLow
  Right channel data in low portion of FIFO.
- bool leftJust
  Left justify data in FIFO.
- bool pdmData
  Data source is the D-Mic subsystem.

- bool sckPol
  SCK polarity.
- bool wsPol
  WS polarity.
- uint16_t divider
  Flexcomm function clock divider (1 - 4096)
- bool oneChannel
  true mono, false stereo
- uint8_t dataLength
  Data length (4 - 32)
- uint16_t frameLength
  Frame width (4 - 512)
- uint16_t position
  Data position in the frame.
- uint8_t watermark
  FIFO trigger level.
- bool txEmptyZero
  Transmit zero when buffer becomes empty or last item.
- bool pack48
  Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)

17.8.2.2 struct i2s_transfer_t

Data Fields

- volatile uint8_t * data
  Pointer to data buffer.
- volatile size_t dataSize
  Buffer size in bytes.

17.8.2.2.0.29 Field Documentation

17.8.2.2.0.29.1 volatile uint8_t* i2s_transfer_t::data

17.8.2.2.0.29.2 volatile size_t i2s_transfer_t::dataSize

17.8.2.3 struct _i2s_handle

Transactional state of the intialized transfer or receive I2S operation.

Data Fields

- uint32_t state
  State of transfer.
- i2s_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.
- bool oneChannel

true mono, false stereo
- uint8_t dataLength
  Data length (4 - 32)
- bool pack48
  Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)
- bool useFifo48H
  When dataLength 17-24: true use FIFOWR48H, false use FIFOWR.
- volatile i2s_transfer_t i2sQueue [I2S_NUM_BUFFERS]
  Transfer queue storing transfer buffers.
- volatile uint8_t queueUser
  Queue index where user's next transfer will be stored.
- volatile uint8_t queueDriver
  Queue index of buffer actually used by the driver.
- volatile uint32_t errorCount
  Number of buffer underruns/overruns.
- volatile uint32_t transferCount
  Number of bytes transferred.
- volatile uint8_t watermark
  FIFO trigger level.

17.8.3 Macro Definition Documentation

17.8.3.1 #define FSL_I2S_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Current version: 2.0.0

Change log:

- Version 2.0.0
  - initial version

17.8.3.2 #define I2S_NUM_BUFFERS (4)

17.8.4 Typedef Documentation

17.8.4.1 typedef void(∗ i2s_transfer_callback_t)(I2S_Type ∗base, i2s_handle_t ∗handle, status_t completionStatus, void ∗userData)

Parameters

base	I2S base pointer.

handle	pointer to I2S transaction.
completion-Status	status of the transaction.
userData	optional pointer to user arguments data.

17.8.5 Enumeration Type Documentation

17.8.5.1 enum _i2s_status

Enumerator

kStatus_I2S_BufferComplete Transfer from/into a single buffer has completed.
kStatus_I2S_Done All buffers transfers have completed.
kStatus_I2S_Busy Already performing a transfer and cannot queue another buffer.

17.8.5.2 enum i2s_flags_t

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kI2S_TxErrorFlag TX error interrupt.
kI2S_TxLevelFlag TX level interrupt.
kI2S_RxErrorFlag RX error interrupt.
kI2S_RxLevelFlag RX level interrupt.

17.8.5.3 enum i2s_master_slave_t

Enumerator

kI2S_MasterSlaveNormalSlave Normal slave.
kI2S_MasterSlaveWsSyncMaster WS synchronized master.
kI2S_MasterSlaveExtSckMaster Master using existing SCK.
kI2S_MasterSlaveNormalMaster Normal master.

17.8.5.4 enum i2s_mode_t

Enumerator

kI2S_ModeI2sClassic I2S classic mode.

kI2S_ModeDspWs50 DSP mode, WS having 50% duty cycle.
kI2S_ModeDspWsShort DSP mode, WS having one clock long pulse.
kI2S_ModeDspWsLong DSP mode, WS having one data slot long pulse.

17.8.6 Function Documentation

17.8.6.1 void I2S_TxInit (I2S_Type * *base*, const i2s_config_t * *config*)

Ungates the FLEXCOMM clock and configures the module for I2S transmission using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_TxGetDefaultConfig().

Note

This API should be called at the beginning of the application to use the I2S driver.

Parameters

base	I2S base pointer.
config	pointer to I2S configuration structure.

17.8.6.2 void I2S_RxInit (I2S_Type * *base*, const i2s_config_t * *config*)

Ungates the FLEXCOMM clock and configures the module for I2S receive using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_RxGetDefaultConfig().

Note

This API should be called at the beginning of the application to use the I2S driver.

Parameters

base	I2S base pointer.
config	pointer to I2S configuration structure.

17.8.6.3 void I2S_TxGetDefaultConfig (i2s_config_t * *config*)

This API initializes the configuration structure for use in I2S_TxInit(). The initialized structure can remain unchanged in I2S_TxInit(), or it can be modified before calling I2S_TxInit(). Example:

```
i2s_config_t config;
I2S_TxGetDefaultConfig(&config);
```

Default values:

```
*   config->masterSlave = kI2S_MasterSlaveNormalMaster;
*   config->mode = kI2S_ModeI2sClassic;
*   config->rightLow = false;
*   config->leftJust = false;
*   config->pdmData = false;
*   config->sckPol = false;
*   config->wsPol = false;
*   config->divider = 1;
*   config->oneChannel = false;
*   config->dataLength = 16;
*   config->frameLength = 32;
*   config->position = 0;
*   config->watermark = 4;
*   config->txEmptyZero = true;
*   config->pack48 = false;
*
```

Parameters

config	pointer to I2S configuration structure.

17.8.6.4 void I2S_RxGetDefaultConfig (i2s_config_t * *config*)

This API initializes the configuration structure for use in I2S_RxInit(). The initialized structure can remain unchanged in I2S_RxInit(), or it can be modified before calling I2S_RxInit(). Example:

```
i2s_config_t config;
I2S_RxGetDefaultConfig(&config);
```

Default values:

```
*   config->masterSlave = kI2S_MasterSlaveNormalSlave;
*   config->mode = kI2S_ModeI2sClassic;
*   config->rightLow = false;
*   config->leftJust = false;
*   config->pdmData = false;
*   config->sckPol = false;
*   config->wsPol = false;
*   config->divider = 1;
*   config->oneChannel = false;
*   config->dataLength = 16;
*   config->frameLength = 32;
*   config->position = 0;
*   config->watermark = 4;
*   config->txEmptyZero = false;
*   config->pack48 = false;
*
```

Parameters

config	pointer to I2S configuration structure.

17.8.6.5 void I2S_Deinit (I2S_Type * *base*)

This API gates the FLEXCOMM clock. The I2S module can't operate unless I2S_TxInit or I2S_RxInit is called to enable the clock.

Parameters

base	I2S base pointer.

17.8.6.6 void I2S_TxTransferCreateHandle (I2S_Type * *base*, i2s_handle_t * *handle*, i2s_transfer_callback_t *callback*, void * *userData*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
callback	function to be called back when transfer is done or fails.
userData	pointer to data passed to callback.

17.8.6.7 status_t I2S_TxTransferNonBlocking (I2S_Type * *base*, i2s_handle_t * *handle*, i2s_transfer_t *transfer*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
transfer	data buffer.

Return values

kStatus_Success	
kStatus_I2S_Busy	if all queue slots are occupied with unsent buffers.

17.8.6.8 void I2S_TxTransferAbort (I2S_Type * *base*, i2s_handle_t * *handle*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.

17.8.6.9 void I2S_RxTransferCreateHandle (I2S_Type * *base*, i2s_handle_t * *handle*, i2s_transfer_callback_t *callback*, void * *userData*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
callback	function to be called back when transfer is done or fails.
userData	pointer to data passed to callback.

17.8.6.10 status_t I2S_RxTransferNonBlocking (I2S_Type * *base*, i2s_handle_t * *handle*, i2s_transfer_t *transfer*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
transfer	data buffer.

Return values

kStatus_Success	

kStatus_I2S_Busy	if all queue slots are occupied with buffers which are not full.

17.8.6.11 void I2S_RxTransferAbort (I2S_Type * *base,* i2s_handle_t * *handle*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.

17.8.6.12 status_t I2S_TransferGetCount (I2S_Type * *base,* i2s_handle_t * *handle,* size_t * *count*)

Parameters

	base	I2S base pointer.
	handle	pointer to handle structure.
out	*count*	number of bytes transferred so far by the non-blocking transaction.

Return values

kStatus_Success	
kStatus_NoTransferInProgress	there is no non-blocking transaction currently in progress.

17.8.6.13 status_t I2S_TransferGetErrorCount (I2S_Type * *base,* i2s_handle_t * *handle,* size_t * *count*)

Parameters

	base	I2S base pointer.
	handle	pointer to handle structure.
out	*count*	number of transmit errors encountered so far by the non-blocking transaction.

Return values

kStatus_Success	
kStatus_NoTransferInProgress	there is no non-blocking transaction currently in progress.

17.8.6.14 static void I2S_Enable (I2S_Type * *base*) [inline], [static]

Parameters

base	I2S base pointer.

17.8.6.15 static void I2S_Disable (I2S_Type * *base*) [inline], [static]

Parameters

base	I2S base pointer.

17.8.6.16 static void I2S_EnableInterrupts (I2S_Type * *base*, uint32_t *interruptMask*) [inline], [static]

Parameters

base	I2S base pointer.
interruptMask	bit mask of interrupts to enable. See i2s_flags_t for the set of constants that should be OR'd together to form the bit mask.

17.8.6.17 static void I2S_DisableInterrupts (I2S_Type * *base*, uint32_t *interruptMask*) [inline], [static]

Parameters

base	I2S base pointer.

interruptMask	bit mask of interrupts to enable. See i2s_flags_t for the set of constants that should be OR'd together to form the bit mask.

17.8.6.18 static uint32_t I2S_GetEnabledInterrupts (I2S_Type * *base*) [inline], [static]

Parameters

base	I2S base pointer.

Returns

A bitmask composed of i2s_flags_t enumerators OR'd together to indicate the set of enabled interrupts.

17.8.6.19 void I2S_TxHandleIRQ (I2S_Type * *base*, i2s_handle_t * *handle*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.

17.8.6.20 void I2S_RxHandleIRQ (I2S_Type * *base*, i2s_handle_t * *handle*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.

17.9 I2S DMA Driver

17.9.1 Overview

Files

- file fsl_i2s_dma.h

Data Structures

- struct i2s_dma_handle_t
  Members not to be accessed / modified outside of the driver. More...

Typedefs

- typedef void(∗ i2s_dma_transfer_callback_t)(I2S_Type ∗base, i2s_dma_handle_t ∗handle, status_t completionStatus, void ∗userData)
  Callback function invoked from DMA API on completion.

Driver version

- #define FSL_I2S_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  I2S DMA driver version 2.0.0.

DMA API

- void I2S_TxTransferCreateHandleDMA (I2S_Type ∗base, i2s_dma_handle_t ∗handle, dma_handle_t ∗dmaHandle, i2s_dma_transfer_callback_t callback, void ∗userData)
  Initializes handle for transfer of audio data.
- status_t I2S_TxTransferSendDMA (I2S_Type ∗base, i2s_dma_handle_t ∗handle, i2s_transfer_t transfer)
  Begins or queue sending of the given data.
- void I2S_TransferAbortDMA (I2S_Type ∗base, i2s_dma_handle_t ∗handle)
  Aborts transfer of data.
- void I2S_RxTransferCreateHandleDMA (I2S_Type ∗base, i2s_dma_handle_t ∗handle, dma_handle_t ∗dmaHandle, i2s_dma_transfer_callback_t callback, void ∗userData)
  Initializes handle for reception of audio data.
- status_t I2S_RxTransferReceiveDMA (I2S_Type ∗base, i2s_dma_handle_t ∗handle, i2s_transfer_t transfer)
  Begins or queue reception of data into given buffer.
- void I2S_DMACallback (dma_handle_t ∗handle, void ∗userData, bool transferDone, uint32_t tcds)
  Invoked from DMA interrupt handler.

17.9.2 Data Structure Documentation

17.9.2.1 struct _i2s_dma_handle

Data Fields

- uint32_t state
  Internal state of I2S DMA transfer.
- i2s_dma_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.
- dma_handle_t * dmaHandle
  DMA handle.
- volatile i2s_transfer_t i2sQueue [I2S_NUM_BUFFERS]
  Transfer queue storing transfer buffers.
- volatile uint8_t queueUser
  Queue index where user's next transfer will be stored.
- volatile uint8_t queueDriver
  Queue index of buffer actually used by the driver.

17.9.3 Macro Definition Documentation

17.9.3.1 #define FSL_I2S_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Current version: 2.0.0

Change log:

- Version 2.0.0
  - initial version

17.9.4 Typedef Documentation

17.9.4.1 typedef void(* i2s_dma_transfer_callback_t)(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)

Parameters

base	I2S base pointer.

handle	pointer to I2S transaction.
completion-Status	status of the transaction.
userData	optional pointer to user arguments data.

17.9.5 Function Documentation

17.9.5.1 void I2S_TxTransferCreateHandleDMA (I2S_Type * *base*, i2s_dma_handle_t * *handle*, dma_handle_t * *dmaHandle*, i2s_dma_transfer_callback_t *callback*, void * *userData*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
dmaHandle	pointer to dma handle structure.
callback	function to be called back when transfer is done or fails.
userData	pointer to data passed to callback.

17.9.5.2 status_t I2S_TxTransferSendDMA (I2S_Type * *base*, i2s_dma_handle_t * *handle*, i2s_transfer_t *transfer*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
transfer	data buffer.

Return values

kStatus_Success	
kStatus_I2S_Busy	if all queue slots are occupied with unsent buffers.

17.9.5.3 void I2S_TransferAbortDMA (I2S_Type * *base*, i2s_dma_handle_t * *handle*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.

17.9.5.4 void I2S_RxTransferCreateHandleDMA (I2S_Type * *base*, i2s_dma_handle_t * *handle*, dma_handle_t * *dmaHandle*, i2s_dma_transfer_callback_t *callback*, void * *userData*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
dmaHandle	pointer to dma handle structure.
callback	function to be called back when transfer is done or fails.
userData	pointer to data passed to callback.

17.9.5.5 status_t I2S_RxTransferReceiveDMA (I2S_Type * *base*, i2s_dma_handle_t * *handle*, i2s_transfer_t *transfer*)

Parameters

base	I2S base pointer.
handle	pointer to handle structure.
transfer	data buffer.

Return values

kStatus_Success	
kStatus_I2S_Busy	if all queue slots are occupied with buffers which are not full.

17.9.5.6 void I2S_DMACallback (dma_handle_t * *handle*, void * *userData*, bool *transferDone*, uint32_t *tcds*)

Parameters

handle	pointer to DMA handle structure.
userData	argument for user callback.
transferDone	if transfer was done.
tcds	

Chapter 18
SPI: Serial Peripheral Interface Driver

18.1 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.

18.2 Typical use case

18.2.1 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, srcClock_Hz);
    SPI_MasterTransferCreateHandle(SPI0, &spiHandle, SPI_UserCallback, NULL);

    // Prepare to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
```

```
    xfer.dataSize = sizeof(sendData);

    // Send out.
    SPI_MasterTransferNonBlocking(SPI0, &spiHandle, &xfer);

    // Wait send finished.
    while (!isFinished)
    {
    }

    // ...
}
```

18.2.2 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;

/* SPI/DMA buffers MUST be always array of 4B (32 bit) words */
uint32_t sendData[BUFFER_LEN] = ...;
uint32_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)
{
    //...

    // Initialize DMA peripheral
    DMA_Init(DMA0);

    // Initialize SPI peripheral
    SPI_MasterGetDefaultConfig(&masterConfig);
    masterConfig.sselNum = SPI_SSEL;
    SPI_MasterInit(SPI0, &masterConfig, srcClock_Hz);

    // Enable DMA channels connected to SPI0 Tx/SPI0 Rx request lines
    DMA_EnableChannel(SPI0, SPI_MASTER_TX_CHANNEL);
    DMA_EnableChannel(SPI0, SPI_MASTER_RX_CHANNEL);

    // Set DMA channels priority
    DMA_SetChannelPriority(SPI0, SPI_MASTER_TX_CHANNEL,
      kDMA_ChannelPriority3);
    DMA_SetChannelPriority(SPI0, SPI_MASTER_RX_CHANNEL,
      kDMA_ChannelPriority2);

    // Creates the DMA handle.
    DMA_CreateHandle(&masterTxHandle, SPI0, SPI_MASTER_TX_CHANNEL);
    DMA_CreateHandle(&masterRxHandle, SPI0, SPI_MASTER_RX_CHANNEL);

    // Create SPI DMA handle
    SPI_MasterTransferCreateHandleDMA(SPI0, spiHandle, SPI_UserCallback,
      NULL, &g_spiTxDmaHandle, &g_spiRxDmaHandle);

    // Prepares to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
```

```
    xfer.dataSize = sizeof(sendData);

    // Sends out.
    SPI_MasterTransferDMA(SPI0, &spiHandle, &xfer);

    // Waits for send to complete.
    while (!isFinished)
    {
    }

    // ...
}
```

Modules

- SPI DMA Driver
- SPI Driver
- SPI FreeRTOS driver

18.3 SPI Driver

18.3.1 Overview

This section describes the programming interface of the 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_config_t
  Internal configuration structure used in 'spi' and 'spi_dma' driver. More...
- struct spi_master_handle_t
  SPI transfer handle structure. More...

Typedefs

- typedef spi_master_handle_t spi_slave_handle_t
  Slave handle type.
- 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 slave callback for finished transmit.

Enumerations

- enum spi_xfer_option_t {
  kSPI_FrameDelay = (SPI_FIFOWR_EOF_MASK),
  kSPI_FrameAssert = (SPI_FIFOWR_EOT_MASK) }
  SPI transfer option.
- enum spi_shift_direction_t {
  kSPI_MsbFirst = 0U,
  kSPI_LsbFirst = 1U }
  SPI data shifter direction options.

- 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_txfifo_watermark_t {
  kSPI_TxFifo0 = 0,
  kSPI_TxFifo1 = 1,
  kSPI_TxFifo2 = 2,
  kSPI_TxFifo3 = 3,
  kSPI_TxFifo4 = 4,
  kSPI_TxFifo5 = 5,
  kSPI_TxFifo6 = 6,
  kSPI_TxFifo7 = 7 }
  txFIFO watermark values
- enum spi_rxfifo_watermark_t {
  kSPI_RxFifo1 = 0,
  kSPI_RxFifo2 = 1,
  kSPI_RxFifo3 = 2,
  kSPI_RxFifo4 = 3,
  kSPI_RxFifo5 = 4,
  kSPI_RxFifo6 = 5,
  kSPI_RxFifo7 = 6,
  kSPI_RxFifo8 = 7 }
  rxFIFO watermark values
- enum spi_data_width_t {
  kSPI_Data4Bits = 3,
  kSPI_Data5Bits = 4,
  kSPI_Data6Bits = 5,
  kSPI_Data7Bits = 6,
  kSPI_Data8Bits = 7,
  kSPI_Data9Bits = 8,
  kSPI_Data10Bits = 9,
  kSPI_Data11Bits = 10,
  kSPI_Data12Bits = 11,
  kSPI_Data13Bits = 12,
  kSPI_Data14Bits = 13,
  kSPI_Data15Bits = 14,
  kSPI_Data16Bits = 15 }
  Transfer data width.
- enum spi_ssel_t {

kSPI_Ssel0 = 0,
kSPI_Ssel1 = 1,
kSPI_Ssel2 = 2,
kSPI_Ssel3 = 3 }
Slave select.

- enum _spi_status {
kStatus_SPI_Busy = MAKE_STATUS(kStatusGroup_LPC_SPI, 0),
kStatus_SPI_Idle = MAKE_STATUS(kStatusGroup_LPC_SPI, 1),
kStatus_SPI_Error = MAKE_STATUS(kStatusGroup_LPC_SPI, 2),
kStatus_SPI_BaudrateNotSupport }
SPI transfer status.
- enum _spi_interrupt_enable {
kSPI_RxLvlIrq = SPI_FIFOINTENSET_RXLVL_MASK,
kSPI_TxLvlIrq = SPI_FIFOINTENSET_TXLVL_MASK }
SPI interrupt sources.
- enum _spi_statusflags {
kSPI_TxEmptyFlag = SPI_FIFOSTAT_TXEMPTY_MASK,
kSPI_TxNotFullFlag = SPI_FIFOSTAT_TXNOTFULL_MASK,
kSPI_RxNotEmptyFlag = SPI_FIFOSTAT_RXNOTEMPTY_MASK,
kSPI_RxFullFlag = SPI_FIFOSTAT_RXFULL_MASK }
SPI status flags.

Functions

- uint32_t SPI_GetInstance (SPI_Type *base)
Returns instance number for SPI peripheral base address.

Driver version

- #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
USART 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.
- status_t 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.
- status_t 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 (SPI_Type *base, bool enable)
  Enable or disable the SPI Master or Slave.

Status

- static uint32_t SPI_GetStatusFlags (SPI_Type *base)
  Gets the status flag.

Interrupts

- static void SPI_EnableInterrupts (SPI_Type *base, uint32_t irqs)
  Enables the interrupt for the SPI.
- static void SPI_DisableInterrupts (SPI_Type *base, uint32_t irqs)
  Disables the interrupt for the SPI.

DMA Control

- void SPI_EnableTxDMA (SPI_Type *base, bool enable)
  Enables the DMA request from SPI txFIFO.
- void SPI_EnableRxDMA (SPI_Type *base, bool enable)
  Enables the DMA request from SPI rxFIFO.

Bus Operations

- status_t SPI_MasterSetBaud (SPI_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
  Sets the baud rate for SPI transfer.
- void SPI_WriteData (SPI_Type *base, uint16_t data, uint32_t configFlags)
  Writes a data into the SPI data register.
- static uint32_t SPI_ReadData (SPI_Type *base)
  Gets a data from the SPI data register.

Transactional

- status_t 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 master transfer count.

- void SPI_MasterTransferAbort (SPI_Type *base, spi_master_handle_t *handle)
  SPI master aborts a transfer using an interrupt.
- void SPI_MasterTransferHandleIRQ (SPI_Type *base, spi_master_handle_t *handle)
  Interrupts the handler for the SPI.
- static status_t 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 slave transfer count.
- static void SPI_SlaveTransferAbort (SPI_Type *base, spi_slave_handle_t *handle)
  SPI slave aborts a transfer using an interrupt.
- static void SPI_SlaveTransferHandleIRQ (SPI_Type *base, spi_slave_handle_t *handle)
  Interrupts a handler for the SPI slave.

18.3.2 Data Structure Documentation

18.3.2.1 struct spi_master_config_t

Data Fields

- bool enableLoopback
  Enable loopback for test purpose.
- bool enableMaster
  Enable SPI at initialization time.
- spi_clock_polarity_t polarity
  Clock polarity.
- spi_clock_phase_t phase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.
- uint32_t baudRate_Bps
  Baud Rate for SPI in Hz.
- spi_data_width_t dataWidth
  Width of the data.
- spi_ssel_t sselNum
  Slave select number.
- spi_txfifo_watermark_t txWatermark
  txFIFO watermark
- spi_rxfifo_watermark_t rxWatermark
  rxFIFO watermark

18.3.2.2 struct spi_slave_config_t

Data Fields

- bool enableSlave
  Enable SPI at initialization time.
- spi_clock_polarity_t polarity
  Clock polarity.
- spi_clock_phase_t phase
  Clock phase.
- spi_shift_direction_t direction
  MSB or LSB.
- spi_data_width_t dataWidth
  Width of the data.
- spi_txfifo_watermark_t txWatermark
  txFIFO watermark
- spi_rxfifo_watermark_t rxWatermark
  rxFIFO watermark

18.3.2.3 struct spi_transfer_t

Data Fields

- uint8_t * txData
  Send buffer.
- uint8_t * rxData
  Receive buffer.
- uint32_t configFlags
  Additional option to control transfer.
- size_t dataSize
  Transfer bytes.

18.3.2.4 struct spi_config_t

18.3.2.5 struct _spi_master_handle

Master handle type.

Data Fields

- uint8_t *volatile txData
  Transfer buffer.
- uint8_t *volatile rxData
  Receive buffer.
- volatile size_t txRemainingBytes
  Number of data to be transmitted [in bytes].
- volatile size_t rxRemainingBytes
  Number of data to be received [in bytes].

- volatile size_t toReceiveCount
  Receive data remaining in bytes.
- size_t totalByteCount
  A number of transfer bytes.
- volatile uint32_t state
  SPI internal state.
- spi_master_callback_t callback
  SPI callback.
- void * userData
  Callback parameter.
- uint8_t dataWidth
  Width of the data [Valid values: 1 to 16].
- uint8_t sselNum
  Slave select number to be asserted when transferring data [Valid values: 0 to 3].
- uint32_t configFlags
  Additional option to control transfer.
- spi_txfifo_watermark_t txWatermark
  txFIFO watermark
- spi_rxfifo_watermark_t rxWatermark
  rxFIFO watermark

18.3.3 Macro Definition Documentation

18.3.3.1 #define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

18.3.4 Enumeration Type Documentation

18.3.4.1 enum spi_xfer_option_t

Enumerator

kSPI_FrameDelay Delay chip select.
kSPI_FrameAssert When transfer ends, assert chip select.

18.3.4.2 enum spi_shift_direction_t

Enumerator

kSPI_MsbFirst Data transfers start with most significant bit.
kSPI_LsbFirst Data transfers start with least significant bit.

18.3.4.3 enum spi_clock_polarity_t

Enumerator

kSPI_ClockPolarityActiveHigh Active-high SPI clock (idles low).

kSPI_ClockPolarityActiveLow Active-low SPI clock (idles high).

18.3.4.4 enum spi_clock_phase_t

Enumerator

kSPI_ClockPhaseFirstEdge First edge on SCK occurs at the middle of the first cycle of a data transfer.

kSPI_ClockPhaseSecondEdge First edge on SCK occurs at the start of the first cycle of a data transfer.

18.3.4.5 enum spi_txfifo_watermark_t

Enumerator

kSPI_TxFifo0 SPI tx watermark is empty.
kSPI_TxFifo1 SPI tx watermark at 1 item.
kSPI_TxFifo2 SPI tx watermark at 2 items.
kSPI_TxFifo3 SPI tx watermark at 3 items.
kSPI_TxFifo4 SPI tx watermark at 4 items.
kSPI_TxFifo5 SPI tx watermark at 5 items.
kSPI_TxFifo6 SPI tx watermark at 6 items.
kSPI_TxFifo7 SPI tx watermark at 7 items.

18.3.4.6 enum spi_rxfifo_watermark_t

Enumerator

kSPI_RxFifo1 SPI rx watermark at 1 item.
kSPI_RxFifo2 SPI rx watermark at 2 items.
kSPI_RxFifo3 SPI rx watermark at 3 items.
kSPI_RxFifo4 SPI rx watermark at 4 items.
kSPI_RxFifo5 SPI rx watermark at 5 items.
kSPI_RxFifo6 SPI rx watermark at 6 items.
kSPI_RxFifo7 SPI rx watermark at 7 items.
kSPI_RxFifo8 SPI rx watermark at 8 items.

18.3.4.7 enum spi_data_width_t

Enumerator

kSPI_Data4Bits 4 bits data width

kSPI_Data5Bits 5 bits data width
kSPI_Data6Bits 6 bits data width
kSPI_Data7Bits 7 bits data width
kSPI_Data8Bits 8 bits data width
kSPI_Data9Bits 9 bits data width
kSPI_Data10Bits 10 bits data width
kSPI_Data11Bits 11 bits data width
kSPI_Data12Bits 12 bits data width
kSPI_Data13Bits 13 bits data width
kSPI_Data14Bits 14 bits data width
kSPI_Data15Bits 15 bits data width
kSPI_Data16Bits 16 bits data width

18.3.4.8 enum spi_ssel_t

Enumerator

kSPI_Ssel0 Slave select 0.
kSPI_Ssel1 Slave select 1.
kSPI_Ssel2 Slave select 2.
kSPI_Ssel3 Slave select 3.

18.3.4.9 enum _spi_status

Enumerator

kStatus_SPI_Busy SPI bus is busy.
kStatus_SPI_Idle SPI is idle.
kStatus_SPI_Error SPI error.
kStatus_SPI_BaudrateNotSupport Baudrate is not support in current clock source.

18.3.4.10 enum _spi_interrupt_enable

Enumerator

kSPI_RxLvlIrq Rx level interrupt.
kSPI_TxLvlIrq Tx level interrupt.

18.3.4.11 enum _spi_statusflags

Enumerator

kSPI_TxEmptyFlag txFifo is empty

kSPI_TxNotFullFlag txFifo is not full
kSPI_RxNotEmptyFlag rxFIFO is not empty
kSPI_RxFullFlag rxFIFO is full

18.3.5 Function Documentation

18.3.5.1 uint32_t SPI_GetInstance (SPI_Type * *base*)

18.3.5.2 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

18.3.5.3 status_t 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.

18.3.5.4 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

18.3.5.5 status_t 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 = flexSPIClockPolarity_ActiveHigh;
.phase = flexSPIClockPhase_FirstEdge;
.direction = flexSPIMsbFirst;
...
};
SPI_SlaveInit(SPI0, &config);
```

Parameters

base	SPI base pointer
config	pointer to slave configuration structure

18.3.5.6 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

18.3.5.7 static void SPI_Enable (SPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SPI base pointer
enable	or disable (true = enable, false = disable)

18.3.5.8 static uint32_t SPI_GetStatusFlags (SPI_Type * *base*) [inline], [static]

Parameters

base	SPI base pointer

Returns

SPI Status, use status flag to AND _spi_statusflags could get the related status.

18.3.5.9 static void SPI_EnableInterrupts (SPI_Type * *base*, uint32_t *irqs*) [inline], [static]

Parameters

base	SPI base pointer
irqs	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxLvlIrq • kSPI_TxLvlIrq

18.3.5.10 static void SPI_DisableInterrupts (SPI_Type * *base*, uint32_t *irqs*) [inline], [static]

Parameters

base	SPI base pointer
irqs	SPI interrupt source. The parameter can be any combination of the following values: • kSPI_RxLvlIrq • kSPI_TxLvlIrq

18.3.5.11 void SPI_EnableTxDMA (SPI_Type * *base*, bool *enable*)

Parameters

base	SPI base pointer
enable	True means enable DMA, false means disable DMA

18.3.5.12 void SPI_EnableRxDMA (SPI_Type * *base*, bool *enable*)

Parameters

base	SPI base pointer
enable	True means enable DMA, false means disable DMA

18.3.5.13 status_t SPI_MasterSetBaud (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.

18.3.5.14 void SPI_WriteData (SPI_Type * *base*, uint16_t *data*, uint32_t *configFlags*)

Parameters

base	SPI base pointer
data	needs to be write.
configFlags	transfer configuration options spi_xfer_option_t

18.3.5.15 static uint32_t SPI_ReadData (SPI_Type * *base*) [inline], [static]

Parameters

base	SPI base pointer

Returns

Data in the register.

18.3.5.16 status_t 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.

18.3.5.17 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.

18.3.5.18 status_t SPI_MasterTransferNonBlocking (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_transfer_t * *xfer*)

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.

18.3.5.19 status_t SPI_MasterTransferGetCount (SPI_Type * *base*, spi_master_handle_t * *handle*, size_t * *count*)

This function gets the master transfer count.

Parameters

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

Returns

status of status_t.

18.3.5.20 void SPI_MasterTransferAbort (SPI_Type * *base,* spi_master_handle_t * *handle*)

This function aborts a transfer using an interrupt.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_master_handle_t structure which stores the transfer state.

18.3.5.21 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.

18.3.5.22 static status_t 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.

18.3.5.23 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.

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.

18.3.5.24 static status_t SPI_SlaveTransferGetCount (SPI_Type * *base*, spi_slave_handle_t * *handle*, size_t * *count*) [inline], [static]

This function gets the slave transfer count.

Parameters

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

Returns

status of status_t.

18.3.5.25 static void SPI_SlaveTransferAbort (SPI_Type * *base*, spi_slave_handle_t * *handle*) [inline], [static]

This function aborts a transfer using an interrupt.

Parameters

base	SPI peripheral base address.
handle	Pointer to the spi_slave_handle_t structure which stores the transfer state.

18.3.5.26 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

18.4 SPI DMA Driver

18.4.1 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

- status_t 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.
- static status_t 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.
- 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)
  Gets the master DMA transfer remaining bytes.
- 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)
  Gets the slave DMA transfer remaining bytes.

18.4.2 Data Structure Documentation

18.4.2.1 struct _spi_dma_handle

Data Fields

- volatile bool txInProgress
  Send transfer finished.
- volatile 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.

18.4.3 Typedef Documentation

18.4.3.1 typedef void(* spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)

18.4.4 Function Documentation

18.4.4.1 status_t 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.

18.4.4.2 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.

18.4.4.3 static status_t 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.

18.4.4.4 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.

18.4.4.5 void SPI_MasterTransferAbortDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*)

Parameters

base	SPI peripheral base address.

handle	SPI DMA handle pointer.

18.4.4.6 status_t SPI_MasterTransferGetCountDMA (SPI_Type * *base,* spi_dma_handle_t * *handle,* size_t * *count*)

This function gets the master DMA transfer remaining bytes.

Parameters

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

Returns

status of status_t.

18.4.4.7 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.

18.4.4.8 static status_t SPI_SlaveTransferGetCountDMA (SPI_Type * *base,* spi_dma_handle_t * *handle,* size_t * *count*) [inline], [static]

This function gets the slave DMA transfer remaining bytes.

Parameters

base	SPI peripheral base address.
handle	A pointer to the spi_dma_handle_t structure which stores the transfer state.

count	A number of bytes transferred by the non-blocking transaction.

Returns

status of status_t.

18.5 SPI FreeRTOS driver

18.5.1 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...

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.

18.5.2 Data Structure Documentation

18.5.2.1 struct spi_rtos_handle_t

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.

18.5.3 Function Documentation

18.5.3.1 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.

18.5.3.2 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.

18.5.3.3 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 19
USART: Universal Asynchronous Receiver/Transmitter Driver

19.1 Overview

The SDK provides a peripheral UART driver for the Universal Synchronous Receiver/Transmitter (USART) module of LPC devices. Driver does not support synchronous mode !

The USART driver includes two parts: functional APIs and transactional APIs.

Functional APIs are used for USART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the USART 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. USART 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 usart_handle_t as the second parameter. Initialize the handle by calling the USART_TransferCreateHandle() API.

Transactional APIs support asynchronous transfer, which means that the functions USART_TransferSendNonBlocking() and USART_TransferReceiveNonBlocking() set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus_USART_TxIdle and kStatus_USART_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 USART_TransferCreateHandle(). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The USART_TransferReceiveNonBlocking() function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the kStatus_USART_RxIdle.

If the receive ring buffer is full, the upper layer is informed through a callback with the kStatus_USART_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:

```
USART_TransferCreateHandle(USART0, &handle, USART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

19.2 Typical use case

19.2.1 USART Send/receive using a polling method

```
uint8_t ch;
USART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

USART_Init(USART1,&user_config,120000000U);

while(1)
{
    USART_ReadBlocking(USART1, &ch, 1);
    USART_WriteBlocking(USART1, &ch, 1);
}
```

19.2.2 USART Send/receive using an interrupt method

```
usart_handle_t g_usartHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);
    USART_TransferCreateHandle(USART1, &g_usartHandle, USART_UserCallback, NULL);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData);
    txFinished = false;

    // Send out.
    USART_TransferSendNonBlocking(USART1, &g_usartHandle, &sendXfer);
```

```
    // Wait send finished.
    while (!txFinished)
    {
    }

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData);
    rxFinished = false;

    // Receive.
    USART_TransferReceiveNonBlocking(USART1, &g_usartHandle, &receiveXfer,
      NULL);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

19.2.3 USART Receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE     32

usart_handle_t g_usartHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);
    USART_TransferCreateHandle(USART1, &g_usartHandle, USART_UserCallback, NULL);
    USART_TransferStartRingBuffer(USART1, &g_usartHandle, ringBuffer,
      RING_BUFFER_SIZE);
    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData);
```

```
    rxFinished = false;

    // Receive.
    USART_TransferReceiveNonBlocking(USART1, &g_usartHandle, &receiveXfer);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data. */
    {
        ;
    }
    else
    {
        if (bytesRead) /* Received some data, process first. */
        {
            ;
        }

        // Wait receive finished.
        while (!rxFinished)
        {
        }
    }

    // ...
}
```

19.2.4 USART Send/Receive using the DMA method

```
usart_handle_t g_usartHandle;
dma_handle_t g_usartTxDmaHandle;
dma_handle_t g_usartRxDmaHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = ['H', 'e', 'l', 'l', 'o'];
uint8_t receiveData[32];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);

    // Set up the DMA
```

```
    DMA_Init(DMA0);
    DMA_EnableChannel(DMA0, USART_TX_DMA_CHANNEL);
    DMA_EnableChannel(DMA0, USART_RX_DMA_CHANNEL);

    DMA_CreateHandle(&g_usartTxDmaHandle, DMA0, USART_TX_DMA_CHANNEL);
    DMA_CreateHandle(&g_usartRxDmaHandle, DMA0, USART_RX_DMA_CHANNEL);

    USART_TransferCreateHandleDMA(USART1, &g_usartHandle, USART_UserCallback,
      NULL, &g_usartTxDmaHandle, &g_usartRxDmaHandle);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData);
    txFinished = false;

    // Send out.
    USART_TransferSendDMA(USART1, &g_usartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {
    }

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData);
    rxFinished = false;

    // Receive.
    USART_TransferReceiveDMA(USART1, &g_usartHandle, &receiveXfer);

    // Wait receive finished.
    while (!rxFinished)
    {
    }

    // ...
}
```

Modules

- USART DMA Driver
- USART Driver
- USART FreeRTOS Driver

19.3 USART Driver

19.3.1 Overview

Data Structures

- struct usart_config_t
  USART configuration structure. More...
- struct usart_transfer_t
  USART transfer structure. More...
- struct usart_handle_t
  USART handle structure. More...

Typedefs

- typedef void(∗ usart_transfer_callback_t)(USART_Type ∗base, usart_handle_t ∗handle, status_t status, void ∗userData)
  USART transfer callback function.

Enumerations

- enum _usart_status {
  kStatus_USART_TxBusy = MAKE_STATUS(kStatusGroup_LPC_USART, 0),
  kStatus_USART_RxBusy = MAKE_STATUS(kStatusGroup_LPC_USART, 1),
  kStatus_USART_TxIdle = MAKE_STATUS(kStatusGroup_LPC_USART, 2),
  kStatus_USART_RxIdle = MAKE_STATUS(kStatusGroup_LPC_USART, 3),
  kStatus_USART_TxError = MAKE_STATUS(kStatusGroup_LPC_USART, 7),
  kStatus_USART_RxError = MAKE_STATUS(kStatusGroup_LPC_USART, 9),
  kStatus_USART_RxRingBufferOverrun = MAKE_STATUS(kStatusGroup_LPC_USART, 8),
  kStatus_USART_NoiseError = MAKE_STATUS(kStatusGroup_LPC_USART, 10),
  kStatus_USART_FramingError = MAKE_STATUS(kStatusGroup_LPC_USART, 11),
  kStatus_USART_ParityError = MAKE_STATUS(kStatusGroup_LPC_USART, 12),
  kStatus_USART_BaudrateNotSupport }
  Error codes for the USART driver.
- enum usart_parity_mode_t {
  kUSART_ParityDisabled = 0x0U,
  kUSART_ParityEven = 0x2U,
  kUSART_ParityOdd = 0x3U }
  USART parity mode.
- enum usart_stop_bit_count_t {
  kUSART_OneStopBit = 0U,
  kUSART_TwoStopBit = 1U }
  USART stop bit count.
- enum usart_data_len_t {
  kUSART_7BitsPerChar = 0U,

kUSART_8BitsPerChar = 1U }
USART data size.
- enum usart_txfifo_watermark_t {
kUSART_TxFifo0 = 0,
kUSART_TxFifo1 = 1,
kUSART_TxFifo2 = 2,
kUSART_TxFifo3 = 3,
kUSART_TxFifo4 = 4,
kUSART_TxFifo5 = 5,
kUSART_TxFifo6 = 6,
kUSART_TxFifo7 = 7 }
txFIFO watermark values
- enum usart_rxfifo_watermark_t {
kUSART_RxFifo1 = 0,
kUSART_RxFifo2 = 1,
kUSART_RxFifo3 = 2,
kUSART_RxFifo4 = 3,
kUSART_RxFifo5 = 4,
kUSART_RxFifo6 = 5,
kUSART_RxFifo7 = 6,
kUSART_RxFifo8 = 7 }
rxFIFO watermark values
- enum _usart_interrupt_enable
USART interrupt configuration structure, default settings all disabled.
- enum _usart_flags {
kUSART_TxError = (USART_FIFOSTAT_TXERR_MASK),
kUSART_RxError = (USART_FIFOSTAT_RXERR_MASK),
kUSART_TxFifoEmptyFlag = (USART_FIFOSTAT_TXEMPTY_MASK),
kUSART_TxFifoNotFullFlag = (USART_FIFOSTAT_TXNOTFULL_MASK),
kUSART_RxFifoNotEmptyFlag = (USART_FIFOSTAT_RXNOTEMPTY_MASK),
kUSART_RxFifoFullFlag = (USART_FIFOSTAT_RXFULL_MASK) }
USART status flags.

Functions

- uint32_t USART_GetInstance (USART_Type ∗base)
Returns instance number for USART peripheral base address.

Driver version

- #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
USART driver version 2.0.0.

Initialization and deinitialization

- status_t USART_Init (USART_Type *base, const usart_config_t *config, uint32_t srcClock_Hz)
  Initializes a USART instance with user configuration structure and peripheral clock.
- void USART_Deinit (USART_Type *base)
  Deinitializes a USART instance.
- void USART_GetDefaultConfig (usart_config_t *config)
  Gets the default configuration structure.
- status_t USART_SetBaudRate (USART_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
  Sets the USART instance baud rate.

Status

- static uint32_t USART_GetStatusFlags (USART_Type *base)
  Get USART status flags.
- static void USART_ClearStatusFlags (USART_Type *base, uint32_t mask)
  Clear USART status flags.

Interrupts

- static void USART_EnableInterrupts (USART_Type *base, uint32_t mask)
  Enables USART interrupts according to the provided mask.
- static void USART_DisableInterrupts (USART_Type *base, uint32_t mask)
  Disables USART interrupts according to a provided mask.
- static void USART_EnableTxDMA (USART_Type *base, bool enable)
  Enable DMA for Tx.
- static void USART_EnableRxDMA (USART_Type *base, bool enable)
  Enable DMA for Rx.

Bus Operations

- static void USART_WriteByte (USART_Type *base, uint8_t data)
  Writes to the FIFOWR register.
- static uint8_t USART_ReadByte (USART_Type *base)
  Reads the FIFORD register directly.
- void USART_WriteBlocking (USART_Type *base, const uint8_t *data, size_t length)
  Writes to the TX register using a blocking method.
- status_t USART_ReadBlocking (USART_Type *base, uint8_t *data, size_t length)
  Read RX data register using a blocking method.

Transactional

- status_t USART_TransferCreateHandle (USART_Type *base, usart_handle_t *handle, usart_transfer_callback_t callback, void *userData)

Initializes the USART handle.

- status_t USART_TransferSendNonBlocking (USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer)
  Transmits a buffer of data using the interrupt method.
- void USART_TransferStartRingBuffer (USART_Type *base, usart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
  Sets up the RX ring buffer.
- void USART_TransferStopRingBuffer (USART_Type *base, usart_handle_t *handle)
  Aborts the background transfer and uninstalls the ring buffer.
- void USART_TransferAbortSend (USART_Type *base, usart_handle_t *handle)
  Aborts the interrupt-driven data transmit.
- status_t USART_TransferGetSendCount (USART_Type *base, usart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been written to USART TX register.
- status_t USART_TransferReceiveNonBlocking (USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer, size_t *receivedBytes)
  Receives a buffer of data using an interrupt method.
- void USART_TransferAbortReceive (USART_Type *base, usart_handle_t *handle)
  Aborts the interrupt-driven data receiving.
- status_t USART_TransferGetReceiveCount (USART_Type *base, usart_handle_t *handle, uint32_t *count)
  Get the number of bytes that have been received.
- void USART_TransferHandleIRQ (USART_Type *base, usart_handle_t *handle)
  USART IRQ handle function.

19.3.2 Data Structure Documentation

19.3.2.1 struct usart_config_t

Data Fields

- uint32_t baudRate_Bps
  USART baud rate.
- usart_parity_mode_t parityMode
  Parity mode, disabled (default), even, odd.
- usart_stop_bit_count_t stopBitCount
  Number of stop bits, 1 stop bit (default) or 2 stop bits.
- usart_data_len_t bitCountPerChar
  Data length - 7 bit, 8 bit.
- bool loopback
  Enable peripheral loopback.
- bool enableRx
  Enable RX.
- bool enableTx
  Enable TX.
- usart_txfifo_watermark_t txWatermark
  txFIFO watermark
- usart_rxfifo_watermark_t rxWatermark
  rxFIFO watermark

19.3.2.2 struct usart_transfer_t

Data Fields

- uint8_t * data
  The buffer of data to be transfer.
- size_t dataSize
  The byte count to be transfer.

