Texas Instruments

F28004x Peripheral Driver Library 1.04.00.00

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 1.04.00.00 of this document, last updated on Tue Jun 26 03:09:07 CDT 2018.

Table of Contents

Copyright . . . 1

Revision Information . . . 1

1 Introduction . . . 3

2 Revision History . . . 4

3 Programming Model . . . 6
3.1 Introduction . . . 6
3.2 Direct Register Access Model . . . 6
3.3 Software Driver Model . . . 7
3.4 Combining The Models . . . 7

4 Driver Library Usage . . . 9
4.1 Introduction . . . 9
4.2 Code Composer Studio Tips . . . 9
4.3 ASSERT Macro . . . 10
4.4 Driver Library Optimization . . . 11

5 ADC Module . . . 12
5.1 ADC Introduction . . . 12
5.2 API Functions . . . 12

6 ASysCtl Module . . . 37
6.1 ASysCtl Introduction . . . 37
6.2 API Functions . . . 37

7 CAN Module . . . 46
7.1 CAN Introduction . . . 46
7.2 API Functions . . . 46

8 CLA Module . . . 67
8.1 CLA Introduction . . . 67
8.2 API Functions . . . 67

9 CLAPROMCRC Module . . . 86
9.1 CLAPROMCRC Introduction . . . 86
9.2 API Functions . . . 86

10 CMPSS Module . . . 94
10.1 CMPSS Introduction . . . 94
10.2 API Functions . . . 94

11 CPU Timer . . . 110
11.1 CPU Timer Introduction . . . 110
11.2 API Functions . . . 110

12 DAC Module . . . 118
12.1 DAC Introduction . . . 118
12.2 API Functions . . . 118

13 DCC Module . . . 127
13.1 DCC Introduction . . . 127
13.2 API Functions . . . 127

14 DCSM Module . . . 140
14.1 DCSM Introduction . . . 140
14.2 API Functions . . . 140

15 DMA Module . . . **152**
15.1 DMA Introduction . . . 152
15.2 API Functions . . . 152
16 ECAP Module . . . **165**
16.1 ECAP Introduction . . . 165
16.2 API Functions . . . 165
17 HRCAP Module . . . **189**
17.1 HRCAP Introduction . . . 189
17.2 API Functions . . . 189
18 EPWM Module . . . **197**
18.1 EPWM Introduction . . . 197
18.2 API Functions . . . 197
19 HRPWM Module . . . **311**
19.1 HRPWM Introduction . . . 311
19.2 API Functions . . . 311
20 EQEP Module . . . **327**
20.1 EQEP Introduction . . . 327
20.2 API Functions . . . 327
21 Flash Module . . . **349**
21.1 Flash Introduction . . . 349
21.2 API Functions . . . 349
22 FSI Module . . . **371**
22.1 FSI Introduction . . . 371
22.2 API Functions . . . 371
23 GPIO Module . . . **412**
23.1 GPIO Introduction . . . 412
23.2 API Functions . . . 412
24 I2C Module . . . **431**
24.1 I2C Introduction . . . 431
24.2 API Functions . . . 431
25 Interrupt Module . . . **451**
25.1 Interrupt Introduction . . . 451
25.2 API Functions . . . 451
26 LIN Module . . . **457**
26.1 LIN Introduction . . . 457
26.2 API Functions . . . 457
27 MemCfg Module . . . **506**
27.1 MemCfg Introduction . . . 506
27.2 API Functions . . . 506
28 PGA Module . . . **522**
28.1 PGA Introduction . . . 522
28.2 API Functions . . . 522
29 PMBus Module . . . **527**
29.1 PMBus Introduction . . . 527
29.2 PMBus Message Types . . . 527
29.3 API Functions . . . 534
30 SCI Module . . . **552**

30.1 SCI Introduction . . . 552
30.2 API Functions . . . 552

31 SDFM Module . . . 573
31.1 SDFM Introduction . . . 573
31.2 API Functions . . . 573

32 SPI Module . . . 609
32.1 SPI Introduction . . . 609
32.2 API Functions . . . 609

33 SysCtl Module . . . 629
33.1 SysCtl Introduction . . . 629
33.2 API Functions . . . 629

34 Version Module . . . 666
34.1 Version Introduction . . . 666
34.2 API Functions . . . 666

35 X-BAR Module . . . 668
35.1 X-BAR Introduction . . . 668
35.2 API Functions . . . 668

IMPORTANT NOTICE . . . 678

1 Introduction

The F28004x Peripheral Driver Library is a set of drivers for accessing the peripherals found on the F28004x microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a software layer to facilitate a slightly higher level of programming than direct register accesses.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C except where absolutely not possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They are intended to make code more portable across other C2000 devices.
- Code written with these APIs will be more readable than code written using many direct register accesses.

Some consequences of this are that the drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

Minimum Requirements: CCSv6.2.0.00050 and C2000 Compiler v16.9.1.LTS

Source Code Overview

The following is an overview of the organization of the peripheral driver library source code.

`driverlib/`	This directory contains the source code for the drivers.
`driverlib/inc/`	This directory holds the peripheral, interrupt, and register access header files used for the direct register access programming model.
`hw_*.h`	Header files, one per peripheral, that describe all the registers and the bit fields within those registers for each peripheral. These header files are used by the drivers to directly access a peripheral, and can be used by application code to bypass the peripheral driver library API.

2 Revision History

v1.04.00.00

- dac.c - Corrected DAC_tuneOffsetTrim() function
- asysctl.h - Fixed missing typecast of uint32$_{t}inASysCtl_selectCMPHPMuxandASysCtl_selectCMPLPMuxAPIsbeforea16-bitshift.can.c-UpdatedCAN_readMessage()functiontousebaseinsteadofCANA_{B}ASEparameterwhereeverhardcoded.$

- cla.h - Updated CLA triggers sources
- epwm.h - Added APIs for DC Edge Filter configurations.
- fsi.c - Fixed FSI_configRxDelayLine() function.
- flash.h - Adding Flash$_{c}learLowErrorPosition()andFlash_{c}learHighErrorPosition()functionshw_erad.h-AddedERADregisters$
- hw_memmap.h - Added ERAD base addresses
- hw_types.h - Added header guards for float types

v1.03.00.00

- IMPORTANT: can.h - Changed interrupt numbering from 1 and 2 to 0 and 1
- IMPORTANT: Removed Low Power Mode Standby (Removed SysCtl_enterStandbyMode(), SysCtl_setStandbyQualificationPeriod(), SysCtl_enableWatchdogStandbyWakeup(), SysCtl_disableWatchdogStandbyWakeup())
- hrpwm.h - Removed HRPWM_enableSelfSync and HRPWM_disableSelfSync functions
- pga.h - Updated enum fields for PGA_LowPassResistorValue
- xbar.h - Corrected ASSERT values
- dac.h - New DAC_tuneOffsetTrim() function
- flash.h - Added pragmas for functions in RAM when building for C++
- epwm.h - New functions: EPWM_enableValleyCapture(), EPWM_disableValleyCapture(), EPWM_startValleyCapture(), EPWM_setValleyTriggerSource(), EPWM_setValleyTriggerEdgeCounts(), EPWM_enableValleyHWDelay(), EPWM_disableValleyHWDelay(), EPWM_setValleySWDelayValue(), EPWM_setValleyDelayDivider(), EPWM_getValleyEdgeStatus(), EPWM_getValleyCount(), EPWM_getValleyHWDelay()

v1.02.00.00

- hrcap.h - Fixed HRCAP_getScaleFactor() where the HRCAP base variable was incorrect
- asysctl.h - New DCDC functions: ASysCtl_enableDCDC(), ASysCtl_disableDCDC(), ASysCtl_getInductorFaultStatus(), ASysCtl_getSwitchSequenceStatus(), ASysCtl_lockDCDC()
- lin.h - Correct LIN_disableModule() Bitrate calculation comment
- fsi.h - New FSI SPI mode functions: FSI_enableTxSPIMode(), FSI_disableTxSPIMode(), FSI_enableRxSPIPairing(), FSI_disableRxSPIPairing(), FSI_enableRxSPIMode(),FSI_disableRxSPIMode()
- can.c - Fixed issue when setting up, sending, or receiving CAN messages that message object 32 would get enabled. Additionally, this fixes issues when optimizing.
- adc.h - Added clarifications to comments for ADC_setVREF()

- adc.h - New temperature sensor functions: ADC_getTemperatureC(), ADC_getTemperatureK()

v1.01.00.00

- IMPORTANT: sdfm.h and hw_sdfm.h - Renamed macros containing "SDIPARMx" to "SDDPARMx" and renamed "FILRESEN" to "SDSYNCEN"
- clapromcrc.h - Corrected return value for CLAPROMCRC_checkStatus()
- can.c - Fixed issue where CAN_readMessage() wasn't clearing the NewData bit field
- can.c - Removed clears to interface registers in CAN_setupMessageObject() causing optimization issues
- can.h - Removed macros for CAN_STATUS_PDA and CAN_STATUS_WAKE_UP
- hw_can.h - Renamed incorrect "Name" field in the CAN_GLB_INT_FLG register to INT0_FLG
- hw_ecap.h - Added ECAPSYNCINSEL register
- hw_hrcap.h - Added ECAPSYNCINSEL register
- hw_fsi.h - Removed bit fields related to SPI mode

v1.00.00.00

- Initial release

3 Programming Model

Introduction 6
Direct Register Access Model 6
Software Driver Model 7
Combining The Models 7

3.1 Introduction

The peripheral driver library provides support for two programming models: the direct register access model and the software driver model. Each model can be used independently or combined, based on the needs of the application or the programming environment desired by the developer.

Each programming model has advantages and disadvantages. Use of the direct register access model generally results in smaller and more efficient code than using the software driver model. However, the direct register access model requires detailed knowledge of the operation of each register and bit field, as well as their interactions and any sequencing required for proper operation of the peripheral; the developer is somewhat more insulated from these details by the software driver model, generally requiring less time to develop applications. The software driver model also results in more readable code.

3.2 Direct Register Access Model

In the direct register access model, the peripherals are programmed by the application by writing values directly into the peripheral's registers. A set of macros is provided that simplifies this process. These macros are stored in several header files contained in the `inc` directory. By including the header files `inc/hw_types.h` and `inc/hw_memmap.h`, macros are available for accessing all registers. Individual bitfield accesses can easily be added by simply including the `inc/hw_peripheral.h` header file for the desired peripheral.

The defines used by the direct register access model follow a naming convention that makes it easier to know how to use a particular macro. The rules are as follows:

- Values that end in `_BASE` and are found in `inc/hw_memmap.h` are module instance base addresses. For example, `SPIA_BASE` and `SPIB_BASE` are the base addresses of instances A and B of the SPI module respectively.
- Values that contain an `_O_` are register address offsets used to access the value of a register. For example, `SPI_O_CCR` is used to access the `CCR` register in a SPI module. These can be added to the base address values to get the register address.
- Values that end in `_M` represent the mask for a multi-bit field in a register. For example, `SPI_CCR_SPICHAR_M` is a mask for the `SPICHAR` field in the `CCR` register. Note that fields that are the whole width of the register are not given masks.
- Values that end in `_S` represent the number of bits to shift a value in order to align it with a multi-bit field. These values match the macro with the same base name but ending with `_M`.
- All others are single-bit field masks. For example, `SPI_CCR_SPILBK` corresponds to the `SPILBK` bit in the `CCR` register.

The `inc\hw_types.h` file contains macros to access a register. They are as follows where `x` is the address to be accessed:

- `HWREG(x)` is used for 32-bit accesses, such as reading a value from a 32-bit counter register.
- `HWREGH(x)` is used for 16-bit accesses. This can be used to access a 16-bit register or the upper or lower words of a 32-bit register. This is usually the most efficient.
- `HWREGB(x)` is used for 8-bit accesses using the __byte() intrinsic (see the TMS320C28x Optimizing C/C++ Compiler User's Guide). It typically should only be used when an 8-bit access is required by the hardware. Otherwise, use `HWREGH()` and mask and shift out the unwanted bits.
- `HWREG_BP(x)` is another macro used for 32-bit accesses, but it uses the __byte_peripheral_32() compiler intrinsic. This is intended for use with peripherals that use a special addressing scheme to support byte accesses such as CAN or USB.