19.3.2.2.0.30 Field Documentation

19.3.2.2.0.30.1 uint8_t* usart_transfer_t::data

19.3.2.2.0.30.2 size_t usart_transfer_t::dataSize

19.3.2.3 struct _usart_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.
- usart_transfer_callback_t callback
  Callback function.
- void * userData
  USART callback function parameter.
- volatile uint8_t txState
  TX transfer state.
- volatile uint8_t rxState
  RX transfer state.
- usart_txfifo_watermark_t txWatermark
  txFIFO watermark
- usart_rxfifo_watermark_t rxWatermark
  rxFIFO watermark

19.3.2.3.0.31 Field Documentation

19.3.2.3.0.31.1 uint8_t* volatile usart_handle_t::txData

19.3.2.3.0.31.2 volatile size_t usart_handle_t::txDataSize

19.3.2.3.0.31.3 size_t usart_handle_t::txDataSizeAll

19.3.2.3.0.31.4 uint8_t* volatile usart_handle_t::rxData

19.3.2.3.0.31.5 volatile size_t usart_handle_t::rxDataSize

19.3.2.3.0.31.6 size_t usart_handle_t::rxDataSizeAll

19.3.2.3.0.31.7 uint8_t* usart_handle_t::rxRingBuffer

19.3.2.3.0.31.8 size_t usart_handle_t::rxRingBufferSize

19.3.2.3.0.31.9 volatile uint16_t usart_handle_t::rxRingBufferHead

19.3.2.3.0.31.10 volatile uint16_t usart_handle_t::rxRingBufferTail

19.3.2.3.0.31.11 usart_transfer_callback_t usart_handle_t::callback

19.3.2.3.0.31.12 void* usart_handle_t::userData

19.3.2.3.0.31.13 volatile uint8_t usart_handle_t::txState

19.3.3 Macro Definition Documentation

19.3.3.1 #define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

19.3.4 Typedef Documentation

19.3.4.1 typedef void(* usart_transfer_callback_t)(USART_Type *base, usart_handle_t *handle, status_t status, void *userData)

19.3.5 Enumeration Type Documentation

19.3.5.1 enum _usart_status

Enumerator

kStatus_USART_TxBusy Transmitter is busy.
kStatus_USART_RxBusy Receiver is busy.
kStatus_USART_TxIdle USART transmitter is idle.
kStatus_USART_RxIdle USART receiver is idle.
kStatus_USART_TxError Error happens on txFIFO.

kStatus_USART_RxError Error happens on rxFIFO.
kStatus_USART_RxRingBufferOverrun Error happens on rx ring buffer.
kStatus_USART_NoiseError USART noise error.
kStatus_USART_FramingError USART framing error.
kStatus_USART_ParityError USART parity error.
kStatus_USART_BaudrateNotSupport Baudrate is not support in current clock source.

19.3.5.2 enum usart_parity_mode_t

Enumerator

kUSART_ParityDisabled Parity disabled.
kUSART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kUSART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

19.3.5.3 enum usart_stop_bit_count_t

Enumerator

kUSART_OneStopBit One stop bit.
kUSART_TwoStopBit Two stop bits.

19.3.5.4 enum usart_data_len_t

Enumerator

kUSART_7BitsPerChar Seven bit mode.
kUSART_8BitsPerChar Eight bit mode.

19.3.5.5 enum usart_txfifo_watermark_t

Enumerator

kUSART_TxFifo0 USART tx watermark is empty.
kUSART_TxFifo1 USART tx watermark at 1 item.
kUSART_TxFifo2 USART tx watermark at 2 items.
kUSART_TxFifo3 USART tx watermark at 3 items.
kUSART_TxFifo4 USART tx watermark at 4 items.
kUSART_TxFifo5 USART tx watermark at 5 items.
kUSART_TxFifo6 USART tx watermark at 6 items.
kUSART_TxFifo7 USART tx watermark at 7 items.

19.3.5.6 enum usart_rxfifo_watermark_t

Enumerator

kUSART_RxFifo1 USART rx watermark at 1 item.
kUSART_RxFifo2 USART rx watermark at 2 items.
kUSART_RxFifo3 USART rx watermark at 3 items.
kUSART_RxFifo4 USART rx watermark at 4 items.
kUSART_RxFifo5 USART rx watermark at 5 items.
kUSART_RxFifo6 USART rx watermark at 6 items.
kUSART_RxFifo7 USART rx watermark at 7 items.
kUSART_RxFifo8 USART rx watermark at 8 items.

19.3.5.7 enum _usart_flags

This provides constants for the USART status flags for use in the USART functions.

Enumerator

kUSART_TxError TEERR bit, sets if TX buffer is error.
kUSART_RxError RXERR bit, sets if RX buffer is error.
kUSART_TxFifoEmptyFlag TXEMPTY bit, sets if TX buffer is empty.
kUSART_TxFifoNotFullFlag TXNOTFULL bit, sets if TX buffer is not full.
kUSART_RxFifoNotEmptyFlag RXNOEMPTY bit, sets if RX buffer is not empty.
kUSART_RxFifoFullFlag RXFULL bit, sets if RX buffer is full.

19.3.6 Function Documentation

19.3.6.1 uint32_t USART_GetInstance (USART_Type * *base*)

19.3.6.2 status_t USART_Init (USART_Type * *base*, const usart_config_t * *config*, uint32_t *srcClock_Hz*)

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the USART_GetDefaultConfig() function. Example below shows how to use this API to configure USART.

```
*  usart_config_t usartConfig;
*  usartConfig.baudRate_Bps = 115200U;
*  usartConfig.parityMode = kUSART_ParityDisabled;
*  usartConfig.stopBitCount = kUSART_OneStopBit;
*  USART_Init(USART1, &usartConfig, 20000000U);
*
```

Parameters

base	USART peripheral base address.
config	Pointer to user-defined configuration structure.
srcClock_Hz	USART clock source frequency in HZ.

Return values

kStatus_USART_BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_InvalidArgument	USART base address is not valid
kStatus_Success	Status USART initialize succeed

19.3.6.3 void USART_Deinit (USART_Type * *base*)

This function waits for TX complete, disables TX and RX, and disables the USART clock.

Parameters

base	USART peripheral base address.

19.3.6.4 void USART_GetDefaultConfig (usart_config_t * *config*)

This function initializes the USART configuration structure to a default value. The default values are: usartConfig->baudRate_Bps = 115200U; usartConfig->parityMode = kUSART_ParityDisabled; usartConfig->stopBitCount = kUSART_OneStopBit; usartConfig->bitCountPerChar = kUSART_8BitsPerChar; usartConfig->loopback = false; usartConfig->enableTx = false; usartConfig->enableRx = false;

Parameters

config	Pointer to configuration structure.

19.3.6.5 status_t USART_SetBaudRate (USART_Type * *base*, uint32_t *baudrate_Bps*, uint32_t *srcClock_Hz*)

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the USART_Init.

```
*  USART_SetBaudRate(USART1, 115200U, 20000000U);
*
```

Parameters

base	USART peripheral base address.
baudrate_Bps	USART baudrate to be set.
srcClock_Hz	USART clock source freqency in HZ.

Return values

kStatus_USART_BaudrateNotSupport	Baudrate is not support in current clock source.
kStatus_Success	Set baudrate succeed.
kStatus_InvalidArgument	One or more arguments are invalid.

19.3.6.6 static uint32_t USART_GetStatusFlags (USART_Type * *base*) [inline], [static]

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators _usart_flags. To check a specific status, compare the return value with enumerators in _usart_flags. For example, to check whether the TX is empty:

```
*     if (kUSART_TxFifoNotFullFlag &
      USART_GetStatusFlags(USART1))
*     {
*         ...
*     }
*
```

Parameters

base	USART peripheral base address.

Returns

USART status flags which are ORed by the enumerators in the _usart_flags.

19.3.6.7 static void USART_ClearStatusFlags (USART_Type * *base*, uint32_t *mask*) [inline], [static]

This function clear supported USART status flags Flags that can be cleared or set are: kUSART_TxError kUSART_RxError For example:

```
*     USART_ClearStatusFlags(USART1, kUSART_TxError |
      kUSART_RxError)
*
```

Parameters

base	USART peripheral base address.
mask	status flags to be cleared.

19.3.6.8 static void USART_EnableInterrupts (USART_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See _usart_interrupt_enable. For example, to enable TX empty interrupt and RX full interrupt:

```
*     USART_EnableInterrupts(USART1, kUSART_TxLevelInterruptEnable |
      kUSART_RxLevelInterruptEnable);
*
```

Parameters

base	USART peripheral base address.
mask	The interrupts to enable. Logical OR of _usart_interrupt_enable.

19.3.6.9 static void USART_DisableInterrupts (USART_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See _usart_interrupt_enable. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
*     USART_DisableInterrupts(USART1, kUSART_TxLevelInterruptEnable |
      kUSART_RxLevelInterruptEnable);
*
```

Parameters

base	USART peripheral base address.
mask	The interrupts to disable. Logical OR of _usart_interrupt_enable.

19.3.6.10 static void USART_WriteByte (USART_Type * *base*, uint8_t *data*) [inline], [static]

This function writes data to the txFIFO directly. The upper layer must ensure that txFIFO has space for data to write before calling this function.

Parameters

base	USART peripheral base address.
data	The byte to write.

19.3.6.11 static uint8_t USART_ReadByte (USART_Type * *base*) [inline], [static]

This function reads data from the rxFIFO directly. The upper layer must ensure that the rxFIFO is not empty before calling this function.

Parameters

base	USART peripheral base address.

Returns

The byte read from USART data register.

19.3.6.12 void USART_WriteBlocking (USART_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.

Parameters

base	USART peripheral base address.
data	Start address of the data to write.
length	Size of the data to write.

19.3.6.13 status_t USART_ReadBlocking (USART_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	USART peripheral base address.
data	Start address of the buffer to store the received data.
length	Size of the buffer.

Return values

kStatus_USART_FramingError	Receiver overrun happened while receiving data.
kStatus_USART_ParityError	Noise error happened while receiving data.
kStatus_USART_NoiseError	Framing error happened while receiving data.
kStatus_USART_RxError	Overflow or underflow rxFIFO happened.
kStatus_Success	Successfully received all data.

19.3.6.14 status_t USART_TransferCreateHandle (USART_Type * *base*, usart_handle_t * *handle*, usart_transfer_callback_t *callback*, void * *userData*)

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

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

19.3.6.15 status_t USART_TransferSendNonBlocking (USART_Type * *base*, usart_handle_t * *handle*, usart_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 IRQ handler, the USART driver calls the callback function and passes the kStatus_USART_TxIdle as status parameter.

Note

The kStatus_USART_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 kUSART-_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.
xfer	USART transfer structure. See usart_transfer_t.

Return values

kStatus_Success	Successfully start the data transmission.
kStatus_USART_TxBusy	Previous transmission still not finished, data not all written to TX register yet.
kStatus_InvalidArgument	Invalid argument.

19.3.6.16 void USART_TransferStartRingBuffer (USART_Type * *base*, usart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

This function sets up the RX ring buffer to a specific USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the USART_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	USART peripheral base address.
handle	USART 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.

19.3.6.17 void USART_TransferStopRingBuffer (USART_Type * *base,* usart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

19.3.6.18 void USART_TransferAbortSend (USART_Type * *base,* usart_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	USART peripheral base address.
handle	USART handle pointer.

19.3.6.19 status_t USART_TransferGetSendCount (USART_Type * *base,* usart_handle_t * *handle,* uint32_t * *count*)

This function gets the number of bytes that have been written to USART TX register by interrupt method.

Parameters

base	USART peripheral base address.
handle	USART 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`;

19.3.6.20 status_t USART_TransferReceiveNonBlocking (USART_Type * *base*, usart_handle_t * *handle*, usart_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 USART driver. When the new data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter kStatus_USART_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 USART 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	USART peripheral base address.
handle	USART handle pointer.
xfer	USART transfer structure, see usart_transfer_t.
receivedBytes	Bytes received from the ring buffer directly.

Return values

kStatus_Success	Successfully queue the transfer into transmit queue.
kStatus_USART_RxBusy	Previous receive request is not finished.
kStatus_InvalidArgument	Invalid argument.

19.3.6.21 void USART_TransferAbortReceive (USART_Type * *base,* usart_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	USART peripheral base address.
handle	USART handle pointer.

19.3.6.22 status_t USART_TransferGetReceiveCount (USART_Type * *base*, usart_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	USART peripheral base address.
handle	USART 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`;

19.3.6.23 void USART_TransferHandleIRQ (USART_Type * *base*, usart_handle_t * *handle*)

This function handles the USART transmit and receive IRQ request.

Parameters

base	USART peripheral base address.
handle	USART handle pointer.

19.4 USART DMA Driver

19.4.1 Overview

Files

- file fsl_usart_dma.h

Data Structures

- struct usart_dma_handle_t
  UART DMA handle. More...

Typedefs

- typedef void(∗ usart_dma_transfer_callback_t)(USART_Type ∗base, usart_dma_handle_t ∗handle, status_t status, void ∗userData)
  UART transfer callback function.

DMA transactional

- status_t USART_TransferCreateHandleDMA (USART_Type ∗base, usart_dma_handle_t ∗handle, usart_dma_transfer_callback_t callback, void ∗userData, dma_handle_t ∗txDmaHandle, dma_handle_t ∗rxDmaHandle)
  Initializes the USART handle which is used in transactional functions.
- status_t USART_TransferSendDMA (USART_Type ∗base, usart_dma_handle_t ∗handle, usart_transfer_t ∗xfer)
  Sends data using DMA.
- status_t USART_TransferReceiveDMA (USART_Type ∗base, usart_dma_handle_t ∗handle, usart_transfer_t ∗xfer)
  Receives data using DMA.
- void USART_TransferAbortSendDMA (USART_Type ∗base, usart_dma_handle_t ∗handle)
  Aborts the sent data using DMA.
- void USART_TransferAbortReceiveDMA (USART_Type ∗base, usart_dma_handle_t ∗handle)
  Aborts the received data using DMA.
- status_t USART_TransferGetReceiveCountDMA (USART_Type ∗base, usart_dma_handle_t ∗handle, uint32_t ∗count)
  Get the number of bytes that have been received.

19.4.2 Data Structure Documentation

19.4.2.1 struct _usart_dma_handle

Data Fields

- USART_Type * base
  UART peripheral base address.
- usart_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.

19.4.2.1.0.32 Field Documentation

19.4.2.1.0.32.1 USART_Type* usart_dma_handle_t::base

19.4.2.1.0.32.2 usart_dma_transfer_callback_t usart_dma_handle_t::callback

19.4.2.1.0.32.3 void* usart_dma_handle_t::userData

19.4.2.1.0.32.4 size_t usart_dma_handle_t::rxDataSizeAll

19.4.2.1.0.32.5 size_t usart_dma_handle_t::txDataSizeAll

19.4.2.1.0.32.6 dma_handle_t* usart_dma_handle_t::txDmaHandle

19.4.2.1.0.32.7 dma_handle_t* usart_dma_handle_t::rxDmaHandle

19.4.2.1.0.32.8 volatile uint8_t usart_dma_handle_t::txState

19.4.3 Typedef Documentation

19.4.3.1 typedef void(* usart_dma_transfer_callback_t)(USART_Type *base, usart_dma_handle_t *handle, status_t status, void *userData)

19.4.4 Function Documentation

19.4.4.1 status_t USART_TransferCreateHandleDMA (USART_Type * *base*, usart_dma_handle_t * *handle*, usart_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *txDmaHandle*, dma_handle_t * *rxDmaHandle*)

Parameters

base	USART peripheral base address.
handle	Pointer to usart_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.

19.4.4.2 status_t USART_TransferSendDMA (USART_Type * *base*, usart_dma_handle_t * *handle*, usart_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	USART peripheral base address.
handle	USART handle pointer.
xfer	USART DMA transfer structure. See usart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_USART_TxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

19.4.4.3 status_t USART_TransferReceiveDMA (USART_Type * *base*, usart_dma_handle_t * *handle*, usart_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	USART peripheral base address.
handle	Pointer to usart_dma_handle_t structure.
xfer	USART DMA transfer structure. See usart_transfer_t.

Return values

kStatus_Success	if succeed, others failed.
kStatus_USART_RxBusy	Previous transfer on going.
kStatus_InvalidArgument	Invalid argument.

19.4.4.4 void USART_TransferAbortSendDMA (USART_Type * *base*, usart_dma_handle_t * *handle*)

This function aborts send data using DMA.

Parameters

base	USART peripheral base address
handle	Pointer to usart_dma_handle_t structure

19.4.4.5 void USART_TransferAbortReceiveDMA (USART_Type * *base*, usart_dma_handle_t * *handle*)

This function aborts the received data using DMA.

Parameters

base	USART peripheral base address
handle	Pointer to usart_dma_handle_t structure

19.4.4.6 status_t USART_TransferGetReceiveCountDMA (USART_Type * *base*, usart_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been received.

Parameters

base	USART peripheral base address.
handle	USART 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`;

19.5 USART FreeRTOS Driver

19.5.1 Overview

Files

- file fsl_usart_freertos.h

Data Structures

- struct rtos_usart_config
  FLEX USART configuration structure. More...
- struct usart_rtos_handle_t
  FLEX USART FreeRTOS handle. More...

USART RTOS Operation

- int USART_RTOS_Init (usart_rtos_handle_t *handle, usart_handle_t *t_handle, const struct rtos_usart_config *cfg)
  Initializes a USART instance for operation in RTOS.
- int USART_RTOS_Deinit (usart_rtos_handle_t *handle)
  Deinitializes a USART instance for operation.

USART transactional Operation

- int USART_RTOS_Send (usart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
  Sends data in the background.
- int USART_RTOS_Receive (usart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
  Receives data.

19.5.2 Data Structure Documentation

19.5.2.1 struct rtos_usart_config

Data Fields

- USART_Type * base
  USART base address.
- uint32_t srcclk
  USART source clock in Hz.
- uint32_t baudrate
  Desired communication speed.
- usart_parity_mode_t parity

Parity setting.
- usart_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.

19.5.2.2 struct usart_rtos_handle_t

Data Fields

- USART_Type * base
  USART base address.
- usart_transfer_t txTransfer
  TX transfer structure.
- usart_transfer_t rxTransfer
  RX transfer structure.
- SemaphoreHandle_t rxSemaphore
  RX semaphore for resource sharing.
- SemaphoreHandle_t txSemaphore
  TX semaphore for resource sharing.
- EventGroupHandle_t rxEvent
  RX completion event.
- EventGroupHandle_t txEvent
  TX completion event.
- void * t_state
  Transactional state of the underlying driver.

19.5.3 Function Documentation

19.5.3.1 int USART_RTOS_Init (usart_rtos_handle_t * *handle*, usart_handle_t * *t_handle*, const struct rtos_usart_config * *cfg*)

Parameters

handle	The RTOS USART 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 USART after initialization.

Returns

0 succeed, others fail.

19.5.3.2 int USART_RTOS_Deinit (usart_rtos_handle_t * *handle*)

This function deinitializes the USART module, sets all register values to reset value, and releases the resources.

Parameters

handle	The RTOS USART handle.

19.5.3.3 int USART_RTOS_Send (usart_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 USART handle.
buffer	The pointer to buffer to send.
length	The number of bytes to send.

19.5.3.4 int USART_RTOS_Receive (usart_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

This function receives data from USART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

Parameters

handle	The RTOS USART handle.
buffer	The pointer to buffer where to write received data.
length	The number of bytes to receive.
received	The pointer to a variable of size_t where the number of received data is filled.

Chapter 20
FMC: Hardware flash signature generator

20.1 Overview

The KSDK provides a peripheral driver for the Flash Signature generator module of LPC devices.

The flash module contains a built-in signature generator. This generator can produce a 128-bit signature from a range of flash memory. A typical usage is to verify the flashed contents against a calculated signature (e.g. during programming). The signature generator can also be accessed via an IAP function call or ISP command.

20.2 Generate flash signature

1. FMC_GenerateFlashSignature() function generates flash signature for a specified address range.

This example code shows how to generate 128-bit flash signature using the FMC driver.

```
{
    fmc_config_t config;
    fmc_flash_signature_t hardSignature;

    FMC_GetDefaultConfig(&config);
    FMC_Init(FMC, &config);

    FMC_GenerateFlashSignature(FMC, startAddress, length, &hardSignature);

    /* print data. */

    PRINTF(" Generate hardware signature: 0x%x %x %x %x\r\n", hardSignature.word3, hardSignature.word2,
      hardSignature.word1, hardSignature.word0);
}
```

Modules

- Fmc_driver

Functions

- void FMC_Init (FMC_Type *base, fmc_config_t *config)
  Initialize FMC module.
- void FMC_Deinit (FMC_Type *base)
  Deinit FMC module.
- void FMC_GetDefaultConfig (fmc_config_t *config)
  Provides default configuration for fmc module.
- void FMC_GenerateFlashSignature (FMC_Type *base, uint32_t startAddress, uint32_t length, fmc_flash_signature_t *flashSignature)
  Generate hardware flash signature.

Driver version

- #define FSL_FMC_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 0U))
  Driver version 2.0.0.

20.3 Macro Definition Documentation

20.3.1 #define FSL_FMC_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 0U))

20.4 Function Documentation

20.4.1 void FMC_Init (FMC_Type * *base*, fmc_config_t * *config*)

This function initialize FMC module with user configuration

Parameters

base	The FMC peripheral base address.
config	pointer to user configuration structure.

20.4.2 void FMC_Deinit (FMC_Type * *base*)

This function De-initialize FMC module.

Parameters

base	The FMC peripheral base address.

20.4.3 void FMC_GetDefaultConfig (fmc_config_t * *config*)

This function provides default configuration for fmc module, the default wait states value is 5.

Parameters

config	pointer to user configuration structure.

20.4.4 void FMC_GenerateFlashSignature (FMC_Type * *base*, uint32_t *startAddress*, uint32_t *length*, fmc_flash_signature_t * *flashSignature*)

This function generates hardware flash signature for specified address range.

Note

This function needs to be excuted out of flash memory.

Parameters

base	The FMC peripheral base address.
startAddress	Flash start address for signature generation.
length	Length of address range.
flashSignature	Pointer which stores the generated flash signarue.

Chapter 21
FMEAS: Frequency Measure Driver

21.1 Overview

The SDK provides a peripheral driver for the Frequency Measure function of LPC devices' SYSCON module.

It measures frequency of any on-chip or off-chip clock signal. The more precise and higher accuracy clock is selected as a reference clock. The resulting frequency is internally computed from the ratio of value of selected target and reference clock counters.

21.2 Frequency Measure Driver operation

INPUTMUX_AttachSignal() function has to be used to select reference and target clock signal sources.

FMEAS_StartMeasure() function starts the measurement cycle.

FMEAS_IsMeasureComplete() can be polled to check if the measurement cycle has finished.

FMEAS_GetFrequency() returns the frequency of the target clock. Frequency of the reference clock has to be provided as a parameter.

21.3 Typical use case

```
uint32_t freqRef = ...;
uint32_t freq;

/* Setup reference clock */
INPUTMUX_AttachSignal(INPUTMUX, EXAMPLE_REFERENCE_CLOCK_REGISTRY_INDEX,
      EXAMPLE_REFERENCE_CLOCK);

/* Setup to measure the selected target */
INPUTMUX_AttachSignal(INPUTMUX, EXAMPLE_TARGET_CLOCK_REGISTRY_INDEX,
      EXAMPLE_TARGET_CLOCK);

/* Start a measurement cycle and wait for it to complete. If the target
   clock is not running, the measurement cycle will remain active
   forever, so a timeout may be necessary if the target clock can stop. */
FMEAS_StartMeasure(SYSCON);
while (!FMEAS_IsMeasureComplete(SYSCON)) {}

/* Get computed frequency */
freq = FMEAS_GetFrequency(SYSCON, freqRef);
```

Files

- file fsl_fmeas.h

Driver version

- #define FSL_FMEAS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Defines LPC Frequency Measure driver version 2.0.0.

FMEAS Functional Operation

- static void FMEAS_StartMeasure (SYSCON_Type ∗base)
  Starts a frequency measurement cycle.
- static bool FMEAS_IsMeasureComplete (SYSCON_Type ∗base)
  Indicates when a frequency measurement cycle is complete.
- uint32_t FMEAS_GetFrequency (SYSCON_Type ∗base, uint32_t refClockRate)
  Returns the computed value for a frequency measurement cycle.

21.4 Macro Definition Documentation

21.4.1 #define FSL_FMEAS_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Change log:

- Version 2.0.0
  - initial version

21.5 Function Documentation

21.5.1 static void FMEAS_StartMeasure (SYSCON_Type ∗ *base*) [inline], [static]

Parameters

base	: SYSCON peripheral base address.

21.5.2 static bool FMEAS_IsMeasureComplete (SYSCON_Type ∗ *base*) [inline], [static]

Parameters

base	: SYSCON peripheral base address.

Returns

true if a measurement cycle is active, otherwise false.

21.5.3 uint32_t FMEAS_GetFrequency (SYSCON_Type ∗ *base,* uint32_t *refClockRate*)

Parameters

base	: SYSCON peripheral base address.
refClockRate	: Reference clock rate used during the frequency measurement cycle.

Returns

Frequency in Hz.

Chapter 22
GINT: Group GPIO Input Interrupt Driver

22.1 Overview

The SDK provides a driver for the Group GPIO Input Interrupt (GINT).

It can configure one or more pins to generate a group interrupt when the pin conditions are met. The pins do not have to be configured as gpio pins.

22.2 Group GPIO Input Interrupt Driver operation

GINT_SetCtrl() and GINT_ConfigPins() functions configure the pins.

GINT_EnableCallback() function enables the callback functionality. Callback function is called when the pin conditions are met.

22.3 Typical use case

```
void gint_callback(void)
{
    /* Take action for gint event */;
}

    /* Initialize GINT */
    GINT_Init(GINT0);

    /* Setup GINT for edge trigger, "OR" mode. */
    GINT_SetCtrl(GINT0, kGINT_CombOr, kGINT_TrigEdge, gint_callback);

    /* Select pins & polarity for GINT0 */
    GINT_ConfigPins(GINT0, GINT_PORT, GINT_POL_MASK, GINT_ENA_MASK);

    /* Enable callback for GINT */
    GINT_EnableCallback(GINT0);
```

Files

- file fsl_gint.h

Typedefs

- typedef void(* gint_cb_t)(void)
  GINT Callback function.

Enumerations

- enum gint_comb_t {
  kGINT_CombineOr = 0U,
  kGINT_CombineAnd = 1U }
  GINT combine inputs type.

Enumeration Type Documentation

- enum gint_trig_t {
  kGINT_TrigEdge = 0U,
  kGINT_TrigLevel = 1U }
  GINT trigger type.

Functions

- void GINT_Init (GINT_Type ∗base)
  Initialize GINT peripheral.
- void GINT_SetCtrl (GINT_Type ∗base, gint_comb_t comb, gint_trig_t trig, gint_cb_t callback)
  Setup GINT peripheral control parameters.
- void GINT_GetCtrl (GINT_Type ∗base, gint_comb_t ∗comb, gint_trig_t ∗trig, gint_cb_t ∗callback)
  Get GINT peripheral control parameters.
- void GINT_ConfigPins (GINT_Type ∗base, gint_port_t port, uint32_t polarityMask, uint32_t enableMask)
  Configure GINT peripheral pins.
- void GINT_GetConfigPins (GINT_Type ∗base, gint_port_t port, uint32_t ∗polarityMask, uint32_t ∗enableMask)
  Get GINT peripheral pin configuration.
- void GINT_EnableCallback (GINT_Type ∗base)
  Enable callback.
- void GINT_DisableCallback (GINT_Type ∗base)
  Disable callback.
- static void GINT_ClrStatus (GINT_Type ∗base)
  Clear GINT status.
- static uint32_t GINT_GetStatus (GINT_Type ∗base)
  Get GINT status.
- void GINT_Deinit (GINT_Type ∗base)
  Deinitialize GINT peripheral.

Driver version

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

22.4 Macro Definition Documentation

22.4.1 #define FSL_GINT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

22.5 Typedef Documentation

22.5.1 typedef void(∗ gint_cb_t)(void)

22.6 Enumeration Type Documentation

22.6.1 enum gint_comb_t

Enumerator

kGINT_CombineOr A grouped interrupt is generated when any one of the enabled inputs is active.

kGINT_CombineAnd A grouped interrupt is generated when all enabled inputs are active.

22.6.2 enum gint_trig_t

Enumerator

kGINT_TrigEdge Edge triggered based on polarity.
kGINT_TrigLevel Level triggered based on polarity.

22.7 Function Documentation

22.7.1 void GINT_Init (GINT_Type * *base*)

This function initializes the GINT peripheral and enables the clock.

Parameters

base	Base address of the GINT peripheral.

Return values

None.	

22.7.2 void GINT_SetCtrl (GINT_Type * *base,* gint_comb_t *comb,* gint_trig_t *trig,* gint_cb_t *callback*)

This function sets the control parameters of GINT peripheral.

Parameters

base	Base address of the GINT peripheral.
comb	Controls if the enabled inputs are logically ORed or ANDed for interrupt generation.
trig	Controls if the enabled inputs are level or edge sensitive based on polarity.
callback	This function is called when configured group interrupt is generated.

Return values

None.	

22.7.3 void GINT_GetCtrl (GINT_Type * *base*, gint_comb_t * *comb*, gint_trig_t * *trig*, gint_cb_t * *callback*)

This function returns the control parameters of GINT peripheral.

Parameters

base	Base address of the GINT peripheral.
comb	Pointer to store combine input value.
trig	Pointer to store trigger value.
callback	Pointer to store callback function.

Return values

None.	

22.7.4 void GINT_ConfigPins (GINT_Type * *base*, gint_port_t *port*, uint32_t *polarityMask*, uint32_t *enableMask*)

This function enables and controls the polarity of enabled pin(s) of a given port.

Parameters

base	Base address of the GINT peripheral.
port	Port number.
polarityMask	Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
enableMask	Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None.	

22.7.5 void GINT_GetConfigPins (GINT_Type * *base*, gint_port_t *port*, uint32_t * *polarityMask*, uint32_t * *enableMask*)

This function returns the pin configuration of a given port.

Parameters

base	Base address of the GINT peripheral.
port	Port number.
polarityMask	Pointer to store the polarity mask Each bit position indicates the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
enableMask	Pointer to store the enable mask. Each bit position indicates if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None.	

22.7.6 void GINT_EnableCallback (GINT_Type * *base*)

This function enables the interrupt for the selected GINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

base	Base address of the GINT peripheral.

Return values

None.	

22.7.7 void GINT_DisableCallback (GINT_Type * *base*)

This function disables the interrupt for the selected GINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

base	Base address of the peripheral.

Return values

None.	

22.7.8 static void GINT_ClrStatus (GINT_Type * *base*) [inline], [static]

This function clears the GINT status bit.

Parameters

base	Base address of the GINT peripheral.

Return values

None.	

22.7.9 static uint32_t GINT_GetStatus (GINT_Type * *base*) [inline], [static]

This function returns the GINT status.

Parameters

base	Base address of the GINT peripheral.

Return values

status	= 0 No group interrupt request. = 1 Group interrupt request active.

22.7.10 void GINT_Deinit (GINT_Type * *base*)

This function disables the GINT clock.

Parameters

base	Base address of the GINT peripheral.

Return values

None.	

Chapter 23
GPIO: General Purpose I/O

23.1 Overview

The SDK provides Peripheral driver for the General Purpose I/O (GPIO) module of LPC devices.

23.2 Function groups

23.2.1 Initialization and deinitialization

The function GPIO_PinInit() initializes the GPIO with specified configuration.

23.2.2 Pin manipulation

The function GPIO_WritePinOutput() set output state of selected GPIO pin. The function GPIO_ReadPinInput() read input value of selected GPIO pin.

23.2.3 Port manipulation

The function GPIO_SetPinsOutput() sets the output level of selected GPIO pins to the logic 1. The function GPIO_ClearPinsOutput() sets the output level of selected GPIO pins to the logic 1. The function GPIO_TogglePinsOutput() reverse the output level of selected GPIO pins. The function GPIO_ReadPinsInput() read input value of selected port.

23.2.4 Port masking

The function GPIO_SetPortMask() set port mask, only pins masked by 0 will be enabled in following functions. The function GPIO_WriteMPort() sets the state of selected GPIO port, only pins masked by 0 will be affected. The function GPIO_ReadMPort() reads the state of selected GPIO port, only pins masked by 0 are enabled for read, pins masked by 1 are read as 0.

23.3 Typical use case

Example use of GPIO API.

```
int main(void)
{
    uint32_t port_state = 0;

    /* Define the init structure for the output LED pin*/
```

Typical use case

```
    gpio_pin_config_t led_config = {
        kGPIO_DigitalOutput, 0,
    };

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();


    /* Init output LED GPIO. */
    GPIO_PinInit(GPIO, BOARD_LED_GREEN_GPIO_PORT, BOARD_LED_GREEN_GPIO_PIN, &led_config);
    GPIO_WritePinOutput(GPIO, BOARD_LED_GREEN_GPIO_PORT, BOARD_LED_GREEN_GPIO_PIN, 1);

    GPIO_PinInit(GPIO, BOARD_LED_GPIO_PORT, BOARD_LED_GPIO_PIN, &led_config);
    GPIO_WritePinOutput(GPIO, BOARD_LED_GPIO_PORT, BOARD_LED_GPIO_PIN, 1);

    GPIO_PinInit(GPIO, BOARD_LED_BLUE_GPIO_PORT, BOARD_LED_BLUE_GPIO_PIN, &led_config);
    GPIO_WritePinOutput(GPIO, BOARD_LED_BLUE_GPIO_PORT, BOARD_LED_BLUE_GPIO_PIN, 1);

    GPIO_ClearPinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN | 1 <<
      BOARD_LED_BLUE_GPIO_PIN);
    GPIO_SetPinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN | 1 <<
      BOARD_LED_BLUE_GPIO_PIN);

    GPIO_ClearPinsOutput(GPIO, 1, 1 << BOARD_LED_BLUE_GPIO_PIN);
    GPIO_SetPinsOutput(GPIO, 1, 1 << BOARD_LED_BLUE_GPIO_PIN);

    GPIO_TogglePinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN | 1 <<
      BOARD_LED_BLUE_GPIO_PIN);
    GPIO_TogglePinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN | 1 <<
      BOARD_LED_BLUE_GPIO_PIN);

    GPIO_TogglePinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN);
    GPIO_TogglePinsOutput(GPIO, 1, 1 << BOARD_LED_GREEN_GPIO_PIN);

    GPIO_TogglePinsOutput(GPIO, BOARD_LED_GPIO_PORT, 1 << BOARD_LED_GPIO_PIN);
    GPIO_TogglePinsOutput(GPIO, BOARD_LED_GPIO_PORT, 1 << BOARD_LED_GPIO_PIN);

    /* Port masking */
    GPIO_SetPortMask(GPIO, BOARD_LED_GPIO_PORT, 0x0000ffff);
    GPIO_WriteMPort(GPIO, BOARD_LED_GPIO_PORT, 0xffffffff);
    port_state = GPIO_ReadPinsInput(GPIO, 0);
    port_state = GPIO_ReadMPort(GPIO, 0);

    while (1)
    {
        port_state = GPIO_ReadPinsInput(GPIO, 0);
        if (!(port_state & (1 << BOARD_SW1_GPIO_PIN)))
        {
            GPIO_TogglePinsOutput(GPIO, BOARD_LED_GPIO_PORT, 1u << BOARD_LED_GPIO_PIN)
      ;
        }

        if (!GPIO_ReadPinInput(GPIO, BOARD_SW2_GPIO_PORT, BOARD_SW2_GPIO_PIN))
        {
            GPIO_TogglePinsOutput(GPIO, BOARD_LED_GREEN_GPIO_PORT, 1u <<
      BOARD_LED_GREEN_GPIO_PIN);
        }
        delay();
    }
}
```

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 }
  LPC GPIO direction definition.

Functions

- static void GPIO_SetPinsOutput (GPIO_Type *base, uint32_t port, 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 port, 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 port, uint32_t mask)
  Reverses current output logic of the multiple GPIO pins.

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LPC GPIO driver version 2.0.0.

GPIO Configuration

- void GPIO_PinInit (GPIO_Type *base, uint32_t port, 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 port, uint32_t pin, uint8_t output)
  Sets the output level of the one GPIO pin to the logic 1 or 0.

GPIO Input Operations

- static uint32_t GPIO_ReadPinInput (GPIO_Type *base, uint32_t port, uint32_t pin)
  Reads the current input value of the GPIO PIN.

23.4 Data Structure Documentation

23.4.1 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.

Data Fields

- gpio_pin_direction_t pinDirection
  GPIO direction, input or output.
- uint8_t outputLogic
  Set default output logic, no use in input.

23.5 Macro Definition Documentation

23.5.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

23.6 Enumeration Type Documentation

23.6.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.
kGPIO_DigitalOutput Set current pin as digital output.

23.7 Function Documentation

23.7.1 void GPIO_PinInit (GPIO_Type * *base*, uint32_t *port*, 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(Typically GPIO)
port	GPIO port number
pin	GPIO pin number
config	GPIO pin configuration pointer

23.7.2 static void GPIO_WritePinOutput (GPIO_Type * *base*, uint32_t *port*, uint32_t *pin*, uint8_t *output*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
pin	GPIO pin number
output	GPIO pin output logic level. • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

23.7.3 static uint32_t GPIO_ReadPinInput (GPIO_Type * *base*, uint32_t *port*, uint32_t *pin*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
pin	GPIO pin number

Return values

GPIO	port input value • 0: corresponding pin input low-logic level. • 1: corresponding pin input high-logic level.

23.7.4 static void GPIO_SetPinsOutput (GPIO_Type * *base,* uint32_t *port,* uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

23.7.5 static void GPIO_ClearPinsOutput (GPIO_Type * *base,* uint32_t *port,* uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

23.7.6 static void GPIO_TogglePinsOutput (GPIO_Type * *base,* uint32_t *port,* uint32_t *mask*) [inline], [static]

Parameters

base	GPIO peripheral base pointer(Typically GPIO)
port	GPIO port number
mask	GPIO pin number macro

Chapter 24
INPUTMUX: Input Multiplexing Driver

24.1 Overview

The SDK provides a driver for the Input multiplexing (INPUTMUX).

It configures the inputs to the pin interrupt block, DMA trigger and the frequency measure function. Once configured the clock is not needed for the inputmux.

24.2 Input Multiplexing Driver operation

INPUTMUX_AttachSignal function configures the specified input

24.3 Typical use case

```
INPUTMUX_Init(INPUTMUX);
INPUTMUX_AttachSignal(INPUT_MUX, kPINT_PinInt0,
      kINPUTMUX_GpioPort0Pin0ToPintsel);
/* Disable clock to save power */
INPUTMUX_Deinit(INPUTMUX)
```

Files

- file fsl_inputmux.h
- file fsl_inputmux_connections.h

Functions

- void INPUTMUX_Init (INPUTMUX_Type *base)
  Initialize INPUTMUX peripheral.
- void INPUTMUX_AttachSignal (INPUTMUX_Type *base, uint32_t index, inputmux_connection_t connection)
  Attaches a signal.
- void INPUTMUX_Deinit (INPUTMUX_Type *base)
  Deinitialize INPUTMUX peripheral.

Driver version

- #define FSL_INPUTMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Group interrupt driver version for SDK.

Input multiplexing connections

- enum inputmux_connection_t {
  kINPUTMUX_SctGpi0ToSct0 = 0U + (SCT0_PMUX_ID << PMUX_SHIFT) ,
  kINPUTMUX_I2sS7clkToSct0 = 24U + (SCT0_PMUX_ID << PMUX_SHIFT) ,
  kINPUTMUX_FreqmeGpioClk_b = 6U + (FREQMEAS_PMUX_ID << PMUX_SHIFT) ,
  kINPUTMUX_GpioPort1Pin31ToPintsel = 63U + (PINTSEL_PMUX_ID << PMUX_SHIFT) ,

kINPUTMUX_Otrig3ToDma = 19U + (DMA_TRIG0_PMUX_ID << PMUX_SHIFT) }
INPUTMUX connections type.
- #define SCT0_PMUX_ID 0x00U
  Periphinmux IDs.
- #define **PINTSEL_PMUX_ID** 0xC0U
- #define **DMA_TRIG0_PMUX_ID** 0xE0U
- #define **DMA_OTRIG_PMUX_ID** 0x160U
- #define **FREQMEAS_PMUX_ID** 0x180U
- #define **PMUX_SHIFT** 20U

24.4 Macro Definition Documentation

24.4.1 #define FSL_INPUTMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

24.5 Enumeration Type Documentation

24.5.1 enum inputmux_connection_t

Enumerator

kINPUTMUX_SctGpi0ToSct0 SCT INMUX.
kINPUTMUX_I2sS7clkToSct0 Frequency measure.
kINPUTMUX_FreqmeGpioClk_b Pin Interrupt.
kINPUTMUX_GpioPort1Pin31ToPintsel DMA ITRIG.
kINPUTMUX_Otrig3ToDma DMA OTRIG.

24.6 Function Documentation

24.6.1 void INPUTMUX_Init (INPUTMUX_Type * *base*)

This function enables the INPUTMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.

Return values

None.	

24.6.2 void INPUTMUX_AttachSignal (INPUTMUX_Type * *base*, uint32_t *index*, inputmux_connection_t *connection*)

This function gates the INPUTPMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.
index	Destination peripheral to attach the signal to.
connection	Selects connection.

Return values

None.	

24.6.3 void INPUTMUX_Deinit (INPUTMUX_Type * *base*)

This function disables the INPUTMUX clock.

Parameters

base	Base address of the INPUTMUX peripheral.

Return values

None.	

Chapter 25
IOCON: I/O pin configuration

25.1 Overview

The SDK provides Peripheral driver for the I/O pin configuration (IOCON) module of LPC devices.

25.2 Function groups

25.2.1 Pin mux set

The function IOCONPinMuxSet() set pinmux for single pin according to selected configuration.

25.2.2 Pin mux set

The function IOCON_SetPinMuxing() set pinmux for group of pins according to selected configuration.

25.3 Typical use case

Example use of IOCON API to selection of GPIO mode.