Given these definitions, the `CCR` register can be programmed as follows:

```
// Enable loopback mode on SPI A
HWREGH(SPIA_BASE + SPI_O_CCR) |= SPI_CCR_SPILBK;

// Change the number of bits that make up a character to 8
// - First clear the field
// - Then shift the new value into place and write it into the register
HWREGH(SPIA_BASE + SPI_O_CCR) &= ~SPI_CCR_SPICHAR_M;
HWREGH(SPIA_BASE + SPI_O_CCR) |= 8 << SPI_CCR_SPICHAR_S;
```

Extracting the value of the `SPICHAR` field in the `CCR` register is as follows:

```
x = (HWREGH(SPIA_BASE + SPI_O_CCR) & SPI_CCR_SPICHAR_M) >> SPI_CCR_SPICHAR_S;
```

3.3 Software Driver Model

In the software driver model, the API provided by the peripheral driver library is used by applications to control the peripherals. Because these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This method provides for rapid development of the application without requiring detailed knowledge of the registers.

The following function call programs the `SPICHAR` field of `CCR` register mentioned in the direct register access model as well as a few other fields and registers.

```
    SPI_setConfig(SPIA_BASE,  100000000,  SPI_PROT_POL0PHA0,
                  SPI_MODE_MASTER,  500000,  16);
```

The drivers in the peripheral driver library are described in the remaining chapters in this document. They combine to form the software driver model.

3.4 Combining The Models

The direct register access model and software driver model can be used together in a single application, allowing the most appropriate model to be applied as needed to any particular situation within the application. For example, the software driver model can be used to configure the peripherals (because this is not performance critical) and the direct register access model can be used for operation of the peripheral (which may be more performance critical). Or, the software driver model can be used for peripherals that are not performance critical (such as SCI used for data logging) and the direct register access model for performance critical peripherals.

Additionally, the direct register access model can be used when there is no suitable driver library API for the desired task. Although an API may be available that performs a specific function on an individual bit or register, it could be more beneficial to use the direct register access programming model when performing tasks on entire registers or multiple bits at a given time. However, if there is an API available for the intended task it should be used as it will provide for more rapid development of the application without going into depth on programming the peripherals.

4 Driver Library Usage

Introduction . . . 9
Code Composer Studio Tips . . . 9
ASSERT Macro . . . 10
Driver Library Optimization . . . 11

4.1 Introduction

To develop with the peripheral driver library more efficiently, Code Composer Studio (CCS) offers several project and workspace features that can help maximize development time and device application execution. As previously discussed in the programming model chapter, there are advantages and disadvantages to each programming model. This chapter will explain optimization tips that should be used in conjuction with the APIs provided by the peripheral driver library to overcome and minimize those disadvantages.

4.2 Code Composer Studio Tips

This section will detail some Code Composer Studio (CCS) tips that can be used to help effectively use the driver library during development.

4.2.1 Content Assist

In CCS, the Content Assist feature can be used to offer suggestions for completing function and parameter names. This feature may be auto-activated while typing or it can be activated by hitting Ctrl+Space. To get the desired preferences, adjust the settings under C/C++ -> Editor -> Content Assist. The figure below shows the Content Assist in use.

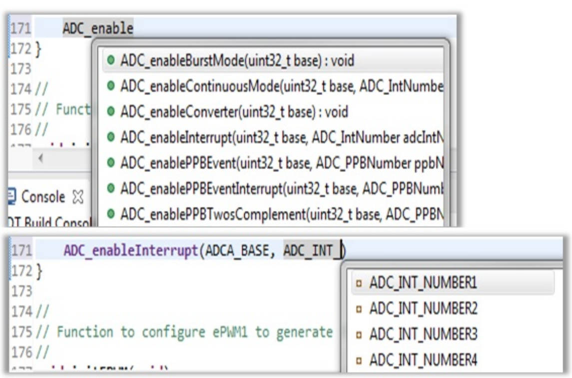

Figure 4.1: Content Assist

If you can't tell what an appropriate parameter is just from looking at the function prototype and the Content Assist list, hover over the function to view its description.

4.2.2 CCS Outline View

With a driver header file open, it is useful to take advantage of the CCS Outline view to get a complete list of functions, enumerations, and macros. The Outline view can be opened by selecting Window -> Show view -> Outline. The figure below shows the outline view in use.

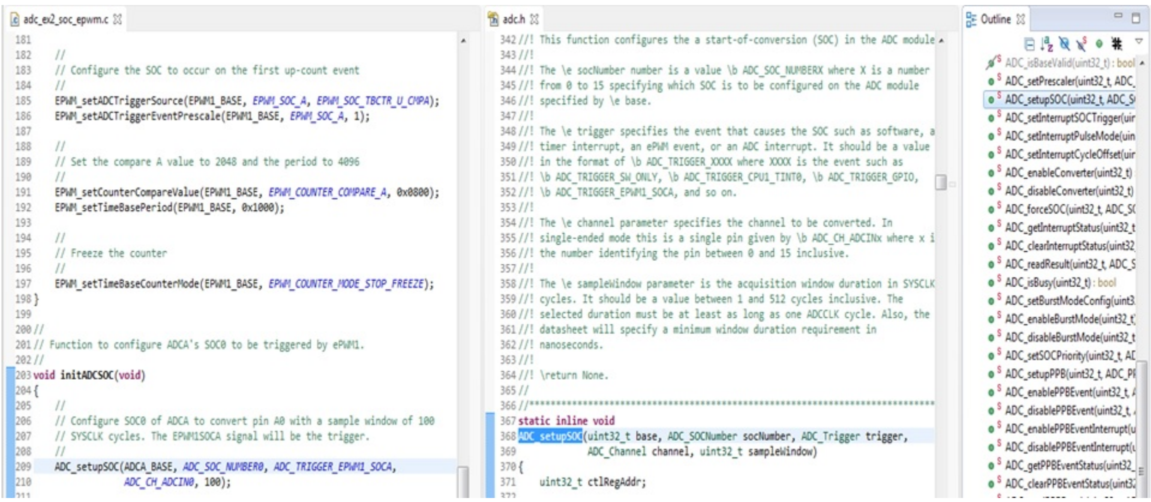

Figure 4.2: CCS Outline View

Similarly, you can split screen between application code and the API Reference Guide in the Resource Explorer.

Additionally, the function prototype in a driver header file can be viewed by holding Ctrl and clicking on the function name in the application code.

For more information on any of the tips provided, refer to the CCS Online Help section for details (CCS menu Help -> Help Contents and search for Content Assist).

4.3 ASSERT Macro

An ASSERT macro is defined in the `driverlib/debug.h` file as a method of checking the validity of function arguments and other error conditions. When the symbol DEBUG is defined, ASSERT(*expr*) will call a user-defined error function __error__() when Boolean expression *expr* is evaluated to false. To use the macro, an application must provide an __error__() function with the following prototype:

```
void __error__(char *filename, uint32_t line);
```

The *filename* and *line* parameters indicate which ASSERT resulted in the error condition. It is up to the application to decide what action the __error__() function should take to report the error.

The default Debug build configuration for the driverlib.lib project and the Driverlib example projects have turned on ASSERT by putting DEBUG in the projects' predefined symbols. Removing the DEBUG symbol from the projects will cause the ASSERT macro to compile to nothing, meaning it will add no code size or cycles to the application when it is turned off.

4.4 Driver Library Optimization

When using the software driver programming model it is important to note that there is a price to abstraction and making functions generic. Some of the drawbacks include the overhead time of the function call and the calculation time required to access a specific register offset or bit field within the register.

To help overcome these shortcomings, it is important to consider the use of inline functions. Using inline functions eliminates the need for function calls since the function is essentially treated like a macro. If constants are being passed into the function's parameters, much of its code may be evaluated at compile time. In order to utilize inline functions you must turn on optimization for it to take effect. If optimization is desired without the use of inline functions, use the –no_inling (-pi) option. This option can be set in the CCS project properties under Build -> C2000 Compiler -> Advanced Options -> Language Options.

In addition to inline functions, using the "generating function subsection" compiler option(–gen_func_subsections=on, -mo) is important. By default, the library project provided with the peripheral driver library project has this option turned on. When this option is selected, the compiler places each driver library function into its own subsection. This allows only the functions that are referenced in the application to be linked into the final executable. This can result in an overall code size reduction. This compiler option can be set by accessing the CCS project properties under Build -> C2000 Compiler -> Advanced Options -> Runtime Model Options.

The optimization options can be found in the CCS project properties which is accessed by right-clicking on the project in the project explorer and selecting properties. In the resulting window, the optimization settings are found in Build -> C2000 Compiler -> Optimization.

5 ADC Module

Introduction . . . 12
API Functions . . . 12

5.1 ADC Introduction

The analog to digital converter (ADC) API provides a set of functions for programming the digital circuits of the converter, referred to as the ADC wrapper. Functions are provided to configure the conversions, read the data conversion result registers, configure the post-processing blocks (PPB), and set up and handle interrupts and events.

5.2 API Functions

Enumerations

- enum ADC_ClkPrescale {
ADC_CLK_DIV_1_0, ADC_CLK_DIV_2_0, ADC_CLK_DIV_2_5, ADC_CLK_DIV_3_0, ADC_CLK_DIV_3_5, ADC_CLK_DIV_4_0, ADC_CLK_DIV_4_5, ADC_CLK_DIV_5_0, ADC_CLK_DIV_5_5, ADC_CLK_DIV_6_0, ADC_CLK_DIV_6_5, ADC_CLK_DIV_7_0, ADC_CLK_DIV_7_5, ADC_CLK_DIV_8_0, ADC_CLK_DIV_8_5 }
- enum ADC_Trigger {
ADC_TRIGGER_SW_ONLY, ADC_TRIGGER_CPU1_TINT0, ADC_TRIGGER_CPU1_TINT1, ADC_TRIGGER_CPU1_TINT2, ADC_TRIGGER_GPIO, ADC_TRIGGER_EPWM1_SOCA, ADC_TRIGGER_EPWM1_SOCB, ADC_TRIGGER_EPWM2_SOCA, ADC_TRIGGER_EPWM2_SOCB, ADC_TRIGGER_EPWM3_SOCA, ADC_TRIGGER_EPWM3_SOCB, ADC_TRIGGER_EPWM4_SOCA, ADC_TRIGGER_EPWM4_SOCB, ADC_TRIGGER_EPWM5_SOCA, ADC_TRIGGER_EPWM5_SOCB, ADC_TRIGGER_EPWM6_SOCA, ADC_TRIGGER_EPWM6_SOCB, ADC_TRIGGER_EPWM7_SOCA, ADC_TRIGGER_EPWM7_SOCB, ADC_TRIGGER_EPWM8_SOCA, ADC_TRIGGER_EPWM8_SOCB }
- enum ADC_Channel {
ADC_CH_ADCIN0, ADC_CH_ADCIN1, ADC_CH_ADCIN2, ADC_CH_ADCIN3, ADC_CH_ADCIN4, ADC_CH_ADCIN5, ADC_CH_ADCIN6, ADC_CH_ADCIN7, ADC_CH_ADCIN8, ADC_CH_ADCIN9, ADC_CH_ADCIN10, ADC_CH_ADCIN11, ADC_CH_ADCIN12, ADC_CH_ADCIN13, ADC_CH_ADCIN14, ADC_CH_ADCIN15 }
- enum ADC_PulseMode { ADC_PULSE_END_OF_ACQ_WIN, ADC_PULSE_END_OF_CONV }
- enum ADC_IntNumber { ADC_INT_NUMBER1, ADC_INT_NUMBER2, ADC_INT_NUMBER3, ADC_INT_NUMBER4 }
- enum ADC_PPBNumber { ADC_PPB_NUMBER1, ADC_PPB_NUMBER2, ADC_PPB_NUMBER3, ADC_PPB_NUMBER4 }
- enum ADC_SOCNumber {
ADC_SOC_NUMBER0, ADC_SOC_NUMBER1, ADC_SOC_NUMBER2, ADC_SOC_NUMBER3,
ADC_SOC_NUMBER4, ADC_SOC_NUMBER5, ADC_SOC_NUMBER6,