```
int main(void)
{
    /* enable clock for IOCON */
    CLOCK_EnableClock(kCLOCK_Iocon);

    /* Set pin mux for single pin */
    IOCON_PinMuxSet(IOCON, 0, 29, IOCON_FUNC0 |
      IOCON_GPIO_MODE | IOCON_DIGITAL_EN |
      IOCON_INPFILT_OFF);

    /* Set pin mux for group of pins */
    const iocon_group_t gpio_pins[] = {
    {0, 24, (IOCON_FUNC0 | IOCON_GPIO_MODE |
      IOCON_DIGITAL_EN | IOCON_INPFILT_OFF)},
    {0, 31, (IOCON_FUNC0 | IOCON_GPIO_MODE |
      IOCON_DIGITAL_EN | IOCON_INPFILT_OFF)},
    };

    Chip_IOCON_SetPinMuxing(IOCON, gpio_pins, sizeof(gpio_pins)/sizeof(gpio_pins[0]));

}
```

Files

- file fsl_iocon.h

Data Structures

- struct iocon_group_t
  Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format. More...

Typical use case

Macros

- #define IOCON_FUNC0 0x0
  IOCON function and mode selection definitions.
- #define IOCON_FUNC1 0x1
  Selects pin function 1.
- #define IOCON_FUNC2 0x2
  Selects pin function 2.
- #define IOCON_FUNC3 0x3
  Selects pin function 3.
- #define IOCON_FUNC4 0x4
  Selects pin function 4.
- #define IOCON_FUNC5 0x5
  Selects pin function 5.
- #define IOCON_FUNC6 0x6
  Selects pin function 6.
- #define IOCON_FUNC7 0x7
  Selects pin function 7.
- #define IOCON_MODE_INACT (0x0 << 3)
  No addition pin function.
- #define IOCON_MODE_PULLDOWN (0x1 << 3)
  Selects pull-down function.
- #define IOCON_MODE_PULLUP (0x2 << 3)
  Selects pull-up function.
- #define IOCON_MODE_REPEATER (0x3 << 3)
  Selects pin repeater function.
- #define IOCON_HYS_EN (0x1 << 5)
  Enables hysteresis.
- #define IOCON_GPIO_MODE (0x1 << 5)
  GPIO Mode.
- #define IOCON_I2C_SLEW (0x1 << 5)
  I2C Slew Rate Control.
- #define IOCON_INV_EN (0x1 << 6)
  Enables invert function on input.
- #define IOCON_ANALOG_EN (0x0 << 7)
  Enables analog function by setting 0 to bit 7.
- #define IOCON_DIGITAL_EN (0x1 << 7)
  Enables digital function by setting 1 to bit 7(default)
- #define IOCON_STDI2C_EN (0x1 << 8)
  I2C standard mode/fast-mode.
- #define IOCON_FASTI2C_EN (0x3 << 8)
  I2C Fast-mode Plus and high-speed slave.
- #define IOCON_INPFILT_OFF (0x1 << 8)
  Input filter Off for GPIO pins.
- #define IOCON_INPFILT_ON (0x0 << 8)
  Input filter On for GPIO pins.
- #define IOCON_OPENDRAIN_EN (0x1 << 10)
  Enables open-drain function.
- #define IOCON_S_MODE_0CLK (0x0 << 11)
  Bypass input filter.
- #define IOCON_S_MODE_1CLK (0x1 << 11)
  Input pulses shorter than 1 filter clock are rejected.

- #define IOCON_S_MODE_2CLK (0x2 << 11)
  Input pulses shorter than 2 filter clock2 are rejected.
- #define IOCON_S_MODE_3CLK (0x3 << 11)
  Input pulses shorter than 3 filter clock2 are rejected.
- #define IOCON_S_MODE(clks) ((clks) << 11)
  Select clocks for digital input filter mode.
- #define IOCON_CLKDIV(div) ((div) << 13)
  Select peripheral clock divider for input filter sampling clock, 2^n, n=0-6.

Functions

- __STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t port, uint8_t pin, uint32_t modefunc)
  Sets I/O Control pin mux.
- __STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base, const iocon_group_t *pinArray, uint32_t arrayLength)
  Set all I/O Control pin muxing.

Driver version

- #define LPC_IOCON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  IOCON driver version 2.0.0.

25.4 Data Structure Documentation

25.4.1 struct iocon_group_t

25.5 Macro Definition Documentation

25.5.1 #define LPC_IOCON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

25.5.2 #define IOCON_FUNC0 0x0

Note

See the User Manual for specific modes and functions supported by the various pins.Selects pin function 0

25.6 Function Documentation

25.6.1 __STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type * *base,* uint8_t *port,* uint8_t *pin,* uint32_t *modefunc*)

Function Documentation

Parameters

base	: The base of IOCON peripheral on the chip
port	: GPIO port to mux
pin	: GPIO pin to mux
modefunc	: OR'ed values of type IOCON_∗

Returns

Nothing

25.6.2 __STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type ∗ *base,* const iocon_group_t ∗ *pinArray,* uint32_t *arrayLength*)

Parameters

base	: The base of IOCON peripheral on the chip
pinArray	: Pointer to array of pin mux selections
arrayLength	: Number of entries in pinArray

Returns

Nothing

Chapter 26
LCDC: LCD Controller Driver

26.1 Overview

The KSDK provides a Peripheral driver for the LCD controller (LCD) of LPC devices.

The LCD driver supports TFT and STN panel. It also supports hardware cursor, which makes software easy.

26.2 Typical use case

26.2.1 Update framebuffer dynamically

The function LCDC_SetPanelAddr is used to set the new framebuffer address. After this function, the new framebuffer address is not loaded to current register until the vertical synchroization. When the address is loaded, the interrupt kLCDC_BaseAddrUpdateInterrupt occurs then upper layer could set the new framebuffer. In this example, there are two buffers. When the active buffer is displayed, upper layer could modify the inactive buffer.

```
#if (defined(__CC_ARM) || defined(__GNUC__))
__attribute__((aligned(8)))
#elif defined(__ICCARM__)
#pragma data_alignment = 8
#else
#error Toolchain not support.
#endif
static uint16_t s_frameBufs[2][IMG_HEIGHT][IMG_WIDTH];

/* The index of the inactive buffer. */
static volatile uint8_t s_inactiveBufsIdx;

/* The new frame address already loaded to the LCD controller. */
static volatile bool s_frameAddrUpdated = false;

void LCD_IRQHandler(void)
{
    uint32_t intStatus = LCDC_GetEnabledInterruptsPendingStatus(LCD);

    LCDC_ClearInterruptsStatus(LCD, intStatus);

    if (intStatus & kLCDC_BaseAddrUpdateInterrupt)
    {
        s_frameAddrUpdated = true;
    }
}

/* This function fills the framebuffer. */
static void APP_FillBuffer(void *buffer);

int main(void)
{
    lcdc_config_t lcdConfig;
```

```
    /* Setup the LCD input clock here. */
    BOARD_InitHardware();

    s_frameAddrUpdated = false;

    /* s_frameBufs[0] is displayed first. */
    s_inactiveBufsIdx = 1;

    /* Fill the s_frameBufs[0]. */
    APP_FillBuffer((void *)(s_frameBufs[0]));

    LCDC_GetDefaultConfig(&lcdConfig);

    lcdConfig.panelClock_Hz = LCD_PANEL_CLK;
    lcdConfig.ppl = LCD_PPL;
    lcdConfig.hsw = LCD_HSW;
    lcdConfig.hfp = LCD_HFP;
    lcdConfig.hbp = LCD_HBP;
    lcdConfig.lpp = LCD_LPP;
    lcdConfig.vsw = LCD_VSW;
    lcdConfig.vfp = LCD_VFP;
    lcdConfig.vbp = LCD_VBP;
    lcdConfig.polarityFlags = LCD_POL_FLAGS;
    lcdConfig.upperPanelAddr = (uint32_t)s_frameBufs[0];
    lcdConfig.bpp = kLCDC_16BPP565;
    lcdConfig.display = kLCDC_DisplayTFT;
    lcdConfig.swapRedBlue = false;

    LCDC_Init(LCD, &lcdConfig, LCD_INPUT_CLK_FREQ);

    LCDC_EnableInterrupts(LCD,
      kLCDC_BaseAddrUpdateInterrupt);
    NVIC_EnableIRQ(LCD_IRQn);

    LCDC_Start(LCD);
    LCDC_PowerUp(LCD);

    while (1)
    {
        /*
         * Fill the inactive buffer.
         */
        APP_FillBuffer((void *)s_frameBufs[s_inactiveBufsIdx]);

        while (!s_frameAddrUpdated)
        {
        }
        /*
         * The buffer address has been loaded to the LCD controller, now
         * set the inactive buffer to active buffer.
         */
        LCDC_SetPanelAddr(LCD, kLCDC_UpperPanel, (uint32_t)(s_frameBufs[
      s_inactiveBufsIdx]));

        s_frameAddrUpdated = false;
        s_inactiveBufsIdx ^= 1U;
    }
}
```

26.2.2 Hardware cursor

This example shows how to show a 32x32 pixel cursor and change its position.

```
lcdc_cursor_config_t cursorConfig;
```

```
int32_t cursorPosX = 0;
int32_t cursorPosY = 0;

/* Init the LCD here. */
// ....

/* Setup the Cursor. */
LCDC_CursorGetDefaultConfig(&cursorConfig);

cursorConfig.size = kLCDC_CursorSize32;
cursorConfig.syncMode = kLCDC_CursorSync;
cursorConfig.image[0] = (uint32_t *)cursor32Img0;

LCDC_SetCursorConfig(LCD, &cursorConfig);

LCDC_ChooseCursor(LCD, 0);

LCDC_SetCursorPosition(LCD, 0, 0);

LCDC_EnableCursor(LCD, true);

while (1)
{
    // Do something else here

    // Update cursorPosX and cursorPosY

    LCDC_SetCursorPosition(LCD, cursorPosX, cursorPosY);
}
```

Data Structures

- struct lcdc_config_t
  LCD configuration structure. More...
- struct lcdc_cursor_palette_t
  LCD hardware cursor palette. More...
- struct lcdc_cursor_config_t
  LCD hardware cursor configuration structure. More...

Macros

- #define LCDC_CURSOR_COUNT 4
  How many hardware cursors supports.
- #define LCDC_CURSOR_IMG_BPP 2
  LCD cursor image bits per pixel.
- #define LCDC_CURSOR_IMG_32X32_WORDS (32 * 32 * LCDC_CURSOR_IMG_BPP / (8 * sizeof(uint32_t)))
  LCD 32x32 cursor image size in word(32-bit).
- #define LCDC_CURSOR_IMG_64X64_WORDS (64 * 64 * LCDC_CURSOR_IMG_BPP / (8 * sizeof(uint32_t)))
  LCD 64x64 cursor image size in word(32-bit).
- #define LCDC_PALETTE_SIZE_WORDS (ARRAY_SIZE(((LCD_Type *)0)->PAL))
  LCD palette size in words(32-bit).

Typical use case

Enumerations

- enum _lcdc_polarity_flags {
  kLCDC_InvertVsyncPolarity = LCD_POL_IVS_MASK,
  kLCDC_InvertHsyncPolarity = LCD_POL_IHS_MASK,
  kLCDC_InvertClkPolarity = LCD_POL_IPC_MASK,
  kLCDC_InvertDePolarity = LCD_POL_IOE_MASK }
  LCD sigal polarity flags.
- enum lcdc_bpp_t {
  kLCDC_1BPP = 0U,
  kLCDC_2BPP = 1U,
  kLCDC_4BPP = 2U,
  kLCDC_8BPP = 3U,
  kLCDC_16BPP = 4U,
  kLCDC_24BPP = 5U,
  kLCDC_16BPP565 = 6U,
  kLCDC_12BPP = 7U }
  LCD bits per pixel.
- enum lcdc_display_t {
  kLCDC_DisplayTFT = LCD_CTRL_LCDTFT_MASK,
  kLCDC_DisplaySingleMonoSTN4Bit = LCD_CTRL_LCDBW_MASK,
  kLCDC_DisplaySingleMonoSTN8Bit,
  kLCDC_DisplayDualMonoSTN4Bit,
  kLCDC_DisplayDualMonoSTN8Bit,
  kLCDC_DisplaySingleColorSTN8Bit = 0U,
  kLCDC_DisplayDualColorSTN8Bit = LCD_CTRL_LCDDUAL_MASK }
  The types of display panel.
- enum lcdc_data_format_t {
  kLCDC_LittleEndian = 0U,
  kLCDC_BigEndian = LCD_CTRL_BEPO_MASK | LCD_CTRL_BEBO_MASK,
  kLCDC_WinCeMode = LCD_CTRL_BEPO_MASK }
  LCD panel buffer data format.
- enum lcdc_vertical_compare_interrupt_mode_t {
  kLCDC_StartOfVsync,
  kLCDC_StartOfBackPorch,
  kLCDC_StartOfActiveVideo,
  kLCDC_StartOfFrontPorch }
  LCD vertical compare interrupt mode.
- enum _lcdc_interrupts {
  kLCDC_CursorInterrupt = LCD_CRSR_INTMSK_CRSRIM_MASK,
  kLCDC_FifoUnderflowInterrupt = LCD_INTMSK_FUFIM_MASK,
  kLCDC_BaseAddrUpdateInterrupt = LCD_INTMSK_LNBUIM_MASK,
  kLCDC_VerticalCompareInterrupt = LCD_INTMSK_VCOMPIM_MASK,
  kLCDC_AhbErrorInterrupt = LCD_INTMSK_BERIM_MASK }
  LCD interrupts.
- enum lcdc_panel_t {
  kLCDC_UpperPanel,

kLCDC_LowerPanel }
LCD panel frame.
- enum lcdc_cursor_size_t { kLCDC_CursorSize32, kLCDC_CursorSize64 }
  LCD hardware cursor size.
- enum lcdc_cursor_sync_mode_t { kLCDC_CursorAsync, kLCDC_CursorSync }
  LCD hardware cursor frame synchronization mode.

Variables

- uint32_t lcdc_config_t::panelClock_Hz
  Panel clock in Hz.
- uint16_t lcdc_config_t::ppl
  Pixels per line, it must could be divided by 16.
- uint8_t lcdc_config_t::hsw
  HSYNC pulse width.
- uint8_t lcdc_config_t::hfp
  Horizontal front porch.
- uint8_t lcdc_config_t::hbp
  Horizontal back porch.
- uint16_t lcdc_config_t::lpp
  Lines per panal.
- uint8_t lcdc_config_t::vsw
  VSYNC pulse width.
- uint8_t lcdc_config_t::vfp
  Vrtical front porch.
- uint8_t lcdc_config_t::vbp
  Vertical back porch.
- uint8_t lcdc_config_t::acBiasFreq
  The number of line clocks between AC bias pin toggling.
- uint16_t lcdc_config_t::polarityFlags
  OR'ed value of _lcdc_polarity_flags, used to contol the signal polarity.
- bool lcdc_config_t::enableLineEnd
  Enable line end or not, the line end is a positive pulse with 4 panel clock.
- uint8_t lcdc_config_t::lineEndDelay
  The panel clocks between the last pixel of line and the start of line end.
- uint32_t lcdc_config_t::upperPanelAddr
  LCD upper panel base address, must be double-word(64-bit) align.
- uint32_t lcdc_config_t::lowerPanelAddr
  LCD lower panel base address, must be double-word(64-bit) align.
- lcdc_bpp_t lcdc_config_t::bpp
  LCD bits per pixel.
- lcdc_data_format_t lcdc_config_t::dataFormat
  Data format.
- bool lcdc_config_t::swapRedBlue
  Set true to use BGR format, set false to choose RGB format.
- lcdc_display_t lcdc_config_t::display
  The display type.

Typical use case

- uint8_t lcdc_cursor_palette_t::red
  Red color component.
- uint8_t lcdc_cursor_palette_t::green
  Red color component.
- uint8_t lcdc_cursor_palette_t::blue
  Red color component.
- lcdc_cursor_size_t lcdc_cursor_config_t::size
  Cursor size.
- lcdc_cursor_sync_mode_t lcdc_cursor_config_t::syncMode
  Cursor synchronization mode.
- lcdc_cursor_palette_t lcdc_cursor_config_t::palette0
  Cursor palette 0.
- lcdc_cursor_palette_t lcdc_cursor_config_t::palette1
  Cursor palette 1.
- uint32_t * lcdc_cursor_config_t::image [LCDC_CURSOR_COUNT]
  Pointer to cursor image data.

Driver version

- #define LPC_LCDC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  LCDC driver version 2.0.0.

Initialization and Deinitialization

- status_t LCDC_Init (LCD_Type *base, const lcdc_config_t *config, uint32_t srcClock_Hz)
  Initialize the LCD module.
- void LCDC_Deinit (LCD_Type *base)
  Deinitialize the LCD module.
- void LCDC_GetDefaultConfig (lcdc_config_t *config)
  Gets default pre-defined settings for initial configuration.

Start and stop

- static void LCDC_Start (LCD_Type *base)
  Start to output LCD timing signal.
- static void LCDC_Stop (LCD_Type *base)
  Stop the LCD timing signal.
- static void LCDC_PowerUp (LCD_Type *base)
  Power up the LCD and output the pixel signal.
- static void LCDC_PowerDown (LCD_Type *base)
  Power down the LCD and disable the output pixel signal.

LCD control

- void LCDC_SetPanelAddr (LCD_Type *base, lcdc_panel_t panel, uint32_t addr)
  Sets panel frame base address.
- void LCDC_SetPalette (LCD_Type *base, const uint32_t *palette, uint8_t count_words)
  Sets palette.

Interrupts

- static void LCDC_SetVerticalInterruptMode (LCD_Type *base, lcdc_vertical_compare_interrupt_mode_t mode)
  Sets the vertical compare interrupt mode.
- void LCDC_EnableInterrupts (LCD_Type *base, uint32_t mask)
  Enable LCD interrupts.
- void LCDC_DisableInterrupts (LCD_Type *base, uint32_t mask)
  Disable LCD interrupts.
- uint32_t LCDC_GetInterruptsPendingStatus (LCD_Type *base)
  Get LCD interrupt pending status.
- uint32_t LCDC_GetEnabledInterruptsPendingStatus (LCD_Type *base)
  Get LCD enabled interrupt pending status.
- void LCDC_ClearInterruptsStatus (LCD_Type *base, uint32_t mask)
  Clear LCD interrupts pending status.

Hardware cursor

- void LCDC_SetCursorConfig (LCD_Type *base, const lcdc_cursor_config_t *config)
  Set the hardware cursor configuration.
- void LCDC_CursorGetDefaultConfig (lcdc_cursor_config_t *config)
  Get the hardware cursor default configuration.
- static void LCDC_EnableCursor (LCD_Type *base, bool enable)
  Enable or disable the cursor.
- static void LCDC_ChooseCursor (LCD_Type *base, uint8_t index)
  Choose which cursor to display.
- void LCDC_SetCursorPosition (LCD_Type *base, int32_t positionX, int32_t positionY)
  Set the position of cursor.
- void LCDC_SetCursorImage (LCD_Type *base, lcdc_cursor_size_t size, uint8_t index, const uint32_t *image)
  Set the cursor image.

26.3 Data Structure Documentation

26.3.1 struct lcdc_config_t

Data Fields

- uint32_t panelClock_Hz
  Panel clock in Hz.
- uint16_t ppl
  Pixels per line, it must could be divided by 16.
- uint8_t hsw
  HSYNC pulse width.
- uint8_t hfp
  Horizontal front porch.
- uint8_t hbp
  Horizontal back porch.
- uint16_t lpp
  Lines per panal.

- uint8_t vsw
  VSYNC pulse width.
- uint8_t vfp
  Vrtical front porch.
- uint8_t vbp
  Vertical back porch.
- uint8_t acBiasFreq
  The number of line clocks between AC bias pin toggling.
- uint16_t polarityFlags
  OR'ed value of _lcdc_polarity_flags, used to contol the signal polarity.
- bool enableLineEnd
  Enable line end or not, the line end is a positive pulse with 4 panel clock.
- uint8_t lineEndDelay
  The panel clocks between the last pixel of line and the start of line end.
- uint32_t upperPanelAddr
  LCD upper panel base address, must be double-word(64-bit) align.
- uint32_t lowerPanelAddr
  LCD lower panel base address, must be double-word(64-bit) align.
- lcdc_bpp_t bpp
  LCD bits per pixel.
- lcdc_data_format_t dataFormat
  Data format.
- bool swapRedBlue
  Set true to use BGR format, set false to choose RGB format.
- lcdc_display_t display
  The display type.

26.3.2 struct lcdc_cursor_palette_t

Data Fields

- uint8_t red
  Red color component.
- uint8_t green
  Red color component.
- uint8_t blue
  Red color component.

26.3.3 struct lcdc_cursor_config_t

Data Fields

- lcdc_cursor_size_t size
  Cursor size.
- lcdc_cursor_sync_mode_t syncMode
  Cursor synchronization mode.
- lcdc_cursor_palette_t palette0

Cursor palette 0.

- lcdc_cursor_palette_t palette1
  Cursor palette 1.
- uint32_t * image [LCDC_CURSOR_COUNT]
  Pointer to cursor image data.

26.4 Macro Definition Documentation

26.4.1 #define LPC_LCDC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

26.4.2 #define LCDC_CURSOR_COUNT 4

26.4.3 #define LCDC_CURSOR_IMG_BPP 2

26.4.4 #define LCDC_CURSOR_IMG_32X32_WORDS (32 * 32 * LCDC_CURSOR_IMG_BPP / (8 * sizeof(uint32_t)))

26.4.5 #define LCDC_CURSOR_IMG_64X64_WORDS (64 * 64 * LCDC_CURSOR_IMG_BPP / (8 * sizeof(uint32_t)))

26.4.6 #define LCDC_PALETTE_SIZE_WORDS (ARRAY_SIZE(((LCD_Type *)0)->PAL))

26.5 Enumeration Type Documentation

26.5.1 enum _lcdc_polarity_flags

Enumerator

kLCDC_InvertVsyncPolarity Invert the VSYNC polarity, set to active low.
kLCDC_InvertHsyncPolarity Invert the HSYNC polarity, set to active low.
kLCDC_InvertClkPolarity Invert the panel clock polarity, set to drive data on falling edge.
kLCDC_InvertDePolarity Invert the data enable (DE) polarity, set to active low.

26.5.2 enum lcdc_bpp_t

Enumerator

kLCDC_1BPP 1 bpp.
kLCDC_2BPP 2 bpp.
kLCDC_4BPP 4 bpp.
kLCDC_8BPP 8 bpp.
kLCDC_16BPP 16 bpp.

kLCDC_24BPP 24 bpp, TFT panel only.
kLCDC_16BPP565 16 bpp, 5:6:5 mode.
kLCDC_12BPP 12 bpp, 4:4:4 mode.

26.5.3 enum lcdc_display_t

Enumerator

kLCDC_DisplayTFT Active matrix TFT panels with up to 24-bit bus interface.
kLCDC_DisplaySingleMonoSTN4Bit Single-panel monochrome STN (4-bit bus interface).
kLCDC_DisplaySingleMonoSTN8Bit Single-panel monochrome STN (8-bit bus interface).
kLCDC_DisplayDualMonoSTN4Bit Dual-panel monochrome STN (4-bit bus interface).
kLCDC_DisplayDualMonoSTN8Bit Dual-panel monochrome STN (8-bit bus interface).
kLCDC_DisplaySingleColorSTN8Bit Single-panel color STN (8-bit bus interface).
kLCDC_DisplayDualColorSTN8Bit Dual-panel coor STN (8-bit bus interface).

26.5.4 enum lcdc_data_format_t

Enumerator

kLCDC_LittleEndian Little endian byte, little endian pixel.
kLCDC_BigEndian Big endian byte, big endian pixel.
kLCDC_WinCeMode little-endian byte, big-endian pixel for Windows CE mode.

26.5.5 enum lcdc_vertical_compare_interrupt_mode_t

Enumerator

kLCDC_StartOfVsync Generate vertical compare interrupt at start of VSYNC.
kLCDC_StartOfBackPorch Generate vertical compare interrupt at start of back porch.
kLCDC_StartOfActiveVideo Generate vertical compare interrupt at start of active video.
kLCDC_StartOfFrontPorch Generate vertical compare interrupt at start of front porch.

26.5.6 enum _lcdc_interrupts

Enumerator

kLCDC_CursorInterrupt Cursor image read finished interrupt.
kLCDC_FifoUnderflowInterrupt FIFO underflow interrupt.
kLCDC_BaseAddrUpdateInterrupt Panel frame base address update interrupt.
kLCDC_VerticalCompareInterrupt Vertical compare interrupt.
kLCDC_AhbErrorInterrupt AHB master error interrupt.

26.5.7 enum lcdc_panel_t

Enumerator

kLCDC_UpperPanel Upper panel frame.
kLCDC_LowerPanel Lower panel frame.

26.5.8 enum lcdc_cursor_size_t

Enumerator

kLCDC_CursorSize32 32x32 pixel cursor.
kLCDC_CursorSize64 64x64 pixel cursor.

26.5.9 enum lcdc_cursor_sync_mode_t

Enumerator

kLCDC_CursorAsync Cursor change will be displayed immediately.
kLCDC_CursorSync Cursor change will be displayed in next frame.

26.6 Function Documentation

26.6.1 status_t LCDC_Init (LCD_Type * *base*, const lcdc_config_t * *config*, uint32_t *srcClock_Hz*)

Parameters

base	LCD peripheral base address.
config	Pointer to configuration structure, see to lcdc_config_t.
srcClock_Hz	The LCD input clock (LCDCLK) frequency in Hz.

Return values

kStatus_Success	LCD is initialized successfully.
kStatus_InvalidArgument	Initlialize failed because of invalid argument.

26.6.2 void LCDC_Deinit (LCD_Type * *base*)

Parameters

base	LCD peripheral base address.

26.6.3 void LCDC_GetDefaultConfig (lcdc_config_t * *config*)

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

```
config->panelClock_Hz = 0U;
config->ppl = 0U;
config->hsw = 0U;
config->hfp = 0U;
config->hbp = 0U;
config->lpp = 0U;
config->vsw = 0U;
config->vfp = 0U;
config->vbp = 0U;
config->acBiasFreq = 1U;
config->polarityFlags = 0U;
config->enableLineEnd = false;
config->lineEndDelay = 0U;
config->upperPanelAddr = 0U;
config->lowerPanelAddr = 0U;
config->bpp = kLCDC_1BPP;
config->dataFormat = kLCDC_LittleEndian;
config->swapRedBlue = false;
config->display = kLCDC_DisplayTFT;
```

Parameters

config	Pointer to configuration structure.

26.6.4 static void LCDC_Start (LCD_Type * *base*) [inline], [static]

The LCD power up sequence should be:

1. Apply power to LCD, here all output signals are held low.
2. When LCD power stablized, call LCDC_Start to output the timing signals.
3. Apply contrast voltage to LCD panel. Delay if the display requires.
4. Call LCDC_PowerUp.

Parameters

base	LCD peripheral base address.

26.6.5 static void LCDC_Stop (LCD_Type * *base*) [inline], [static]

The LCD power down sequence should be:

1. Call LCDC_PowerDown.
2. Delay if the display requires. Disable contrast voltage to LCD panel.
3. Call LCDC_Stop to disable the timing signals.
4. Disable power to LCD.

Parameters

base	LCD peripheral base address.

26.6.6 static void LCDC_PowerUp (LCD_Type * *base*) [inline], [static]

Parameters

base	LCD peripheral base address.

26.6.7 static void LCDC_PowerDown (LCD_Type * *base*) [inline], [static]

Parameters

base	LCD peripheral base address.

26.6.8 void LCDC_SetPanelAddr (LCD_Type * *base*, lcdc_panel_t *panel*, uint32_t *addr*)

Parameters

base	LCD peripheral base address.
panel	Which panel to set.
addr	Frame base address, must be doubleword(64-bit) aligned.

26.6.9 void LCDC_SetPalette (LCD_Type * *base,* const uint32_t * *palette,* uint8_t *count_words*)

Parameters

base	LCD peripheral base address.
palette	Pointer to the palette array.
count_words	Length of the palette array to set (how many words), it should not be larger than LCDC_PALETTE_SIZE_WORDS.

26.6.10 static void LCDC_SetVerticalInterruptMode (LCD_Type * *base*, lcdc_vertical_compare_interrupt_mode_t *mode*) [inline], [static]

Parameters

base	LCD peripheral base address.
mode	The vertical compare interrupt mode.

26.6.11 void LCDC_EnableInterrupts (LCD_Type * *base*, uint32_t *mask*)

Example to enable LCD base address update interrupt and vertical compare interrupt:

```
LCDC_EnableInterrupts(LCD, kLCDC_BaseAddrUpdateInterrupt
      | kLCDC_VerticalCompareInterrupt);
```

Parameters

base	LCD peripheral base address.
mask	Interrupts to enable, it is OR'ed value of _lcdc_interrupts.

26.6.12 void LCDC_DisableInterrupts (LCD_Type * *base*, uint32_t *mask*)

Example to disable LCD base address update interrupt and vertical compare interrupt:

```
LCDC_DisableInterrupts(LCD, kLCDC_BaseAddrUpdateInterrupt
       | kLCDC_VerticalCompareInterrupt);
```

Parameters

base	LCD peripheral base address.
mask	Interrupts to disable, it is OR'ed value of _lcdc_interrupts.

26.6.13 uint32_t LCDC_GetInterruptsPendingStatus (LCD_Type * *base*)

Example:

```
uint32_t status;

status = LCDC_GetInterruptsPendingStatus(LCD);

if (kLCDC_BaseAddrUpdateInterrupt & status)
{
    // LCD base address update interrupt occurred.
}

if (kLCDC_VerticalCompareInterrupt & status)
{
    // LCD vertical compare interrupt occurred.
}
```

Parameters

base	LCD peripheral base address.

Returns

Interrupts pending status, it is OR'ed value of _lcdc_interrupts.

26.6.14 uint32_t LCDC_GetEnabledInterruptsPendingStatus (LCD_Type * *base*)

This function is similar with LCDC_GetInterruptsPendingStatus, the only difference is, this function only returns the pending status of the interrupts that have been enabled using LCDC_EnableInterrupts.

Parameters

base	LCD peripheral base address.

Returns

Interrupts pending status, it is OR'ed value of _lcdc_interrupts.

26.6.15 void LCDC_ClearInterruptsStatus (LCD_Type * *base*, uint32_t *mask*)

Example to clear LCD base address update interrupt and vertical compare interrupt pending status:

```
LCDC_ClearInterruptsStatus(LCD,
      kLCDC_BaseAddrUpdateInterrupt |
      kLCDC_VerticalCompareInterrupt);
```

Parameters

base	LCD peripheral base address.
mask	Interrupts to disable, it is OR'ed value of _lcdc_interrupts.

26.6.16 void LCDC_SetCursorConfig (LCD_Type * *base*, const lcdc_cursor_config_t * *config*)

This function should be called before enabling the hardware cursor. It supports initializing multiple cursor images at a time when using 32x32 pixels cursor.

For example:

```
uint32_t cursor0Img[LCDC_CURSOR_IMG_32X32_WORDS] = {...};
uint32_t cursor2Img[LCDC_CURSOR_IMG_32X32_WORDS] = {...};

lcdc_cursor_config_t cursorConfig;

LCDC_CursorGetDefaultConfig(&cursorConfig);

cursorConfig.image[0] = cursor0Img;
cursorConfig.image[2] = cursor2Img;

LCDC_SetCursorConfig(LCD, &cursorConfig);

LCDC_ChooseCursor(LCD, 0);
LCDC_SetCursorPosition(LCD, 0, 0);

LCDC_EnableCursor(LCD);
```

In this example, cursor 0 and cursor 2 image data are initialized, but cursor 1 and cursor 3 image data are not initialized because image[1] and image[2] are all NULL. With this, application could initializes all cursor images it will use at the beginning and call LCDC_SetCursorImage directly to display the one which it needs.

Parameters

base	LCD peripheral base address.
config	Pointer to the hardware cursor configuration structure.

26.6.17 void LCDC_CursorGetDefaultConfig (lcdc_cursor_config_t * *config*)

The default configuration values are:

```
config->size = kLCDC_CursorSize32;
config->syncMode = kLCDC_CursorAsync;
config->palette0.red = 0U;
config->palette0.green = 0U;
config->palette0.blue = 0U;
config->palette1.red = 255U;
config->palette1.green = 255U;
config->palette1.blue = 255U;
config->image[0] = (uint32_t *)0;
config->image[1] = (uint32_t *)0;
config->image[2] = (uint32_t *)0;
config->image[3] = (uint32_t *)0;
```

Parameters

config	Pointer to the hardware cursor configuration structure.

26.6.18 static void LCDC_EnableCursor (LCD_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	LCD peripheral base address.
enable	True to enable, false to disable.

26.6.19 static void LCDC_ChooseCursor (LCD_Type * *base,* uint8_t *index*) [inline], [static]

When using 32x32 cursor, the number of cursors supports is LCDC_CURSOR_COUNT. When using 64x64 cursor, the LCD only supports one cursor. This function selects which cursor to display when using 32x32 cursor. When synchronization mode is kLCDC_CursorSync, the change effects in the next frame. When synchronization mode is * kLCDC_CursorAsync, change effects immediately.

Parameters

base	LCD peripheral base address.
index	Index of the cursor to display.

Note

The function LCDC_SetCursorPosition must be called after this function to show the new cursor.

26.6.20 void LCDC_SetCursorPosition (LCD_Type * *base*, int32_t *positionX*, int32_t *positionY*)

When synchronization mode is kLCDC_CursorSync, position change effects in the next frame. When synchronization mode is kLCDC_CursorAsync, position change effects immediately.

Parameters

base	LCD peripheral base address.
positionX	X ordinate of the cursor top-left measured in pixels
positionY	Y ordinate of the cursor top-left measured in pixels

26.6.21 void LCDC_SetCursorImage (LCD_Type * *base*, lcdc_cursor_size_t *size*, uint8_t *index*, const uint32_t * *image*)

The interrupt kLCDC_CursorInterrupt indicates that last cursor pixel is displayed. When the hardware cursor is enabled,

Parameters

base	LCD peripheral base address.
size	The cursor size.
index	Index of the cursor to set when using 32x32 cursor.
image	Pointer to the cursor image. When using 32x32 cursor, the image size should be LCDC_CURSOR_IMG_32X32_WORDS. When using 64x64 cursor, the image size should be LCDC_CURSOR_IMG_64X64_WORDS.

26.7 Variable Documentation

26.7.1 uint32_t lcdc_config_t::panelClock_Hz

26.7.2 uint16_t lcdc_config_t::ppl

26.7.3 uint8_t lcdc_config_t::hsw

26.7.4 uint8_t lcdc_config_t::hfp

26.7.5 uint8_t lcdc_config_t::hbp

26.7.6 uint16_t lcdc_config_t::lpp

26.7.7 uint8_t lcdc_config_t::vsw

26.7.8 uint8_t lcdc_config_t::vfp

26.7.9 uint8_t lcdc_config_t::vbp

26.7.10 uint8_t lcdc_config_t::acBiasFreq

Only used for STN display.

26.7.11 uint16_t lcdc_config_t::polarityFlags

26.7.12 bool lcdc_config_t::enableLineEnd

26.7.13 uint8_t lcdc_config_t::lineEndDelay

26.7.14 uint32_t lcdc_config_t::upperPanelAddr

26.7.15 uint32_t lcdc_config_t::lowerPanelAddr

26.7.16 lcdc_bpp_t lcdc_config_t::bpp

26.7.17 lcdc_data_format_t lcdc_config_t::dataFormat

26.7.18 bool lcdc_config_t::swapRedBlue

26.7.19 lcdc_display_t lcdc_config_t::display

26.7.20 uint8_t lcdc_cursor_palette_t::red

26.7.21 uint8_t lcdc_cursor_palette_t::green

26.7.22 uint8_t lcdc_cursor_palette_t::blue

26.7.23 lcdc_cursor_size_t lcdc_cursor_config_t::size

26.7.24 lcdc_cursor_sync_mode_t lcdc_cursor_config_t::syncMode

26.7.25 lcdc_cursor_palette_t lcdc_cursor_config_t::palette0

26.7.26 lcdc_cursor_palette_t lcdc_cursor_config_t::palette1

26.7.27 uint32_t* lcdc_cursor_config_t::image[LCDC_CURSOR_COUNT]

Chapter 27
MCAN: Controller Area Network Driver

27.1 Overview

The KSDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of Kinetis devices.

Data Structures

- struct mcan_tx_buffer_frame_t
  MCAN Tx Buffer structure. More...
- struct mcan_rx_buffer_frame_t
  MCAN Rx FIFO/Buffer structure. More...
- struct mcan_rx_fifo_config_t
  MCAN Rx FIFO configuration. More...
- struct mcan_rx_buffer_config_t
  MCAN Rx Buffer configuration. More...
- struct mcan_tx_fifo_config_t
  MCAN Tx Event FIFO configuration. More...
- struct mcan_tx_buffer_config_t
  MCAN Tx Buffer configuration. More...
- struct mcan_std_filter_element_config_t
  MCAN Standard Message ID Filter Element. More...
- struct mcan_ext_filter_element_config_t
  MCAN Extended Message ID Filter Element. More...
- struct mcan_frame_filter_config_t
  MCAN Rx filter configuration. More...
- struct mcan_config_t
  MCAN module configuration structure. More...
- struct mcan_timing_config_t
  MCAN protocol timing characteristic configuration structure. More...
- struct mcan_buffer_transfer_t
  MCAN Buffer transfer. More...
- struct mcan_fifo_transfer_t
  MCAN Rx FIFO transfer. More...
- struct mcan_handle_t
  MCAN handle structure. More...

Typedefs

- typedef void(∗ mcan_transfer_callback_t)(CAN_Type ∗base, mcan_handle_t ∗handle, status_t status, uint32_t result, void ∗userData)
  MCAN transfer callback function.

Enumerations

- enum _mcan_status {
  kStatus_MCAN_TxBusy = MAKE_STATUS(kStatusGroup_MCAN, 0),
  kStatus_MCAN_TxIdle = MAKE_STATUS(kStatusGroup_MCAN, 1),
  kStatus_MCAN_RxBusy = MAKE_STATUS(kStatusGroup_MCAN, 2),
  kStatus_MCAN_RxIdle = MAKE_STATUS(kStatusGroup_MCAN, 3),
  kStatus_MCAN_RxFifo0New = MAKE_STATUS(kStatusGroup_MCAN, 4),
  kStatus_MCAN_RxFifo0Idle = MAKE_STATUS(kStatusGroup_MCAN, 5),
  kStatus_MCAN_RxFifo0Watermark = MAKE_STATUS(kStatusGroup_MCAN, 6),
  kStatus_MCAN_RxFifo0Full = MAKE_STATUS(kStatusGroup_MCAN, 7),
  kStatus_MCAN_RxFifo0Lost = MAKE_STATUS(kStatusGroup_MCAN, 8),
  kStatus_MCAN_RxFifo1New = MAKE_STATUS(kStatusGroup_MCAN, 9),
  kStatus_MCAN_RxFifo1Idle = MAKE_STATUS(kStatusGroup_MCAN, 10),
  kStatus_MCAN_RxFifo1Watermark = MAKE_STATUS(kStatusGroup_MCAN, 11),
  kStatus_MCAN_RxFifo1Full = MAKE_STATUS(kStatusGroup_MCAN, 12),
  kStatus_MCAN_RxFifo1Lost = MAKE_STATUS(kStatusGroup_MCAN, 13),
  kStatus_MCAN_RxFifo0Busy = MAKE_STATUS(kStatusGroup_MCAN, 14),
  kStatus_MCAN_RxFifo1Busy = MAKE_STATUS(kStatusGroup_MCAN, 15),
  kStatus_MCAN_ErrorStatus = MAKE_STATUS(kStatusGroup_MCAN, 16),
  kStatus_MCAN_UnHandled = MAKE_STATUS(kStatusGroup_MCAN, 17) }
  MCAN transfer status.
- enum _mcan_flags {
  kMCAN_AccesstoRsvdFlag = CAN_IR_ARA_MASK,
  kMCAN_ProtocolErrDIntFlag = CAN_IR_PED_MASK,
  kMCAN_ProtocolErrAIntFlag = CAN_IR_PEA_MASK,
  kMCAN_BusOffIntFlag = CAN_IR_BO_MASK,
  kMCAN_ErrorWarningIntFlag = CAN_IR_EW_MASK,
  kMCAN_ErrorPassiveIntFlag = CAN_IR_EP_MASK }
  MCAN status flags.
- enum _mcan_rx_fifo_flags {
  kMCAN_RxFifo0NewFlag = CAN_IR_RF0N_MASK,
  kMCAN_RxFifo0WatermarkFlag = CAN_IR_RF0W_MASK,
  kMCAN_RxFifo0FullFlag = CAN_IR_RF0F_MASK,
  kMCAN_RxFifo0LostFlag = CAN_IR_RF0L_MASK,
  kMCAN_RxFifo1NewFlag = CAN_IR_RF1N_MASK,
  kMCAN_RxFifo1WatermarkFlag = CAN_IR_RF1W_MASK,
  kMCAN_RxFifo1FullFlag = CAN_IR_RF1F_MASK,
  kMCAN_RxFifo1LostFlag = CAN_IR_RF1L_MASK }
  MCAN Rx FIFO status flags.
- enum _mcan_tx_flags {

kMCAN_TxTransmitCompleteFlag = CAN_IR_TC_MASK,
kMCAN_TxTransmitCancelFinishFlag = CAN_IR_TCF_MASK,
kMCAN_TxEventFifoLostFlag = CAN_IR_TEFL_MASK,
kMCAN_TxEventFifoFullFlag = CAN_IR_TEFF_MASK,
kMCAN_TxEventFifoWatermarkFlag = CAN_IR_TEFW_MASK,
kMCAN_TxEventFifoNewFlag = CAN_IR_TEFN_MASK,
kMCAN_TxEventFifoEmptyFlag = CAN_IR_TFE_MASK }
MCAN Tx status flags.

- enum _mcan_interrupt_enable {
kMCAN_BusOffInterruptEnable = CAN_IE_BOE_MASK,
kMCAN_ErrorInterruptEnable = CAN_IE_EPE_MASK,
kMCAN_WarningInterruptEnable = CAN_IE_EWE_MASK }
MCAN interrupt configuration structure, default settings all disabled.
- enum mcan_frame_idformat_t {
kMCAN_FrameIDStandard = 0x0U,
kMCAN_FrameIDExtend = 0x1U }
MCAN frame format.
- enum mcan_frame_type_t {
kMCAN_FrameTypeData = 0x0U,
kMCAN_FrameTypeRemote = 0x1U }
MCAN frame type.
- enum mcan_bytes_in_datafield_t {
kMCAN_8ByteDatafield = 0x0U,
kMCAN_12ByteDatafield = 0x1U,
kMCAN_16ByteDatafield = 0x2U,
kMCAN_20ByteDatafield = 0x3U,
kMCAN_24ByteDatafield = 0x4U,
kMCAN_32ByteDatafield = 0x5U,
kMCAN_48ByteDatafield = 0x6U,
kMCAN_64ByteDatafield = 0x7U }
MCAN frame datafield size.
- enum mcan_fifo_type_t {
kMCAN_Fifo0 = 0x0U,
kMCAN_Fifo1 = 0x1U }
MCAN Rx FIFO block number.
- enum mcan_fifo_opmode_config_t {
kMCAN_FifoBlocking = 0,
kMCAN_FifoOverwrite = 1 }
MCAN FIFO Operation Mode.
- enum mcan_txmode_config_t {
kMCAN_txFifo = 0,
kMCAN_txQueue = 1 }
MCAN Tx FIFO/Queue Mode.
- enum mcan_remote_frame_config_t {
kMCAN_filterFrame = 0,
kMCAN_rejectFrame = 1 }
MCAN remote frames treatment.

- enum mcan_nonmasking_frame_config_t {
  kMCAN_acceptinFifo0 = 0,
  kMCAN_acceptinFifo1 = 1,
  kMCAN_reject0 = 2,
  kMCAN_reject1 = 3 }
  MCAN non-masking frames treatment.
- enum mcan_fec_config_t {
  kMCAN_disable = 0,
  kMCAN_storeinFifo0 = 1,
  kMCAN_storeinFifo1 = 2,
  kMCAN_reject = 3,
  kMCAN_setprio = 4,
  kMCAN_setpriofifo0 = 5,
  kMCAN_setpriofifo1 = 6,
  kMCAN_storeinbuffer = 7 }
  MCAN Filter Element Configuration.
- enum mcan_filter_type_t {
  kMCAN_range = 0,
  kMCAN_dual = 1,
  kMCAN_classic = 2,
  kMCAN_disableORrange2 = 3 }
  MCAN Filter Type.

Driver version

- #define MCAN_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  MCAN driver version 2.0.0.

Initialization and deinitialization

- void MCAN_Init (CAN_Type *base, const mcan_config_t *config, uint32_t sourceClock_Hz)
  Initializes a MCAN instance.
- void MCAN_GetDefaultConfig (mcan_config_t *config)
  Gets the default configuration structure.
- void MCAN_EnterNormalMode (CAN_Type *base)
  MCAN enters normal mode.

Configuration.

- static void MCAN_SetMsgRAMBase (CAN_Type *base, uint32_t value)
  Sets the MCAN Message RAM base address.
- static uint32_t MCAN_GetMsgRAMBase (CAN_Type *base)
  Gets the MCAN Message RAM base address.
- void MCAN_SetArbitrationTimingConfig (CAN_Type *base, const mcan_timing_config_t *config)
  Sets the MCAN protocol arbitration phase timing characteristic.
- void MCAN_SetDataTimingConfig (CAN_Type *base, const mcan_timing_config_t *config)
  Sets the MCAN protocol data phase timing characteristic.

- void MCAN_SetRxFifo0Config (CAN_Type *base, const mcan_rx_fifo_config_t *config)
  Configures a MCAN receive fifo 0 buffer.
- void MCAN_SetRxFifo1Config (CAN_Type *base, const mcan_rx_fifo_config_t *config)
  Configures a MCAN receive fifo 1 buffer.
- void MCAN_SetRxBufferConfig (CAN_Type *base, const mcan_rx_buffer_config_t *config)
  Configures a MCAN receive buffer.
- void MCAN_SetTxEventfifoConfig (CAN_Type *base, const mcan_tx_fifo_config_t *config)
  Configures a MCAN transmit event fifo.
- void MCAN_SetTxBufferConfig (CAN_Type *base, const mcan_tx_buffer_config_t *config)
  Configures a MCAN transmit buffer.
- void MCAN_SetFilterConfig (CAN_Type *base, const mcan_frame_filter_config_t *config)
  Set filter configuration.
- void MCAN_SetSTDFilterElement (CAN_Type *base, const mcan_frame_filter_config_t *config, const mcan_std_filter_element_config_t *filter, uint8_t idx)
  Set filter configuration.
- void MCAN_SetEXTFilterElement (CAN_Type *base, const mcan_frame_filter_config_t *config, const mcan_ext_filter_element_config_t *filter, uint8_t idx)
  Set filter configuration.

Status

- static uint32_t MCAN_GetStatusFlag (CAN_Type *base, uint32_t mask)
  Gets the MCAN module interrupt flags.
- static void MCAN_ClearStatusFlag (CAN_Type *base, uint32_t mask)
  Clears the MCAN module interrupt flags.
- static bool MCAN_GetRxBufferStatusFlag (CAN_Type *base, uint8_t idx)
  Gets the new data flag of specific Rx Buffer.
- static void MCAN_ClearRxBufferStatusFlag (CAN_Type *base, uint8_t idx)
  Clears the new data flag of specific Rx Buffer.

Interrupts

- static void MCAN_EnableInterrupts (CAN_Type *base, uint32_t line, uint32_t mask)
  Enables MCAN interrupts according to the provided interrupt line and mask.
- static void MCAN_EnableTransmitBufferInterrupts (CAN_Type *base, uint8_t idx)
  Enables MCAN Tx Buffer interrupts according to the provided index.
- static void MCAN_DisableTransmitBufferInterrupts (CAN_Type *base, uint8_t idx)
  Disables MCAN Tx Buffer interrupts according to the provided index.
- static void MCAN_DisableInterrupts (CAN_Type *base, uint32_t mask)
  Disables MCAN interrupts according to the provided mask.

Bus Operations

- status_t MCAN_WriteTxBuffer (CAN_Type *base, uint8_t idx, const mcan_tx_buffer_frame_t *txFrame)
  Writes a MCAN Message to the Transmit Buffer.
- status_t MCAN_ReadRxFifo (CAN_Type *base, uint8_t fifoBlock, mcan_rx_buffer_frame_t *rxFrame)
  Reads a MCAN Message from Rx FIFO.

Transactional

- static void MCAN_TransmitAddRequest (CAN_Type *base, uint8_t idx)
  Tx Buffer add request to send message out.
- static void MCAN_TransmitCancelRequest (CAN_Type *base, uint8_t idx)
  Tx Buffer cancel sending request.
- status_t MCAN_TransferSendBlocking (CAN_Type *base, uint8_t idx, mcan_tx_buffer_frame_t *txFrame)
  Performs a polling send transaction on the CAN bus.
- status_t MCAN_TransferReceiveFifoBlocking (CAN_Type *base, uint8_t fifoBlock, mcan_rx_buffer_frame_t *rxFrame)
  Performs a polling receive transaction from Rx FIFO on the CAN bus.
- void MCAN_TransferCreateHandle (CAN_Type *base, mcan_handle_t *handle, mcan_transfer_callback_t callback, void *userData)
  Initializes the MCAN handle.
- status_t MCAN_TransferSendNonBlocking (CAN_Type *base, mcan_handle_t *handle, mcan_buffer_transfer_t *xfer)
  Sends a message using IRQ.
- status_t MCAN_TransferReceiveFifoNonBlocking (CAN_Type *base, uint8_t fifoBlock, mcan_handle_t *handle, mcan_fifo_transfer_t *xfer)
  Receives a message from Rx FIFO using IRQ.
- void MCAN_TransferAbortSend (CAN_Type *base, mcan_handle_t *handle, uint8_t bufferIdx)
  Aborts the interrupt driven message send process.
- void MCAN_TransferAbortReceiveFifo (CAN_Type *base, uint8_t fifoBlock, mcan_handle_t *handle)
  Aborts the interrupt driven message receive from Rx FIFO process.
- void MCAN_TransferHandleIRQ (CAN_Type *base, mcan_handle_t *handle)
  MCAN IRQ handle function.

27.2 Data Structure Documentation

27.2.1 struct mcan_tx_buffer_frame_t

27.2.1.0.0.33 Field Documentation

27.2.1.0.0.33.1 uint32_t mcan_tx_buffer_frame_t::id

27.2.1.0.0.33.2 uint32_t mcan_tx_buffer_frame_t::rtr

27.2.1.0.0.33.3 uint32_t mcan_tx_buffer_frame_t::xtd

27.2.1.0.0.33.4 uint32_t mcan_tx_buffer_frame_t::esi

27.2.1.0.0.33.5 uint32_t mcan_tx_buffer_frame_t::dlc

27.2.1.0.0.33.6 uint32_t mcan_tx_buffer_frame_t::brs

27.2.1.0.0.33.7 uint32_t mcan_tx_buffer_frame_t::fdf

27.2.1.0.0.33.8 uint32_t mcan_tx_buffer_frame_t::__pad1__

27.2.1.0.0.33.9 uint32_t mcan_tx_buffer_frame_t::efc

27.2.1.0.0.33.10 uint32_t mcan_tx_buffer_frame_t::mm

27.2.2 struct mcan_rx_buffer_frame_t

27.2.2.0.0.34 Field Documentation

27.2.2.0.0.34.1 uint32_t mcan_rx_buffer_frame_t::id

27.2.2.0.0.34.2 uint32_t mcan_rx_buffer_frame_t::rtr

27.2.2.0.0.34.3 uint32_t mcan_rx_buffer_frame_t::xtd

27.2.2.0.0.34.4 uint32_t mcan_rx_buffer_frame_t::esi

27.2.2.0.0.34.5 uint32_t mcan_rx_buffer_frame_t::rxts

27.2.2.0.0.34.6 uint32_t mcan_rx_buffer_frame_t::dlc

27.2.2.0.0.34.7 uint32_t mcan_rx_buffer_frame_t::brs

27.2.2.0.0.34.8 uint32_t mcan_rx_buffer_frame_t::fdf

27.2.2.0.0.34.9 uint32_t mcan_rx_buffer_frame_t::__pad0__

27.2.2.0.0.34.10 uint32_t mcan_rx_buffer_frame_t::fidx

27.2.2.0.0.34.11 uint32_t mcan_rx_buffer_frame_t::anmf

27.2.3 struct mcan_rx_fifo_config_t

FIFOn start address.
- uint32_t elementSize
  FIFOn element number.
- uint32_t watermark
  FIFOn watermark level.
- mcan_fifo_opmode_config_t opmode
  FIFOn blocking/overwrite mode.
- mcan_bytes_in_datafield_t datafieldSize
  Data field size per frame, size>8 is for CANFD.

27.2.3.0.0.35 Field Documentation

27.2.3.0.0.35.1 uint32_t mcan_rx_fifo_config_t::address

27.2.3.0.0.35.2 uint32_t mcan_rx_fifo_config_t::elementSize

27.2.3.0.0.35.3 uint32_t mcan_rx_fifo_config_t::watermark

27.2.3.0.0.35.4 mcan_fifo_opmode_config_t mcan_rx_fifo_config_t::opmode

27.2.3.0.0.35.5 mcan_bytes_in_datafield_t mcan_rx_fifo_config_t::datafieldSize

27.2.4 struct mcan_rx_buffer_config_t

Data Fields

- uint32_t address
  Rx Buffer start address.
- mcan_bytes_in_datafield_t datafieldSize
  Data field size per frame, size>8 is for CANFD.

27.2.4.0.0.36 Field Documentation

27.2.4.0.0.36.1 uint32_t mcan_rx_buffer_config_t::address

27.2.4.0.0.36.2 mcan_bytes_in_datafield_t mcan_rx_buffer_config_t::datafieldSize

27.2.5 struct mcan_tx_fifo_config_t

Data Fields

- uint32_t address
  Event fifo start address.
- uint32_t elementSize
  FIFOn element number.
- uint32_t watermark
  FIFOn watermark level.

27.2.5.0.0.37 Field Documentation

27.2.5.0.0.37.1 uint32_t mcan_tx_fifo_config_t::address

27.2.5.0.0.37.2 uint32_t mcan_tx_fifo_config_t::elementSize

27.2.5.0.0.37.3 uint32_t mcan_tx_fifo_config_t::watermark

27.2.6 struct mcan_tx_buffer_config_t

Data Fields

- uint32_t address
  Tx Buffers Start Address.
- uint32_t dedicatedSize
  Number of Dedicated Transmit Buffers.
- uint32_t fqSize
  Transmit FIFO/Queue Size.
- mcan_txmode_config_t mode
  Tx FIFO/Queue Mode.
- mcan_bytes_in_datafield_t datafieldSize
  Data field size per frame, size>8 is for CANFD.

27.2.6.0.0.38 Field Documentation

27.2.6.0.0.38.1 uint32_t mcan_tx_buffer_config_t::address

27.2.6.0.0.38.2 uint32_t mcan_tx_buffer_config_t::dedicatedSize

27.2.6.0.0.38.3 uint32_t mcan_tx_buffer_config_t::fqSize

27.2.6.0.0.38.4 mcan_txmode_config_t mcan_tx_buffer_config_t::mode

27.2.6.0.0.38.5 mcan_bytes_in_datafield_t mcan_tx_buffer_config_t::datafieldSize

27.2.7 struct mcan_std_filter_element_config_t

Data Fields

- uint32_t sfid2: 11
  Standard Filter ID 2.
- uint32_t __pad0__: 5
  Reserved.
- uint32_t sfid1: 11
  Standard Filter ID 1.
- mcan_fec_config_t sfec: 3
  Standard Filter Element Configuration.
- mcan_filter_type_t sft: 2
  Standard Filter Type/.

27.2.7.0.0.39 Field Documentation

27.2.7.0.0.39.1 uint32_t mcan_std_filter_element_config_t::sfid2

27.2.7.0.0.39.2 uint32_t mcan_std_filter_element_config_t::__pad0__

27.2.7.0.0.39.3 uint32_t mcan_std_filter_element_config_t::sfid1

27.2.7.0.0.39.4 mcan_fec_config_t mcan_std_filter_element_config_t::sfec

27.2.8 struct mcan_ext_filter_element_config_t

Data Fields

- uint32_t efid1: 29
  Extended Filter ID 1.
- mcan_fec_config_t efec: 3
  Extended Filter Element Configuration.
- uint32_t efid2: 29
  Extended Filter ID 2.
- uint32_t __pad0__: 1
  Reserved.
- mcan_filter_type_t eft: 2
  Extended Filter Type.

27.2.8.0.0.40 Field Documentation

27.2.8.0.0.40.1 uint32_t mcan_ext_filter_element_config_t::efid1

27.2.8.0.0.40.2 mcan_fec_config_t mcan_ext_filter_element_config_t::efec

27.2.8.0.0.40.3 uint32_t mcan_ext_filter_element_config_t::efid2

27.2.8.0.0.40.4 uint32_t mcan_ext_filter_element_config_t::__pad0__

27.2.8.0.0.40.5 mcan_filter_type_t mcan_ext_filter_element_config_t::eft

27.2.9 struct mcan_frame_filter_config_t

Data Fields

- uint32_t address
  Filter start address.
- uint32_t listSize
  Filter list size.
- mcan_frame_idformat_t idFormat
  Frame format.
- mcan_remote_frame_config_t remFrame
  Remote frame treatment.

- mcan_nonmasking_frame_config_t nmFrame
  Non-masking frame treatment.

27.2.9.0.0.41 Field Documentation

27.2.9.0.0.41.1 uint32_t mcan_frame_filter_config_t::address

27.2.9.0.0.41.2 uint32_t mcan_frame_filter_config_t::listSize

27.2.9.0.0.41.3 mcan_frame_idformat_t mcan_frame_filter_config_t::idFormat

27.2.9.0.0.41.4 mcan_remote_frame_config_t mcan_frame_filter_config_t::remFrame

27.2.9.0.0.41.5 mcan_nonmasking_frame_config_t mcan_frame_filter_config_t::nmFrame

27.2.10 struct mcan_config_t

Data Fields

- uint32_t baudRateA
  Baud rate of Arbitration phase in bps.
- uint32_t baudRateD
  Baud rate of Data phase in bps.
- bool enableCanfdNormal
  Enable or Disable CANFD normal.
- bool enableCanfdSwitch
  Enable or Disable CANFD with baudrate switch.
- bool enableLoopBackInt
  Enable or Disable Internal Back.
- bool enableLoopBackExt
  Enable or Disable External Loop Back.
- bool enableBusMon
  Enable or Disable Bus Monitoring Mode.

27.2.10.0.0.42 Field Documentation

27.2.10.0.0.42.1 uint32_t mcan_config_t::baudRateA

27.2.10.0.0.42.2 uint32_t mcan_config_t::baudRateD

27.2.10.0.0.42.3 bool mcan_config_t::enableCanfdNormal

27.2.10.0.0.42.4 bool mcan_config_t::enableCanfdSwitch

27.2.10.0.0.42.5 bool mcan_config_t::enableLoopBackInt

27.2.10.0.0.42.6 bool mcan_config_t::enableLoopBackExt

27.2.10.0.0.42.7 bool mcan_config_t::enableBusMon

27.2.11 struct mcan_timing_config_t

Data Fields

- uint16_t preDivider
  Clock Pre-scaler Division Factor.
- uint8_t rJumpwidth
  Re-sync Jump Width.
- uint8_t seg1
  Data Time Segment 1.
- uint8_t seg2
  Data Time Segment 2.

27.2.11.0.0.43 Field Documentation

27.2.11.0.0.43.1 uint16_t mcan_timing_config_t::preDivider

27.2.11.0.0.43.2 uint8_t mcan_timing_config_t::rJumpwidth

27.2.11.0.0.43.3 uint8_t mcan_timing_config_t::seg1

27.2.11.0.0.43.4 uint8_t mcan_timing_config_t::seg2

27.2.12 struct mcan_buffer_transfer_t

Data Fields

- mcan_tx_buffer_frame_t * frame
  The buffer of CAN Message to be transfer.
- uint8_t bufferIdx
  The index of Message buffer used to transfer Message.

27.2.12.0.0.44 Field Documentation

27.2.12.0.0.44.1 mcan_tx_buffer_frame_t* mcan_buffer_transfer_t::frame

27.2.12.0.0.44.2 uint8_t mcan_buffer_transfer_t::bufferIdx

27.2.13 struct mcan_fifo_transfer_t

Data Fields

- mcan_rx_buffer_frame_t * frame
  The buffer of CAN Message to be received from Rx FIFO.

27.2.13.0.0.45 Field Documentation

27.2.13.0.0.45.1 mcan_rx_buffer_frame_t* mcan_fifo_transfer_t::frame

27.2.14 struct _mcan_handle

MCAN handle structure definition.

Data Fields

- mcan_transfer_callback_t callback
  Callback function.
- void * userData
  MCAN callback function parameter.
- mcan_tx_buffer_frame_t *volatile bufferFrameBuf [64]
  The buffer for received data from Buffers.
- mcan_rx_buffer_frame_t *volatile rxFifoFrameBuf
  The buffer for received data from Rx FIFO.
- volatile uint8_t txbufferIdx
  Message Buffer transfer state.
- volatile uint8_t bufferState [64]
  Message Buffer transfer state.
- volatile uint8_t rxFifoState
  Rx FIFO transfer state.

27.2.14.0.0.46 Field Documentation

27.2.14.0.0.46.1 mcan_transfer_callback_t mcan_handle_t::callback

27.2.14.0.0.46.2 void* mcan_handle_t::userData

27.2.14.0.0.46.3 mcan_tx_buffer_frame_t* volatile mcan_handle_t::bufferFrameBuf[64]

27.2.14.0.0.46.4 mcan_rx_buffer_frame_t* volatile mcan_handle_t::rxFifoFrameBuf

27.2.14.0.0.46.5 volatile uint8_t mcan_handle_t::txbufferIdx

27.2.14.0.0.46.6 volatile uint8_t mcan_handle_t::bufferState[64]

27.2.14.0.0.46.7 volatile uint8_t mcan_handle_t::rxFifoState

27.3 Macro Definition Documentation

27.3.1 #define MCAN_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

27.4 Typedef Documentation

27.4.1 typedef void(* mcan_transfer_callback_t)(CAN_Type *base, mcan_handle_t *handle, status_t status, uint32_t result, void *userData)

The MCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_MCAN_ErrorStatus, the result parameter is the Content of MCAN status register which can be used to get the working status(or error status) of MCAN module. If the status equals to other MCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other MCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

27.5 Enumeration Type Documentation

27.5.1 enum _mcan_status

Enumerator

kStatus_MCAN_TxBusy Tx Buffer is Busy.
kStatus_MCAN_TxIdle Tx Buffer is Idle.
kStatus_MCAN_RxBusy Rx Buffer is Busy.
kStatus_MCAN_RxIdle Rx Buffer is Idle.
kStatus_MCAN_RxFifo0New New message written to Rx FIFO 0.
kStatus_MCAN_RxFifo0Idle Rx FIFO 0 is Idle.
kStatus_MCAN_RxFifo0Watermark Rx FIFO 0 fill level reached watermark.
kStatus_MCAN_RxFifo0Full Rx FIFO 0 full.
kStatus_MCAN_RxFifo0Lost Rx FIFO 0 message lost.
kStatus_MCAN_RxFifo1New New message written to Rx FIFO 1.
kStatus_MCAN_RxFifo1Idle Rx FIFO 1 is Idle.

kStatus_MCAN_RxFifo1Watermark Rx FIFO 1 fill level reached watermark.
kStatus_MCAN_RxFifo1Full Rx FIFO 1 full.
kStatus_MCAN_RxFifo1Lost Rx FIFO 1 message lost.
kStatus_MCAN_RxFifo0Busy Rx FIFO 0 is busy.
kStatus_MCAN_RxFifo1Busy Rx FIFO 1 is busy.
kStatus_MCAN_ErrorStatus MCAN Module Error and Status.
kStatus_MCAN_UnHandled UnHadled Interrupt asserted.

27.5.2 enum _mcan_flags

This provides constants for the MCAN status flags for use in the MCAN functions. Note: The CPU read action clears MCAN_ErrorFlag, therefore user need to read MCAN_ErrorFlag and distinguish which error is occur using _mcan_error_flags enumerations.

Enumerator

kMCAN_AccesstoRsvdFlag CAN Synchronization Status.
kMCAN_ProtocolErrDIntFlag Tx Warning Interrupt Flag.
kMCAN_ProtocolErrAIntFlag Rx Warning Interrupt Flag.
kMCAN_BusOffIntFlag Tx Error Warning Status.
kMCAN_ErrorWarningIntFlag Rx Error Warning Status.
kMCAN_ErrorPassiveIntFlag Rx Error Warning Status.

27.5.3 enum _mcan_rx_fifo_flags

The MCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO.

Enumerator

kMCAN_RxFifo0NewFlag Rx FIFO 0 new message flag.
kMCAN_RxFifo0WatermarkFlag Rx FIFO 0 watermark reached flag.
kMCAN_RxFifo0FullFlag Rx FIFO 0 full flag.
kMCAN_RxFifo0LostFlag Rx FIFO 0 message lost flag.
kMCAN_RxFifo1NewFlag Rx FIFO 0 new message flag.
kMCAN_RxFifo1WatermarkFlag Rx FIFO 0 watermark reached flag.
kMCAN_RxFifo1FullFlag Rx FIFO 0 full flag.
kMCAN_RxFifo1LostFlag Rx FIFO 0 message lost flag.

27.5.4 enum _mcan_tx_flags

The MCAN Tx Status enumerations are used to determine the status of the Tx Buffer/Event FIFO.

Enumerator

kMCAN_TxTransmitCompleteFlag Transmission completed flag.
kMCAN_TxTransmitCancelFinishFlag Transmission cancellation finished flag.
kMCAN_TxEventFifoLostFlag Tx Event FIFO element lost.
kMCAN_TxEventFifoFullFlag Tx Event FIFO full.
kMCAN_TxEventFifoWatermarkFlag Tx Event FIFO fill level reached watermark.
kMCAN_TxEventFifoNewFlag Tx Handler wrote Tx Event FIFO element flag.
kMCAN_TxEventFifoEmptyFlag Tx FIFO empty flag.

27.5.5 enum _mcan_interrupt_enable

This structure contains the settings for all of the MCAN Module interrupt configurations.

Enumerator

kMCAN_BusOffInterruptEnable Bus Off interrupt.
kMCAN_ErrorInterruptEnable Error interrupt.
kMCAN_WarningInterruptEnable Rx Warning interrupt.

27.5.6 enum mcan_frame_idformat_t

Enumerator

kMCAN_FrameIDStandard Standard frame format attribute.
kMCAN_FrameIDExtend Extend frame format attribute.

27.5.7 enum mcan_frame_type_t

Enumerator

kMCAN_FrameTypeData Data frame type attribute.
kMCAN_FrameTypeRemote Remote frame type attribute.

27.5.8 enum mcan_bytes_in_datafield_t

Enumerator

kMCAN_8ByteDatafield 8 byte data field.
kMCAN_12ByteDatafield 12 byte data field.
kMCAN_16ByteDatafield 16 byte data field.

kMCAN_20ByteDatafield 20 byte data field.
kMCAN_24ByteDatafield 24 byte data field.
kMCAN_32ByteDatafield 32 byte data field.
kMCAN_48ByteDatafield 48 byte data field.
kMCAN_64ByteDatafield 64 byte data field.

27.5.9 enum mcan_fifo_type_t

Enumerator

kMCAN_Fifo0 CAN Rx FIFO 0.
kMCAN_Fifo1 CAN Rx FIFO 1.

27.5.10 enum mcan_fifo_opmode_config_t

Enumerator

kMCAN_FifoBlocking FIFO blocking mode.
kMCAN_FifoOverwrite FIFO overwrite mode.

27.5.11 enum mcan_txmode_config_t

Enumerator

kMCAN_txFifo Tx FIFO operation.
kMCAN_txQueue Tx Queue operation.

27.5.12 enum mcan_remote_frame_config_t

Enumerator

kMCAN_filterFrame Filter remote frames.
kMCAN_rejectFrame Reject all remote frames.

27.5.13 enum mcan_nonmasking_frame_config_t

Enumerator

kMCAN_acceptinFifo0 Accept non-masking frames in Rx FIFO 0.

kMCAN_acceptinFifo1 Accept non-masking frames in Rx FIFO 1.
kMCAN_reject0 Reject non-masking frames.
kMCAN_reject1 Reject non-masking frames.

27.5.14 enum mcan_fec_config_t

Enumerator

kMCAN_disable Disable filter element.
kMCAN_storeinFifo0 Store in Rx FIFO 0 if filter matches.
kMCAN_storeinFifo1 Store in Rx FIFO 1 if filter matches.
kMCAN_reject Reject ID if filter matches.
kMCAN_setprio Set priority if filter matches.
kMCAN_setpriofifo0 Set priority and store in FIFO 0 if filter matches.
kMCAN_setpriofifo1 Set priority and store in FIFO 1 if filter matches.
kMCAN_storeinbuffer Store into Rx Buffer or as debug message.

27.5.15 enum mcan_filter_type_t

Enumerator

kMCAN_range Range filter from SFID1 to SFID2.
kMCAN_dual Dual ID filter for SFID1 or SFID2.
kMCAN_classic Classic filter: SFID1 = filter, SFID2 = mask.
kMCAN_disableORrange2 Filter element disabled for standard filter or Range filter, XIDAM mask not applied for extended filter.

27.6 Function Documentation

27.6.1 void MCAN_Init (CAN_Type * *base,* const mcan_config_t * *config,* uint32_t *sourceClock_Hz*)

This function initializes the MCAN module with user-defined settings. This example shows how to set up the mcan_config_t parameters and how to call the MCAN_Init function by passing in these parameters.

```
*   mcan_config_t config;
*   config->baudRateA = 500000U;
*   config->baudRateD = 500000U;
*   config->enableCanfdNormal = false;
*   config->enableCanfdSwitch = false;
*   config->enableLoopBackInt = false;
*   config->enableLoopBackExt = false;
*   config->enableBusMon = false;
*   MCAN_Init(CANFD0, &config, 8000000UL);
*
```

Parameters

base	MCAN peripheral base address.
config	Pointer to the user-defined configuration structure.
sourceClock_Hz	MCAN Protocol Engine clock source frequency in Hz.

27.6.2 void MCAN_GetDefaultConfig (mcan_config_t * *config*)

This function initializes the MCAN configuration structure to default values. The default values are as follows. config->baudRateA = 500000U; config->baudRateD = 500000U; config->enableCanfdNormal = false; config->enableCanfdSwitch = false; config->enableLoopBackInt = false; config->enableLoopBackExt = false; config->enableBusMon = false;

Parameters

config	Pointer to the MCAN configuration structure.

27.6.3 void MCAN_EnterNormalMode (CAN_Type * *base*)

After initialization, INIT bit in CCCR register must be cleared to enter normal mode thus synchronizes to the CAN bus and ready for communication.

Parameters

base	MCAN peripheral base address.

27.6.4 static void MCAN_SetMsgRAMBase (CAN_Type * *base,* uint32_t *value*) [inline], [static]

This function sets the Message RAM base address.

Parameters

base	MCAN peripheral base address.

value	Desired Message RAM base.

27.6.5 static uint32_t MCAN_GetMsgRAMBase (CAN_Type * *base*) [inline], [static]

This function gets the Message RAM base address.

Parameters

base	MCAN peripheral base address.

Returns

Message RAM base address.

27.6.6 void MCAN_SetArbitrationTimingConfig (CAN_Type * *base*, const mcan_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 MCAN_Init() and fill the baud rate field with a desired value. This provides the default arbitration phase timing characteristics.

Note that calling MCAN_SetArbitrationTimingConfig() overrides the baud rate set in MCAN_Init().

Parameters

base	MCAN peripheral base address.
config	Pointer to the timing configuration structure.

27.6.7 void MCAN_SetDataTimingConfig (CAN_Type * *base*, const mcan_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 MCAN_Init() and fill the baud rate field with a desired value. This provides the default data phase timing characteristics.

Note that calling MCAN_SetArbitrationTimingConfig() overrides the baud rate set in MCAN_Init().

Parameters

base	MCAN peripheral base address.
config	Pointer to the timing configuration structure.

27.6.8 void MCAN_SetRxFifo0Config (CAN_Type * *base*, const mcan_rx_fifo_config_t * *config*)

This function sets start address, element size, watermark, operation mode and datafield size of the recieve fifo 0.

Parameters

base	MCAN peripheral base address.
config	The receive fifo 0 configuration structure.

27.6.9 void MCAN_SetRxFifo1Config (CAN_Type * *base*, const mcan_rx_fifo_config_t * *config*)

This function sets start address, element size, watermark, operation mode and datafield size of the recieve fifo 1.

Parameters

base	MCAN peripheral base address.
config	The receive fifo 1 configuration structure.

27.6.10 void MCAN_SetRxBufferConfig (CAN_Type * *base*, const mcan_rx_buffer_config_t * *config*)

This function sets start address and datafield size of the recieve buffer.

Parameters

base	MCAN peripheral base address.

config	The receive buffer configuration structure.

27.6.11 void MCAN_SetTxEventfifoConfig (CAN_Type * *base*, const mcan_tx_fifo_config_t * *config*)

This function sets start address, element size, watermark of the transmit event fifo.

Parameters

base	MCAN peripheral base address.
config	The transmit event fifo configuration structure.

27.6.12 void MCAN_SetTxBufferConfig (CAN_Type * *base*, const mcan_tx_buffer_config_t * *config*)

This function sets start address, element size, fifo/queue mode and datafield size of the transmit buffer.

Parameters

base	MCAN peripheral base address.
config	The transmit buffer configuration structure.

27.6.13 void MCAN_SetFilterConfig (CAN_Type * *base*, const mcan_frame_filter_config_t * *config*)

This function sets remote and non masking frames in global filter configuration, also the start address, list size in standard/extended ID filter configuration.

Parameters

base	MCAN peripheral base address.
config	The MCAN filter configuration.

27.6.14 void MCAN_SetSTDFilterElement (CAN_Type * *base,* const mcan_frame_filter_config_t * *config,* const mcan_std_filter_element_config_t * *filter,* uint8_t *idx*)

This function sets remote and non masking frames in global filter configuration, also the start address, list size in standard/extended ID filter configuration.

Parameters

base	MCAN peripheral base address.
config	The MCAN filter configuration.

27.6.15 void MCAN_SetEXTFilterElement (CAN_Type * *base*, const mcan_frame_filter_config_t * *config*, const mcan_ext_filter_element_config_t * *filter*, uint8_t *idx*)

This function sets remote and non masking frames in global filter configuration, also the start address, list size in standard/extended ID filter configuration.

Parameters

base	MCAN peripheral base address.
config	The MCAN filter configuration.

27.6.16 static uint32_t MCAN_GetStatusFlag (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function gets all MCAN interrupt status flags.

Parameters

base	MCAN peripheral base address.
mask	The ORed MCAN interrupt mask.

Returns

MCAN status flags which are ORed.

27.6.17 static void MCAN_ClearStatusFlag (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears MCAN interrupt status flags.

Parameters

base	MCAN peripheral base address.
mask	The ORed MCAN interrupt mask.

27.6.18 static bool MCAN_GetRxBufferStatusFlag (CAN_Type * *base,* uint8_t *idx*) [inline], [static]

This function gets new data flag of specific Rx Buffer.

Parameters

base	MCAN peripheral base address.
idx	Rx Buffer index.

Returns

Rx Buffer new data status flag.

27.6.19 static void MCAN_ClearRxBufferStatusFlag (CAN_Type * *base,* uint8_t *idx*) [inline], [static]

This function clears new data flag of specific Rx Buffer.

Parameters

base	MCAN peripheral base address.
idx	Rx Buffer index.

27.6.20 static void MCAN_EnableInterrupts (CAN_Type * *base,* uint32_t *line,* uint32_t *mask*) [inline], [static]

This function enables the MCAN interrupts according to the provided interrupt line and mask. The mask is a logical OR of enumeration members.

Parameters

base	MCAN peripheral base address.
line	Interrupt line number, 0 or 1.
mask	The interrupts to enable.

27.6.21 static void MCAN_EnableTransmitBufferInterrupts (CAN_Type * *base,* uint8_t *idx*) [inline], [static]

This function enables the MCAN Tx Buffer interrupts.

Parameters

base	MCAN peripheral base address.
idx	Tx Buffer index.

27.6.22 static void MCAN_DisableTransmitBufferInterrupts (CAN_Type * *base,* uint8_t *idx*) [inline], [static]

This function disables the MCAN Tx Buffer interrupts.

Parameters

base	MCAN peripheral base address.
idx	Tx Buffer index.

27.6.23 static void MCAN_DisableInterrupts (CAN_Type * *base,* uint32_t *mask*) [inline], [static]

This function disables the MCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members.

Parameters

base	MCAN peripheral base address.
mask	The interrupts to disable.

27.6.24 status_t MCAN_WriteTxBuffer (CAN_Type * *base*, uint8_t *idx*, const mcan_tx_buffer_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	MCAN peripheral base address.
idx	The MCAN Tx Buffer index.
txFrame	Pointer to CAN message frame to be sent.

27.6.25 status_t MCAN_ReadRxFifo (CAN_Type * *base*, uint8_t *fifoBlock*, mcan_rx_buffer_frame_t * *rxFrame*)

This function reads a CAN message from the Rx FIFO in the Message RAM.

Parameters

base	MCAN peripheral base address.
fifoBlock	Rx FIFO block 0 or 1.
rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Read Message from Rx FIFO successfully.

27.6.26 static void MCAN_TransmitAddRequest (CAN_Type * *base*, uint8_t *idx*) [inline], [static]

This function add sending request to corresponding Tx Buffer.

Parameters

base	MCAN peripheral base address.
idx	Tx Buffer index.

27.6.27 static void MCAN_TransmitCancelRequest (CAN_Type * *base*, uint8_t *idx*) [inline], [static]

This function clears Tx buffer request pending bit.

Parameters

base	MCAN peripheral base address.
idx	Tx Buffer index.

27.6.28 status_t MCAN_TransferSendBlocking (CAN_Type * *base*, uint8_t *idx*, mcan_tx_buffer_frame_t * *txFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	MCAN peripheral base pointer.
idx	The MCAN 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.

27.6.29 status_t MCAN_TransferReceiveFifoBlocking (CAN_Type * *base*, uint8_t *fifoBlock*, mcan_rx_buffer_frame_t * *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

base	MCAN peripheral base pointer.
fifoBlock	Rx FIFO block, 0 or 1.
rxFrame	Pointer to CAN message frame structure for reception.

Return values

kStatus_Success	- Read Message from Rx FIFO successfully.
kStatus_Fail	- No new message in Rx FIFO.

27.6.30 void MCAN_TransferCreateHandle (CAN_Type * *base,* mcan_handle_t * *handle,* mcan_transfer_callback_t *callback,* void * *userData*)

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

Parameters

base	MCAN peripheral base address.
handle	MCAN handle pointer.
callback	The callback function.
userData	The parameter of the callback function.

27.6.31 status_t MCAN_TransferSendNonBlocking (CAN_Type * *base,* mcan_handle_t * *handle,* mcan_buffer_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	MCAN peripheral base address.
handle	MCAN handle pointer.

xfer	MCAN Buffer transfer structure. See the mcan_buffer_transfer_t.

Return values

kStatus_Success	Start Tx Buffer sending process successfully.
kStatus_Fail	Write Tx Buffer failed.
kStatus_MCAN_TxBusy	Tx Buffer is in use.

27.6.32 status_t MCAN_TransferReceiveFifoNonBlocking (CAN_Type * *base*, uint8_t *fifoBlock*, mcan_handle_t * *handle*, mcan_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	MCAN peripheral base address.
handle	MCAN handle pointer.
fifoBlock	Rx FIFO block, 0 or 1.
xfer	MCAN Rx FIFO transfer structure. See the mcan_fifo_transfer_t.

Return values

kStatus_Success	- Start Rx FIFO receiving process successfully.
kStatus_MCAN_RxFifo0-Busy	- Rx FIFO 0 is currently in use.
kStatus_MCAN_RxFifo1-Busy	- Rx FIFO 1 is currently in use.

27.6.33 void MCAN_TransferAbortSend (CAN_Type * *base*, mcan_handle_t * *handle*, uint8_t *bufferIdx*)

This function aborts the interrupt driven message send process.

Parameters

base	MCAN peripheral base address.
handle	MCAN handle pointer.
bufferIdx	The MCAN Buffer index.

27.6.34 void MCAN_TransferAbortReceiveFifo (CAN_Type * *base*, uint8_t *fifoBlock*, mcan_handle_t * *handle*)

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

base	MCAN peripheral base address.
fifoBlock	MCAN Fifo block, 0 or 1.
handle	MCAN handle pointer.

27.6.35 void MCAN_TransferHandleIRQ (CAN_Type * *base*, mcan_handle_t * *handle*)

This function handles the MCAN Error, the Buffer, and the Rx FIFO IRQ request.

Parameters

base	MCAN peripheral base address.
handle	MCAN handle pointer.

Chapter 28
MRT: Multi-Rate Timer

28.1 Overview

The SDK provides a driver for the Multi-Rate Timer (MRT) of LPC devices.

28.2 Function groups

The MRT driver supports operating the module as a time counter.

28.2.1 Initialization and deinitialization

The function MRT_Init() initializes the MRT with specified configurations. The function MRT_GetDefaultConfig() gets the default configurations. The initialization function configures the MRT operating mode.

The function MRT_Deinit() stops the MRT timers and disables the module clock.

28.2.2 Timer period Operations

The function MRT_UpdateTimerPeriod() is used to update the timer period in units of count. The new value will be immediately loaded or will be loaded at the end of the current time interval.

The function MRT_GetCurrentTimerCount() reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. User can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds

28.2.3 Start and Stop timer operations

The function MRT_StartTimer() starts the timer counting. After calling this function, the timer loads the period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function MRT_StopTimer() stops the timer counting.

28.2.4 Get and release channel

These functions can be used to reserve and release a channel. The function MRT_GetIdleChannel() finds the available channel. This function returns the lowest available channel number. The function MRT_-ReleaseChannel() release the channel when the timer is using the multi-task mode. In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use.

28.2.5 Status

Provides functions to get and clear the PIT status.

28.2.6 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

28.3 Typical use case

28.3.1 MRT tick example

Updates the MRT period and toggles an LED periodically.

```
int main(void)
{
    uint32_t mrt_clock;

    /* Structure of initialize MRT */
    mrt_config_t mrtConfig;

    /* Initialize and enable LED */
    LED_RED_INIT(1);

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    mrt_clock = CLOCK_GetFreq(kCLOCK_BusClk);

    /* mrtConfig.enableMultiTask = false; */
    MRT_GetDefaultConfig(&mrtConfig);

    /* Init mrt module */
    MRT_Init(MRT0, &mrtConfig);

    /* Setup Channel 0 to be repeated */
    MRT_SetupChannelMode(MRT0, kMRT_Channel_0,
      kMRT_RepeatMode);

    /* Enable timer interrupts for channel 0 */
    MRT_EnableInterrupts(MRT0, kMRT_Channel_0,
      kMRT_TimerInterruptEnable);

    /* Enable at the NVIC */
    EnableIRQ(MRT0_IRQn);

    /* Start channel 0 */
```

```
    PRINTF("\r\nStarting channel No.0 ...");
    MRT_StartTimer(MRT0, kMRT_Channel_0,
      USEC_TO_COUNT(250000U, mrt_clock));

    while (true)
    {
        /* Check whether occur interupt and toggle LED */
        if (true == mrtIsrFlag)
        {
            PRINTF("\r\n Channel No.0 interrupt is occured !");
            LED_RED_TOGGLE();
            mrtIsrFlag = false;
        }
    }
}
```

Files

- file fsl_mrt.h

Data Structures

- struct mrt_config_t
  MRT configuration structure. *More...*

Enumerations

- enum mrt_chnl_t {
  kMRT_Channel_0 = 0U,
  kMRT_Channel_1,
  kMRT_Channel_2,
  kMRT_Channel_3 }
  List of MRT channels.
- enum mrt_timer_mode_t {
  kMRT_RepeatMode = (0 << MRT_CHANNEL_CTRL_MODE_SHIFT),
  kMRT_OneShotMode = (1 << MRT_CHANNEL_CTRL_MODE_SHIFT),
  kMRT_OneShotStallMode = (2 << MRT_CHANNEL_CTRL_MODE_SHIFT) }
  List of MRT timer modes.
- enum mrt_interrupt_enable_t { kMRT_TimerInterruptEnable = MRT_CHANNEL_CTRL_INTEN_MASK }
  List of MRT interrupts.
- enum mrt_status_flags_t {
  kMRT_TimerInterruptFlag = MRT_CHANNEL_STAT_INTFLAG_MASK,
  kMRT_TimerRunFlag = MRT_CHANNEL_STAT_RUN_MASK }
  List of MRT status flags.

Driver version

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

Typical use case

Initialization and deinitialization

- void MRT_Init (MRT_Type *base, const mrt_config_t *config)
  Ungates the MRT clock and configures the peripheral for basic operation.
- void MRT_Deinit (MRT_Type *base)
  Gate the MRT clock.
- static void MRT_GetDefaultConfig (mrt_config_t *config)
  Fill in the MRT config struct with the default settings.
- static void MRT_SetupChannelMode (MRT_Type *base, mrt_chnl_t channel, const mrt_timer_mode_t mode)
  Sets up an MRT channel mode.

Interrupt Interface

- static void MRT_EnableInterrupts (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Enables the MRT interrupt.
- static void MRT_DisableInterrupts (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Disables the selected MRT interrupt.
- static uint32_t MRT_GetEnabledInterrupts (MRT_Type *base, mrt_chnl_t channel)
  Gets the enabled MRT interrupts.

Status Interface

- static uint32_t MRT_GetStatusFlags (MRT_Type *base, mrt_chnl_t channel)
  Gets the MRT status flags.
- static void MRT_ClearStatusFlags (MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
  Clears the MRT status flags.

Read and Write the timer period

- void MRT_UpdateTimerPeriod (MRT_Type *base, mrt_chnl_t channel, uint32_t count, bool immediateLoad)
  Used to update the timer period in units of count.
- static uint32_t MRT_GetCurrentTimerCount (MRT_Type *base, mrt_chnl_t channel)
  Reads the current timer counting value.

Timer Start and Stop

- static void MRT_StartTimer (MRT_Type *base, mrt_chnl_t channel, uint32_t count)
  Starts the timer counting.
- static void MRT_StopTimer (MRT_Type *base, mrt_chnl_t channel)
  Stops the timer counting.

Get & release channel

- static uint32_t MRT_GetIdleChannel (MRT_Type *base)
  Find the available channel.
- static void MRT_ReleaseChannel (MRT_Type *base, mrt_chnl_t channel)
  Release the channel when the timer is using the multi-task mode.

28.4 Data Structure Documentation

28.4.1 struct mrt_config_t

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the MRT_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 enableMultiTask
  true: Timers run in multi-task mode; false: Timers run in hardware status mode

28.5 Enumeration Type Documentation

28.5.1 enum mrt_chnl_t

Enumerator

kMRT_Channel_0 MRT channel number 0.
kMRT_Channel_1 MRT channel number 1.
kMRT_Channel_2 MRT channel number 2.
kMRT_Channel_3 MRT channel number 3.

28.5.2 enum mrt_timer_mode_t

Enumerator

kMRT_RepeatMode Repeat Interrupt mode.
kMRT_OneShotMode One-shot Interrupt mode.
kMRT_OneShotStallMode One-shot stall mode.

28.5.3 enum mrt_interrupt_enable_t

Enumerator

kMRT_TimerInterruptEnable Timer interrupt enable.

28.5.4 enum mrt_status_flags_t

Enumerator

kMRT_TimerInterruptFlag Timer interrupt flag.
kMRT_TimerRunFlag Indicates state of the timer.

28.6 Function Documentation

28.6.1 void MRT_Init (MRT_Type * *base*, const mrt_config_t * *config*)

Note

This API should be called at the beginning of the application using the MRT driver.

Parameters

base	Multi-Rate timer peripheral base address
config	Pointer to user's MRT config structure

28.6.2 void MRT_Deinit (MRT_Type * *base*)

Parameters

base	Multi-Rate timer peripheral base address

28.6.3 static void MRT_GetDefaultConfig (mrt_config_t * *config*) [inline], [static]

The default values are:

```
*     config->enableMultiTask = false;
*
```

Parameters

config	Pointer to user's MRT config structure.

28.6.4 static void MRT_SetupChannelMode (MRT_Type * *base*, mrt_chnl_t *channel*, const mrt_timer_mode_t *mode*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Channel that is being configured.
mode	Timer mode to use for the channel.

28.6.5 static void MRT_EnableInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The interrupts to enable. This is a logical OR of members of the enumeration mrt_interrupt_enable_t

28.6.6 static void MRT_DisableInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The interrupts to disable. This is a logical OR of members of the enumeration mrt_interrupt_enable_t

28.6.7 static uint32_t MRT_GetEnabledInterrupts (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration mrt_interrupt_enable_t

28.6.8 static uint32_t MRT_GetStatusFlags (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration mrt_status_flags_t

28.6.9 static void MRT_ClearStatusFlags (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
mask	The status flags to clear. This is a logical OR of members of the enumeration mrt_status_flags_t

28.6.10 void MRT_UpdateTimerPeriod (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *count*, bool *immediateLoad*)

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number
count	Timer period in units of ticks
immediateLoad	true: Load the new value immediately into the TIMER register; false: Load the new value at the end of current timer interval

28.6.11 static uint32_t MRT_GetCurrentTimerCount (MRT_Type * *base*, mrt_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	Multi-Rate timer peripheral base address
channel	Timer channel number

Returns

Current timer counting value in ticks

28.6.12 static void MRT_StartTimer (MRT_Type * *base*, mrt_chnl_t *channel*, uint32_t *count*) [inline], [static]

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note

User can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.
count	Timer period in units of ticks

28.6.13 static void MRT_StopTimer (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

This function stops the timer from counting.

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.

28.6.14 static uint32_t MRT_GetIdleChannel (MRT_Type * *base*) [inline], [static]

This function returns the lowest available channel number.

Parameters

base	Multi-Rate timer peripheral base address

28.6.15 static void MRT_ReleaseChannel (MRT_Type * *base*, mrt_chnl_t *channel*) [inline], [static]

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling MRT_GetIdleChannel() for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

base	Multi-Rate timer peripheral base address
channel	Timer channel number.

Chapter 29
OTP: One-Time Programmable memory and API

29.1 Overview

The KSDK provides a peripheral driver for the OTP module of LPC devices.

Main clock has to be set to a frequency stated in user manual prior to using OTP driver. OTP memory is manipulated by calling provided API stored in ROM. KSDK driver encapsulates this.

29.2 OTP example

This example shows how to write to OTP.