ADC_SOC_NUMBER7,
ADC_SOC_NUMBER8, ADC_SOC_NUMBER9, ADC_SOC_NUMBER10,
ADC_SOC_NUMBER11,
ADC_SOC_NUMBER12, ADC_SOC_NUMBER13, ADC_SOC_NUMBER14,
ADC_SOC_NUMBER15 }
- enum ADC_IntSOCTrigger { ADC_INT_SOC_TRIGGER_NONE,
ADC_INT_SOC_TRIGGER_ADCINT1, ADC_INT_SOC_TRIGGER_ADCINT2 }
- enum ADC_PriorityMode {
ADC_PRI_ALL_ROUND_ROBIN, ADC_PRI_SOC0_HIPRI, ADC_PRI_THRU_SOC1_HIPRI,
ADC_PRI_THRU_SOC2_HIPRI,
ADC_PRI_THRU_SOC3_HIPRI, ADC_PRI_THRU_SOC4_HIPRI,
ADC_PRI_THRU_SOC5_HIPRI, ADC_PRI_THRU_SOC6_HIPRI,
ADC_PRI_THRU_SOC7_HIPRI, ADC_PRI_THRU_SOC8_HIPRI,
ADC_PRI_THRU_SOC9_HIPRI, ADC_PRI_THRU_SOC10_HIPRI,
ADC_PRI_THRU_SOC11_HIPRI, ADC_PRI_THRU_SOC12_HIPRI,
ADC_PRI_THRU_SOC13_HIPRI, ADC_PRI_THRU_SOC14_HIPRI,
ADC_PRI_ALL_HIPRI }
- enum ADC_ReferenceMode
- enum ADC_ReferenceVoltage

Functions

- static void ADC_setPrescaler (uint32_t base, ADC_ClkPrescale clkPrescale)
- static void ADC_setupSOC (uint32_t base, ADC_SOCNumber socNumber, ADC_Trigger trigger, ADC_Channel channel, uint32_t sampleWindow)
- static void ADC_setInterruptSOCTrigger (uint32_t base, ADC_SOCNumber socNumber, ADC_IntSOCTrigger trigger)
- static void ADC_setInterruptPulseMode (uint32_t base, ADC_PulseMode pulseMode)
- static void ADC_setInterruptCycleOffset (uint32_t base, uint16_t cycleOffset)
- static void ADC_enableConverter (uint32_t base)
- static void ADC_disableConverter (uint32_t base)
- static void ADC_forceSOC (uint32_t base, ADC_SOCNumber socNumber)
- static bool ADC_getInterruptStatus (uint32_t base, ADC_IntNumber adcIntNum)
- static void ADC_clearInterruptStatus (uint32_t base, ADC_IntNumber adcIntNum)
- static uint16_t ADC_readResult (uint32_t resultBase, ADC_SOCNumber socNumber)
- static bool ADC_isBusy (uint32_t base)
- static void ADC_setBurstModeConfig (uint32_t base, ADC_Trigger trigger, uint16_t burstSize)
- static void ADC_enableBurstMode (uint32_t base)
- static void ADC_disableBurstMode (uint32_t base)
- static void ADC_setSOCPriority (uint32_t base, ADC_PriorityMode priMode)
- static void ADC_setupPPB (uint32_t base, ADC_PPBNumber ppbNumber, ADC_SOCNumber socNumber)
- static void ADC_enablePPBEvent (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t evtFlags)
- static void ADC_disablePPBEvent (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t evtFlags)
- static void ADC_enablePPBEventInterrupt (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t intFlags)
- static void ADC_disablePPBEventInterrupt (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t intFlags)
- static uint16_t ADC_getPPBEventStatus (uint32_t base, ADC_PPBNumber ppbNumber)
- static void ADC_clearPPBEventStatus (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t evtFlags)
- static int32_t ADC_readPPBResult (uint32_t resultBase, ADC_PPBNumber ppbNumber)

- static uint16_t ADC_getPPBDelayTimeStamp (uint32_t base, ADC_PPBNumber ppbNumber)
- static void ADC_setPPBCalibrationOffset (uint32_t base, ADC_PPBNumber ppbNumber, int16_t offset)
- static void ADC_setPPBReferenceOffset (uint32_t base, ADC_PPBNumber ppbNumber, uint16_t offset)
- static void ADC_enablePPBTwosComplement (uint32_t base, ADC_PPBNumber ppbNumber)
- static void ADC_disablePPBTwosComplement (uint32_t base, ADC_PPBNumber ppbNumber)
- static void ADC_enableInterrupt (uint32_t base, ADC_IntNumber adcIntNum)
- static void ADC_disableInterrupt (uint32_t base, ADC_IntNumber adcIntNum)
- static void ADC_setInterruptSource (uint32_t base, ADC_IntNumber adcIntNum, ADC_SOCNumber socNumber)
- static void ADC_enableContinuousMode (uint32_t base, ADC_IntNumber adcIntNum)
- static void ADC_disableContinuousMode (uint32_t base, ADC_IntNumber adcIntNum)
- static int16_t ADC_getTemperatureC (uint16_t tempResult, float32_t vref)
- static int16_t ADC_getTemperatureK (uint16_t tempResult, float32_t vref)
- void ADC_setVREF (uint32_t base, ADC_ReferenceMode refMode, ADC_ReferenceVoltage refVoltage)
- void ADC_setPPBTripLimits (uint32_t base, ADC_PPBNumber ppbNumber, int32_t tripHiLimit, int32_t tripLoLimit)

5.2.1 Detailed Description

The code for this module is contained in `driverlib/adc.c`, with `driverlib/adc.h` containing the API declarations for use by applications.

5.2.2 Enumeration Type Documentation

5.2.2.1 enum **ADC_ClkPrescale**

Values that can be passed to ADC_setPrescaler() as the *clkPrescale* parameter.

Enumerator

ADC_CLK_DIV_1_0 ADCCLK = (input clock) / 1.0.
ADC_CLK_DIV_2_0 ADCCLK = (input clock) / 2.0.
ADC_CLK_DIV_2_5 ADCCLK = (input clock) / 2.5.
ADC_CLK_DIV_3_0 ADCCLK = (input clock) / 3.0.
ADC_CLK_DIV_3_5 ADCCLK = (input clock) / 3.5.
ADC_CLK_DIV_4_0 ADCCLK = (input clock) / 4.0.
ADC_CLK_DIV_4_5 ADCCLK = (input clock) / 4.5.
ADC_CLK_DIV_5_0 ADCCLK = (input clock) / 5.0.
ADC_CLK_DIV_5_5 ADCCLK = (input clock) / 5.5.
ADC_CLK_DIV_6_0 ADCCLK = (input clock) / 6.0.
ADC_CLK_DIV_6_5 ADCCLK = (input clock) / 6.5.
ADC_CLK_DIV_7_0 ADCCLK = (input clock) / 7.0.
ADC_CLK_DIV_7_5 ADCCLK = (input clock) / 7.5.
ADC_CLK_DIV_8_0 ADCCLK = (input clock) / 8.0.
ADC_CLK_DIV_8_5 ADCCLK = (input clock) / 8.5.

5.2.2.2 enum **ADC_Trigger**

Values that can be passed to ADC_setupSOC() as the *trigger* parameter to specify the event that will trigger a conversion to start. It is also used with ADC_setBurstModeConfig().

Enumerator

ADC_TRIGGER_SW_ONLY Software only.
ADC_TRIGGER_CPU1_TINT0 CPU1 Timer 0, TINT0.
ADC_TRIGGER_CPU1_TINT1 CPU1 Timer 1, TINT1.
ADC_TRIGGER_CPU1_TINT2 CPU1 Timer 2, TINT2.
ADC_TRIGGER_GPIO GPIO, ADCEXTSOC.
ADC_TRIGGER_EPWM1_SOCA ePWM1, ADCSOCA
ADC_TRIGGER_EPWM1_SOCB ePWM1, ADCSOCB
ADC_TRIGGER_EPWM2_SOCA ePWM2, ADCSOCA
ADC_TRIGGER_EPWM2_SOCB ePWM2, ADCSOCB
ADC_TRIGGER_EPWM3_SOCA ePWM3, ADCSOCA
ADC_TRIGGER_EPWM3_SOCB ePWM3, ADCSOCB
ADC_TRIGGER_EPWM4_SOCA ePWM4, ADCSOCA
ADC_TRIGGER_EPWM4_SOCB ePWM4, ADCSOCB
ADC_TRIGGER_EPWM5_SOCA ePWM5, ADCSOCA
ADC_TRIGGER_EPWM5_SOCB ePWM5, ADCSOCB
ADC_TRIGGER_EPWM6_SOCA ePWM6, ADCSOCA
ADC_TRIGGER_EPWM6_SOCB ePWM6, ADCSOCB
ADC_TRIGGER_EPWM7_SOCA ePWM7, ADCSOCA
ADC_TRIGGER_EPWM7_SOCB ePWM7, ADCSOCB
ADC_TRIGGER_EPWM8_SOCA ePWM8, ADCSOCA
ADC_TRIGGER_EPWM8_SOCB ePWM8, ADCSOCB

5.2.2.3 enum **ADC_Channel**

Values that can be passed to ADC_setupSOC() as the *channel* parameter. This is the input pin on which the signal to be converted is located.

Enumerator

ADC_CH_ADCIN0 ADCIN0 is converted.
ADC_CH_ADCIN1 ADCIN1 is converted.
ADC_CH_ADCIN2 ADCIN2 is converted.
ADC_CH_ADCIN3 ADCIN3 is converted.
ADC_CH_ADCIN4 ADCIN4 is converted.
ADC_CH_ADCIN5 ADCIN5 is converted.
ADC_CH_ADCIN6 ADCIN6 is converted.
ADC_CH_ADCIN7 ADCIN7 is converted.
ADC_CH_ADCIN8 ADCIN8 is converted.
ADC_CH_ADCIN9 ADCIN9 is converted.
ADC_CH_ADCIN10 ADCIN10 is converted.
ADC_CH_ADCIN11 ADCIN11 is converted.

ADC_CH_ADCIN12 ADCIN12 is converted.
ADC_CH_ADCIN13 ADCIN13 is converted.
ADC_CH_ADCIN14 ADCIN14 is converted.
ADC_CH_ADCIN15 ADCIN15 is converted.

5.2.2.4 enum **ADC_PulseMode**

Values that can be passed to ADC_setInterruptPulseMode() as the *pulseMode* parameter.

Enumerator

ADC_PULSE_END_OF_ACQ_WIN Occurs at the end of the acquisition window.
ADC_PULSE_END_OF_CONV Occurs at the end of the conversion.

5.2.2.5 enum **ADC_IntNumber**

Values that can be passed to ADC_enableInterrupt(), ADC_disableInterrupt(), and ADC_getInterruptStatus() as the *adcIntNum* parameter.

Enumerator

ADC_INT_NUMBER1 ADCINT1 Interrupt.
ADC_INT_NUMBER2 ADCINT2 Interrupt.
ADC_INT_NUMBER3 ADCINT3 Interrupt.
ADC_INT_NUMBER4 ADCINT4 Interrupt.

5.2.2.6 enum **ADC_PPBNumber**

Values that can be passed in as the *ppbNumber* parameter for several functions.

Enumerator

ADC_PPB_NUMBER1 Post-processing block 1.
ADC_PPB_NUMBER2 Post-processing block 2.
ADC_PPB_NUMBER3 Post-processing block 3.
ADC_PPB_NUMBER4 Post-processing block 4.