```
{
    status_t status;

    CLOCK_EnableClock(kCLOCK_Otp);

    status = OTP_EnableBankWriteMask(kOTP_Bank3);
    if (status != kStatus_Success)
    {
        return;
    }

    /* Unreversible operation */
    status = OTP_ProgramRegister(3U, 1U, 0xA5A5U);
    if (status != kStatus_Success)
    {
        return;
    }
}
```

Enumerations

- enum otp_bank_t {
  kOTP_Bank0 = 0x1U,
  kOTP_Bank1 = 0x2U,
  kOTP_Bank2 = 0x4U,
  kOTP_Bank3 = 0x8U }
  Bank bit flags.
- enum otp_word_t {
  kOTP_Word0 = 0x1U,
  kOTP_Word1 = 0x2U,
  kOTP_Word2 = 0x4U,
  kOTP_Word3 = 0x8U }
  Bank word bit flags.
- enum otp_lock_t {
  kOTP_LockDontLock = 0U,
  kOTP_LockLock = 1U }

Lock modifications of a read or write access to a bank register.

- enum _otp_status {
  kStatus_OTP_WrEnableInvalid = MAKE_STATUS(kStatusGroup_OTP, 0x1U),
  kStatus_OTP_SomeBitsAlreadyProgrammed = MAKE_STATUS(kStatusGroup_OTP, 0x2U),
  kStatus_OTP_AllDataOrMaskZero = MAKE_STATUS(kStatusGroup_OTP, 0x3U),
  kStatus_OTP_WriteAccessLocked = MAKE_STATUS(kStatusGroup_OTP, 0x4U),
  kStatus_OTP_ReadDataMismatch = MAKE_STATUS(kStatusGroup_OTP, 0x5U),
  kStatus_OTP_UsbIdEnabled = MAKE_STATUS(kStatusGroup_OTP, 0x6U),
  kStatus_OTP_EthMacEnabled = MAKE_STATUS(kStatusGroup_OTP, 0x7U),
  kStatus_OTP_AesKeysEnabled = MAKE_STATUS(kStatusGroup_OTP, 0x8U),
  kStatus_OTP_IllegalBank = MAKE_STATUS(kStatusGroup_OTP, 0x9U),
  kStatus_OTP_ShufflerConfigNotValid = MAKE_STATUS(kStatusGroup_OTP, 0xAU),
  kStatus_OTP_ShufflerNotEnabled = MAKE_STATUS(kStatusGroup_OTP, 0xBU),
  kStatus_OTP_ShufflerCanOnlyProgSingleKey,
  kStatus_OTP_IllegalProgramData = MAKE_STATUS(kStatusGroup_OTP, 0xCU),
  kStatus_OTP_ReadAccessLocked = MAKE_STATUS(kStatusGroup_OTP, 0xDU) }
  OTP error codes.

Functions

- static status_t OTP_Init (void)
  Initializes OTP controller.
- static status_t OTP_EnableBankWriteMask (otp_bank_t bankMask)
  Unlock one or more OTP banks for write access.
- static status_t OTP_DisableBankWriteMask (otp_bank_t bankMask)
  Lock one or more OTP banks for write access.
- static status_t OTP_EnableBankWriteLock (uint32_t bankIndex, otp_word_t regEnableMask, otp_word_t regDisableMask, otp_lock_t lockWrite)
  Locks or unlocks write access to a register of an OTP bank and possibly lock un/locking of it.
- static status_t OTP_EnableBankReadLock (uint32_t bankIndex, otp_word_t regEnableMask, otp_word_t regDisableMask, otp_lock_t lockWrite)
  Locks or unlocks read access to a register of an OTP bank and possibly lock un/locking of it.
- static status_t OTP_ProgramRegister (uint32_t bankIndex, uint32_t regIndex, uint32_t value)
  Program a single register in an OTP bank.
- static uint32_t OTP_GetDriverVersion (void)
  Returns the version of the OTP driver in ROM.

Driver version

- #define FSL_OTP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  OTP driver version 2.0.0.

29.3 Macro Definition Documentation

29.3.1 #define FSL_OTP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Current version: 2.0.0

Change log:

- Version 2.0.0
  - Initial version.

29.4 Enumeration Type Documentation

29.4.1 enum otp_bank_t

Enumerator

kOTP_Bank0 Bank 0.
kOTP_Bank1 Bank 1.
kOTP_Bank2 Bank 2.
kOTP_Bank3 Bank 3.

29.4.2 enum otp_word_t

Enumerator

kOTP_Word0 Word 0.
kOTP_Word1 Word 1.
kOTP_Word2 Word 2.
kOTP_Word3 Word 3.

29.4.3 enum otp_lock_t

Enumerator

kOTP_LockDontLock Do not lock.
kOTP_LockLock Lock till reset.

29.4.4 enum _otp_status

Enumerator

kStatus_OTP_WrEnableInvalid Write enable invalid.
kStatus_OTP_SomeBitsAlreadyProgrammed Some bits already programmed.
kStatus_OTP_AllDataOrMaskZero All data or mask zero.
kStatus_OTP_WriteAccessLocked Write access locked.
kStatus_OTP_ReadDataMismatch Read data mismatch.
kStatus_OTP_UsbIdEnabled USB ID enabled.
kStatus_OTP_EthMacEnabled Ethernet MAC enabled.
kStatus_OTP_AesKeysEnabled AES keys enabled.

kStatus_OTP_IllegalBank Illegal bank.
kStatus_OTP_ShufflerConfigNotValid Shuffler config not valid.
kStatus_OTP_ShufflerNotEnabled Shuffler not enabled.
kStatus_OTP_ShufflerCanOnlyProgSingleKey Shuffler can only program single key.
kStatus_OTP_IllegalProgramData Illegal program data.
kStatus_OTP_ReadAccessLocked Read access locked.

29.5 Function Documentation

29.5.1 static status_t OTP_Init (void) [inline], [static]

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.2 static status_t OTP_EnableBankWriteMask (otp_bank_t *bankMask*) [inline], [static]

Parameters

bankMask	bit flag that specifies which banks to unlock.

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.3 static status_t OTP_DisableBankWriteMask (otp_bank_t *bankMask*) [inline], [static]

Parameters

bankMask	bit flag that specifies which banks to lock.

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.4 static status_t OTP_EnableBankWriteLock (uint32_t *bankIndex,* otp_word_t *regEnableMask,* otp_word_t *regDisableMask,* otp_lock_t *lockWrite*) [inline], [static]

Parameters

bankIndex	OTP bank index, 0 = bank 0, 1 = bank 1 etc.
regEnableMask	bit flag that specifies for which words to enable writing.
regDisable-Mask	bit flag that specifies for which words to disable writing.
lockWrite	specifies if access set can be modified or is locked till reset.

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.5 static status_t OTP_EnableBankReadLock (uint32_t *bankIndex,* otp_word_t *regEnableMask,* otp_word_t *regDisableMask,* otp_lock_t *lockWrite*) [inline], [static]

Parameters

bankIndex	OTP bank index, 0 = bank 0, 1 = bank 1 etc.
regEnableMask	bit flag that specifies for which words to enable reading.
regDisable-Mask	bit flag that specifies for which words to disable reading.
lockWrite	specifies if access set can be modified or is locked till reset.

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.6 static status_t OTP_ProgramRegister (uint32_t *bankIndex,* uint32_t *regIndex,* uint32_t *value*) [inline], [static]

Parameters

bankIndex	OTP bank index, 0 = bank 0, 1 = bank 1 etc.
regIndex	OTP register index.
value	value to write.

Returns

kStatus_Success upon successful execution, error status otherwise.

29.5.7 static uint32_t OTP_GetDriverVersion (void) [inline], [static]

Returns

version.

Chapter 30
PINT: Pin Interrupt and Pattern Match Driver

30.1 Overview

The SDK provides a driver for the Pin Interrupt and Pattern match (PINT).

It can configure one or more pins to generate a pin interrupt when the pin or pattern match conditions are met. The pins do not have to be configured as gpio pins however they must be connected to PINT via INPUTMUX. Only the pin interrupt or pattern match function can be active for interrupt generation. If the pin interrupt function is enabled then the pattern match function can be used for wakeup via RXEV.

30.2 Pin Interrupt and Pattern match Driver operation

PINT_PinInterruptConfig() function configures the pins for pin interrupt.

PINT_PatternMatchConfig() function configures the pins for pattern match.

30.2.1 Pin Interrupt use case

```
void pint_intr_callback(pint_pin_int_t pintr, uint32_t pmatch_status)
{
    /* Take action for pin interrupt */
}

    /* Connect trigger sources to PINT */
    INPUTMUX_Init(INPUTMUX);
    INPUTMUX_AttachSignal(INPUTMUX, kPINT_PinInt0, PINT_PIN_INT0_SRC);

    /* Initialize PINT */
    PINT_Init(PINT);

    /* Setup Pin Interrupt 0 for rising edge */
    PINT_PinInterruptConfig(PINT, kPINT_PinInt0,
      kPINT_PinIntEnableRiseEdge, pint_intr_callback);


    /* Enable callbacks for PINT */
    PINT_EnableCallback(PINT);
```

30.2.2 Pattern match use case

```
void pint_intr_callback(pint_pin_int_t pintr, uint32_t pmatch_status)
{
    /* Take action for pin interrupt */
}

    pint_pmatch_cfg_t pmcfg;

    /* Connect trigger sources to PINT */
    INPUTMUX_Init(INPUTMUX);
```

```
INPUTMUX_AttachSignal(INPUTMUX, kPINT_PinInt0, PINT_PIN_INT0_SRC);

/* Initialize PINT */
PINT_Init(PINT);

/* Setup Pattern Match Bit Slice 0 */
pmcfg.bs_src = kPINT_PatternMatchInp0Src;
pmcfg.bs_cfg = kPINT_PatternMatchStickyFall;
pmcfg.callback = pint_intr_callback;
pmcfg.end_point = true;
PINT_PatternMatchConfig(PINT,
  kPINT_PatternMatchBSlice0, &pmcfg);

/* Enable PatternMatch */
PINT_PatternMatchEnable(PINT);

/* Enable callbacks for PINT */
PINT_EnableCallback(PINT);
```

Files

- file fsl_pint.h

Typedefs

- typedef void(∗ pint_cb_t)(pint_pin_int_t pintr, uint32_t pmatch_status)
  PINT Callback function.

Enumerations

- enum pint_pin_enable_t {
  kPINT_PinIntEnableNone = 0U,
  kPINT_PinIntEnableRiseEdge = PINT_PIN_RISE_EDGE,
  kPINT_PinIntEnableFallEdge = PINT_PIN_FALL_EDGE,
  kPINT_PinIntEnableBothEdges = PINT_PIN_BOTH_EDGE,
  kPINT_PinIntEnableLowLevel = PINT_PIN_LOW_LEVEL,
  kPINT_PinIntEnableHighLevel = PINT_PIN_HIGH_LEVEL }
  PINT Pin Interrupt enable type.
- enum pint_pin_int_t { kPINT_PinInt0 = 0U }
  PINT Pin Interrupt type.
- enum pint_pmatch_input_src_t {
  kPINT_PatternMatchInp0Src = 0U,
  kPINT_PatternMatchInp1Src = 1U,
  kPINT_PatternMatchInp2Src = 2U,
  kPINT_PatternMatchInp3Src = 3U,
  kPINT_PatternMatchInp4Src = 4U,
  kPINT_PatternMatchInp5Src = 5U,
  kPINT_PatternMatchInp6Src = 6U,
  kPINT_PatternMatchInp7Src = 7U }
  PINT Pattern Match bit slice input source type.
- enum pint_pmatch_bslice_t { kPINT_PatternMatchBSlice0 = 0U }
  PINT Pattern Match bit slice type.

- enum pint_pmatch_bslice_cfg_t {
  kPINT_PatternMatchAlways = 0U,
  kPINT_PatternMatchStickyRise = 1U,
  kPINT_PatternMatchStickyFall = 2U,
  kPINT_PatternMatchStickyBothEdges = 3U,
  kPINT_PatternMatchHigh = 4U,
  kPINT_PatternMatchLow = 5U,
  kPINT_PatternMatchNever = 6U,
  kPINT_PatternMatchBothEdges = 7U }
  PINT Pattern Match configuration type.

Functions

- void PINT_Init (PINT_Type *base)
  Initialize PINT peripheral.
- void PINT_PinInterruptConfig (PINT_Type *base, pint_pin_int_t intr, pint_pin_enable_t enable, pint_cb_t callback)
  Configure PINT peripheral pin interrupt.
- void PINT_PinInterruptGetConfig (PINT_Type *base, pint_pin_int_t pintr, pint_pin_enable_t *enable, pint_cb_t *callback)
  Get PINT peripheral pin interrupt configuration.
- static void PINT_PinInterruptClrStatus (PINT_Type *base, pint_pin_int_t pintr)
  Clear Selected pin interrupt status.
- static uint32_t PINT_PinInterruptGetStatus (PINT_Type *base, pint_pin_int_t pintr)
  Get Selected pin interrupt status.
- static void PINT_PinInterruptClrStatusAll (PINT_Type *base)
  Clear all pin interrupts status.
- static uint32_t PINT_PinInterruptGetStatusAll (PINT_Type *base)
  Get all pin interrupts status.
- static void PINT_PinInterruptClrFallFlag (PINT_Type *base, pint_pin_int_t pintr)
  Clear Selected pin interrupt fall flag.
- static uint32_t PINT_PinInterruptGetFallFlag (PINT_Type *base, pint_pin_int_t pintr)
  Get selected pin interrupt fall flag.
- static void PINT_PinInterruptClrFallFlagAll (PINT_Type *base)
  Clear all pin interrupt fall flags.
- static uint32_t PINT_PinInterruptGetFallFlagAll (PINT_Type *base)
  Get all pin interrupt fall flags.
- static void PINT_PinInterruptClrRiseFlag (PINT_Type *base, pint_pin_int_t pintr)
  Clear Selected pin interrupt rise flag.
- static uint32_t PINT_PinInterruptGetRiseFlag (PINT_Type *base, pint_pin_int_t pintr)
  Get selected pin interrupt rise flag.
- static void PINT_PinInterruptClrRiseFlagAll (PINT_Type *base)
  Clear all pin interrupt rise flags.
- static uint32_t PINT_PinInterruptGetRiseFlagAll (PINT_Type *base)
  Get all pin interrupt rise flags.
- void PINT_PatternMatchConfig (PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
  Configure PINT pattern match.
- void PINT_PatternMatchGetConfig (PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)

Get PINT pattern match configuration.
- static uint32_t PINT_PatternMatchGetStatus (PINT_Type *base, pint_pmatch_bslice_t bslice)
  Get pattern match bit slice status.
- static uint32_t PINT_PatternMatchGetStatusAll (PINT_Type *base)
  Get status of all pattern match bit slices.
- uint32_t PINT_PatternMatchResetDetectLogic (PINT_Type *base)
  Reset pattern match detection logic.
- static void PINT_PatternMatchEnable (PINT_Type *base)
  Enable pattern match function.
- static void PINT_PatternMatchDisable (PINT_Type *base)
  Disable pattern match function.
- static void PINT_PatternMatchEnableRXEV (PINT_Type *base)
  Enable RXEV output.
- static void PINT_PatternMatchDisableRXEV (PINT_Type *base)
  Disable RXEV output.
- void PINT_EnableCallback (PINT_Type *base)
  Enable callback.
- void PINT_DisableCallback (PINT_Type *base)
  Disable callback.
- void PINT_Deinit (PINT_Type *base)
  Deinitialize PINT peripheral.

Driver version

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

30.3 Typedef Documentation

30.3.1 typedef void(* pint_cb_t)(pint_pin_int_t pintr, uint32_t pmatch_status)

30.4 Enumeration Type Documentation

30.4.1 enum pint_pin_enable_t

Enumerator

kPINT_PinIntEnableNone Do not generate Pin Interrupt.
kPINT_PinIntEnableRiseEdge Generate Pin Interrupt on rising edge.
kPINT_PinIntEnableFallEdge Generate Pin Interrupt on falling edge.
kPINT_PinIntEnableBothEdges Generate Pin Interrupt on both edges.
kPINT_PinIntEnableLowLevel Generate Pin Interrupt on low level.
kPINT_PinIntEnableHighLevel Generate Pin Interrupt on high level.

30.4.2 enum pint_pin_int_t

Enumerator

kPINT_PinInt0 Pin Interrupt 0.

30.4.3 enum pint_pmatch_input_src_t

Enumerator

kPINT_PatternMatchInp0Src Input source 0.
kPINT_PatternMatchInp1Src Input source 1.
kPINT_PatternMatchInp2Src Input source 2.
kPINT_PatternMatchInp3Src Input source 3.
kPINT_PatternMatchInp4Src Input source 4.
kPINT_PatternMatchInp5Src Input source 5.
kPINT_PatternMatchInp6Src Input source 6.
kPINT_PatternMatchInp7Src Input source 7.

30.4.4 enum pint_pmatch_bslice_t

Enumerator

kPINT_PatternMatchBSlice0 Bit slice 0.

30.4.5 enum pint_pmatch_bslice_cfg_t

Enumerator

kPINT_PatternMatchAlways Always Contributes to product term match.
kPINT_PatternMatchStickyRise Sticky Rising edge.
kPINT_PatternMatchStickyFall Sticky Falling edge.
kPINT_PatternMatchStickyBothEdges Sticky Rising or Falling edge.
kPINT_PatternMatchHigh High level.
kPINT_PatternMatchLow Low level.
kPINT_PatternMatchNever Never contributes to product term match.
kPINT_PatternMatchBothEdges Either rising or falling edge.

30.5 Function Documentation

30.5.1 void PINT_Init (PINT_Type * *base*)

This function initializes the PINT peripheral and enables the clock.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.2 void PINT_PinInterruptConfig (PINT_Type * *base*, pint_pin_int_t *intr*, pint_pin_enable_t *enable*, pint_cb_t *callback*)

This function configures a given pin interrupt.

Parameters

base	Base address of the PINT peripheral.
intr	Pin interrupt.
enable	Selects detection logic.
callback	Callback.

Return values

None.	

30.5.3 void PINT_PinInterruptGetConfig (PINT_Type * *base*, pint_pin_int_t *pintr*, pint_pin_enable_t * *enable*, pint_cb_t * *callback*)

This function returns the configuration of a given pin interrupt.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.
enable	Pointer to store the detection logic.

callback	Callback.

Return values

None.	

30.5.4 static void PINT_PinInterruptClrStatus (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function clears the selected pin interrupt status.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

30.5.5 static uint32_t PINT_PinInterruptGetStatus (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt status.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

status	= 0 No pin interrupt request. = 1 Selected Pin interrupt request active.

30.5.6 static void PINT_PinInterruptClrStatusAll (PINT_Type * *base*) [inline], [static]

This function clears the status of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.7 static uint32_t PINT_PinInterruptGetStatusAll (PINT_Type * *base*) [inline], [static]

This function returns the status of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

status	Each bit position indicates the status of corresponding pin interrupt. = 0 No pin interrupt request. = 1 Pin interrupt request active.

30.5.8 static void PINT_PinInterruptClrFallFlag (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function clears the selected pin interrupt fall flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

30.5.9 static uint32_t PINT_PinInterruptGetFallFlag (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt fall flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

flag	= 0 Falling edge has not been detected. = 1 Falling edge has been detected.

30.5.10 static void PINT_PinInterruptClrFallFlagAll (PINT_Type * *base*) [inline], [static]

This function clears the fall flag for all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.11 static uint32_t PINT_PinInterruptGetFallFlagAll (PINT_Type * *base*) [inline], [static]

This function returns the fall flag of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

flags	Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

30.5.12 static void PINT_PinInterruptClrRiseFlag (PINT_Type * *base,* pint_pin_int_t *pintr*) [inline], [static]

This function clears the selected pin interrupt rise flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

None.	

30.5.13 static uint32_t PINT_PinInterruptGetRiseFlag (PINT_Type * *base*, pint_pin_int_t *pintr*) [inline], [static]

This function returns the selected pin interrupt rise flag.

Parameters

base	Base address of the PINT peripheral.
pintr	Pin interrupt.

Return values

flag	= 0 Rising edge has not been detected. = 1 Rising edge has been detected.

30.5.14 static void PINT_PinInterruptClrRiseFlagAll (PINT_Type * *base*) [inline], [static]

This function clears the rise flag for all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.15 static uint32_t PINT_PinInterruptGetRiseFlagAll (PINT_Type * *base*) [inline], [static]

This function returns the rise flag of all pin interrupts.

Parameters

base	Base address of the PINT peripheral.

Return values

flags	Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

30.5.16 void PINT_PatternMatchConfig (PINT_Type * *base*, pint_pmatch_bslice_t *bslice*, pint_pmatch_cfg_t * *cfg*)

This function configures a given pattern match bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.
cfg	Pointer to bit slice configuration.

Return values

None.	

30.5.17 void PINT_PatternMatchGetConfig (PINT_Type * *base*, pint_pmatch_bslice_t *bslice*, pint_pmatch_cfg_t * *cfg*)

This function returns the configuration of a given pattern match bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.
cfg	Pointer to bit slice configuration.

Return values

None.	

30.5.18 static uint32_t PINT_PatternMatchGetStatus (PINT_Type * *base*, pint_pmatch_bslice_t *bslice*) [inline], [static]

This function returns the status of selected bit slice.

Parameters

base	Base address of the PINT peripheral.
bslice	Pattern match bit slice number.

Return values

status	= 0 Match has not been detected. = 1 Match has been detected.

30.5.19 static uint32_t PINT_PatternMatchGetStatusAll (PINT_Type * *base*) [inline], [static]

This function returns the status of all bit slices.

Parameters

base	Base address of the PINT peripheral.

Return values

status	Each bit position indicates the match status of corresponding bit slice. = 0 Match has not been detected. = 1 Match has been detected.

30.5.20 uint32_t PINT_PatternMatchResetDetectLogic (PINT_Type * *base*)

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

base	Base address of the PINT peripheral.

Return values

pmstatus	Each bit position indicates the match status of corresponding bit slice. = 0 Match was detected. = 1 Match was not detected.

30.5.21 static void PINT_PatternMatchEnable (PINT_Type * *base*) [inline], [static]

This function enables the pattern match function.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.22 static void PINT_PatternMatchDisable (PINT_Type * *base*) [inline], [static]

This function disables the pattern match function.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.23 static void PINT_PatternMatchEnableRXEV (PINT_Type * *base*) [inline], [static]

This function enables the pattern match RXEV output.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.24 static void PINT_PatternMatchDisableRXEV (PINT_Type * *base*) [inline], [static]

This function disables the pattern match RXEV output.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.25 void PINT_EnableCallback (PINT_Type * *base*)

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

30.5.26 void PINT_DisableCallback (PINT_Type * *base*)

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

base	Base address of the peripheral.

Return values

None.	

30.5.27 void PINT_Deinit (PINT_Type * *base*)

This function disables the PINT clock.

Parameters

base	Base address of the PINT peripheral.

Return values

None.	

Chapter 31
RIT: Repetitive Interrupt Timer

31.1 Overview

The KSDK provides a driver for the Repetitive Interrupt Timer (RIT) of Kinetis devices.

31.2 Function groups

The RIT driver supports operating the module as a time counter.

31.2.1 Initialization and deinitialization

The function RIT_Init() initializes the RIT with specified configurations. The function RIT_GetDefault-Config() gets the default configurations. The initialization function configures the RIT operation normally in debug mode.

The function RIT_Deinit() disables the RIT timers and disables the module clock.

31.2.2 Timer read and write Operations

The function RIT_SetTimerCompare() sets the timer period in units of count. Timers counts from 0 to the count value set here. The function RIT_SetMaskBit() sets some bit which will be ignored in comparison between the compare and counter register.

The function RIT_GetCurrentTimerCount() reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. User can call the utility macros provided in fsl_common.h to convert to microseconds or milliseconds

31.2.3 Start and Stop timer operations

The function RIT_StartTimer() starts the timer counting. After calling this function, the timer counts up to the counter value set earlier by using the RIT_SetTimerCompare() function. Each time the timer reaches the count value, it generates a trigger pulse and sets the interrupt flag and set the counter to zero/continue counting when RIT_ClearCounter() set the Timer clear enable/disable.

The function RIT_StopTimer() stops the timer counting./∗ resets the timer's counter register.

Function groups

Data Structures

- struct rit_config_t
  RIT config structure. More...

Enumerations

- enum rit_status_flags_t { kRIT_TimerFlag = RIT_CTRL_RITINT_MASK }
  List of RIT status flags.

Driver version

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

Initialization and deinitialization

- void RIT_Init (RIT_Type *base, const rit_config_t *config)
  Ungates the RIT clock, enables the RIT module, and configures the peripheral for basic operations.
- void RIT_Deinit (RIT_Type *base)
  Gates the RIT clock and disables the RIT module.
- void RIT_GetDefaultConfig (rit_config_t *config)
  Fills in the RIT configuration structure with the default settings.

Status Interface

- static uint32_t RIT_GetStatusFlags (RIT_Type *base)
  Gets the RIT status flags.
- static void RIT_ClearStatusFlags (RIT_Type *base, uint32_t mask)
  Clears the RIT status flags.

Read and Write the timer period

- void RIT_SetTimerCompare (RIT_Type *base, uint64_t count)
  Sets the timer period in units of count.
- void RIT_SetMaskBit (RIT_Type *base, uint64_t count)
  Sets the mask bit of count compare.
- uint64_t RIT_GetCompareTimerCount (RIT_Type *base)
  Reads the current timer counting value of compare register.
- uint64_t RIT_GetCounterTimerCount (RIT_Type *base)
  Reads the current timer counting value of counter register.
- uint64_t RIT_GetMaskTimerCount (RIT_Type *base)
  Reads the current timer counting value of mask register.

Timer Start and Stop

- static void RIT_StartTimer (RIT_Type *base)
  Starts the timer counting.
- static void RIT_StopTimer (RIT_Type *base)
  Stops the timer counting.

31.3 Data Structure Documentation

31.3.1 struct rit_config_t

This structure holds the configuration settings for the RIT peripheral. To initialize this structure to reasonable defaults, call the RIT_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: The timer is halted when the processor is halted for debugging.

31.3.1.0.0.47 Field Documentation

31.3.1.0.0.47.1 bool rit_config_t::enableRunInDebug

; false: Debug has no effect on the timer operation.

31.4 Enumeration Type Documentation

31.4.1 enum rit_status_flags_t

Enumerator

kRIT_TimerFlag Timer flag.

31.5 Function Documentation

31.5.1 void RIT_Init (RIT_Type * *base*, const rit_config_t * *config*)

Note

This API should be called at the beginning of the application using the RIT driver.

Parameters

base	RIT peripheral base address
config	Pointer to the user's RIT config structure

31.5.2 void RIT_Deinit (RIT_Type * *base*)

Parameters

base	RIT peripheral base address

31.5.3 void RIT_GetDefaultConfig (rit_config_t * *config*)

The default values are as follows.

```
*     config->enableRunInDebug = false;
*
```

Parameters

config	Pointer to the onfiguration structure.

31.5.4 static uint32_t RIT_GetStatusFlags (RIT_Type * *base*) [inline], [static]

Parameters

base	RIT peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration rit_status_flags_t

31.5.5 static void RIT_ClearStatusFlags (RIT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	RIT peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration rit_-status_flags_t

31.5.6 void RIT_SetTimerCompare (RIT_Type * *base,* uint64_t *count*)

Timers begin counting from the value set by this function until it XXXXXXX, then it counting the value again. Software must stop the counter before reloading it with a new value..

Note

Users can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	RIT peripheral base address
count	Timer period in units of ticks

31.5.7 void RIT_SetMaskBit (RIT_Type * *base*, uint64_t *count*)

Timers begin counting from the value set by this function until it XXXXXXX, then it counting the value again. Software must stop the counter before reloading it with a new value..

Note

Users can call the utility macros provided in fsl_common.h to convert to ticks

Parameters

base	RIT peripheral base address
count	Timer period in units of ticks

31.5.8 uint64_t RIT_GetCompareTimerCount (RIT_Type * *base*)

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	RIT peripheral base address

Returns

Current timer counting value in ticks

Function Documentation

31.5.9 uint64_t RIT_GetCounterTimerCount (RIT_Type * *base*)

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	RIT peripheral base address

Returns

Current timer counting value in ticks

31.5.10 uint64_t RIT_GetMaskTimerCount (RIT_Type * *base*)

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec

Parameters

base	RIT peripheral base address

Returns

Current timer counting value in ticks

31.5.11 static void RIT_StartTimer (RIT_Type * *base*) [inline], [static]

After calling this function, timers load initial value(0U), count up to desired value or over-flow then the counter will count up again. Each time a timer reaches desired value, it generates a XXXXXXX and sets XXXXXXX.

Parameters

base	RIT peripheral base address

31.5.12 static void RIT_StopTimer (RIT_Type * *base*) [inline], [static]

This function stop timer counting. Timer reload their new value after the next time they call the RIT_-StartTimer.

Parameters

base	RIT peripheral base address
channel	Timer channel number.

Chapter 32
RNG: Random Number Generator

32.1 Overview

The KSDK provides a peripheral driver for the Random Number Generator module of LPC devices.

The Random Number Generator is a hardware module that generates 32-bit random numbers. Internally it is accessed by calling ROM API. A typical consumer is a pseudo random number generator (PRNG) which can be implemented to achieve both true randomness and cryptographic strength random numbers using the RNG output as its entropy seed. The data generated by a RNG is intended for direct use by functions that generate secret keys, per-message secrets, random challenges, and other similar quantities used in cryptographic algorithms.

32.2 Get random data from RNG

1. RNG_GetRandomData() function gets random data from the RNG module.

This example code shows how to get 128-bit random data from the RNG driver.

```
{
    uint32_t number;
    uint32_t skip;
    uint32_t data[4];

    /* Initialisation is done automatically by ROM API. */

    /* Get Random data */
    for (number = 0; number < 4; number++)
    {
        data[number] = RNG_GetRandomData();

        /* Skip next 32 random numbers for better entropy */
        for (skip = 0; skip < 32; skip++)
        {
            RNG_GetRandomData();
        }
    }

    /* Print data */
    PRINTF("0x");
    for (number = 0; number < 4; number++)
    {
        PRINTF("%08X", data[number]);
    }
    PRINTF("\r\n");
}
```

Functions

- static uint32_t RNG_GetRandomData (void)
  Gets random data.

Driver version

- #define FSL_RNG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  RNG driver version 2.0.0.

32.3 Macro Definition Documentation

32.3.1 #define FSL_RNG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Current version: 2.0.0

Change log:

- Version 2.0.0
  - Initial version.

32.4 Function Documentation

32.4.1 static uint32_t RNG_GetRandomData (void) [inline], [static]

This function returns single 32 bit random number.

Returns

random data

Chapter 33
RTC: Real Time Clock

33.1 Overview

The SDK provides a driver for the Real Time Clock (RTC).

33.2 Function groups

The RTC driver supports operating the module as a time counter.

33.2.1 Initialization and deinitialization

The function RTC_Init() initializes the RTC with specified configurations. The function RTC_GetDefault-Config() gets the default configurations.

The function RTC_Deinit() disables the RTC timer and disables the module clock.

33.2.2 Set & Get Datetime

The function RTC_SetDatetime() sets the timer period in seconds. User passes in the details in date & time format by using the below data structure.

```
typedef struct _rtc_datetime
{
    uint16_t year;
    uint8_t month;
    uint8_t day;
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
} rtc_datetime_t;
```

The function RTC_GetDatetime() reads the current timer value in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

33.2.3 Set & Get Alarm

The function RTC_SetAlarm() sets the alarm time period in seconds. User passes in the details in date & time format by using the datetime data structure.

The function RTC_GetAlarm() reads the alarm time in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

33.2.4 Start & Stop timer

The function RTC_StartTimer() starts the RTC time counter.

The function RTC_StopTimer() stops the RTC time counter.

33.2.5 Status

Provides functions to get and clear the RTC status.

33.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

33.2.7 High resolution timer

Provides functions to enable high resolution timer and set and get the wake time.

33.3 Typical use case

33.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
    uint32_t sec;
    uint32_t currSeconds;
    rtc_datetime_t date;

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    /* Init RTC */
    RTC_Init(RTC);

    PRINTF("RTC example: set up time to wake up an alarm\r\n");

    /* Set a start date time and start RT */
    date.year = 2014U;
    date.month = 12U;
    date.day = 25U;
    date.hour = 19U;
    date.minute = 0;
    date.second = 0;

    /* RTC time counter has to be stopped before setting the date & time in the TSR register */
    RTC_StopTimer(RTC);

    /* Set RTC time to default */
    RTC_SetDatetime(RTC, &date);
```

```
    /* Enable RTC alarm interrupt */
    RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

    /* Enable at the NVIC */
    EnableIRQ(RTC_IRQn);

    /* Start the RTC time counter */
    RTC_StartTimer(RTC);

    /* This loop will set the RTC alarm */
    while (1)
    {
        busyWait = true;
        /* Get date time */
        RTC_GetDatetime(RTC, &date);

        /* print default time */
        PRINTF("Current datetime: %04d-%02d-%02d %02d:%02d:%02d\r\n",
               date.year,
               date.month,
               date.day,
               date.hour,
               date.minute,
               date.second);

        /* Get alarm time from user */
        sec = 0;
        PRINTF("Please input the number of second to wait for alarm \r\n");
        PRINTF("The second must be positive value\r\n");
        while (sec < 1)
        {
            SCANF("%d", &sec);
        }

        /* Read the RTC seconds register to get current time in seconds */
        currSeconds = RTC->COUNT;

        /* Add alarm seconds to current time */
        currSeconds += sec;

        /* Set alarm time in seconds */
        RTC->MATCH = currSeconds;

        /* Get alarm time */
        RTC_GetAlarm(RTC, &date);

        /* Print alarm time */
        PRINTF("Alarm will occur at: %04d-%02d-%02d %02d:%02d:%02d\r\n",
               date.year,
               date.month,
               date.day,
               date.hour,
               date.minute,
               date.second);

        /* Wait until alarm occurs */
        while (busyWait)
        {
        }

        PRINTF("\r\n Alarm occurs !!!! ");
    }
}
```

Typical use case

Files

- file fsl_rtc.h

Data Structures

- struct rtc_datetime_t
  Structure is used to hold the date and time. More...

Enumerations

- enum rtc_interrupt_enable_t {
  kRTC_AlarmInterruptEnable = RTC_CTRL_ALARMDPD_EN_MASK,
  kRTC_WakeupInterruptEnable = RTC_CTRL_WAKEDPD_EN_MASK }
  List of RTC interrupts.
- enum rtc_status_flags_t {
  kRTC_AlarmFlag = RTC_CTRL_ALARM1HZ_MASK,
  kRTC_WakeupFlag = RTC_CTRL_WAKE1KHZ_MASK }
  List of RTC flags.

Functions

- static void RTC_SetWakeupCount (RTC_Type *base, uint16_t wakeupValue)
  Enable the RTC high resolution timer and set the wake-up time.
- static uint16_t RTC_GetWakeupCount (RTC_Type *base)
  Read actual RTC counter value.
- 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)
  Ungates the RTC clock and enables the RTC oscillator.
- static void RTC_Deinit (RTC_Type *base)
  Stop the timer and gate the RTC clock.

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.
- static 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.

33.4 Data Structure Documentation

33.4.1 struct rtc_datetime_t

Data Fields

- uint16_t year
  Range from 1970 to 2099.
- uint8_t month
  Range from 1 to 12.
- uint8_t day
  Range from 1 to 31 (depending on month).
- uint8_t hour
  Range from 0 to 23.
- uint8_t minute
  Range from 0 to 59.
- uint8_t second
  Range from 0 to 59.

33.4.1.0.0.48 Field Documentation

33.4.1.0.0.48.1 uint16_t rtc_datetime_t::year

33.4.1.0.0.48.2 uint8_t rtc_datetime_t::month

33.4.1.0.0.48.3 uint8_t rtc_datetime_t::day

33.4.1.0.0.48.4 uint8_t rtc_datetime_t::hour

33.4.1.0.0.48.5 uint8_t rtc_datetime_t::minute

33.4.1.0.0.48.6 uint8_t rtc_datetime_t::second

33.5 Enumeration Type Documentation

33.5.1 enum rtc_interrupt_enable_t

Enumerator

kRTC_AlarmInterruptEnable Alarm interrupt.
kRTC_WakeupInterruptEnable Wake-up interrupt.

33.5.2 enum rtc_status_flags_t

Enumerator

kRTC_AlarmFlag Alarm flag.
kRTC_WakeupFlag 1kHz wake-up timer flag

33.6 Function Documentation

33.6.1 void RTC_Init (RTC_Type * *base*)

Note

This API should be called at the beginning of the application using the RTC driver.

Parameters

base	RTC peripheral base address

33.6.2 static void RTC_Deinit (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

33.6.3 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

33.6.4 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.

33.6.5 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

33.6.6 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.

33.6.7 static void RTC_SetWakeupCount (RTC_Type * *base*, uint16_t *wakeupValue*) [inline], [static]

Parameters

base	RTC peripheral base address
wakeupValue	The value to be loaded into the RTC WAKE register

33.6.8 static uint16_t RTC_GetWakeupCount (RTC_Type * *base*) [inline], [static]

Parameters

base	RTC peripheral base address

33.6.9 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

33.6.10 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

33.6.11 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

33.6.12 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

33.6.13 static void RTC_ClearStatusFlags (RTC_Type * *base,* uint32_t *mask*) [inline], [static]

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

33.6.14 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

33.6.15 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

33.6.16 static void RTC_Reset (RTC_Type * *base*) [inline], [static]

This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.

Parameters

base	RTC peripheral base address

Chapter 34
SCTimer: SCTimer/PWM (SCT)

34.1 Overview

The SDK provides a driver for the SCTimer Module (SCT) of LPC devices.

34.2 Function groups

The SCTimer driver supports the generation of PWM signals. The driver also supports enabling events in various states of the SCTimer and the actions that will be triggered when an event occurs.

34.2.1 Initialization and deinitialization

The function SCTIMER_Init() initializes the SCTimer with specified configurations. The function SCTIMER_GetDefaultConfig() gets the default configurations.

The function SCTIMER_Deinit() halts the SCTimer counter and turns off the module clock.

34.2.2 PWM Operations

The function SCTIMER_SetupPwm() sets up SCTimer channels for PWM output. The function can set up the PWM signal properties duty cycle and level-mode (active low or high) to use. However, the same PWM period and PWM mode (edge or center-aligned) 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 1 and 100.

The function SCTIMER_UpdatePwmDutycycle() updates the PWM signal duty cycle of a particular SCTimer channel.

34.2.3 Status

Provides functions to get and clear the SCTimer status.

34.2.4 Interrupt

Provides functions to enable/disable SCTimer interrupts and get current enabled interrupts.

34.3 SCTimer State machine and operations

The SCTimer has 10 states and each state can have a set of events enabled that can trigger a user specified action when the event occurs.

34.3.1 SCTimer event operations

The user can create an event and enable it in the current state using the functions SCTIMER_CreateAndScheduleEvent() and SCTIMER_ScheduleEvent(). SCTIMER_CreateAndScheduleEvent() creates a new event based on the users preference and enables it in the current state. SCTIMER_ScheduleEvent() enables an event created earlier in the current state.

34.3.2 SCTimer state operations

The user can get the current state number by calling SCTIMER_GetCurrentState(), he can use this state number to set state transitions when a particular event is triggered.

Once the user has created and enabled events for the current state he can go to the next state by calling the function SCTIMER_IncreaseState(). The user can then start creating events to be enabled in this new state.

34.3.3 SCTimer action operations

There are a set of functions that decide what action should be taken when an event is triggered. SCTIMER_SetupCaptureAction() sets up which counter to capture and which capture register to read on event trigger. SCTIMER_SetupNextStateAction() sets up which state the SCTimer state machine should transition to on event trigger. SCTIMER_SetupOutputSetAction() sets up which pin to set on event trigger. SCTIMER_SetupOutputClearAction() sets up which pin to clear on event trigger. SCTIMER_SetupOutputToggleAction() sets up which pin to toggle on event trigger. SCTIMER_SetupCounterLimitAction() sets up which counter will be limited on event trigger. SCTIMER_SetupCounterStopAction() sets up which counter will be stopped on event trigger. SCTIMER_SetupCounterStartAction() sets up which counter will be started on event trigger. SCTIMER_SetupCounterHaltAction() sets up which counter will be halted on event trigger. SCTIMER_SetupDmaTriggerAction() sets up which DMA request will be activated on event trigger.

34.4 16-bit counter mode

The SCTimer is configurable to run as two 16-bit counters via the enableCounterUnify flag that is available in the configuration structure passed in to the SCTIMER_Init() function.

When operating in 16-bit mode, it is important the user specify the appropriate counter to use when working with the functions: SCTIMER_StartTimer(), SCTIMER_StopTimer(), SCTIMER_CreateAndScheduleEvent(), SCTIMER_SetupCaptureAction(), SCTIMER_SetupCounterLimitAction(), SCTIM-

ER_SetupCounterStopAction(), SCTIMER_SetupCounterStartAction(), SCTIMER_SetupCounterHalt-Action().

34.5 Typical use case

34.5.1 PWM output

Output a PWM signal on 2 SCTimer channels with different duty cycles.

```
int main(void)
{
    sctimer_config_t sctimerInfo;
    sctimer_pwm_signal_param_t pwmParam;
    uint32_t event;
    uint32_t sctimerClock;

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    sctimerClock = CLOCK_GetFreq(kCLOCK_BusClk);

    /* Print a note to terminal */
    PRINTF("\r\nSCTimer example to output 2 center-aligned PWM signals\r\n");
    PRINTF("\r\nYou will see a change in LED brightness if an LED is connected to the SCTimer output pins")
      ;
    PRINTF("\r\nIf no LED is connected to the pin, then probe the signal using an oscilloscope");

    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    /* Configure first PWM with frequency 24kHZ from output 4 */
    pwmParam.output = kSCTIMER_Out_4;
    pwmParam.level = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam,
      kSCTIMER_CenterAlignedPwm, 24000U, sctimerClock, &event) == kStatus_Fail)
    {
        return -1;
    }

    /* Configure second PWM with different duty cycle but same frequency as before */
    pwmParam.output = kSCTIMER_Out_2;
    pwmParam.level = kSCTIMER_LowTrue;
    pwmParam.dutyCyclePercent = 20;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam,
      kSCTIMER_CenterAlignedPwm, 24000U, sctimerClock, &event) == kStatus_Fail)
    {
        return -1;
    }

    /* Start the timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_L);

    while (1)
    {
    }
}
```

Files

- file fsl_sctimer.h

Typical use case

Data Structures

- struct sctimer_pwm_signal_param_t
  Options to configure a SCTimer PWM signal. More...
- struct sctimer_config_t
  SCTimer configuration structure. More...

Typedefs

- typedef void(∗ sctimer_event_callback_t)(void)
  SCTimer callback typedef.

Enumerations

- enum sctimer_pwm_mode_t {
  kSCTIMER_EdgeAlignedPwm = 0U,
  kSCTIMER_CenterAlignedPwm }
  SCTimer PWM operation modes.
- enum sctimer_counter_t {
  kSCTIMER_Counter_L = 0U,
  kSCTIMER_Counter_H }
  SCTimer counters when working as two independent 16-bit counters.
- enum sctimer_input_t {
  kSCTIMER_Input_0 = 0U,
  kSCTIMER_Input_1,
  kSCTIMER_Input_2,
  kSCTIMER_Input_3,
  kSCTIMER_Input_4,
  kSCTIMER_Input_5,
  kSCTIMER_Input_6,
  kSCTIMER_Input_7 }
  List of SCTimer input pins.
- enum sctimer_out_t {
  kSCTIMER_Out_0 = 0U,
  kSCTIMER_Out_1,
  kSCTIMER_Out_2,
  kSCTIMER_Out_3,
  kSCTIMER_Out_4,
  kSCTIMER_Out_5,
  kSCTIMER_Out_6,
  kSCTIMER_Out_7 }
  List of SCTimer output pins.
- enum sctimer_pwm_level_select_t {
  kSCTIMER_LowTrue = 0U,
  kSCTIMER_HighTrue }
  SCTimer PWM output pulse mode: high-true, low-true or no output.
- enum sctimer_clock_mode_t {

kSCTIMER_System_ClockMode = 0U,
kSCTIMER_Sampled_ClockMode,
kSCTIMER_Input_ClockMode,
kSCTIMER_Asynchronous_ClockMode }
SCTimer clock mode options.

- enum sctimer_clock_select_t {
kSCTIMER_Clock_On_Rise_Input_0 = 0U,
kSCTIMER_Clock_On_Fall_Input_0,
kSCTIMER_Clock_On_Rise_Input_1,
kSCTIMER_Clock_On_Fall_Input_1,
kSCTIMER_Clock_On_Rise_Input_2,
kSCTIMER_Clock_On_Fall_Input_2,
kSCTIMER_Clock_On_Rise_Input_3,
kSCTIMER_Clock_On_Fall_Input_3,
kSCTIMER_Clock_On_Rise_Input_4,
kSCTIMER_Clock_On_Fall_Input_4,
kSCTIMER_Clock_On_Rise_Input_5,
kSCTIMER_Clock_On_Fall_Input_5,
kSCTIMER_Clock_On_Rise_Input_6,
kSCTIMER_Clock_On_Fall_Input_6,
kSCTIMER_Clock_On_Rise_Input_7,
kSCTIMER_Clock_On_Fall_Input_7 }
SCTimer clock select options.
- enum sctimer_conflict_resolution_t {
kSCTIMER_ResolveNone = 0U,
kSCTIMER_ResolveSet,
kSCTIMER_ResolveClear,
kSCTIMER_ResolveToggle }
SCTimer output conflict resolution options.
- enum sctimer_event_t
List of SCTimer event types.
- enum sctimer_interrupt_enable_t {
kSCTIMER_Event0InterruptEnable = (1U << 0),
kSCTIMER_Event1InterruptEnable = (1U << 1),
kSCTIMER_Event2InterruptEnable = (1U << 2),
kSCTIMER_Event3InterruptEnable = (1U << 3),
kSCTIMER_Event4InterruptEnable = (1U << 4),
kSCTIMER_Event5InterruptEnable = (1U << 5),
kSCTIMER_Event6InterruptEnable = (1U << 6),
kSCTIMER_Event7InterruptEnable = (1U << 7),
kSCTIMER_Event8InterruptEnable = (1U << 8),
kSCTIMER_Event9InterruptEnable = (1U << 9),
kSCTIMER_Event10InterruptEnable = (1U << 10),
kSCTIMER_Event11InterruptEnable = (1U << 11),
kSCTIMER_Event12InterruptEnable = (1U << 12) }
List of SCTimer interrupts.