5.2.2.7 enum **ADC_SOCNumber**

Values that can be passed in as the *socNumber* parameter for several functions. This value identifies the start-of-conversion (SOC) that a function is configuring or accessing. Note that in some cases (for example, ADC_setInterruptSource()) *socNumber* is used to refer to the corresponding end-of-conversion (EOC).

Enumerator

ADC_SOC_NUMBER0 SOC/EOC number 0.
ADC_SOC_NUMBER1 SOC/EOC number 1.
ADC_SOC_NUMBER2 SOC/EOC number 2.
ADC_SOC_NUMBER3 SOC/EOC number 3.

ADC_SOC_NUMBER4 SOC/EOC number 4.
ADC_SOC_NUMBER5 SOC/EOC number 5.
ADC_SOC_NUMBER6 SOC/EOC number 6.
ADC_SOC_NUMBER7 SOC/EOC number 7.
ADC_SOC_NUMBER8 SOC/EOC number 8.
ADC_SOC_NUMBER9 SOC/EOC number 9.
ADC_SOC_NUMBER10 SOC/EOC number 10.
ADC_SOC_NUMBER11 SOC/EOC number 11.
ADC_SOC_NUMBER12 SOC/EOC number 12.
ADC_SOC_NUMBER13 SOC/EOC number 13.
ADC_SOC_NUMBER14 SOC/EOC number 14.
ADC_SOC_NUMBER15 SOC/EOC number 15.

5.2.2.8 enum **ADC_IntSOCTrigger**

Values that can be passed in as the *trigger* parameter for the ADC_setInterruptSOCTrigger() function.

Enumerator

ADC_INT_SOC_TRIGGER_NONE No ADCINT will trigger the SOC.
ADC_INT_SOC_TRIGGER_ADCINT1 ADCINT1 will trigger the SOC.
ADC_INT_SOC_TRIGGER_ADCINT2 ADCINT2 will trigger the SOC.

5.2.2.9 enum **ADC_PriorityMode**

Values that can be passed to ADC_setSOCPriority() as the *priMode* parameter.

Enumerator

ADC_PRI_ALL_ROUND_ROBIN Round robin mode is used for all.
ADC_PRI_SOC0_HIPRI SOC 0 hi pri, others in round robin.
ADC_PRI_THRU_SOC1_HIPRI SOC 0-1 hi pri, others in round robin.
ADC_PRI_THRU_SOC2_HIPRI SOC 0-2 hi pri, others in round robin.
ADC_PRI_THRU_SOC3_HIPRI SOC 0-3 hi pri, others in round robin.
ADC_PRI_THRU_SOC4_HIPRI SOC 0-4 hi pri, others in round robin.
ADC_PRI_THRU_SOC5_HIPRI SOC 0-5 hi pri, others in round robin.
ADC_PRI_THRU_SOC6_HIPRI SOC 0-6 hi pri, others in round robin.
ADC_PRI_THRU_SOC7_HIPRI SOC 0-7 hi pri, others in round robin.
ADC_PRI_THRU_SOC8_HIPRI SOC 0-8 hi pri, others in round robin.
ADC_PRI_THRU_SOC9_HIPRI SOC 0-9 hi pri, others in round robin.
ADC_PRI_THRU_SOC10_HIPRI SOC 0-10 hi pri, others in round robin.
ADC_PRI_THRU_SOC11_HIPRI SOC 0-11 hi pri, others in round robin.
ADC_PRI_THRU_SOC12_HIPRI SOC 0-12 hi pri, others in round robin.
ADC_PRI_THRU_SOC13_HIPRI SOC 0-13 hi pri, others in round robin.
ADC_PRI_THRU_SOC14_HIPRI SOC 0-14 hi pri, SOC15 in round robin.
ADC_PRI_ALL_HIPRI All priorities based on SOC number.

5.2.2.10 enum **ADC_ReferenceMode**

Values that can be passed to ADC_getTemperatureC(), ADC_getTemperatureK(), and ADC_setVREF() as the *refMode* parameter.

5.2.3 Function Documentation

5.2.3.1 static void ADC_setPrescaler (uint32_t *base,* **ADC_ClkPrescale** *clkPrescale*) `[inline], [static]`

Configures the analog-to-digital converter module prescaler.

Parameters

base	is the base address of the ADC module.
clkPrescale	is the ADC clock prescaler.

This function configures the ADC module's ADCCLK.

The *clkPrescale* parameter specifies the value by which the input clock is divided to make the ADCCLK. The value can be specified with the value **ADC_CLK_DIV_1_0**, **ADC_CLK_DIV_2_0**, **ADC_CLK_DIV_2_5**, ..., **ADC_CLK_DIV_7_5**, **ADC_CLK_DIV_8_0**, or **ADC_CLK_DIV_8_5**.

Returns

None.

5.2.3.2 static void ADC_setupSOC (uint32_t *base,* **ADC_SOCNumber** *socNumber,* **ADC_Trigger** *trigger,* **ADC_Channel** *channel,* uint32_t *sampleWindow*) `[inline], [static]`

Configures a start-of-conversion (SOC) in the ADC.

Parameters

base	is the base address of the ADC module.
socNumber	is the number of the start-of-conversion.
trigger	the source that will cause the SOC.
channel	is the number associated with the input signal.
sampleWindow	is the acquisition window duration.

This function configures the a start-of-conversion (SOC) in the ADC module.

The *socNumber* number is a value **ADC_SOC_NUMBERX** where X is a number from 0 to 15 specifying which SOC is to be configured on the ADC module specified by *base*.

The *trigger* specifies the event that causes the SOC such as software, a timer interrupt, an ePWM event, or an ADC interrupt. It should be a value in the format of **ADC_TRIGGER_XXXX** where XXXX is the event such as **ADC_TRIGGER_SW_ONLY**, **ADC_TRIGGER_CPU1_TINT0**, **ADC_TRIGGER_GPIO**, **ADC_TRIGGER_EPWM1_SOCA**, and so on.