Typical use case

- enum sctimer_status_flags_t {
  kSCTIMER_Event0Flag = (1U << 0),
  kSCTIMER_Event1Flag = (1U << 1),
  kSCTIMER_Event2Flag = (1U << 2),
  kSCTIMER_Event3Flag = (1U << 3),
  kSCTIMER_Event4Flag = (1U << 4),
  kSCTIMER_Event5Flag = (1U << 5),
  kSCTIMER_Event6Flag = (1U << 6),
  kSCTIMER_Event7Flag = (1U << 7),
  kSCTIMER_Event8Flag = (1U << 8),
  kSCTIMER_Event9Flag = (1U << 9),
  kSCTIMER_Event10Flag = (1U << 10),
  kSCTIMER_Event11Flag = (1U << 11),
  kSCTIMER_Event12Flag = (1U << 12),
  kSCTIMER_BusErrorLFlag,
  kSCTIMER_BusErrorHFlag }
  List of SCTimer flags.

Driver version

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

Initialization and deinitialization

- status_t SCTIMER_Init (SCT_Type *base, const sctimer_config_t *config)
  Ungates the SCTimer clock and configures the peripheral for basic operation.
- void SCTIMER_Deinit (SCT_Type *base)
  Gates the SCTimer clock.
- void SCTIMER_GetDefaultConfig (sctimer_config_t *config)
  Fills in the SCTimer configuration structure with the default settings.

PWM setup operations

- status_t SCTIMER_SetupPwm (SCT_Type *base, const sctimer_pwm_signal_param_t *pwmParams, sctimer_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, uint32_t *event)
  Configures the PWM signal parameters.
- void SCTIMER_UpdatePwmDutycycle (SCT_Type *base, sctimer_out_t output, uint8_t dutyCyclePercent, uint32_t event)
  Updates the duty cycle of an active PWM signal.

Interrupt Interface

- static void SCTIMER_EnableInterrupts (SCT_Type *base, uint32_t mask)
  Enables the selected SCTimer interrupts.
- static void SCTIMER_DisableInterrupts (SCT_Type *base, uint32_t mask)
  Disables the selected SCTimer interrupts.

- static uint32_t SCTIMER_GetEnabledInterrupts (SCT_Type *base)
  Gets the enabled SCTimer interrupts.

Status Interface

- static uint32_t SCTIMER_GetStatusFlags (SCT_Type *base)
  Gets the SCTimer status flags.
- static void SCTIMER_ClearStatusFlags (SCT_Type *base, uint32_t mask)
  Clears the SCTimer status flags.

Counter Start and Stop

- static void SCTIMER_StartTimer (SCT_Type *base, sctimer_counter_t countertoStart)
  Starts the SCTimer counter.
- static void SCTIMER_StopTimer (SCT_Type *base, sctimer_counter_t countertoStop)
  Halts the SCTimer counter.

Functions to create a new event and manage the state logic

- status_t SCTIMER_CreateAndScheduleEvent (SCT_Type *base, sctimer_event_t howToMonitor, uint32_t matchValue, uint32_t whichIO, sctimer_counter_t whichCounter, uint32_t *event)
  Create an event that is triggered on a match or IO and schedule in current state.
- void SCTIMER_ScheduleEvent (SCT_Type *base, uint32_t event)
  Enable an event in the current state.
- status_t SCTIMER_IncreaseState (SCT_Type *base)
  Increase the state by 1.
- uint32_t SCTIMER_GetCurrentState (SCT_Type *base)
  Provides the current state.

Actions to take in response to an event

- status_t SCTIMER_SetupCaptureAction (SCT_Type *base, sctimer_counter_t whichCounter, uint32_t *captureRegister, uint32_t event)
  Setup capture of the counter value on trigger of a selected event.
- void SCTIMER_SetCallback (SCT_Type *base, sctimer_event_callback_t callback, uint32_t event)
  Receive noticification when the event trigger an interrupt.
- static void SCTIMER_SetupNextStateAction (SCT_Type *base, uint32_t nextState, uint32_t event)
  Transition to the specified state.
- static void SCTIMER_SetupOutputSetAction (SCT_Type *base, uint32_t whichIO, uint32_t event)
  Set the Output.
- static void SCTIMER_SetupOutputClearAction (SCT_Type *base, uint32_t whichIO, uint32_t event)
  Clear the Output.
- void SCTIMER_SetupOutputToggleAction (SCT_Type *base, uint32_t whichIO, uint32_t event)
  Toggle the output level.
- static void SCTIMER_SetupCounterLimitAction (SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)
  Limit the running counter.
- static void SCTIMER_SetupCounterStopAction (SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)

Stop the running counter.
- static void SCTIMER_SetupCounterStartAction (SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)
  Re-start the stopped counter.
- static void SCTIMER_SetupCounterHaltAction (SCT_Type *base, sctimer_counter_t whichCounter, uint32_t event)
  Halt the running counter.
- static void SCTIMER_SetupDmaTriggerAction (SCT_Type *base, uint32_t dmaNumber, uint32_t event)
  Generate a DMA request.
- void SCTIMER_EventHandleIRQ (SCT_Type *base)
  SCTimer interrupt handler.

34.6 Data Structure Documentation

34.6.1 struct sctimer_pwm_signal_param_t

Data Fields

- sctimer_out_t output
  The output pin to use to generate the PWM signal.
- sctimer_pwm_level_select_t level
  PWM output active level select.
- uint8_t dutyCyclePercent
  PWM pulse width, value should be between 1 to 100 100 = always active signal (100% duty cycle).

34.6.1.0.0.49 Field Documentation

34.6.1.0.0.49.1 sctimer_pwm_level_select_t sctimer_pwm_signal_param_t::level

34.6.1.0.0.49.2 uint8_t sctimer_pwm_signal_param_t::dutyCyclePercent

34.6.2 struct sctimer_config_t

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the SCTMR_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

- bool enableCounterUnify
  true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters
- sctimer_clock_mode_t clockMode
  SCT clock mode value.
- sctimer_clock_select_t clockSelect
  SCT clock select value.

- bool enableBidirection_l
  true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter
- bool enableBidirection_h
  true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter.
- uint8_t prescale_l
  Prescale value to produce the L or unified counter clock.
- uint8_t prescale_h
  Prescale value to produce the H counter clock.
- uint8_t outInitState
  Defines the initial output value.

34.6.2.0.0.50 Field Documentation

34.6.2.0.0.50.1 bool sctimer_config_t::enableBidirection_h

This field is used only if the enableCounterUnify is set to false

34.6.2.0.0.50.2 uint8_t sctimer_config_t::prescale_h

This field is used only if the enableCounterUnify is set to false

34.7 Typedef Documentation

34.7.1 typedef void(∗ sctimer_event_callback_t)(void)

34.8 Enumeration Type Documentation

34.8.1 enum sctimer_pwm_mode_t

Enumerator

kSCTIMER_EdgeAlignedPwm Edge-aligned PWM.
kSCTIMER_CenterAlignedPwm Center-aligned PWM.

34.8.2 enum sctimer_counter_t

Enumerator

kSCTIMER_Counter_L Counter L.
kSCTIMER_Counter_H Counter H.

34.8.3 enum sctimer_input_t

Enumerator

kSCTIMER_Input_0 SCTIMER input 0.
kSCTIMER_Input_1 SCTIMER input 1.
kSCTIMER_Input_2 SCTIMER input 2.
kSCTIMER_Input_3 SCTIMER input 3.
kSCTIMER_Input_4 SCTIMER input 4.
kSCTIMER_Input_5 SCTIMER input 5.
kSCTIMER_Input_6 SCTIMER input 6.
kSCTIMER_Input_7 SCTIMER input 7.

34.8.4 enum sctimer_out_t

Enumerator

kSCTIMER_Out_0 SCTIMER output 0.
kSCTIMER_Out_1 SCTIMER output 1.
kSCTIMER_Out_2 SCTIMER output 2.
kSCTIMER_Out_3 SCTIMER output 3.
kSCTIMER_Out_4 SCTIMER output 4.
kSCTIMER_Out_5 SCTIMER output 5.
kSCTIMER_Out_6 SCTIMER output 6.
kSCTIMER_Out_7 SCTIMER output 7.

34.8.5 enum sctimer_pwm_level_select_t

Enumerator

kSCTIMER_LowTrue Low true pulses.
kSCTIMER_HighTrue High true pulses.

34.8.6 enum sctimer_clock_mode_t

Enumerator

kSCTIMER_System_ClockMode System Clock Mode.
kSCTIMER_Sampled_ClockMode Sampled System Clock Mode.
kSCTIMER_Input_ClockMode SCT Input Clock Mode.
kSCTIMER_Asynchronous_ClockMode Asynchronous Mode.

34.8.7 enum sctimer_clock_select_t

Enumerator

kSCTIMER_Clock_On_Rise_Input_0 Rising edges on input 0.
kSCTIMER_Clock_On_Fall_Input_0 Falling edges on input 0.
kSCTIMER_Clock_On_Rise_Input_1 Rising edges on input 1.
kSCTIMER_Clock_On_Fall_Input_1 Falling edges on input 1.
kSCTIMER_Clock_On_Rise_Input_2 Rising edges on input 2.
kSCTIMER_Clock_On_Fall_Input_2 Falling edges on input 2.
kSCTIMER_Clock_On_Rise_Input_3 Rising edges on input 3.
kSCTIMER_Clock_On_Fall_Input_3 Falling edges on input 3.
kSCTIMER_Clock_On_Rise_Input_4 Rising edges on input 4.
kSCTIMER_Clock_On_Fall_Input_4 Falling edges on input 4.
kSCTIMER_Clock_On_Rise_Input_5 Rising edges on input 5.
kSCTIMER_Clock_On_Fall_Input_5 Falling edges on input 5.
kSCTIMER_Clock_On_Rise_Input_6 Rising edges on input 6.
kSCTIMER_Clock_On_Fall_Input_6 Falling edges on input 6.
kSCTIMER_Clock_On_Rise_Input_7 Rising edges on input 7.
kSCTIMER_Clock_On_Fall_Input_7 Falling edges on input 7.

34.8.8 enum sctimer_conflict_resolution_t

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Enumerator

kSCTIMER_ResolveNone No change.
kSCTIMER_ResolveSet Set output.
kSCTIMER_ResolveClear Clear output.
kSCTIMER_ResolveToggle Toggle output.

34.8.9 enum sctimer_interrupt_enable_t

Enumerator

kSCTIMER_Event0InterruptEnable Event 0 interrupt.
kSCTIMER_Event1InterruptEnable Event 1 interrupt.
kSCTIMER_Event2InterruptEnable Event 2 interrupt.
kSCTIMER_Event3InterruptEnable Event 3 interrupt.
kSCTIMER_Event4InterruptEnable Event 4 interrupt.
kSCTIMER_Event5InterruptEnable Event 5 interrupt.

kSCTIMER_Event6InterruptEnable Event 6 interrupt.
kSCTIMER_Event7InterruptEnable Event 7 interrupt.
kSCTIMER_Event8InterruptEnable Event 8 interrupt.
kSCTIMER_Event9InterruptEnable Event 9 interrupt.
kSCTIMER_Event10InterruptEnable Event 10 interrupt.
kSCTIMER_Event11InterruptEnable Event 11 interrupt.
kSCTIMER_Event12InterruptEnable Event 12 interrupt.

34.8.10 enum sctimer_status_flags_t

Enumerator

kSCTIMER_Event0Flag Event 0 Flag.
kSCTIMER_Event1Flag Event 1 Flag.
kSCTIMER_Event2Flag Event 2 Flag.
kSCTIMER_Event3Flag Event 3 Flag.
kSCTIMER_Event4Flag Event 4 Flag.
kSCTIMER_Event5Flag Event 5 Flag.
kSCTIMER_Event6Flag Event 6 Flag.
kSCTIMER_Event7Flag Event 7 Flag.
kSCTIMER_Event8Flag Event 8 Flag.
kSCTIMER_Event9Flag Event 9 Flag.
kSCTIMER_Event10Flag Event 10 Flag.
kSCTIMER_Event11Flag Event 11 Flag.
kSCTIMER_Event12Flag Event 12 Flag.
kSCTIMER_BusErrorLFlag Bus error due to write when L counter was not halted.
kSCTIMER_BusErrorHFlag Bus error due to write when H counter was not halted.

34.9 Function Documentation

34.9.1 status_t SCTIMER_Init (SCT_Type * *base*, const sctimer_config_t * *config*)

Note

This API should be called at the beginning of the application using the SCTimer driver.

Parameters

base	SCTimer peripheral base address
config	Pointer to the user configuration structure.

Returns

kStatus_Success indicates success; Else indicates failure.

34.9.2 void SCTIMER_Deinit (SCT_Type * *base*)

Parameters

base	SCTimer peripheral base address

34.9.3 void SCTIMER_GetDefaultConfig (sctimer_config_t * *config*)

The default values are:

```
*  config->enableCounterUnify = true;
*  config->clockMode = kSCTIMER_System_ClockMode;
*  config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
*  config->enableBidirection_l = false;
*  config->enableBidirection_h = false;
*  config->prescale_l = 0;
*  config->prescale_h = 0;
*  config->outInitState = 0;
*
```

Parameters

config	Pointer to the user configuration structure.

34.9.4 status_t SCTIMER_SetupPwm (SCT_Type * *base*, const sctimer_pwm_signal_param_t * *pwmParams*, sctimer_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*, uint32_t * *event*)

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function SCTIMER_GetCurrentStateNumber(). The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

Note

When setting PWM output from multiple output pins, they all should use the same PWM mode i.e all PWM's should be either edge-aligned or center-aligned.

Parameters

base	SCTimer peripheral base address
pwmParams	PWM parameters to configure the output
mode	PWM operation mode, options available in enumeration sctimer_pwm_mode_t
pwmFreq_Hz	PWM signal frequency in Hz
srcClock_Hz	SCTimer counter clock in Hz
event	Pointer to a variable where the PWM period event number is stored

Returns

kStatus_Success on success kStatus_Fail If we have hit the limit in terms of number of events created or if an incorrect PWM dutycylce is passed in.

34.9.5 void SCTIMER_UpdatePwmDutycycle (SCT_Type * *base*, sctimer_out_t *output*, uint8_t *dutyCyclePercent*, uint32_t *event*)

Parameters

base	SCTimer peripheral base address
output	The output to configure
dutyCycle-Percent	New PWM pulse width; the value should be between 1 to 100
event	Event number associated with this PWM signal. This was returned to the user by the function SCTIMER_SetupPwm().

34.9.6 static void SCTIMER_EnableInterrupts (SCT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration sctimer_interrupt_enable_t

34.9.7 static void SCTIMER_DisableInterrupts (SCT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address
mask	The interrupts to enable. This is a logical OR of members of the enumeration sctimer_interrupt_enable_t

34.9.8 static uint32_t SCTIMER_GetEnabledInterrupts (SCT_Type * *base*) [inline], [static]

Parameters

base	SCTimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration sctimer_interrupt_enable_t

34.9.9 static uint32_t SCTIMER_GetStatusFlags (SCT_Type * *base*) [inline], [static]

Parameters

base	SCTimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration sctimer_status_flags_t

34.9.10 static void SCTIMER_ClearStatusFlags (SCT_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SCTimer peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration sctimer_status_flags_t

34.9.11 static void SCTIMER_StartTimer (SCT_Type * *base*, sctimer_counter_t *countertoStart*) [inline], [static]

Parameters

base	SCTimer peripheral base address
countertoStart	SCTimer counter to start; if unify mode is set then function always writes to HALT_L bit

34.9.12 static void SCTIMER_StopTimer (SCT_Type * *base*, sctimer_counter_t *countertoStop*) [inline], [static]

Parameters

base	SCTimer peripheral base address
countertoStop	SCTimer counter to stop; if unify mode is set then function always writes to HALT_L bit

34.9.13 status_t SCTIMER_CreateAndScheduleEvent (SCT_Type * *base*, sctimer_event_t *howToMonitor*, uint32_t *matchValue*, uint32_t *whichIO*, sctimer_counter_t *whichCounter*, uint32_t * *event*)

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

Parameters

base	SCTimer peripheral base address
howToMonitor	Event type; options are available in the enumeration sctimer_interrupt_enable_t
matchValue	The match value that will be programmed to a match register
whichIO	The input or output that will be involved in event triggering. This field is ignored if the event type is "match only"
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as we have only 1 unified counter; hence ignored.
event	Pointer to a variable where the new event number is stored

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

34.9.14 void SCTIMER_ScheduleEvent (SCT_Type * *base*, uint32_t *event*)

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function SCTIMER_SetupPwm() or function SCTIMER_CreateAndScheduleEvent() .

Parameters

base	SCTimer peripheral base address
event	Event number to enable in the current state

34.9.15 status_t SCTIMER_IncreaseState (SCT_Type * *base*)

All future events created by calling the function SCTIMER_ScheduleEvent() will be enabled in this new state.

Parameters

base	SCTimer peripheral base address

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of states used

34.9.16 uint32_t SCTIMER_GetCurrentState (SCT_Type * *base*)

User can use this to set the next state by calling the function SCTIMER_SetupNextStateAction().

Parameters

base	SCTimer peripheral base address

Returns

The current state

34.9.17 status_t SCTIMER_SetupCaptureAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t * *captureRegister*, uint32_t *event*)

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
captureRegister	Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
event	Event number that will trigger the capture

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

34.9.18 void SCTIMER_SetCallback (SCT_Type * *base*, sctimer_event_callback_t *callback*, uint32_t *event*)

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

Parameters

base	SCTimer peripheral base address

event	Event number that will trigger the interrupt
callback	Function to invoke when the event is triggered

34.9.19 static void SCTIMER_SetupNextStateAction (SCT_Type * *base,* uint32_t *nextState,* uint32_t *event*) [inline], [static]

This transition will be triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
nextState	The next state SCTimer will transition to
event	Event number that will trigger the state transition

34.9.20 static void SCTIMER_SetupOutputSetAction (SCT_Type * *base,* uint32_t *whichIO,* uint32_t *event*) [inline], [static]

This output will be set when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichIO	The output to set
event	Event number that will trigger the output change

34.9.21 static void SCTIMER_SetupOutputClearAction (SCT_Type * *base,* uint32_t *whichIO,* uint32_t *event*) [inline], [static]

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichIO	The output to clear
event	Event number that will trigger the output change

34.9.22 void SCTIMER_SetupOutputToggleAction (SCT_Type * *base,* uint32_t *whichIO,* uint32_t *event*)

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
whichIO	The output to toggle
event	Event number that will trigger the output change

34.9.23 static void SCTIMER_SetupCounterLimitAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be limited

34.9.24 static void SCTIMER_SetupCounterStopAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be stopped

34.9.25 static void SCTIMER_SetupCounterStartAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to re-start

34.9.26 static void SCTIMER_SetupCounterHaltAction (SCT_Type * *base*, sctimer_counter_t *whichCounter*, uint32_t *event*) [inline], [static]

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the SCTIMER_StartTimer() function.

Parameters

base	SCTimer peripheral base address
whichCounter	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
event	Event number that will trigger the counter to be halted

34.9.27 static void SCTIMER_SetupDmaTriggerAction (SCT_Type * *base*, uint32_t *dmaNumber*, uint32_t *event*) [inline], [static]

DMA request will be triggered by the event number that is passed in by the user.

Parameters

base	SCTimer peripheral base address
dmaNumber	The DMA request to generate
event	Event number that will trigger the DMA request

34.9.28 void SCTIMER_EventHandleIRQ (SCT_Type * *base*)

Parameters

base	SCTimer peripheral base address.

Chapter 35
SDIF: SD/MMC/SDIO card interface

35.1 Overview

The KSDK provides a peripheral driver for the SD/MMC/SDIO card interface (sdif) module of LPC devices.

35.2 Typical use case

35.2.1 sdif Operation

```
/* Initializes the sdif. */
    sdif_config_t sdifConfig;
    sdifConfig->responseTimeout = 0x40U;
    sdifConfig->cardDetDebounce_Clock = 0xFFFFFFU;
    sdifConfig->dataTimeout = 0xFFFFFFU;
    SDIF_Init(BOARD_sdif_BASEADDR, sdifConfig);

/* transfer data/commmand in a blocking way */
    /* Internal DMA configuraion */
    sdif_dma_config_t dmaConfig;
    dmaConfig.enableFixBurstLen = true;
    dmaConfig.mode = kSDIF_ChainDMAMode;
    dmaConfig.dmaDesBufferLen = 0x04;/* one contain one descriptor */

    /* config the command to send */
    sdif_command_t command;
    command.index = read/write;
    command.argument = argument;
    command.responseType = command response type;

    /* config the data if need transfer data */
    sdif_data_t data;
    data.autoDataTransferStop = true;
    data.blockSize = 128;
    data.blockCount = 1;
    data.rxData = user define buffer to recieve data;

    /* transfer data in blocking way */
    sdif_transfer_t transfer.;
    transfer.dmaConfig = &dmaConfig;
    transfer.command = &command;
    transfer.data = &data;

    /* need check the status */
    SDIF_TransferBlocking(base, &transfer);

/* transfer data/command in a non-blocking way */
    /* create you call back function */
    sdif_transfer_callback_t callBack;
    callBack.CardInserted = CardInsert;
    callBack.DMADesUnavailable = DMADesUnavailable;
    callBack.CommandReload = CommandReload;
    callBack.TransferComplete = TransferComplete;

    sdif_handle_t handle;
    SDIF_TransferCreateHandle(base, &handle,  &callback, &userData);
```

Typical use case

```
SDIF_TransferNonBlocking(base, &handle, &transfer);

/* do not need to check the status by software, interrupt will be trigger
when error happen */
```

Data Structures

- struct sdif_dma_descriptor_t
  define the internal DMA descriptor More...
- struct sdif_dma_config_t
  Defines the internal DMA config structure. More...
- struct sdif_data_t
  Card data descriptor. More...
- struct sdif_command_t
  Card command descriptor. More...
- struct sdif_transfer_t
  Transfer state. More...
- struct sdif_config_t
  Data structure to initialize the sdif. More...
- struct sdif_capability_t
  SDIF capability information. More...
- struct sdif_transfer_callback_t
  sdif callback functions. More...
- struct sdif_handle_t
  sdif handle More...
- struct sdif_host_t
  sdif host descriptor More...

Macros

- #define SDIF_DriverIRQHandler SDIO_DriverIRQHandler
  convert the name here, due to RM use SDIO
- #define SDIF_SUPPORT_SD_VERSION (0x20)
  define the controller support sd/sdio card version 2.0
- #define SDIF_SUPPORT_MMC_VERSION (0x44)
  define the controller support mmc card version 4.4
- #define SDIF_TIMEOUT_VALUE (65535U)
  define the timeout counter
- #define SDIF_POLL_DEMAND_VALUE (0xFFU)
  this value can be any value
- #define SDIF_DMA_DESCRIPTOR_BUFFER1_SIZE(x) (x & 0x1FFFU)
  DMA descriptor buffer1 size.
- #define SDIF_DMA_DESCRIPTOR_BUFFER2_SIZE(x) ((x & 0x1FFFU) << 13U)
  DMA descriptor buffer2 size.
- #define SDIF_RX_WATERMARK (15U)
  RX water mark value.
- #define SDIF_TX_WATERMARK (16U)
  TX water mark value.
- #define SDIF_INDENTIFICATION_MODE_SAMPLE_DELAY (0X17U)
  SDIOCLKCTRL setting below clock delay setting should meet you board layout user can change it when you meet timing mismatch issue such as: response error/CRC error and so on.

Typedefs

- typedef status_t(∗ sdif_transfer_function_t)(SDIF_Type ∗base, sdif_transfer_t ∗content)
  sdif transfer function.

Enumerations

- enum _sdif_status {
  kStatus_SDIF_DescriptorBufferLenError = MAKE_STATUS(kStatusGroup_SDIF, 0U),
  kStatue_SDIF_InvalidArgument = MAKE_STATUS(kStatusGroup_SDIF, 1U),
  kStatus_SDIF_SyncCmdTimeout = MAKE_STATUS(kStatusGroup_SDIF, 2U) }
  SDIF status.
- enum _sdif_capability_flag {
  kSDIF_SupportHighSpeedFlag = 0x1U,
  kSDIF_SupportDmaFlag = 0x2U,
  kSDIF_SupportSuspendResumeFlag = 0x4U,
  kSDIF_SupportV330Flag = 0x8U,
  kSDIF_Support4BitFlag = 0x10U,
  kSDIF_Support8BitFlag = 0x20U }
  Host controller capabilities flag mask.
- enum _sdif_reset_type {
  kSDIF_ResetController,
  kSDIF_ResetFIFO = SDIF_CTRL_FIFO_RESET_MASK,
  kSDIF_ResetDMAInterface = SDIF_CTRL_DMA_RESET_MASK,
  kSDIF_ResetAll }
  define the reset type
- enum sdif_bus_width_t {
  kSDIF_Bus1BitWidth = 0U,
  kSDIF_Bus4BitWidth = SDIF_CTYPE_CARD_WIDTH0_MASK,
  kSDIF_Bus8BitWidth = SDIF_CTYPE_CARD_WIDTH1_MASK }
  define the card bus width type
- enum _sdif_command_flags {

kSDIF_CmdResponseExpect = SDIF_CMD_RESPONSE_EXPECT_MASK,
kSDIF_CmdResponseLengthLong = SDIF_CMD_RESPONSE_LENGTH_MASK,
kSDIF_CmdCheckResponseCRC = SDIF_CMD_CHECK_RESPONSE_CRC_MASK,
kSDIF_DataExpect = SDIF_CMD_DATA_EXPECTED_MASK,
kSDIF_DataWriteToCard = SDIF_CMD_READ_WRITE_MASK,
kSDIF_DataStreamTransfer = SDIF_CMD_TRANSFER_MODE_MASK,
kSDIF_DataTransferAutoStop = SDIF_CMD_SEND_AUTO_STOP_MASK,
kSDIF_WaitPreTransferComplete,
kSDIF_TransferStopAbort,
kSDIF_SendInitialization,
kSDIF_CmdUpdateClockRegisterOnly,
kSDIF_CmdtoReadCEATADevice = SDIF_CMD_READ_CEATA_DEVICE_MASK,
kSDIF_CmdExpectCCS = SDIF_CMD_CCS_EXPECTED_MASK,
kSDIF_BootModeEnable = SDIF_CMD_ENABLE_BOOT_MASK,
kSDIF_BootModeExpectAck = SDIF_CMD_EXPECT_BOOT_ACK_MASK,
kSDIF_BootModeDisable = SDIF_CMD_DISABLE_BOOT_MASK,
kSDIF_BootModeAlternate = SDIF_CMD_BOOT_MODE_MASK,
kSDIF_CmdVoltageSwitch = SDIF_CMD_VOLT_SWITCH_MASK,
kSDIF_CmdDataUseHoldReg = SDIF_CMD_USE_HOLD_REG_MASK }

define the command flags

- enum _sdif_command_type {
  kCARD_CommandTypeNormal = 0U,
  kCARD_CommandTypeSuspend = 1U,
  kCARD_CommandTypeResume = 2U,
  kCARD_CommandTypeAbort = 3U }

  The command type.
- enum _sdif_response_type {
  kCARD_ResponseTypeNone = 0U,
  kCARD_ResponseTypeR1 = 1U,
  kCARD_ResponseTypeR1b = 2U,
  kCARD_ResponseTypeR2 = 3U,
  kCARD_ResponseTypeR3 = 4U,
  kCARD_ResponseTypeR4 = 5U,
  kCARD_ResponseTypeR5 = 6U,
  kCARD_ResponseTypeR5b = 7U,
  kCARD_ResponseTypeR6 = 8U,
  kCARD_ResponseTypeR7 = 9U }

  The command response type.
- enum _sdif_interrupt_mask {

kSDIF_CardDetect = SDIF_INTMASK_CDET_MASK,
kSDIF_ResponseError = SDIF_INTMASK_RE_MASK,
kSDIF_CommandDone = SDIF_INTMASK_CDONE_MASK,
kSDIF_DataTransferOver = SDIF_INTMASK_DTO_MASK,
kSDIF_WriteFIFORequest = SDIF_INTMASK_TXDR_MASK,
kSDIF_ReadFIFORequest = SDIF_INTMASK_RXDR_MASK,
kSDIF_ResponseCRCError = SDIF_INTMASK_RCRC_MASK,
kSDIF_DataCRCError = SDIF_INTMASK_DCRC_MASK,
kSDIF_ResponseTimeout = SDIF_INTMASK_RTO_MASK,
kSDIF_DataReadTimeout = SDIF_INTMASK_DRTO_MASK,
kSDIF_DataStarvationByHostTimeout = SDIF_INTMASK_HTO_MASK,
kSDIF_FIFOError = SDIF_INTMASK_FRUN_MASK,
kSDIF_HardwareLockError = SDIF_INTMASK_HLE_MASK,
kSDIF_DataStartBitError = SDIF_INTMASK_SBE_MASK,
kSDIF_AutoCmdDone = SDIF_INTMASK_ACD_MASK,
kSDIF_DataEndBitError = SDIF_INTMASK_EBE_MASK,
kSDIF_SDIOInterrupt = SDIF_INTMASK_SDIO_INT_MASK_MASK,
kSDIF_CommandTransferStatus,
kSDIF_DataTransferStatus ,
kSDIF_AllInterruptStatus = 0x1FFFFU }
define the interrupt mask flags

- enum _sdif_dma_status {
  kSDIF_DMATransFinishOneDescriptor = SDIF_IDSTS_TI_MASK,
  kSDIF_DMARecvFinishOneDescriptor = SDIF_IDSTS_RI_MASK,
  kSDIF_DMAFatalBusError = SDIF_IDSTS_FBE_MASK,
  kSDIF_DMADescriptorUnavailable = SDIF_IDSTS_DU_MASK,
  kSDIF_DMACardErrorSummary = SDIF_IDSTS_CES_MASK,
  kSDIF_NormalInterruptSummary = SDIF_IDSTS_NIS_MASK,
  kSDIF_AbnormalInterruptSummary = SDIF_IDSTS_AIS_MASK }
  define the internal DMA status flags
- enum _sdif_dma_descriptor_flag {
  kSDIF_DisableCompleteInterrupt = 0x2U,
  kSDIF_DMADescriptorDataBufferEnd = 0x4U,
  kSDIF_DMADescriptorDataBufferStart = 0x8U,
  kSDIF_DMASecondAddrChained = 0x10U,
  kSDIF_DMADescriptorEnd = 0x20U,
  kSDIF_DMADescriptorOwnByDMA = 0x80000000U }
  define the internal DMA descriptor flag
- enum sdif_dma_mode_t
  define the internal DMA mode
- enum _sdif_card_freq {
  kSDIF_Freq50MHZ = 50000000U,
  kSDIF_Freq400KHZ = 400000U }
  define the card work freq mode
- enum _sdif_clock_pharse_shift {

Typical use case

kSDIF_ClcokPharseShift0,
kSDIF_ClcokPharseShift90,
kSDIF_ClcokPharseShift180,
kSDIF_ClcokPharseShift270 }
define the clock pharse shift

Functions

- void SDIF_Init (SDIF_Type *base, sdif_config_t *config)
  SDIF module initialization function.
- void SDIF_Deinit (SDIF_Type *base)
  SDIF module deinit function.
- bool SDIF_SendCardActive (SDIF_Type *base, uint32_t timeout)
  SDIF send initialize 80 clocks for SD card after initilize.
- static uint32_t SDIF_DetectCardInsert (SDIF_Type *base, bool data3)
  SDIF module detect card insert status function.
- static void SDIF_EnableCardClock (SDIF_Type *base, bool enable)
  SDIF module enable/disable card clock.
- static void SDIF_EnableLowPowerMode (SDIF_Type *base, bool enable)
  SDIF module enable/disable module disable the card clock to enter low power mode when card is idle,for SDIF cards, if interrupts must be detected, clock should not be stopped.
- uint32_t SDIF_SetCardClock (SDIF_Type *base, uint32_t srcClock_Hz, uint32_t target_HZ)
  Sets the card bus clock frequency.
- bool SDIF_Reset (SDIF_Type *base, uint32_t mask, uint32_t timeout)
  reset the different block of the interface.
- static void SDIF_EnableCardPower (SDIF_Type *base, bool enable)
  enable/disable the card power.
- static uint32_t SDIF_GetCardWriteProtect (SDIF_Type *base)
  get the card write protect status
- static void SDIF_SetCardBusWidth (SDIF_Type *base, sdif_bus_width_t type)
  set card data bus width
- static void SDIF_AssertHardwareReset (SDIF_Type *base)
  toggle state on hardware reset PIN This is used which card has a reset PIN typically.
- status_t SDIF_SendCommand (SDIF_Type *base, sdif_command_t *cmd, uint32_t timeout)
  send command to the card
- static void SDIF_EnableGlobalInterrupt (SDIF_Type *base, bool enable)
  SDIF enable/disable global interrupt.
- static void SDIF_EnableInterrupt (SDIF_Type *base, uint32_t mask)
  SDIF enable interrupt.
- static void SDIF_DisableInterrupt (SDIF_Type *base, uint32_t mask)
  SDIF disable interrupt.
- static uint32_t SDIF_GetInterruptStatus (SDIF_Type *base)
  SDIF get interrupt status.
- static void SDIF_ClearInterruptStatus (SDIF_Type *base, uint32_t mask)
  SDIF clear interrupt status.
- void SDIF_TransferCreateHandle (SDIF_Type *base, sdif_handle_t *handle, sdif_transfer_-callback_t *callback, void *userData)
  Creates the SDIF handle.
- static void SDIF_EnableDmaInterrupt (SDIF_Type *base, uint32_t mask)
  SDIF enable DMA interrupt.
- static void SDIF_DisableDmaInterrupt (SDIF_Type *base, uint32_t mask)

SDIF disable DMA interrupt.
- static uint32_t SDIF_GetInternalDMAStatus (SDIF_Type *base)
  SDIF get internal DMA status.
- static void SDIF_ClearInternalDMAStatus (SDIF_Type *base, uint32_t mask)
  SDIF clear internal DMA status.
- status_t SDIF_InternalDMAConfig (SDIF_Type *base, sdif_dma_config_t *config, const uint32_t *data, uint32_t dataSize)
  SDIF internal DMA config function.
- static void SDIF_SendReadWait (SDIF_Type *base)
  SDIF send read wait to SDIF card function.
- bool SDIF_AbortReadData (SDIF_Type *base, uint32_t timeout)
  SDIF abort the read data when SDIF card is in suspend state Once assert this bit,data state machine will be reset which is waiting for the next blocking data,used in SDIO card suspend sequence,should call after suspend cmd send.
- static void SDIF_EnableCEATAInterrupt (SDIF_Type *base, bool enable)
  SDIF enable/disable CE-ATA card interrupt this bit should set together with the card register.
- status_t SDIF_TransferNonBlocking (SDIF_Type *base, sdif_handle_t *handle, sdif_dma_config_t *dmaConfig, sdif_transfer_t *transfer)
  SDIF transfer function data/cmd in a non-blocking way this API should be use in interrupt mode, when use this API user must call SDIF_TransferCreateHandle first, all status check through interrupt.
- status_t SDIF_TransferBlocking (SDIF_Type *base, sdif_dma_config_t *dmaConfig, sdif_transfer_t *transfer)
  SDIF transfer function data/cmd in a blocking way.
- status_t SDIF_ReleaseDMADescriptor (SDIF_Type *base, sdif_dma_config_t *dmaConfig)
  SDIF release the DMA descriptor to DMA engine this function should be called when DMA descriptor unavailable status occurs.
- void SDIF_GetCapability (SDIF_Type *base, sdif_capability_t *capability)
  SDIF return the controller capability.
- static uint32_t SDIF_GetControllerStatus (SDIF_Type *base)
  SDIF return the controller status.
- static void SDIF_SendCCSD (SDIF_Type *base, bool withAutoStop)
  SDIF send command complete signal disable to CE-ATA card.
- void SDIF_ConfigClockDelay (uint32_t target_HZ, uint32_t divider)
  SDIF config the clock delay This function is used to config the cclk_in delay to sample and drvive the data ,should meet the min setup time and hold time, and user need to config this paramter according to your board setting.

Driver version

- #define FSL_SDIF_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 0U))
  Driver version 2.0.0.

35.3 Data Structure Documentation

35.3.1 struct sdif_dma_descriptor_t

Data Fields

- uint32_t dmaDesAttribute

internal DMA attribute control and status
- uint32_t dmaDataBufferSize
  internal DMA transfer buffer size control
- const uint32_t * dmaDataBufferAddr0
  internal DMA buffer 0 addr ,the buffer size must be 32bit aligned
- const uint32_t * dmaDataBufferAddr1
  internal DMA buffer 1 addr ,the buffer size must be 32bit aligned

35.3.2 struct sdif_dma_config_t

Data Fields

- bool enableFixBurstLen
  `fix burst len enable/disable flag,When set, the AHB will`
  use only SINGLE, INCR4, INCR8 or INCR16 during start of normal burst transfers.
- sdif_dma_mode_t mode
  define the DMA mode
- uint8_t dmaDesSkipLen
  define the descriptor skip length ,the length between two descriptor this field is special for dual DMA mode
- uint32_t * dmaDesBufferStartAddr
  internal DMA descriptor start address
- uint32_t dmaDesBufferLen
  internal DMA buffer descriptor buffer len ,user need to pay attention to the dma descriptor buffer length if it is bigger enough for your transfer

35.3.2.0.0.51 Field Documentation

35.3.2.0.0.51.1 bool sdif_dma_config_t::enableFixBurstLen

When reset, the AHB will use SINGLE and INCR burst transfer operations

35.3.3 struct sdif_data_t

Data Fields

- bool streamTransfer
  indicate this is a stream data transfer command
- bool enableAutoCommand12
  indicate if auto stop will send when data transfer over
- bool enableIgnoreError
  indicate if enable ignore error when transfer data
- size_t blockSize
  Block size, take care when config this parameter.
- uint32_t blockCount
  Block count.
- uint32_t * rxData
  data buffer to recieve

- const uint32_t * txData
  data buffer to transfer

35.3.4 struct sdif_command_t

Define card command-related attribute.

Data Fields

- uint32_t index
  Command index.
- uint32_t argument
  Command argument.
- uint32_t response [4U]
  Response for this command.
- uint32_t type
  define the command type
- uint32_t responseType
  Command response type.
- uint32_t flags
  Cmd flags.
- uint32_t responseErrorFlags
  response error flags, need to check the flags when recieve the cmd response

35.3.5 struct sdif_transfer_t

Data Fields

- sdif_data_t * data
  Data to transfer.
- sdif_command_t * command
  Command to send.

35.3.6 struct sdif_config_t

Data Fields

- uint8_t responseTimeout
  command reponse timeout value
- uint32_t cardDetDebounce_Clock
  define the debounce clock count which will used in card detect logic,typical value is 5-25ms
- uint32_t endianMode
  `define endian mode ,this field is not used in this`
  module actually, keep for compatible with middleware

- uint32_t dataTimeout
  data timeout value

35.3.7 struct sdif_capability_t

Defines a structure to get the capability information of SDIF.

Data Fields

- uint32_t sdVersion
  support SD card/sdio version
- uint32_t mmcVersion
  support emmc card version
- uint32_t maxBlockLength
  Maximum block length united as byte.
- uint32_t maxBlockCount
  Maximum byte count can be transfered.
- uint32_t flags
  Capability flags to indicate the support information.

35.3.8 struct sdif_transfer_callback_t

Data Fields

- void(∗ SDIOInterrupt)(void)
  SDIO card interrupt occurs.
- void(∗ DMADesUnavailable)(void)
  DMA descriptor unavailable.
- void(∗ CommandReload)(void)
  command buffer full,need re-load
- void(∗ TransferComplete)(SDIF_Type ∗base, void ∗handle, status_t status, void ∗userData)
  Transfer complete callback.

35.3.9 struct sdif_handle_t

Defines the structure to save the sdif 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 sdif_interrupt_flag_t;

Note

All the fields except interruptFlags and transferredWords must be allocated by the user.

Data Fields

- sdif_data_t *volatile data
  Data to transfer.
- sdif_command_t *volatile command
  Command to send.
- volatile uint32_t interruptFlags
  Interrupt flags of last transaction.
- volatile uint32_t dmaInterruptFlags
  DMA interrupt flags of last transaction.
- volatile uint32_t transferredWords
  Words transferred by polling way.
- sdif_transfer_callback_t callback
  Callback function.
- void * userData
  Parameter for transfer complete callback.

35.3.10 struct sdif_host_t

Data Fields

- SDIF_Type * base
  sdif peripheral base address
- uint32_t sourceClock_Hz
  sdif source clock frequency united in Hz
- sdif_config_t config
  sdif configuration
- sdif_transfer_function_t transfer
  sdif transfer function
- sdif_capability_t capability
  sdif capability information

35.4 Macro Definition Documentation

35.4.1 #define FSL_SDIF_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 0U))

35.5 Typedef Documentation

35.5.1 typedef status_t(* sdif_transfer_function_t)(SDIF_Type *base, sdif_transfer_t *content)

35.6 Enumeration Type Documentation

35.6.1 enum _sdif_status

Enumerator

kStatus_SDIF_DescriptorBufferLenError Set DMA descriptor failed.

kStatue_SDIF_InvalidArgument invalid argument status
kStatus_SDIF_SyncCmdTimeout sync command to CIU timeout status

35.6.2 enum _sdif_capability_flag

Enumerator

kSDIF_SupportHighSpeedFlag Support high-speed.
kSDIF_SupportDmaFlag Support DMA.
kSDIF_SupportSuspendResumeFlag Support suspend/resume.
kSDIF_SupportV330Flag Support voltage 3.3V.
kSDIF_Support4BitFlag Support 4 bit mode.
kSDIF_Support8BitFlag Support 8 bit mode.

35.6.3 enum _sdif_reset_type

Enumerator

kSDIF_ResetController reset controller,will reset: BIU/CIU interface CIU and state machine,ABORT_READ_DATA,SEND_IRQ_RESPONSE and READ_WAIT bits of control register,START_CMD bit of the command register
kSDIF_ResetFIFO reset data FIFO
kSDIF_ResetDMAInterface reset DMA interface
kSDIF_ResetAll reset all

35.6.4 enum sdif_bus_width_t

Enumerator

kSDIF_Bus1BitWidth 1bit bus width, 1bit mode and 4bit mode share one register bit
kSDIF_Bus4BitWidth 4bit mode mask
kSDIF_Bus8BitWidth support 8 bit mode

35.6.5 enum _sdif_command_flags

Enumerator

kSDIF_CmdResponseExpect command request response
kSDIF_CmdResponseLengthLong command response length long
kSDIF_CmdCheckResponseCRC request check command response CRC

kSDIF_DataExpect request data transfer,ethier read/write
kSDIF_DataWriteToCard data transfer direction
kSDIF_DataStreamTransfer data transfer mode :stream/block transfer command
kSDIF_DataTransferAutoStop data transfer with auto stop at the end of
kSDIF_WaitPreTransferComplete wait pre transfer complete before sending this cmd
kSDIF_TransferStopAbort when host issue stop or abort cmd to stop data transfer ,this bit should set so that cmd/data state-machines of CIU can return to idle correctly
kSDIF_SendInitialization send initaliztion 80 clocks for SD card after power on
kSDIF_CmdUpdateClockRegisterOnly send cmd update the CIU clock register only
kSDIF_CmdtoReadCEATADevice host is perform read access to CE-ATA device
kSDIF_CmdExpectCCS command expect command completion signal signal
kSDIF_BootModeEnable this bit should only be set for mandatory boot mode
kSDIF_BootModeExpectAck boot mode expect ack
kSDIF_BootModeDisable when software set this bit along with START_CMD, CIU terminates the boot operation
kSDIF_BootModeAlternate select boot mode ,alternate or mandatory
kSDIF_CmdVoltageSwitch this bit set for CMD11 only
kSDIF_CmdDataUseHoldReg cmd and data send to card through the HOLD register

35.6.6 enum _sdif_command_type

Enumerator

kCARD_CommandTypeNormal Normal command.
kCARD_CommandTypeSuspend Suspend command.
kCARD_CommandTypeResume Resume command.
kCARD_CommandTypeAbort Abort command.

35.6.7 enum _sdif_response_type

Define the command response type from card to host controller.

Enumerator

kCARD_ResponseTypeNone Response type: none.
kCARD_ResponseTypeR1 Response type: R1.
kCARD_ResponseTypeR1b Response type: R1b.
kCARD_ResponseTypeR2 Response type: R2.
kCARD_ResponseTypeR3 Response type: R3.
kCARD_ResponseTypeR4 Response type: R4.
kCARD_ResponseTypeR5 Response type: R5.
kCARD_ResponseTypeR5b Response type: R5b.

kCARD_ResponseTypeR6 Response type: R6.
kCARD_ResponseTypeR7 Response type: R7.

35.6.8 enum _sdif_interrupt_mask

Enumerator

kSDIF_CardDetect mask for card detect
kSDIF_ResponseError command response error
kSDIF_CommandDone command transfer over
kSDIF_DataTransferOver data transfer over flag
kSDIF_WriteFIFORequest write FIFO request
kSDIF_ReadFIFORequest read FIFO request
kSDIF_ResponseCRCError reponse CRC error
kSDIF_DataCRCError data CRC error
kSDIF_ResponseTimeout response timeout
kSDIF_DataReadTimeout read data timeout
kSDIF_DataStarvationByHostTimeout data starvation by host time out
kSDIF_FIFOError indicate the FIFO underrun or overrun error
kSDIF_HardwareLockError hardware lock write error
kSDIF_DataStartBitError start bit error
kSDIF_AutoCmdDone indicate the auto command done
kSDIF_DataEndBitError end bit error
kSDIF_SDIOInterrupt interrupt from the SDIO card
kSDIF_CommandTransferStatus command transfer status collection
kSDIF_DataTransferStatus data transfer status collection
kSDIF_AllInterruptStatus all interrupt mask

35.6.9 enum _sdif_dma_status

Enumerator

kSDIF_DMATransFinishOneDescriptor DMA transfer finished for one DMA descriptor.
kSDIF_DMARecvFinishOneDescriptor DMA revieve finished for one DMA descriptor.
kSDIF_DMAFatalBusError DMA fatal bus error.
kSDIF_DMADescriptorUnavailable DMA descriptor unavailable.
kSDIF_DMACardErrorSummary card error summary
kSDIF_NormalInterruptSummary normal interrupt summary
kSDIF_AbnormalInterruptSummary abnormal interrupt summary

35.6.10 enum _sdif_dma_descriptor_flag

Enumerator

kSDIF_DisableCompleteInterrupt disable the complete interrupt flag for the ends in the buffer pointed to by this descriptor
kSDIF_DMADescriptorDataBufferEnd indicate this descriptor contain the last data buffer of data
kSDIF_DMADescriptorDataBufferStart indicate this descriptor contain the first data buffer of data,if first buffer size is 0,next descriptor contain the begaining of the data
kSDIF_DMASecondAddrChained indicate that the second addr in the descriptor is the next descriptor addr not the data buffer
kSDIF_DMADescriptorEnd indicate that the descriptor list reached its final descriptor
kSDIF_DMADescriptorOwnByDMA indicate the descriptor is own by SD/MMC DMA

35.6.11 enum _sdif_card_freq

Enumerator

kSDIF_Freq50MHZ 50MHZ mode
kSDIF_Freq400KHZ identificatioin mode

35.6.12 enum _sdif_clock_pharse_shift

Enumerator

kSDIF_ClcokPharseShift0 clock pharse shift 0
kSDIF_ClcokPharseShift90 clock pharse shift 90
kSDIF_ClcokPharseShift180 clock pharse shift 180
kSDIF_ClcokPharseShift270 clock pharse shift 270

35.7 Function Documentation

35.7.1 void SDIF_Init (SDIF_Type * *base*, sdif_config_t * *config*)

Configures the SDIF according to the user configuration.

Parameters

base	SDIF peripheral base address.
config	SDIF configuration information.

35.7.2 void SDIF_Deinit (SDIF_Type * *base*)

user should call this function follow with IP reset

Parameters

base	SDIF peripheral base address.

35.7.3 bool SDIF_SendCardActive (SDIF_Type * *base,* uint32_t *timeout*)

Parameters

base	SDIF peripheral base address.
timeout	value

35.7.4 static uint32_t SDIF_DetectCardInsert (SDIF_Type * *base,* bool *data3*) [inline], [static]

Parameters

base	SDIF peripheral base address.
data3	indicate use data3 as card insert detect pin will return the data3 PIN status in this condition

35.7.5 static void SDIF_EnableCardClock (SDIF_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	SDIF peripheral base address.
enable/disable	flag

35.7.6 static void SDIF_EnableLowPowerMode (SDIF_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SDIF peripheral base address.
enable/disable	flag

35.7.7 uint32_t SDIF_SetCardClock (SDIF_Type * *base*, uint32_t *srcClock_Hz*, uint32_t *target_HZ*)

Parameters

base	SDIF peripheral base address.
srcClock_Hz	SDIF source clock frequency united in Hz.
target_HZ	card bus clock frequency united in Hz.

Returns

The nearest frequency of busClock_Hz configured to SD bus.

35.7.8 bool SDIF_Reset (SDIF_Type * *base*, uint32_t *mask*, uint32_t *timeout*)

Parameters

base	SDIF peripheral base address.
mask	indicate which block to reset.
timeout	value,set to wait the bit self clear

Returns

reset result.

35.7.9 static void SDIF_EnableCardPower (SDIF_Type * *base,* bool *enable*) [inline], [static]

once turn power on, software should wait for regulator/switch ramp-up time before trying to initialize card.

Parameters

base	SDIF peripheral base address.
enable/disable	flag.

35.7.10 static uint32_t SDIF_GetCardWriteProtect (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

35.7.11 static void SDIF_SetCardBusWidth (SDIF_Type * *base,* sdif_bus_width_t *type*) [inline], [static]

Parameters

base	SDIF peripheral base address.
data	bus width type

35.7.12 static void SDIF_AssertHardwareReset (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

35.7.13 status_t SDIF_SendCommand (SDIF_Type * *base,* sdif_command_t * *cmd,* uint32_t *timeout*)

Parameters

base	SDIF peripheral base address.
command	configuration collection
timeout	value

Returns

command excute status

35.7.14 static void SDIF_EnableGlobalInterrupt (SDIF_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	SDIF peripheral base address.
enable/disable	flag

35.7.15 static void SDIF_EnableInterrupt (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
interrupt	mask

35.7.16 static void SDIF_DisableInterrupt (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
interrupt	mask

35.7.17 static uint32_t SDIF_GetInterruptStatus (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

35.7.18 static void SDIF_ClearInterruptStatus (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
status	mask to clear

35.7.19 void SDIF_TransferCreateHandle (SDIF_Type * *base,* sdif_handle_t * *handle,* sdif_transfer_callback_t * *callback,* void * *userData*)

register call back function for interrupt and enable the interrupt

Parameters

base	SDIF peripheral base address.
handle	SDIF handle pointer.
callback	Structure pointer to contain all callback functions.
userData	Callback function parameter.

35.7.20 static void SDIF_EnableDmaInterrupt (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
interrupt	mask to set

35.7.21 static void SDIF_DisableDmaInterrupt (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
interrupt	mask to clear

35.7.22 static uint32_t SDIF_GetInternalDMAStatus (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

Returns

the internal DMA status register

35.7.23 static void SDIF_ClearInternalDMAStatus (SDIF_Type * *base,* uint32_t *mask*) [inline], [static]

Parameters

base	SDIF peripheral base address.
status	mask to clear

35.7.24 status_t SDIF_InternalDMAConfig (SDIF_Type * *base,* sdif_dma_config_t * *config,* const uint32_t * *data,* uint32_t *dataSize*)

Parameters

base	SDIF peripheral base address.
internal	DMA configuration collection
data	buffer pointer
data	buffer size

35.7.25 static void SDIF_SendReadWait (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

35.7.26 bool SDIF_AbortReadData (SDIF_Type * *base*, uint32_t *timeout*)

Parameters

base	SDIF peripheral base address.
timeout	value to wait this bit self clear which indicate the data machine reset to idle

35.7.27 static void SDIF_EnableCEATAInterrupt (SDIF_Type * *base*, bool *enable*) [inline], [static]

Parameters

base	SDIF peripheral base address.
enable/disable	flag

35.7.28 status_t SDIF_TransferNonBlocking (SDIF_Type * *base*, sdif_handle_t * *handle*, sdif_dma_config_t * *dmaConfig*, sdif_transfer_t * *transfer*)

Parameters

base	SDIF peripheral base address.
sdif	handle
DMA	config structure This parameter can be config as: 1. NULL In this condition, polling transfer mode is selected 2. avaliable DMA config In this condition, DMA transfer mode is selected
sdif	transfer configuration collection

35.7.29 status_t SDIF_TransferBlocking (SDIF_Type * *base*, sdif_dma_config_t * *dmaConfig*, sdif_transfer_t * *transfer*)

Parameters

base	SDIF peripheral base address.
DMA	config structure 1. NULL In this condition, polling transfer mode is selected 2. avaliable DMA config In this condition, DMA transfer mode is selected
sdif	transfer configuration collection

35.7.30 status_t SDIF_ReleaseDMADescriptor (SDIF_Type * *base*, sdif_dma_config_t * *dmaConfig*)

Parameters

base	SDIF peripheral base address.
sdif	DMA config pointer

35.7.31 void SDIF_GetCapability (SDIF_Type * *base*, sdif_capability_t * *capability*)

Parameters

base	SDIF peripheral base address.
sdif	capability pointer

35.7.32 static uint32_t SDIF_GetControllerStatus (SDIF_Type * *base*) [inline], [static]

Parameters

base	SDIF peripheral base address.

35.7.33 static void SDIF_SendCCSD (SDIF_Type * *base*, bool *withAutoStop*) [inline], [static]

Parameters

base	SDIF peripheral base address.
send	auto stop flag

35.7.34 void SDIF_ConfigClockDelay (uint32_t *target_HZ,* uint32_t *divider*)

Parameters

target	freq work mode
clock	divider which is used to decide if use pharse shift for delay

Chapter 36
SPIFI: SPIFI flash interface driver

36.1 Overview

Modules

- SPIFI DMA Driver
- SPIFI Driver

Data Structures

- struct spifi_command_t
  SPIFI command structure. More...
- struct spifi_config_t
  SPIFI region configuration structure. More...
- struct spifi_transfer_t
  Transfer structure for SPIFI. More...
- struct spifi_dma_handle_t
  SPIFI DMA transfer handle, users should not touch the content of the handle. More...

Typedefs

- typedef void(∗ spifi_dma_callback_t)(SPIFI_Type ∗base, spifi_dma_handle_t ∗handle, status_t status, void ∗userData)
  SPIFI DMA transfer callback function for finish and error.

Enumerations

- enum _status_t {
  kStatus_SPIFI_Idle = MAKE_STATUS(kStatusGroup_SPIFI, 0),
  kStatus_SPIFI_Busy = MAKE_STATUS(kStatusGroup_SPIFI, 1),
  kStatus_SPIFI_Error = MAKE_STATUS(kStatusGroup_SPIFI, 2) }
  Status structure of SPIFI.
- enum spifi_interrupt_enable_t { kSPIFI_CommandFinishInterruptEnable = SPIFI_CTRL_INTEN_MASK }
  SPIFI interrupt source.
- enum spifi_spi_mode_t {
  kSPIFI_SPISckLow = 0x0U,
  kSPIFI_SPISckHigh = 0x1U }
  SPIFI SPI mode select.
- enum spifi_dual_mode_t {
  kSPIFI_QuadMode = 0x0U,
  kSPIFI_DualMode = 0x1U }
  SPIFI dual mode select.

- enum spifi_data_direction_t {
  kSPIFI_DataInput = 0x0U,
  kSPIFI_DataOutput = 0x1U }
  SPIFI data direction.
- enum spifi_command_format_t {
  kSPIFI_CommandAllSerial = 0x0,
  kSPIFI_CommandDataQuad = 0x1U,
  kSPIFI_CommandOpcodeSerial = 0x2U,
  kSPIFI_CommandAllQuad = 0x3U }
  SPIFI command opcode format.
- enum spifi_command_type_t {
  kSPIFI_CommandOpcodeOnly = 0x1U,
  kSPIFI_CommandOpcodeAddrOneByte = 0x2U,
  kSPIFI_CommandOpcodeAddrTwoBytes = 0x3U,
  kSPIFI_CommandOpcodeAddrThreeBytes = 0x4U,
  kSPIFI_CommandOpcodeAddrFourBytes = 0x5U,
  kSPIFI_CommandNoOpcodeAddrThreeBytes = 0x6U,
  kSPIFI_CommandNoOpcodeAddrFourBytes = 0x7U }
  SPIFI command type.
- enum _spifi_status_flags {
  kSPIFI_MemoryCommandWriteFinished = SPIFI_STAT_MCINIT_MASK,
  kSPIFI_CommandWriteFinished = SPIFI_STAT_CMD_MASK,
  kSPIFI_InterruptRequest = SPIFI_STAT_INTRQ_MASK }
  SPIFI status flags.

Functions

- static void SPIFI_EnableDMA (SPIFI_Type *base, bool enable)
  Enable or disable DMA request for SPIFI.
- static uint32_t SPIFI_GetDataRegisterAddress (SPIFI_Type *base)
  Gets the SPIFI data register address.
- static void SPIFI_WriteData (SPIFI_Type *base, uint32_t data)
  Write a word data in address of SPIFI.
- static uint32_t SPIFI_ReadData (SPIFI_Type *base)
  Read data from serial flash.

Driver version

- #define FSL_SPIFI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  SPIFI driver version 2.0.0.

Initialization and deinitialization

- void SPIFI_Init (SPIFI_Type *base, const spifi_config_t *config)
  Initializes the SPIFI with the user configuration structure.
- void SPIFI_GetDefaultConfig (spifi_config_t *config)
  Get SPIFI default configure settings.
- void SPIFI_Deinit (SPIFI_Type *base)
  Deinitializes the SPIFI regions.

Basic Control Operations

- void SPIFI_SetCommand (SPIFI_Type *base, spifi_command_t *cmd)
  Set SPIFI flash command.
- static void SPIFI_SetCommandAddress (SPIFI_Type *base, uint32_t addr)
  Set SPIFI command address.
- static void SPIFI_SetIntermediateData (SPIFI_Type *base, uint32_t val)
  Set SPIFI intermediate data.
- static void SPIFI_SetCacheLimit (SPIFI_Type *base, uint32_t val)
  Set SPIFI Cache limit value.
- static void SPIFI_ResetCommand (SPIFI_Type *base)
  Reset the command field of SPIFI.
- void SPIFI_SetMemoryCommand (SPIFI_Type *base, spifi_command_t *cmd)
  Set SPIFI flash AHB read command.
- static void SPIFI_EnableInterrupt (SPIFI_Type *base, uint32_t mask)
  Enable SPIFI interrupt.
- static void SPIFI_DisableInterrupt (SPIFI_Type *base, uint32_t mask)
  Disable SPIFI interrupt.

Status

- static uint32_t SPIFI_GetStatusFlag (SPIFI_Type *base)
  Get the status of all interrupt flags for SPIFI.

DMA Transactional

- void SPIFI_TransferTxCreateHandleDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, spifi_dma_callback_t callback, void *userData, dma_handle_t *dmaHandle)
  Initializes the SPIFI handle for send which is used in transactional functions and set the callback.
- void SPIFI_TransferRxCreateHandleDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, spifi_dma_callback_t callback, void *userData, dma_handle_t *dmaHandle)
  Initializes the SPIFI handle for receive which is used in transactional functions and set the callback.
- status_t SPIFI_TransferSendDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, spifi_transfer_t *xfer)
  Transfers SPIFI data using an DMA non-blocking method.
- status_t SPIFI_TransferReceiveDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, spifi_transfer_t *xfer)
  Receives data using an DMA non-blocking method.
- void SPIFI_TransferAbortSendDMA (SPIFI_Type *base, spifi_dma_handle_t *handle)
  Aborts the sent data using DMA.
- void SPIFI_TransferAbortReceiveDMA (SPIFI_Type *base, spifi_dma_handle_t *handle)
  Aborts the receive data using DMA.
- status_t SPIFI_TransferGetSendCountDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, size_t *count)
  Gets the transferred counts of send.
- status_t SPIFI_TransferGetReceiveCountDMA (SPIFI_Type *base, spifi_dma_handle_t *handle, size_t *count)
  Gets the status of the receive transfer.

36.2 Data Structure Documentation

36.2.1 struct spifi_command_t

Data Fields

- uint16_t dataLen
  How many data bytes are needed in this command.
- bool isPollMode
  For command need to read data from serial flash.
- spifi_data_direction_t direction
  Data direction of this command.
- uint8_t intermediateBytes
  How many intermediate bytes needed.
- spifi_command_format_t format
  Command format.
- spifi_command_type_t type
  Command type.
- uint8_t opcode
  Command opcode value.

36.2.1.0.0.52 Field Documentation

36.2.1.0.0.52.1 uint16_t spifi_command_t::dataLen

36.2.1.0.0.52.2 spifi_data_direction_t spifi_command_t::direction

36.2.2 struct spifi_config_t

Data Fields

- uint16_t timeout
  SPI transfer timeout, the unit is SCK cycles.
- uint8_t csHighTime
  CS high time cycles.
- bool disablePrefetch
  True means SPIFI will not attempt a speculative prefetch.
- bool disableCachePrefech
  Disable prefetch of cache line.
- bool isFeedbackClock
  Is data sample uses feedback clock.
- spifi_spi_mode_t spiMode
  SPIFI spi mode select.
- bool isReadFullClockCycle
  If enable read full clock cycle.
- spifi_dual_mode_t dualMode
  SPIFI dual mode, dual or quad.

36.2.2.0.0.53 Field Documentation

36.2.2.0.0.53.1 bool spifi_config_t::disablePrefetch

36.2.2.0.0.53.2 bool spifi_config_t::isFeedbackClock

36.2.2.0.0.53.3 bool spifi_config_t::isReadFullClockCycle

36.2.2.0.0.53.4 spifi_dual_mode_t spifi_config_t::dualMode

36.2.3 struct spifi_transfer_t

Data Fields

- uint8_t * data
  Pointer to data to transmit.
- size_t dataSize
  Bytes to be transmit.

36.2.4 struct _spifi_dma_handle

Data Fields

- dma_handle_t * dmaHandle
  DMA handler for SPIFI send.
- size_t transferSize
  Bytes need to transfer.
- uint32_t state
  Internal state for SPIFI DMA transfer.
- spifi_dma_callback_t callback
  Callback for users while transfer finish or error occurred.
- void * userData
  User callback parameter.

36.2.4.0.0.54 Field Documentation

36.2.4.0.0.54.1 size_t spifi_dma_handle_t::transferSize

36.3 Macro Definition Documentation

36.3.1 #define FSL_SPIFI_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

36.4 Enumeration Type Documentation

36.4.1 enum _status_t

Enumerator

kStatus_SPIFI_Idle SPIFI is in idle state.
kStatus_SPIFI_Busy SPIFI is busy.
kStatus_SPIFI_Error Error occurred during SPIFI transfer.

36.4.2 enum spifi_interrupt_enable_t

Enumerator

kSPIFI_CommandFinishInterruptEnable Interrupt while command finished.

36.4.3 enum spifi_spi_mode_t

Enumerator

kSPIFI_SPISckLow SCK low after last bit of command, keeps low while CS high.
kSPIFI_SPISckHigh SCK high after last bit of command and while CS high.

36.4.4 enum spifi_dual_mode_t

Enumerator

kSPIFI_QuadMode SPIFI uses IO3:0.
kSPIFI_DualMode SPIFI uses IO1:0.

36.4.5 enum spifi_data_direction_t

Enumerator

kSPIFI_DataInput Data input from serial flash.
kSPIFI_DataOutput Data output to serial flash.

36.4.6 enum spifi_command_format_t

Enumerator

kSPIFI_CommandAllSerial All fields of command are serial.
kSPIFI_CommandDataQuad Only data field is dual/quad, others are serial.
kSPIFI_CommandOpcodeSerial Only opcode field is serial, others are quad/dual.
kSPIFI_CommandAllQuad All fields of command are dual/quad mode.

36.4.7 enum spifi_command_type_t

Enumerator

kSPIFI_CommandOpcodeOnly Command only have opcode, no address field.
kSPIFI_CommandOpcodeAddrOneByte Command have opcode and also one byte address field.
kSPIFI_CommandOpcodeAddrTwoBytes Command have opcode and also two bytes address field.

kSPIFI_CommandOpcodeAddrThreeBytes Command have opcode and also three bytes address field.
kSPIFI_CommandOpcodeAddrFourBytes Command have opcode and also four bytes address field.

kSPIFI_CommandNoOpcodeAddrThreeBytes Command have no opcode and three bytes address field.
kSPIFI_CommandNoOpcodeAddrFourBytes Command have no opcode and four bytes address field.

36.4.8 enum _spifi_status_flags

Enumerator

kSPIFI_MemoryCommandWriteFinished Memory command write finished.
kSPIFI_CommandWriteFinished Command write finished.
kSPIFI_InterruptRequest CMD flag from 1 to 0, means command execute finished.

36.5 Function Documentation

36.5.1 void SPIFI_Init (SPIFI_Type * *base,* const spifi_config_t * *config*)

This function configures the SPIFI module with the user-defined configuration.

Parameters

base	SPIFI peripheral base address.
config	The pointer to the configuration structure.

36.5.2 void SPIFI_GetDefaultConfig (spifi_config_t * *config*)

Parameters

config	SPIFI config structure pointer.

36.5.3 void SPIFI_Deinit (SPIFI_Type * *base*)

Parameters

base	SPIFI peripheral base address.

36.5.4 void SPIFI_SetCommand (SPIFI_Type * *base,* spifi_command_t * *cmd*)

Parameters

base	SPIFI peripheral base address.
cmd	SPIFI command structure pointer.

36.5.5 static void SPIFI_SetCommandAddress (SPIFI_Type * *base,* uint32_t *addr*) [inline], [static]

Parameters

base	SPIFI peripheral base address.
addr	Address value for the command.

36.5.6 static void SPIFI_SetIntermediateData (SPIFI_Type * *base,* uint32_t *val*) [inline], [static]

Before writing a command wihch needs specific intermediate value, users shall call this function to write it. The main use of this function for current serial flash is to select no-opcode mode and cancelling this mode. As dummy cycle do not care about the value, no need to call this function.

Parameters

base	SPIFI peripheral base address.
val	Intermediate data.

36.5.7 static void SPIFI_SetCacheLimit (SPIFI_Type * *base,* uint32_t *val*) [inline], [static]

SPIFI includes caching of prevously-accessed data to improve performance. Software can write an address to this function, to prevent such caching at and above the address.

Parameters

base	SPIFI peripheral base address.
val	Zero-based upper limit of cacheable memory.

36.5.8 static void SPIFI_ResetCommand (SPIFI_Type * *base*) [inline], [static]

This function is used to abort the current command or memory mode.

Parameters

base	SPIFI peripheral base address.

36.5.9 void SPIFI_SetMemoryCommand (SPIFI_Type * *base,* spifi_command_t * *cmd*)

Call this function means SPIFI enters to memory mode, while users need to use command, a SPIFI_Reset-Command shall be called.

Parameters

base	SPIFI peripheral base address.
cmd	SPIFI command structure pointer.

36.5.10 static void SPIFI_EnableInterrupt (SPIFI_Type * *base,* uint32_t *mask*) [inline], [static]

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

base	SPIFI peripheral base address.
mask	SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: spifi_interrupt_enable_t

36.5.11 static void SPIFI_DisableInterrupt (SPIFI_Type * *base,* uint32_t *mask*) [inline], [static]

The interrupt is triggered only in command mode, and it means the command now is finished.

Parameters

base	SPIFI peripheral base address.
mask	SPIFI interrupt enable mask. It is a logic OR of members the enumeration :: spifi_interrupt_enable_t

36.5.12 static uint32_t SPIFI_GetStatusFlag (SPIFI_Type * *base*) [inline], [static]

Parameters

base	SPIFI peripheral base address.

Returns

SPIFI flag status

36.5.13 static void SPIFI_EnableDMA (SPIFI_Type * *base,* bool *enable*) [inline], [static]

Parameters

base	SPIFI peripheral base address.
enable	True means enable DMA and false means disable DMA.

36.5.14 static uint32_t SPIFI_GetDataRegisterAddress (SPIFI_Type * *base*) [inline], [static]

This API is used to provide a transfer address for the SPIFI DMA transfer configuration.

Parameters

base	SPIFI base pointer

Returns

data register address

36.5.15 static void SPIFI_WriteData (SPIFI_Type * *base,* uint32_t *data*) [inline], [static]

Users can write a page or at least a word data into SPIFI address.

Parameters

base	SPIFI peripheral base address.
data	Data need be write.

36.5.16 static uint32_t SPIFI_ReadData (SPIFI_Type * *base*) [inline], [static]

Users should notice before call this function, the data length field in command register shall larger than 4, otherwise a hardfault will happen.

Parameters

base	SPIFI peripheral base address.

Returns

Data input from flash.

36.5.17 void SPIFI_TransferTxCreateHandleDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, spifi_dma_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

Parameters

base	SPIFI peripheral base address
handle	Pointer to spifi_dma_handle_t structure
callback	SPIFI callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested DMA handle for DMA transfer

36.5.18 void SPIFI_TransferRxCreateHandleDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, spifi_dma_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

Parameters

base	SPIFI peripheral base address
handle	Pointer to spifi_dma_handle_t structure
callback	SPIFI callback, NULL means no callback.
userData	User callback function data.
rxDmaHandle	User requested DMA handle for DMA transfer

36.5.19 status_t SPIFI_TransferSendDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, spifi_transfer_t * *xfer*)

This function writes data to the SPIFI transmit FIFO. This function is non-blocking.

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to spifi_dma_handle_t structure
xfer	SPIFI transfer structure.

36.5.20 status_t SPIFI_TransferReceiveDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, spifi_transfer_t * *xfer*)

This function receive data from the SPIFI receive buffer/FIFO. This function is non-blocking.

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to spifi_dma_handle_t structure
xfer	SPIFI transfer structure.

36.5.21 void SPIFI_TransferAbortSendDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*)

This function aborts the sent data using DMA.

Parameters

base	SPIFI peripheral base address.
handle	Pointer to spifi_dma_handle_t structure

36.5.22 void SPIFI_TransferAbortReceiveDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*)

This function abort receive data which using DMA.

Parameters

base	SPIFI peripheral base address.
handle	Pointer to spifi_dma_handle_t structure

36.5.23 status_t SPIFI_TransferGetSendCountDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to spifi_dma_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.

36.5.24 status_t SPIFI_TransferGetReceiveCountDMA (SPIFI_Type * *base*, spifi_dma_handle_t * *handle*, size_t * *count*)

Parameters

base	Pointer to QuadSPI Type.
handle	Pointer to spifi_dma_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.

36.6 SPIFI Driver

SPIFI driver includes functional APIs.

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

36.6.1 Typical use case

36.6.1.1 SPIFI transfer using an polling method

```
#define PAGE_SIZE (256)
#define SECTOR_SIZE (4096)
/* Initialize SPIFI */
SPIFI_GetDefaultConfig(&config);
SPIFI_Init(EXAMPLE_SPIFI, &config, sourceClockFreq);

/* Set the buffer */
for (i = 0; i < PAGE_SIZE; i ++)
{
    g_buffer[i] = i;

}

/* Write enable */
SPIFI_SetCommand(EXAMPLE_SPIFI, &command[WRITE_ENABLE]);
/* Set address */
SPIFI_SetCommandAddress(EXAMPLE_SPIFI, FSL_FEATURE_SPIFI_START_ADDRESS);
/* Erase sector */
SPIFI_SetCommand(EXAMPLE_SPIFI, &command[ERASE_SECTOR]);
/* Check if finished */
check_if_finish();

/* Program page */

while (page < (SECTOR_SIZE/PAGE_SIZE))
{
    SPIFI_SetCommand(EXAMPLE_SPIFI, &command[WRITE_ENABLE]);
    SPIFI_SetCommandAddress(EXAMPLE_SPIFI, FSL_FEATURE_SPIFI_START_ADDRESS + page *
      PAGE_SIZE);
    SPIFI_SetCommand(EXAMPLE_SPIFI, &command[PROGRAM_PAGE]);
    for (i = 0; i < PAGE_SIZE; i += 4)
    {
        for (j = 0; j < 4; j ++)
        {
            data |= ((uint32_t)(g_buffer[i + j])) << (j * 8);
        }
        SPIFI_WriteData(EXAMPLE_SPIFI, data);
        data = 0;
    }
    page ++;
    check_if_finish();
}
```

36.7 SPIFI DMA Driver

This chapter describes the programming interface of the SPIFI DMA driver. SPIFI DMA driver includes transactional APIs.

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 spifi_handle_t as the first parameter. Initialize the handle by calling the SPIFI_TransferCreateHandleDMA() API.

36.7.1 Typical use case

36.7.1.1 SPIFI Send/receive using a DMA method

```
/* Initialize SPIFI */
#define PAGE_SIZE (256)
#define SECTOR_SIZE (4096)
SPIFI_GetDefaultConfig(&config);
SPIFI_Init(EXAMPLE_SPIFI, &config, sourceClockFreq);
SPIFI_TransferRxCreateHandleDMA(EXAMPLE_SPIFI, &handle, callback, NULL, &
      s_DmaHandle);

/* Set the buffer */
for (i = 0; i < PAGE_SIZE; i ++)
{
    g_buffer[i] = i;
}

/* Write enable */
SPIFI_SetCommand(EXAMPLE_SPIFI, &command[WRITE_ENABLE]);
/* Set address */
SPIFI_SetCommandAddress(EXAMPLE_SPIFI, FSL_FEATURE_SPIFI_START_ADDRESS);
/* Erase sector */
SPIFI_SetCommand(EXAMPLE_SPIFI, &command[ERASE_SECTOR]);

/* Check if finished */
check_if_finish();

/* Program page */
while (page < (SECTOR_SIZE/PAGE_SIZE))
{
    SPIFI_SetCommand(EXAMPLE_SPIFI, &command[WRITE_ENABLE]);
    SPIFI_SetCommandAddress(EXAMPLE_SPIFI, FSL_FEATURE_SPIFI_START_ADDRESS + page *
      PAGE_SIZE);
    SPIFI_SetCommand(EXAMPLE_SPIFI, &command[PROGRAM_PAGE]);
    xfer.data = g_buffer;
    xfer.dataSize = PAGE_SIZE;
    SPIFI_TransferSendDMA(EXAMPLE_SPIFI, &handle, &xfer);
    while (!finished)
    {}
    finished = false;
    page ++;
    check_if_finish();
}
```

Chapter 37
SYSCON: System Configuration

37.1 Overview

The SDK provides a peripheral clock and power driver for the SYSCON module of LPC devices. For furter details, see corresponding chapter.

Modules

- Clock driver

37.2 Clock driver

37.2.1 Overview

The SDK provides a peripheral clock driver for the SYSCON module of LPC devices.

37.2.2 Function description

Clock driver provides these functions:

- Functions to initialize the Core clock to given frequency
- Functions to configure the clock selection muxes.
- Functions to setup peripheral clock dividers
- Functions to set the flash wait states for the input freuqency
- Functions to get the frequency of the selected clock
- Functions to set PLL frequency

37.2.2.1 SYSCON Clock frequency functions

SYSCON clock module provides clocks, such as MCLKCLK, ADCCLK, DMICCLK, MCGFLLCLK, FXCOMCLK,WDTOSC, RTCOSC, USBCLK and SYSPLL. The functions CLOCK_EnableClock() and CLOCK_DisableClock() enables and disables the various clocks. CLOCK_SetupFROClocking() initializes the FRO to 12MHz, 48 MHz or 96 MHz frequency. CLOCK_SetupPLLData(), CLOCK_SetupSystemPLLPrec(), and CLOCK_SetPLLFreq() functions are used to setup the PLL. The SYSCON clock driver provides functions to get the frequency of these clocks, such as CLOCK_GetFreq(), CLOCK_GetFro12MFreq(), CLOCK_GetExtClkFreq(), CLOCK_GetWdtOscFreq(), CLOCK_GetFroHfFreq(), CLOCK_GetPllOutFreq(), CLOCK_GetOsc32KFreq() , CLOCK_GetCoreSysClkFreq(), CLOCK_GetI2SMClkFreq(),CLOCK_GetFlexCommClkFreq and CLOCK_GetAsyncApbClkFreq.

37.2.2.2 SYSCON clock Selection Muxes

The SYSCON clock driver provides the function to configure the clock selected. The function CLOCK_AttachClk() is implemented for this. The function selects the clock source for a particular peripheral like MAINCLK, DMIC, FLEXCOMM, USB, ADC and PLL.

37.2.2.3 SYSCON clock dividers

The SYSCON clock module provides the function to setup the peripheral clock dividers. The function CLOCK_SetClkDiv() configures the CLKDIV registers for various periperals like USB, DMIC, I2S, SYSTICK, AHB, ADC and also for CLKOUT and TRACE functions.

37.2.2.4 SYSCON flash wait states

The SYSCON clock driver provides the function CLOCK_SetFLASHAccessCyclesForFreq() that configures FLASHCFG register with a selected FLASHTIM value.

37.2.3 Typical use case

```
POWER_DisablePD(kPDRUNCFG_PD_FRO_EN); /*!< Ensure FRO is on so that we can switch to its 12MH
```

Files

- file fsl_clock.h

Data Structures

- struct pll_config_t
  PLL configuration structure. More...
- struct pll_setup_t
  PLL setup structure This structure can be used to pre-build a PLL setup configuration at run-time and quickly set the PLL to the configuration. More...
- struct usb_pll_setup_t
  PLL setup structure This structure can be used to pre-build a USB PLL setup configuration at run-time and quickly set the usb PLL to the configuration. More...

Macros

- #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0
  Configure whether driver controls clock.
- #define ADC_CLOCKS
  Clock ip name array for ROM.
- #define ROM_CLOCKS
  Clock ip name array for ROM.
- #define SRAM_CLOCKS
  Clock ip name array for SRAM.
- #define FLASH_CLOCKS
  Clock ip name array for FLASH.
- #define FMC_CLOCKS
  Clock ip name array for FMC.
- #define EEPROM_CLOCKS
  Clock ip name array for EEPROM.
- #define SPIFI_CLOCKS
  Clock ip name array for SPIFI.
- #define INPUTMUX_CLOCKS
  Clock ip name array for INPUTMUX.
- #define IOCON_CLOCKS

Clock ip name array for IOCON.

- #define GPIO_CLOCKS
  Clock ip name array for GPIO.
- #define PINT_CLOCKS
  Clock ip name array for PINT.
- #define GINT_CLOCKS
  Clock ip name array for GINT.
- #define DMA_CLOCKS
  Clock ip name array for DMA.
- #define CRC_CLOCKS
  Clock ip name array for CRC.
- #define WWDT_CLOCKS
  Clock ip name array for WWDT.
- #define RTC_CLOCKS
  Clock ip name array for RTC.
- #define ADC0_CLOCKS
  Clock ip name array for ADC0.
- #define MRT_CLOCKS
  Clock ip name array for MRT.
- #define RIT_CLOCKS
  Clock ip name array for RIT.
- #define SCT_CLOCKS
  Clock ip name array for SCT0.
- #define MCAN_CLOCKS
  Clock ip name array for MCAN.
- #define UTICK_CLOCKS
  Clock ip name array for UTICK.
- #define FLEXCOMM_CLOCKS
  Clock ip name array for FLEXCOMM.
- #define LPUART_CLOCKS
  Clock ip name array for LPUART.
- #define BI2C_CLOCKS
  Clock ip name array for BI2C.
- #define LPSI_CLOCKS
  Clock ip name array for LSPI.
- #define FLEXI2S_CLOCKS
  Clock ip name array for FLEXI2S.
- #define DMIC_CLOCKS
  Clock ip name array for DMIC.
- #define CTIMER_CLOCKS
  Clock ip name array for CT32B.
- #define LCD_CLOCKS
  Clock ip name array for LCD.
- #define SDIO_CLOCKS
  Clock ip name array for SDIO.
- #define USBRAM_CLOCKS
  Clock ip name array for USBRAM.
- #define EMC_CLOCKS
  Clock ip name array for EMC.
- #define ETH_CLOCKS
  Clock ip name array for ETH.

- #define AES_CLOCKS
  Clock ip name array for AES.
- #define OTP_CLOCKS
  Clock ip name array for OTP.
- #define RNG_CLOCKS
  Clock ip name array for RNG.
- #define USBHMR0_CLOCKS
  Clock ip name array for USBHMR0.
- #define USBHSL0_CLOCKS
  Clock ip name array for USBHSL0.
- #define SHA0_CLOCKS
  Clock ip name array for SHA0.
- #define SMARTCARD_CLOCKS
  Clock ip name array for SMARTCARD.
- #define USBD_CLOCKS
  Clock ip name array for USBD.
- #define USBH_CLOCKS
  Clock ip name array for USBH.
- #define CLK_GATE_REG_OFFSET_SHIFT 8U
  Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.
- #define MUX_A(m, choice) (((m) << 0) | ((choice + 1) << 8))
  Clock Mux Switches The encoding is as follows each connection identified is 64bits wide starting from LSB upwards.
- #define PLL_CONFIGFLAG_USEINRATE (1 << 0)
  PLL configuration structure flags for 'flags' field These flags control how the PLL configuration function sets up the PLL setup structure.
- #define PLL_CONFIGFLAG_FORCENOFRACT
  Force non-fractional output mode, PLL output will not use the fractional, automatic bandwidth, or SS \\\ \\\\\\\\\\\\\\\\ hardware.
- #define PLL_SETUPFLAG_POWERUP (1 << 0)
  PLL setup structure flags for 'flags' field These flags control how the PLL setup function sets up the PLL.
- #define PLL_SETUPFLAG_WAITLOCK (1 << 1)
  Setup will wait for PLL lock, implies the PLL will be pwoered on.
- #define PLL_SETUPFLAG_ADGVOLT (1 << 2)
  Optimize system voltage for the new PLL rate.

Enumerations

- enum clock_ip_name_t
  Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.
- enum clock_name_t {

kCLOCK_CoreSysClk,
kCLOCK_BusClk,
kCLOCK_ClockOut,
kCLOCK_FroHf,
kCLOCK_SpiFi,
kCLOCK_Adc,
kCLOCK_Usb0,
kCLOCK_Usb1,
kCLOCK_UsbPll,
kCLOCK_Mclk,
kCLOCK_Sct,
kCLOCK_SDio,
kCLOCK_EMC,
kCLOCK_LCD,
kCLOCK_MCAN0,
kCLOCK_MCAN1,
kCLOCK_Fro12M,
kCLOCK_ExtClk,
kCLOCK_PllOut,
kCLOCK_UsbClk,
kClock_WdtOsc,
kCLOCK_Frg,
kCLOCK_Dmic,
kCLOCK_AsyncApbClk,
kCLOCK_FlexI2S,
kCLOCK_Flexcomm0,
kCLOCK_Flexcomm1,
kCLOCK_Flexcomm2,
kCLOCK_Flexcomm3,
kCLOCK_Flexcomm4,
kCLOCK_Flexcomm5,
kCLOCK_Flexcomm6,
kCLOCK_Flexcomm7,
kCLOCK_Flexcomm8,
kCLOCK_Flexcomm9 }

Clock name used to get clock frequency.

- enum async_clock_src_t {
  kCLOCK_AsyncMainClk = 0,
  kCLOCK_AsyncFro12Mhz }
- enum clock_flashtim_t {

kCLOCK_Flash1Cycle = 0,
kCLOCK_Flash2Cycle,
kCLOCK_Flash3Cycle,
kCLOCK_Flash4Cycle,
kCLOCK_Flash5Cycle,
kCLOCK_Flash6Cycle,
kCLOCK_Flash7Cycle,
kCLOCK_Flash8Cycle }

FLASH Access time definitions.

- enum ss_progmodfm_t {
  kSS_MF_512 = (0 << 20),
  kSS_MF_384 = (1 << 20),
  kSS_MF_256 = (2 << 20),
  kSS_MF_128 = (3 << 20),
  kSS_MF_64 = (4 << 20),
  kSS_MF_32 = (5 << 20),
  kSS_MF_24 = (6 << 20),
  kSS_MF_16 = (7 << 20) }

  PLL Spread Spectrum (SS) Programmable modulation frequency See (MF) field in the SYSPLLSSCTRL1 register in the UM.
- enum ss_progmoddp_t {
  kSS_MR_K0 = (0 << 23),
  kSS_MR_K1 = (1 << 23),
  kSS_MR_K1_5 = (2 << 23),
  kSS_MR_K2 = (3 << 23),
  kSS_MR_K3 = (4 << 23),
  kSS_MR_K4 = (5 << 23),
  kSS_MR_K6 = (6 << 23),
  kSS_MR_K8 = (7 << 23) }

  PLL Spread Spectrum (SS) Programmable frequency modulation depth See (MR) field in the SYSPLLSSCTRL1 register in the UM.
- enum ss_modwvctrl_t {
  kSS_MC_NOC = (0 << 26),
  kSS_MC_RECC = (2 << 26),
  kSS_MC_MAXC = (3 << 26) }

  PLL Spread Spectrum (SS) Modulation waveform control See (MC) field in the SYSPLLSSCTRL1 register in the UM.
- enum pll_error_t {
  kStatus_PLL_Success = MAKE_STATUS(kStatusGroup_Generic, 0),
  kStatus_PLL_OutputTooLow = MAKE_STATUS(kStatusGroup_Generic, 1),
  kStatus_PLL_OutputTooHigh = MAKE_STATUS(kStatusGroup_Generic, 2),
  kStatus_PLL_InputTooLow = MAKE_STATUS(kStatusGroup_Generic, 3),
  kStatus_PLL_InputTooHigh = MAKE_STATUS(kStatusGroup_Generic, 4),
  kStatus_PLL_OutsideIntLimit = MAKE_STATUS(kStatusGroup_Generic, 5),
  kStatus_PLL_CCOTooLow = MAKE_STATUS(kStatusGroup_Generic, 6),
  kStatus_PLL_CCOTooHigh = MAKE_STATUS(kStatusGroup_Generic, 7) }

PLL status definitions.

- enum clock_usb_src_t {
  kCLOCK_UsbSrcFro = (uint32_t)kCLOCK_FroHf,
  kCLOCK_UsbSrcSystemPll = (uint32_t)kCLOCK_PllOut,
  kCLOCK_UsbSrcMainClock = (uint32_t)kCLOCK_CoreSysClk,
  kCLOCK_UsbSrcUsbPll = (uint32_t)kCLOCK_UsbPll,
  kCLOCK_UsbSrcNone = SYSCON_USB0CLKSEL_SEL(7) }
  USB clock source definition.
- enum usb_pll_psel
  USB PDEL Divider.

Functions

- static void CLOCK_SetFLASHAccessCycles (clock_flashtim_t clks)
  Set FLASH memory access time in clocks.
- status_t CLOCK_SetupFROClocking (uint32_t iFreq)
  Initialize the Core clock to given frequency (12, 48 or 96 MHz). Turns on FRO and uses default CCO, if freq is 12000000, then high speed output is off, else high speed output is enabled.
- void CLOCK_AttachClk (clock_attach_id_t connection)
  Configure the clock selection muxes.
- void CLOCK_SetClkDiv (clock_div_name_t div_name, uint32_t divided_by_value, bool reset)
  Setup peripheral clock dividers.
- void CLOCK_SetFLASHAccessCyclesForFreq (uint32_t iFreq)
  Set the flash wait states for the input freuqency.
- uint32_t CLOCK_GetFreq (clock_name_t clockName)
  Return Frequency of selected clock.
- uint32_t CLOCK_GetFro12MFreq (void)
  Return Frequency of FRO 12MHz.
- uint32_t CLOCK_GetClockOutClkFreq (void)
  Return Frequency of ClockOut.
- uint32_t CLOCK_GetSpifiClkFreq (void)
  Return Frequency of Spifi Clock.
- uint32_t CLOCK_GetAdcClkFreq (void)
  Return Frequency of Adc Clock.
- uint32_t CLOCK_GetUsb0ClkFreq (void)
  Return Frequency of Usb0 Clock.
- uint32_t CLOCK_GetUsb1ClkFreq (void)
  Return Frequency of Usb1 Clock.
- uint32_t CLOCK_GetMclkClkFreq (void)
  Return Frequency of MClk Clock.
- uint32_t CLOCK_GetSctClkFreq (void)
  Return Frequency of SCTimer Clock.
- uint32_t CLOCK_GetSdioClkFreq (void)
  Return Frequency of SDIO Clock.
- uint32_t CLOCK_GetLcdClkFreq (void)
  Return Frequency of LCD Clock.
- uint32_t CLOCK_GetLcdClkIn (void)
  Return Frequency of LCD CLKIN Clock.
- uint32_t CLOCK_GetExtClkFreq (void)

Return Frequency of External Clock.
- uint32_t CLOCK_GetWdtOscFreq (void)
  Return Frequency of Watchdog Oscillator.
- uint32_t CLOCK_GetFroHfFreq (void)
  Return Frequency of High-Freq output of FRO.
- uint32_t CLOCK_GetPllOutFreq (void)
  Return Frequency of PLL.
- uint32_t CLOCK_GetUsbPllOutFreq (void)
  Return Frequency of USB PLL.
- uint32_t CLOCK_GetAudioPllOutFreq (void)
  Return Frequency of AUDIO PLL.
- uint32_t CLOCK_GetOsc32KFreq (void)
  Return Frequency of 32kHz osc.
- uint32_t CLOCK_GetCoreSysClkFreq (void)
  Return Frequency of Core System.
- uint32_t CLOCK_GetI2SMClkFreq (void)
  Return Frequency of I2S MCLK Clock.
- uint32_t CLOCK_GetFlexCommClkFreq (uint32_t id)
  Return Frequency of Flexcomm functional Clock.
- __STATIC_INLINE async_clock_src_t CLOCK_GetAsyncApbClkSrc (void)
  Return Asynchronous APB Clock source.
- uint32_t CLOCK_GetAsyncApbClkFreq (void)
  Return Frequency of Asynchronous APB Clock.
- uint32_t CLOCK_GetAudioPLLInClockRate (void)
  Return Audio PLL input clock rate.
- uint32_t CLOCK_GetSystemPLLInClockRate (void)
  Return System PLL input clock rate.
- uint32_t CLOCK_GetSystemPLLOutClockRate (bool recompute)
  Return System PLL output clock rate.
- uint32_t CLOCK_GetAudioPLLOutClockRate (bool recompute)
  Return System AUDIO PLL output clock rate.
- uint32_t CLOCK_GetUSbPLLOutClockRate (bool recompute)
  Return System USB PLL output clock rate.
- __STATIC_INLINE void CLOCK_SetBypassPLL (bool bypass)
  Enables and disables PLL bypass mode.
- __STATIC_INLINE bool CLOCK_IsSystemPLLLocked (void)
  Check if PLL is locked or not.
- __STATIC_INLINE bool CLOCK_IsUsbPLLLocked (void)
  Check if USB PLL is locked or not.
- __STATIC_INLINE bool CLOCK_IsAudioPLLLocked (void)
  Check if AUDIO PLL is locked or not.
- __STATIC_INLINE void CLOCK_Enable_SysOsc (bool enable)
  Enables and disables SYS OSC.
- void CLOCK_SetStoredPLLClockRate (uint32_t rate)
  Store the current PLL rate.
- void CLOCK_SetStoredAudioPLLClockRate (uint32_t rate)
  Store the current AUDIO PLL rate.
- uint32_t CLOCK_GetSystemPLLOutFromSetup (pll_setup_t *pSetup)
  Return System PLL output clock rate from setup structure.
- uint32_t CLOCK_GetAudioPLLOutFromSetup (pll_setup_t *pSetup)
  Return System AUDIO PLL output clock rate from setup structure.

- uint32_t CLOCK_GetUsbPLLOutFromSetup (const usb_pll_setup_t *pSetup)
  Return System USB PLL output clock rate from setup structure.
- pll_error_t CLOCK_SetupPLLData (pll_config_t *pControl, pll_setup_t *pSetup)
  Set PLL output based on the passed PLL setup data.
- pll_error_t CLOCK_SetupAudioPLLData (pll_config_t *pControl, pll_setup_t *pSetup)
  Set AUDIO PLL output based on the passed AUDIO PLL setup data.
- pll_error_t CLOCK_SetupSystemPLLPrec (pll_setup_t *pSetup, uint32_t flagcfg)
  Set PLL output from PLL setup structure (precise frequency)
- pll_error_t CLOCK_SetupAudioPLLPrec (pll_setup_t *pSetup, uint32_t flagcfg)
  Set AUDIO PLL output from AUDIOPLL setup structure (precise frequency)
- pll_error_t CLOCK_SetPLLFreq (const pll_setup_t *pSetup)
  Set PLL output from PLL setup structure (precise frequency)
- pll_error_t CLOCK_SetUsbPLLFreq (const usb_pll_setup_t *pSetup)
  Set USB PLL output from USB PLL setup structure (precise frequency)
- void CLOCK_SetupSystemPLLMult (uint32_t multiply_by, uint32_t input_freq)
  Set PLL output based on the multiplier and input frequency.
- static void CLOCK_DisableUsbDevicefs0Clock (clock_ip_name_t clk)
  Disable USB clock.
- bool CLOCK_EnableUsbfs0DeviceClock (clock_usb_src_t src, uint32_t freq)
  Enable USB Device FS clock.
- bool CLOCK_EnableUsbfs0HostClock (clock_usb_src_t src, uint32_t freq)
  Enable USB HOST FS clock.
- bool CLOCK_EnableUsbhs0DeviceClock (clock_usb_src_t src, uint32_t freq)
  Enable USB Device HS clock.
- bool CLOCK_EnableUsbhs0HostClock (clock_usb_src_t src, uint32_t freq)
  Enable USB HOST HS clock.

37.2.4 Data Structure Documentation

37.2.4.1 struct pll_config_t

This structure can be used to configure the settings for a PLL setup structure. Fill in the desired configuration for the PLL and call the PLL setup function to fill in a PLL setup structure.

Data Fields

- uint32_t desiredRate
  Desired PLL rate in Hz.
- uint32_t inputRate
  PLL input clock in Hz, only used if PLL_CONFIGFLAG_USEINRATE flag is set.
- uint32_t flags
  PLL configuration flags, Or'ed value of PLL_CONFIGFLAG_* definitions.
- ss_progmodfm_t ss_mf
  SS Programmable modulation frequency, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag.
- ss_progmoddp_t ss_mr
  SS Programmable frequency modulation depth, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag.

- ss_modwvctrl_t ss_mc
  SS Modulation waveform control, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag.
- bool mfDither
  false for fixed modulation frequency or true for dithering, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

37.2.4.2 struct pll_setup_t

It can be populated with the PLL setup function. If powering up or waiting for PLL lock, the PLL input clock source should be configured prior to PLL setup.

Data Fields

- uint32_t syspllctrl
  PLL control register SYSPLLCTRL.
- uint32_t syspllndec
  PLL NDEC register SYSPLLNDEC.
- uint32_t syspllpdec
  PLL PDEC register SYSPLLPDEC.
- uint32_t syspllmdec
  PLL MDEC registers SYSPLLPDEC.
- uint32_t pllRate
  Acutal PLL rate.
- uint32_t flags
  PLL setup flags, Or'ed value of PLL_SETUPFLAG_∗ definitions.

37.2.4.3 struct usb_pll_setup_t

It can be populated with the USB PLL setup function. If powering up or waiting for USB PLL lock, the PLL input clock source should be configured prior to USB PLL setup.

Data Fields

- uint8_t msel
  USB PLL control register msel:1U-256U.
- uint8_t psel
  USB PLL control register psel:only support inter 1U 2U 4U 8U.
- uint8_t nsel
  USB PLL control register nsel:only suppoet inter 1U 2U 3U 4U.
- bool direct
  USB PLL CCO output control.
- bool bypass
  USB PLL inout clock bypass control.
- bool fbsel
  USB PLL ineter mode and non-integer mode control.
- uint32_t inputRate

USB PLL input rate.

37.2.5 Macro Definition Documentation

37.2.5.1 #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could contol the clock out of the driver.

Note

All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

37.2.5.2 #define ADC_CLOCKS

Value:

```
{               \
        kCLOCK_Adc0 \
    }
```

37.2.5.3 #define ROM_CLOCKS

Value:

```
{               \
        kCLOCK_Rom \
    }
```

37.2.5.4 #define SRAM_CLOCKS

Value:

```
{                \
        kCLOCK_Sram1, kCLOCK_Sram2, kCLOCK_Sram3 \
    }
```

37.2.5.5 #define FLASH_CLOCKS

Value:

```
{                \
        kCLOCK_Flash \
    }
```

37.2.5.6 #define FMC_CLOCKS

Value:

```
{              \
        kCLOCK_Fmc \
    }
```

37.2.5.7 #define EEPROM_CLOCKS

Value:

```
{                  \
        kCLOCK_Eeprom  \
    }
```

37.2.5.8 #define SPIFI_CLOCKS

Value:

```
{                  \
        kCLOCK_Spifi  \
    }
```

37.2.5.9 #define INPUTMUX_CLOCKS

Value:

```
{                       \
        kCLOCK_InputMux      \
    }
```

37.2.5.10 #define IOCON_CLOCKS

Value:

```
{                       \
        kCLOCK_Iocon          \
    }
```

37.2.5.11 #define GPIO_CLOCKS

Value:

```
{                       \
        kCLOCK_Gpio0,kLOCK_Gpio1, kCLOCK_Gpio2, kCLOCK_Gpio3, kCLOCK_Gpio4, kCLOCK_Gpio5     \
    }
```

37.2.5.12 #define PINT_CLOCKS

Value:

```
{                       \
        kCLOCK_Pint           \
    }
```

37.2.5.13 #define GINT_CLOCKS

Value:

```
{                       \
        kCLOCK_Gint, kCLOCK_Gint          \
    }
```

37.2.5.14 #define DMA_CLOCKS

Value:

```
{                       \
        kCLOCK_Dma           \
    }
```

37.2.5.15 #define CRC_CLOCKS

Value:

```
{                       \
        kCLOCK_Crc          \
    }
```

37.2.5.16 #define WWDT_CLOCKS

Value:

```
{                       \
        kCLOCK_Wwdt         \
    }
```

37.2.5.17 #define RTC_CLOCKS

Value:

```
{                       \
        kCLOCK_Rtc          \
    }
```

37.2.5.18 #define ADC0_CLOCKS

Value:

```
{                       \
        kCLOCK_Adc0         \
    }
```

37.2.5.19 #define MRT_CLOCKS

Value:

```
{                       \
        kCLOCK_Mrt          \
    }
```

37.2.5.20 #define RIT_CLOCKS

Value:

```
{                       \
        kCLOCK_Rit          \
    }
```

37.2.5.21 #define SCT_CLOCKS

Value:

```
{                       \
        kCLOCK_Sct0         \
    }
```

37.2.5.22 #define MCAN_CLOCKS

Value:

```
{                       \
        kCLOCK_Mcan0, kCLOCK_Mcan1          \
    }
```

37.2.5.23 #define UTICK_CLOCKS

Value:

```
{                       \
        kCLOCK_Utick        \
    }
```

37.2.5.24 #define FLEXCOMM_CLOCKS

Value:

```
{                                                                       \
        kCLOCK_FlexComm0, kCLOCK_FlexComm1, kCLOCK_FlexComm2, kCLOCK_FlexComm3, \
                    kCLOCK_FlexComm4, kCLOCK_FlexComm5, kCLOCK_FlexComm6, kCLOCK_FlexComm7, \
                                      kCLOCK_FlexComm8, kCLOCK_FlexComm9 \
    }
```

37.2.5.25 #define LPUART_CLOCKS

Value:

```
{                                                                                                   \
        kCLOCK_MinUart0, kCLOCK_MinUart1, kCLOCK_MinUart2, kCLOCK_MinUart3, kCLOCK_MinUart4,
      kCLOCK_MinUart5, \
            kCLOCK_MinUart6, kCLOCK_MinUart7, kCLOCK_MinUart8,kCLOCK_MinUart9     \
    }
```

37.2.5.26 #define BI2C_CLOCKS

Value:

```
{
                \
        kCLOCK_BI2c0, kCLOCK_BI2c1, kCLOCK_BI2c2, kCLOCK_BI2c3, kCLOCK_BI2c4, kCLOCK_BI2c5, kCLOCK_BI2c6,
      kCLOCK_BI2c7, \
        kCLOCK_BI2c8, kCLOCK_BI2c9  \
    }
```

37.2.5.27 #define LPSI_CLOCKS

Value:

```
{
                \
        kCLOCK_LSpi0, kCLOCK_LSpi1, kCLOCK_LSpi2, kCLOCK_LSpi3, kCLOCK_LSpi4, kCLOCK_LSpi5, kCLOCK_LSpi6,
      kCLOCK_LSpi7, \
        kCLOCK_LSpi8, kCLOCK_LSpi9  \
    }
```

37.2.5.28 #define FLEXI2S_CLOCKS

Value:

```
{                                                                                                   \
        kCLOCK_FlexI2s0, kCLOCK_FlexI2s1, kCLOCK_FlexI2s2, kCLOCK_FlexI2s3, kCLOCK_FlexI2s4,
      kCLOCK_FlexI2s5, \
            kCLOCK_FlexI2s6, kCLOCK_FlexI2s7, kCLOCK_FlexI2s8, kCLOCK_FlexI2s9
                                            \
    }
```

37.2.5.29 #define DMIC_CLOCKS

Value:

```
{               \
        kCLOCK_DMic \
    }
```

37.2.5.30 #define CTIMER_CLOCKS

Value:

```
{                                                                                \
        kCLOCK_Ct32b0, kCLOCK_Ct32b1, kCLOCK_Ct32b2, kCLOCK_Ct32b3, kCLOCK_Ct32b4   \
    }
```

37.2.5.31 #define LCD_CLOCKS

Value:

```
{               \
        kCLOCK_Lcd  \
    }
```

37.2.5.32 #define SDIO_CLOCKS

Value:

```
{                \
        kCLOCK_Sdio  \
    }
```

37.2.5.33 #define USBRAM_CLOCKS

Value:

```
{                    \
        kCLOCK_UsbRam1   \
    }
```

37.2.5.34 #define EMC_CLOCKS

Value:

```
{                    \
        kCLOCK_Emc       \
    }
```

37.2.5.35 #define ETH_CLOCKS

Value:

```
{                   \
        kCLOCK_Eth      \
    }
```

37.2.5.36 #define AES_CLOCKS

Value:

```
{                   \
        kCLOCK_Aes      \
    }
```

37.2.5.37 #define OTP_CLOCKS

Value:

```
{                   \
        kCLOCK_Otp      \
    }
```

37.2.5.38 #define RNG_CLOCKS

Value:

```
{                   \
        kCLOCK_Rng      \
    }
```

37.2.5.39 #define USBHMR0_CLOCKS

Value:

```
{                       \
        kCLOCK_Usbhmr0      \
    }
```

37.2.5.40 #define USBHSL0_CLOCKS

Value:

```
{                          \
        kCLOCK_Usbhs10        \
    }
```

37.2.5.41 #define SHA0_CLOCKS

Value:

```
{                       \
        kCLOCK_Sha0        \
    }
```

37.2.5.42 #define SMARTCARD_CLOCKS

Value:

```
{                       \
        kCLOCK_SmartCard0, kCLOCK_SmartCard1 \
    }
```

37.2.5.43 #define USBD_CLOCKS

Value:

```
{                  \
        kCLOCK_Usbd0, kCLOCK_Usbh1, kCLOCK_Usbd1 \
    }
```

37.2.5.44 #define USBH_CLOCKS

Value:

```
{                  \
        kCLOCK_Usbh1 \
    }
```

37.2.5.45 #define CLK_GATE_REG_OFFSET_SHIFT 8U

37.2.5.46 #define MUX_A(*m, choice*) (((m) << 0) | ((choice + 1) << 8))

[4 bits for choice, where 1 is A, 2 is B, 3 is C and 4 is D, 0 means end of descriptor] [8 bits mux ID]*

37.2.5.47 #define PLL_CONFIGFLAG_USEINRATE (1 << 0)

When the PLL_CONFIGFLAG_USEINRATE flag is selected, the 'InputRate' field in the configuration structure must be assigned with the expected PLL frequency. If the PLL_CONFIGFLAG_USEINRATE is not used, 'InputRate' is ignored in the configuration function and the driver will determine the PLL rate from the currently selected PLL source. This flag might be used to configure the PLL input clock more accurately when using the WDT oscillator or a more dyanmic CLKIN source.

When the PLL_CONFIGFLAG_FORCENOFRACT flag is selected, the PLL hardware for the automatic bandwidth selection, Spread Spectrum (SS) support, and fractional M-divider are not used.

Flag to use InputRate in PLL configuration structure for setup

37.2.5.48 #define PLL_SETUPFLAG_POWERUP (1 << 0)

Setup will power on the PLL after setup

37.2.6 Enumeration Type Documentation

37.2.6.1 enum clock_ip_name_t

37.2.6.2 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock (aka MAIN_CLK)
kCLOCK_BusClk Bus clock (AHB clock)
kCLOCK_ClockOut CLOCKOUT.
kCLOCK_FroHf FRO48/96.
kCLOCK_SpiFi SPIFI.
kCLOCK_Adc ADC.
kCLOCK_Usb0 USB0.
kCLOCK_Usb1 USB1.
kCLOCK_UsbPll USB1 PLL.
kCLOCK_Mclk MCLK.
kCLOCK_Sct SCT.
kCLOCK_SDio SDIO.
kCLOCK_EMC EMC.
kCLOCK_LCD LCD.

kCLOCK_MCAN0 MCAN0.
kCLOCK_MCAN1 MCAN1.
kCLOCK_Fro12M FRO12M.
kCLOCK_ExtClk External Clock.
kCLOCK_PllOut PLL Output.
kCLOCK_UsbClk USB input.
kClock_WdtOsc Watchdog Oscillator.
kCLOCK_Frg Frg Clock.
kCLOCK_Dmic Digital Mic clock.
kCLOCK_AsyncApbClk Async APB clock.
kCLOCK_FlexI2S FlexI2S clock.
kCLOCK_Flexcomm0 Flexcomm0Clock.
kCLOCK_Flexcomm1 Flexcomm1Clock.
kCLOCK_Flexcomm2 Flexcomm2Clock.
kCLOCK_Flexcomm3 Flexcomm3Clock.
kCLOCK_Flexcomm4 Flexcomm4Clock.
kCLOCK_Flexcomm5 Flexcomm5Clock.
kCLOCK_Flexcomm6 Flexcomm6Clock.
kCLOCK_Flexcomm7 Flexcomm7Clock.
kCLOCK_Flexcomm8 Flexcomm8Clock.
kCLOCK_Flexcomm9 Flexcomm9Clock.

37.2.6.3 enum async_clock_src_t

Clock source selections for the asynchronous APB clock

Enumerator

kCLOCK_AsyncMainClk Main System clock.
kCLOCK_AsyncFro12Mhz 12MHz FRO

37.2.6.4 enum clock_flashtim_t

Enumerator

kCLOCK_Flash1Cycle Flash accesses use 1 CPU clocks.
kCLOCK_Flash2Cycle Flash accesses use 2 CPU clocks.
kCLOCK_Flash3Cycle Flash accesses use 3 CPU clocks.
kCLOCK_Flash4Cycle Flash accesses use 4 CPU clocks.
kCLOCK_Flash5Cycle Flash accesses use 5 CPU clocks.
kCLOCK_Flash6Cycle Flash accesses use 6 CPU clocks.
kCLOCK_Flash7Cycle Flash accesses use 7 CPU clocks.
kCLOCK_Flash8Cycle Flash accesses use 8 CPU clocks.

37.2.6.5 enum ss_progmodfm_t

Enumerator

kSS_MF_512 Nss = 512 (fm ? 3.9 - 7.8 kHz)
kSS_MF_384 Nss ?= 384 (fm ? 5.2 - 10.4 kHz)
kSS_MF_256 Nss = 256 (fm ? 7.8 - 15.6 kHz)
kSS_MF_128 Nss = 128 (fm ? 15.6 - 31.3 kHz)
kSS_MF_64 Nss = 64 (fm ? 32.3 - 64.5 kHz)
kSS_MF_32 Nss = 32 (fm ? 62.5- 125 kHz)
kSS_MF_24 Nss ?= 24 (fm ? 83.3- 166.6 kHz)
kSS_MF_16 Nss = 16 (fm ? 125- 250 kHz)

37.2.6.6 enum ss_progmoddp_t

Enumerator

kSS_MR_K0 k = 0 (no spread spectrum)
kSS_MR_K1 k = 1
kSS_MR_K1_5 k = 1.5
kSS_MR_K2 k = 2
kSS_MR_K3 k = 3
kSS_MR_K4 k = 4
kSS_MR_K6 k = 6
kSS_MR_K8 k = 8

37.2.6.7 enum ss_modwvctrl_t

Compensation for low pass filtering of the PLL to get a triangular modulation at the output of the PLL, giving a flat frequency spectrum.

Enumerator

kSS_MC_NOC no compensation
kSS_MC_RECC recommended setting
kSS_MC_MAXC max. compensation

37.2.6.8 enum pll_error_t

Enumerator

kStatus_PLL_Success PLL operation was successful.
kStatus_PLL_OutputTooLow PLL output rate request was too low.
kStatus_PLL_OutputTooHigh PLL output rate request was too high.

kStatus_PLL_InputTooLow PLL input rate is too low.
kStatus_PLL_InputTooHigh PLL input rate is too high.
kStatus_PLL_OutsideIntLimit Requested output rate isn't possible.
kStatus_PLL_CCOTooLow Requested CCO rate isn't possible.
kStatus_PLL_CCOTooHigh Requested CCO rate isn't possible.

37.2.6.9 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcFro Use FRO 96 or 48 MHz.
kCLOCK_UsbSrcSystemPll Use System PLL output.
kCLOCK_UsbSrcMainClock Use Main clock.
kCLOCK_UsbSrcUsbPll Use USB PLL clock.
kCLOCK_UsbSrcNone Use None, this may be selected in order to reduce power when no output is needed.

37.2.6.10 enum usb_pll_psel

37.2.7 Function Documentation

37.2.7.1 static void CLOCK_SetFLASHAccessCycles (clock_flashtim_t *clks*) [inline], [static]

Parameters

clks	: Clock cycles for FLASH access

Returns

Nothing

37.2.7.2 status_t CLOCK_SetupFROClocking (uint32_t *iFreq*)

Parameters

iFreq	: Desired frequency (must be one of #CLK_FRO_12MHZ or #CLK_FRO_48MHZ or #CLK_FRO_96MHZ)

Returns

returns success or fail status.

37.2.7.3 void CLOCK_AttachClk (clock_attach_id_t *connection*)

Parameters

connection	: Clock to be configured.

Returns

Nothing

37.2.7.4 void CLOCK_SetClkDiv (clock_div_name_t *div_name,* uint32_t *divided_by_value,* bool *reset*)

Parameters

div_name	: Clock divider name
divided_by_value,:	Value to be divided
reset	: Whether to reset the divider counter.

Returns

Nothing

37.2.7.5 void CLOCK_SetFLASHAccessCyclesForFreq (uint32_t *iFreq*)

Parameters

iFreq	: Input frequency

Returns

Nothing

37.2.7.6 uint32_t CLOCK_GetFreq (clock_name_t *clockName*)

Returns

Frequency of selected clock

37.2.7.7 uint32_t CLOCK_GetFro12MFreq (void)

Returns

Frequency of FRO 12MHz

37.2.7.8 uint32_t CLOCK_GetClockOutClkFreq (void)

Returns

Frequency of ClockOut

37.2.7.9 uint32_t CLOCK_GetSpifiClkFreq (void)

Returns

Frequency of Spifi.

37.2.7.10 uint32_t CLOCK_GetAdcClkFreq (void)

Returns

Frequency of Adc Clock.

37.2.7.11 uint32_t CLOCK_GetUsb0ClkFreq (void)

Returns

Frequency of Usb0 Clock.

37.2.7.12 uint32_t CLOCK_GetUsb1ClkFreq (void)

Returns

Frequency of Usb1 Clock.

37.2.7.13 uint32_t CLOCK_GetMclkClkFreq (void)

Returns

Frequency of MClk Clock.

37.2.7.14 uint32_t CLOCK_GetSctClkFreq (void)

Returns

Frequency of SCTimer Clock.

37.2.7.15 uint32_t CLOCK_GetSdioClkFreq (void)

Returns

Frequency of SDIO Clock.

37.2.7.16 uint32_t CLOCK_GetLcdClkFreq (void)

Returns

Frequency of LCD Clock.

37.2.7.17 uint32_t CLOCK_GetLcdClkIn (void)

Returns

Frequency of LCD CLKIN Clock.

37.2.7.18 uint32_t CLOCK_GetExtClkFreq (void)

Returns

Frequency of External Clock. If no external clock is used returns 0.

37.2.7.19 uint32_t CLOCK_GetWdtOscFreq (void)

Returns

Frequency of Watchdog Oscillator

37.2.7.20 uint32_t CLOCK_GetFroHfFreq (void)

Returns

Frequency of High-Freq output of FRO

37.2.7.21 uint32_t CLOCK_GetPllOutFreq (void)

Returns

Frequency of PLL

37.2.7.22 uint32_t CLOCK_GetUsbPllOutFreq (void)

Returns

Frequency of PLL

37.2.7.23 uint32_t CLOCK_GetAudioPllOutFreq (void)

Returns

Frequency of PLL

37.2.7.24 uint32_t CLOCK_GetOsc32KFreq (void)

Returns

Frequency of 32kHz osc

37.2.7.25 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Frequency of Core System

37.2.7.26 uint32_t CLOCK_GetI2SMClkFreq (void)

Returns

Frequency of I2S MCLK Clock

37.2.7.27 uint32_t CLOCK_GetFlexCommClkFreq (uint32_t *id*)

Returns

Frequency of Flexcomm functional Clock

37.2.7.28 __STATIC_INLINE async_clock_src_t CLOCK_GetAsyncApbClkSrc (void)

Returns

Asynchronous APB CLock source

37.2.7.29 uint32_t CLOCK_GetAsyncApbClkFreq (void)

Returns

Frequency of Asynchronous APB Clock Clock

37.2.7.30 uint32_t CLOCK_GetAudioPLLInClockRate (void)

Returns

Audio PLL input clock rate

37.2.7.31 uint32_t CLOCK_GetSystemPLLInClockRate (void)

Returns

System PLL input clock rate

37.2.7.32 uint32_t CLOCK_GetSystemPLLOutClockRate (bool *recompute*)

Parameters

recompute	: Forces a PLL rate recomputation if true

Returns

System PLL output clock rate

Note

The PLL rate is cached in the driver in a variable as the rate computation function can take some time to perform. It is recommended to use 'false' with the 'recompute' parameter.

37.2.7.33 uint32_t CLOCK_GetAudioPLLOutClockRate (bool *recompute*)

Parameters

recompute	: Forces a AUDIO PLL rate recomputation if true

Returns

System AUDIO PLL output clock rate

Note

The AUDIO PLL rate is cached in the driver in a variable as the rate computation function can take some time to perform. It is recommended to use 'false' with the 'recompute' parameter.

37.2.7.34 uint32_t CLOCK_GetUSbPLLOutClockRate (bool *recompute*)

Parameters

recompute	: Forces a USB PLL rate recomputation if true

Returns

System USB PLL output clock rate

Note

The USB PLL rate is cached in the driver in a variable as the rate computation function can take some time to perform. It is recommended to use 'false' with the 'recompute' parameter.

37.2.7.35 __STATIC_INLINE void CLOCK_SetBypassPLL (bool *bypass*)

bypass : true to bypass PLL (PLL output = PLL input, false to disable bypass

Returns

System PLL output clock rate

37.2.7.36 __STATIC_INLINE bool CLOCK_IsSystemPLLLocked (void)

Returns

true if the PLL is locked, false if not locked

37.2.7.37 __STATIC_INLINE bool CLOCK_IsUsbPLLLocked (void)

Returns

true if the USB PLL is locked, false if not locked

37.2.7.38 __STATIC_INLINE bool CLOCK_IsAudioPLLLocked (void)

Returns

true if the AUDIO PLL is locked, false if not locked

37.2.7.39 __STATIC_INLINE void CLOCK_Enable_SysOsc (bool *enable*)

enable : true to enable SYS OSC, false to disable SYS OSC

37.2.7.40 void CLOCK_SetStoredPLLClockRate (uint32_t *rate*)

Parameters

rate,:	Current rate of the PLL

Returns

Nothing

37.2.7.41 void CLOCK_SetStoredAudioPLLClockRate (uint32_t *rate*)

Parameters

rate,:	Current rate of the PLL

Returns

Nothing

37.2.7.42 uint32_t CLOCK_GetSystemPLLOutFromSetup (pll_setup_t * *pSetup*)

Parameters

pSetup	: Pointer to a PLL setup structure

Returns

System PLL output clock rate the setup structure will generate

37.2.7.43 uint32_t CLOCK_GetAudioPLLOutFromSetup (pll_setup_t * *pSetup*)

Parameters

pSetup	: Pointer to a PLL setup structure

Returns

System PLL output clock rate the setup structure will generate

37.2.7.44 uint32_t CLOCK_GetUsbPLLOutFromSetup (const usb_pll_setup_t * *pSetup*)

Parameters

pSetup	: Pointer to a PLL setup structure

Returns

System PLL output clock rate the setup structure will generate

37.2.7.45 pll_error_t CLOCK_SetupPLLData (pll_config_t * *pControl,* pll_setup_t * *pSetup*)

Parameters

pControl	: Pointer to populated PLL control structure to generate setup with
pSetup	: Pointer to PLL setup structure to be filled

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

Note

Actual frequency for setup may vary from the desired frequency based on the accuracy of input clocks, rounding, non-fractional PLL mode, etc.

37.2.7.46 pll_error_t CLOCK_SetupAudioPLLData (pll_config_t * *pControl,* pll_setup_t * *pSetup*)

Parameters

pControl	: Pointer to populated PLL control structure to generate setup with
pSetup	: Pointer to PLL setup structure to be filled

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

Note

Actual frequency for setup may vary from the desired frequency based on the accuracy of input clocks, rounding, non-fractional PLL mode, etc.

37.2.7.47 pll_error_t CLOCK_SetupSystemPLLPrec (pll_setup_t * *pSetup,* uint32_t *flagcfg*)

Parameters

pSetup	: Pointer to populated PLL setup structure
flagcfg	: Flag configuration for PLL config structure

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

Note

This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main systen clock) that may use the PLL, so these should be setup prior to and after exiting the function.

37.2.7.48 pll_error_t CLOCK_SetupAudioPLLPrec (pll_setup_t * *pSetup,* uint32_t *flagcfg*)

Parameters

pSetup	: Pointer to populated PLL setup structure
flagcfg	: Flag configuration for PLL config structure

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

Note

This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the AUDIO PLL, wait for PLL lock, and adjust system voltages to the new AUDIOPLL rate. The function will not alter any source clocks (ie, main systen clock) that may use the AUDIO PLL, so these should be setup prior to and after exiting the function.

37.2.7.49 pll_error_t CLOCK_SetPLLFreq (const pll_setup_t * *pSetup*)

Parameters

pSetup	: Pointer to populated PLL setup structure

Returns

kStatus_PLL_Success on success, or PLL setup error code

Note

This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main systen clock) that may use the PLL, so these should be setup prior to and after exiting the function.

37.2.7.50 pll_error_t CLOCK_SetUsbPLLFreq (const usb_pll_setup_t * *pSetup*)

Parameters

pSetup	: Pointer to populated USB PLL setup structure

Returns

kStatus_PLL_Success on success, or USB PLL setup error code

Note

This function will power off the USB PLL, setup the PLL with the new setup data, and then optionally powerup the USB PLL, wait for USB PLL lock, and adjust system voltages to the new USB PLL rate. The function will not alter any source clocks (ie, usb pll clock) that may use the USB PLL, so these should be setup prior to and after exiting the function.

37.2.7.51 void CLOCK_SetupSystemPLLMult (uint32_t *multiply_by,* uint32_t *input_freq*)

Parameters

multiply_by	: multiplier
input_freq	: Clock input frequency of the PLL

Returns

Nothing

Note

Unlike the Chip_Clock_SetupSystemPLLPrec() function, this function does not disable or enable PLL power, wait for PLL lock, or adjust system voltages. These must be done in the application. The function will not alter any source clocks (ie, main systen clock) that may use the PLL, so these should be setup prior to and after exiting the function.

37.2.7.52 static void CLOCK_DisableUsbDevicefs0Clock (clock_ip_name_t *clk*) [inline], [static]

Disable USB clock.

37.2.7.53 bool CLOCK_EnableUsbfs0DeviceClock (clock_usb_src_t *src,* uint32_t *freq*)

Parameters

src	: clock source
freq,:	clock frequency Enable USB Device Full Speed clock.

37.2.7.54 bool CLOCK_EnableUsbfs0HostClock (clock_usb_src_t *src,* uint32_t *freq*)

Parameters

src	: clock source
freq,:	clock frequency Enable USB HOST Full Speed clock.

37.2.7.55 bool CLOCK_EnableUsbhs0DeviceClock (clock_usb_src_t *src,* uint32_t *freq*)

Parameters

src	: clock source
freq,:	clock frequency Enable USB Device High Speed clock.

37.2.7.56 bool CLOCK_EnableUsbhs0HostClock (clock_usb_src_t *src,* uint32_t *freq*)

Parameters

src	: clock source
freq,:	clock frequency Enable USB HOST High Speed clock.

Chapter 38
UTICK: MictoTick Timer Driver

38.1 Overview

The SDK provides Peripheral driver for the UTICK module of LPC devices.

UTICK driver is created to help user to operate the UTICK module. The UTICK timer can be used as a low power timer. The APIs can be used to enable the UTICK module, initialize it and set the time. UTICK can be used as a wake up source from low power mode.

38.2 Typical use case

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

 /* Running FRO = 12 MHz*/
 BOARD_BootClockVLPR();

 /* Power up Watchdog oscillator*/
 POWER_DisablePD(kPDRUNCFG_PD_WDT_OSC);

 /* Intiialize UTICK */
 UTICK_Init(UTICK0);

 /* Set the UTICK timer to wake up the device from reduced power mode */
 UTICK_SetTick(UTICK0, kUTICK_Repeat, UTICK_TIME, NULL);
 while (1)
 {
 }
```

Files

- file fsl_utick.h

Typedefs

- typedef void(∗ utick_callback_t)(void)
  UTICK callback function.

Enumerations

- enum utick_mode_t {
  kUTICK_Onetime = 0x0U,
  kUTICK_Repeat = 0x1U }
  UTICK timer operational mode.

Driver version

- #define FSL_UTICK_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  UTICK driver version 2.0.0.

Initialization and deinitialization

- void UTICK_Init (UTICK_Type *base)
  Initializes an UTICK by turning its bus clock on.
- void UTICK_Deinit (UTICK_Type *base)
  Deinitializes a UTICK instance.
- uint32_t UTICK_GetStatusFlags (UTICK_Type *base)
  Get Status Flags.
- void UTICK_ClearStatusFlags (UTICK_Type *base)
  Clear Status Interrupt Flags.
- void UTICK_SetTick (UTICK_Type *base, utick_mode_t mode, uint32_t count, utick_callback_t cb)
  Starts UTICK.
- void UTICK_HandleIRQ (UTICK_Type *base, utick_callback_t cb)
  UTICK Interrupt Service Handler.

38.3 Macro Definition Documentation

38.3.1 #define FSL_UTICK_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

38.4 Typedef Documentation

38.4.1 typedef void(* utick_callback_t)(void)

38.5 Enumeration Type Documentation

38.5.1 enum utick_mode_t

Enumerator

kUTICK_Onetime Trigger once.
kUTICK_Repeat Trigger repeatedly.

38.6 Function Documentation

38.6.1 void UTICK_Init (UTICK_Type * *base*)

38.6.2 void UTICK_Deinit (UTICK_Type * *base*)

This function shuts down Utick bus clock

Parameters

base	UTICK peripheral base address.

38.6.3 uint32_t UTICK_GetStatusFlags (UTICK_Type * *base*)

This returns the status flag

Parameters

base	UTICK peripheral base address.

Returns

status register value

38.6.4 void UTICK_ClearStatusFlags (UTICK_Type * *base*)

This clears intr status flag

Parameters

base	UTICK peripheral base address.

Returns

none

38.6.5 void UTICK_SetTick (UTICK_Type * *base*, utick_mode_t *mode*, uint32_t *count*, utick_callback_t *cb*)

This function starts a repeat/onetime countdown with an optional callback

Parameters

base	UTICK peripheral base address.

mode	UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
count	UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
cb	UTICK callback (can be left as NULL if none, otherwise should be a void func(void))

Returns

none

38.6.6 void UTICK_HandleIRQ (UTICK_Type * *base*, utick_callback_t *cb*)

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in UTICK_SetTick()). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

base	UTICK peripheral base address.
cb	callback scheduled for this instance of UTICK

Returns

none

Chapter 39
WWDT: Windowed Watchdog Timer Driver

39.1 Overview

The SDK provides a peripheral driver for the Watchdog module (WDOG) of LPC devices.

39.2 Function groups

39.2.1 Initialization and deinitialization

The function WWDT_Init() initializes the watchdog timer with specified configurations. The configurations include timeout value and whether to enable watchdog after iniy. The function WWDT_GetDefaultConfig() gets the default configurations.

The function WWDT_Deinit() disables the watchdog and the module clock.

39.2.2 Status

Provides functions to get and clear the WWDT status.

39.2.3 Interrupt

Provides functions to enable/disable WWDT interrupts and get current enabled interrupts.

39.2.4 Watch dog Refresh

The function WWDT_Refresh() feeds the WWDT.

39.3 Typical use case

```
int main(void)
{
    wwdt_config_t config;
    uint32_t wdtFreq;

    /* Init hardware*/
    BOARD_InitHardware();

    /* Set Red LED to initially be high */
    LED_RED_INIT(1);

    POWER_DisablePD(kPDRUNCFG_PD_WDT_OSC);
```

Typical use case

```
    /* The WDT divides the input frequency into it by 4 */
    wdtFreq = CLOCK_GetFreq(kClock_WdtOsc) / 4;

    NVIC_EnableIRQ(WDT_BOD_IRQn);

    WWDT_GetDefaultConfig(&config);

    /* Check if reset is due to Watchdog */
    if (WWDT_GetStatusFlags(WWDT) & kWWDT_TimeoutFlag) {
        LED_RED_ON();
        PRINTF("Watchdog reset occurred\r\n");
    }

    /*
     * Set watchdog feed time constant to approximately 2s
     * Set watchdog warning time to 512 ticks after feed time constant
     * Set watchdog window time to 1s
     */
    config.timeoutValue = wdtFreq * 2;
    config.warningValue = 512;
    config.windowValue = wdtFreq * 1;
    /* Configure WWDT to reset on timeout */
    config.enableWatchdogReset = true;

    /* wdog refresh test in window mode */
    PRINTF("\r\n--- Window mode refresh test start---\r\n");
    WWDT_Init(WWDT, &config);

    /* First feed will start the watchdog */
    WWDT_Refresh(WWDT);

    while (1)
    {
    }
}
```

Files

- file fsl_wwdt.h

Data Structures

- struct wwdt_config_t
  Describes WWDT configuration structure. More...

Enumerations

- enum _wwdt_status_flags_t {
  kWWDT_TimeoutFlag = WWDT_MOD_WDTOF_MASK,
  kWWDT_WarningFlag = WWDT_MOD_WDINT_MASK }
  WWDT status flags.

Driver version

- #define FSL_WWDT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))
  Defines WWDT driver version 2.0.0.

Refresh sequence

- #define WWDT_FIRST_WORD_OF_REFRESH (0xAAU)
  First word of refresh sequence.
- #define WWDT_SECOND_WORD_OF_REFRESH (0x55U)
  Second word of refresh sequence.

WWDT Initialization and De-initialization

- void WWDT_GetDefaultConfig (wwdt_config_t *config)
  Initializes WWDT configure sturcture.
- void WWDT_Init (WWDT_Type *base, const wwdt_config_t *config)
  Initializes the WWDT.
- void WWDT_Deinit (WWDT_Type *base)
  Shuts down the WWDT.

WWDT Functional Operation

- static void WWDT_Enable (WWDT_Type *base)
  Enables the WWDT module.
- static void WWDT_Disable (WWDT_Type *base)
  Disables the WWDT module.
- static uint32_t WWDT_GetStatusFlags (WWDT_Type *base)
  Gets all WWDT status flags.
- void WWDT_ClearStatusFlags (WWDT_Type *base, uint32_t mask)
  Clear WWDT flag.
- static void WWDT_SetWarningValue (WWDT_Type *base, uint32_t warningValue)
  Set the WWDT warning value.
- static void WWDT_SetTimeoutValue (WWDT_Type *base, uint32_t timeoutCount)
  Set the WWDT timeout value.
- static void WWDT_SetWindowValue (WWDT_Type *base, uint32_t windowValue)
  Sets the WWDT window value.
- void WWDT_Refresh (WWDT_Type *base)
  Refreshes the WWDT timer.

39.4 Data Structure Documentation

39.4.1 struct wwdt_config_t

Data Fields

- bool enableWwdt
  Enables or disables WWDT.
- bool enableWatchdogReset
  true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset
- bool enableWatchdogProtect
  true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time
- bool enableLockOscillator

true: Disabling or powering down the watchdog oscillator is prevented Once set, this bit can only be cleared by a reset false: Do not lock oscillator

- uint32_t windowValue
  Window value, set this to 0xFFFFFF if windowing is not in effect.
- uint32_t timeoutValue
  Timeout value.
- uint32_t warningValue
  Watchdog time counter value that will generate a warning interrupt.

39.4.1.0.0.55 Field Documentation

39.4.1.0.0.55.1 uint32_t wwdt_config_t::warningValue

Set this to 0 for no warning

39.5 Macro Definition Documentation

39.5.1 #define FSL_WWDT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

39.6 Enumeration Type Documentation

39.6.1 enum _wwdt_status_flags_t

This structure contains the WWDT status flags for use in the WWDT functions.

Enumerator

kWWDT_TimeoutFlag Time-out flag, set when the timer times out.
kWWDT_WarningFlag Warning interrupt flag, set when timer is below the value WDWARNINT.

39.7 Function Documentation

39.7.1 void WWDT_GetDefaultConfig (wwdt_config_t * *config*)

This function initializes the WWDT configure structure to default value. The default value are:

```
*  config->enableWwdt = true;
*  config->enableWatchdogReset = false;
*  config->enableWatchdogProtect = false;
*  config->enableLockOscillator = false;
*  config->windowValue = 0xFFFFFFU;
*  config->timeoutValue = 0xFFFFFFU;
*  config->warningValue = 0;
*
```

Parameters

config	Pointer to WWDT config structure.

See Also

wwdt_config_t

39.7.2 void WWDT_Init (WWDT_Type * *base*, const wwdt_config_t * *config*)

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
*   wwdt_config_t config;
*   WWDT_GetDefaultConfig(&config);
*   config.timeoutValue = 0x7ffU;
*   WWDT_Init(wwdt_base,&config);
*
```

Parameters

base	WWDT peripheral base address
config	The configuration of WWDT

39.7.3 void WWDT_Deinit (WWDT_Type * *base*)

This function shuts down the WWDT.

Parameters

base	WWDT peripheral base address

39.7.4 static void WWDT_Enable (WWDT_Type * *base*) [inline], [static]

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

base	WWDT peripheral base address

39.7.5 static void WWDT_Disable (WWDT_Type * *base*) [inline], [static]

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

base	WWDT peripheral base address

39.7.6 static uint32_t WWDT_GetStatusFlags (WWDT_Type * *base*) [inline], [static]

This function gets all status flags.

Example for getting Timeout Flag:

```
*   uint32_t status;
*   status = WWDT_GetStatusFlags(wwdt_base) &
      kWWDT_TimeoutFlag;
*
```

Parameters

base	WWDT peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration _wwdt_status_flags_t

39.7.7 void WWDT_ClearStatusFlags (WWDT_Type * *base*, uint32_t *mask*)

This function clears WWDT status flag.

Example for clearing warning flag:

```
*   WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
*
```

Parameters

base	WWDT peripheral base address
mask	The status flags to clear. This is a logical OR of members of the enumeration _wwdt-_status_flags_t

39.7.8 static void WWDT_SetWarningValue (WWDT_Type * *base*, uint32_t *warningValue*) [inline], [static]

The WDWARNINT register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by WARNINT, an interrupt will be generated after the subsequent WDCLK.

Parameters

base	WWDT peripheral base address
warningValue	WWDT warning value.

39.7.9 static void WWDT_SetTimeoutValue (WWDT_Type * *base*, uint32_t *timeoutCount*) [inline], [static]

This function sets the timeout value. Every time a feed sequence occurs the value in the TC register is loaded into the Watchdog timer. Writing a value below 0xFF will cause 0xFF to be loaded into the TC register. Thus the minimum time-out interval is TWDCLK*256*4. If enableWatchdogProtect flag is true in wwdt_config_t config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the WDTOF flag.

Parameters

base	WWDT peripheral base address
timeoutCount	WWDT timeout value, count of WWDT clock tick.

39.7.10 static void WWDT_SetWindowValue (WWDT_Type * *base*, uint32_t *windowValue*) [inline], [static]

The WINDOW register determines the highest TV value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in WINDOW, a watchdog event will occur. To disable windowing, set windowValue to 0xFFFFFF (maximum possible timer value) so windowing is not in effect.

Parameters

base	WWDT peripheral base address
windowValue	WWDT window value.

39.7.11 void WWDT_Refresh (WWDT_Type * *base*)

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

base	WWDT peripheral base address

39.8 Fmc_driver

39.8.1 Overview

Data Structures

- struct fmc_flash_signature_t
  Defines the generated 128-bit signature. More...
- struct fmc_config_t
  fmc config structure. More...

Enumerations

- enum _fmc_flags { kFMC_SignatureGenerationDoneFlag = FMC_FMSTAT_SIG_DONE_MASK }
  fmc peripheral flag.

39.8.2 Data Structure Documentation

39.8.2.1 struct fmc_flash_signature_t

39.8.2.2 struct fmc_config_t

39.8.3 Enumeration Type Documentation

39.8.3.1 enum _fmc_flags

Enumerator

kFMC_SignatureGenerationDoneFlag Flash signature generation done.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

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

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