The *channel* parameter specifies the channel to be converted. In single-ended mode this is a single pin given by **ADC_CH_ADCINx** where x is the number identifying the pin between 0 and 15 inclusive.

The *sampleWindow* parameter is the acquisition window duration in SYSCLK cycles. It should be a value between 1 and 512 cycles inclusive. The selected duration must be at least as long as one ADCCLK cycle. Also, the datasheet will specify a minimum window duration requirement in nanoseconds.

Returns
None.

5.2.3.3 static void ADC_setInterruptSOCTrigger (uint32_t *base,* **ADC_SOCNumber** *socNumber,* **ADC_IntSOCTrigger** *trigger*) [inline], [static]

Configures the interrupt SOC trigger of an SOC.

Parameters

base	is the base address of the ADC module.
socNumber	is the number of the start-of-conversion.
trigger	the interrupt source that will cause the SOC.

This function configures the an interrupt start-of-conversion trigger in the ADC module.

The *socNumber* number is a value **ADC_SOC_NUMBERX** where X is a number from 0 to 15 specifying which SOC is to be configured on the ADC module specified by *base*.

The *trigger* specifies the interrupt that causes a start of conversion or none. It should be one of the following values.

- **ADC_INT_SOC_TRIGGER_NONE**
- **ADC_INT_SOC_TRIGGER_ADCINT1**
- **ADC_INT_SOC_TRIGGER_ADCINT2**

This functionality is useful for creating continuous conversions.

Returns
None.

5.2.3.4 static void ADC_setInterruptPulseMode (uint32_t *base,* **ADC_PulseMode** *pulseMode*) [inline], [static]

Sets the timing of the end-of-conversion pulse

Parameters

base	is the base address of the ADC module.
pulseMode	is the generation mode of the EOC pulse.

This function configures the end-of-conversion (EOC) pulse generated by the ADC. This pulse will be generated either at the end of the acquisition window (pass **ADC_PULSE_END_OF_ACQ_WIN** into *pulseMode*) or at the end of the voltage conversion, one cycle prior to the ADC result latching into its result register (pass **ADC_PULSE_END_OF_CONV** into *pulseMode*).

Returns
None.

5.2.3.5 static void ADC_setInterruptCycleOffset (uint32_t *base*, uint16_t *cycleOffset*) [inline], [static]

Sets the timing of early interrupt generation.

Parameters

base	is the base address of the ADC module.
cycleOffset	is the cycles from an SOC falling edge to an early interrupt pulse.

This function configures cycle offset between the negative edge of a sample pulse and an early interrupt pulse being generated. This number of cycles is specified with the *cycleOffset* parameter.

This function only applies when early interrupt generation is enabled. That means the ADC_setInterruptPulseMode() function *pulseMode* parameter is configured as **ADC_PULSE_END_OF_ACQ_WIN**.

Returns
None.

5.2.3.6 static void ADC_enableConverter (uint32_t *base*) [inline], [static]

Powers up the analog-to-digital converter core.

Parameters

base	is the base address of the ADC module.

This function powers up the analog circuitry inside the analog core.

Note
Allow at least a 500us delay before sampling after calling this API. If you enable multiple ADCs, you can delay after they all have begun powering up.

Returns
None.

5.2.3.7 static void ADC_disableConverter (uint32_t *base*) [inline], [static]

Powers down the analog-to-digital converter module.

Parameters

base	is the base address of the ADC module.

This function powers down the analog circuitry inside the analog core.

Returns
None.

5.2.3.8 static void ADC_forceSOC (uint32_t *base*, **ADC_SOCNumber** *socNumber*) `[inline]`, `[static]`

Forces a SOC flag to a 1 in the analog-to-digital converter.

Parameters

base	is the base address of the ADC module.
socNumber	is the number of the start-of-conversion.

This function forces the SOC flag associated with the SOC specified by *socNumber*. This initiates a conversion once that SOC is given priority. This software trigger can be used whether or not the SOC has been configured to accept some other specific trigger.

Returns
None.

5.2.3.9 static bool ADC_getInterruptStatus (uint32_t *base,* **ADC_IntNumber** *adcIntNum*) [inline], [static]

Gets the current ADC interrupt status.

Parameters

base	is the base address of the ADC module.
adcIntNum	is interrupt number within the ADC wrapper.

This function returns the interrupt status for the analog-to-digital converter.

Returns
true if the interrupt flag for the specified interrupt number is set and **false** if it is not.

5.2.3.10 static void ADC_clearInterruptStatus (uint32_t *base,* **ADC_IntNumber** *adcIntNum*) [inline], [static]

Clears ADC interrupt sources.

Parameters

base	is the base address of the ADC module.
adcIntNum	is interrupt number within the ADC wrapper.

This function clears the specified ADC interrupt sources so that they no longer assert. If not in continuous mode, this function must be called before any further interrupt pulses may occur.

adcIntNum takes a one of the values **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, **or** ADC_INT_NUMBER4 to express which of the four interrupts of the ADC module should be cleared

Returns
None.

5.2.3.11 static uint16_t ADC_readResult (uint32_t *resultBase,* **ADC_SOCNumber** *socNumber*) [inline], [static]

Reads the conversion result.

Parameters

resultBase	is the base address of the ADC results.
socNumber	is the number of the start-of-conversion.

This function returns the conversion result that corresponds to the base address passed into *resultBase* and the SOC passed into *socNumber*.

The *socNumber* number is a value **ADC_SOC_NUMBERX** where X is a number from 0 to 15 specifying which SOC's result is to be read.

Note

Take care that you are using a base address for the result registers (ADCxRESULT_BASE) and not a base address for the control registers.

Returns

Returns the conversion result.

5.2.3.12 static bool ADC_isBusy (uint32_t *base*) [inline], [static]

Determines whether the ADC is busy or not.

Parameters

base	is the base address of the ADC.

This function allows the caller to determine whether or not the ADC is busy and can sample another channel.

Returns

Returns **true** if the ADC is sampling or **false** if all samples are complete.

5.2.3.13 static void ADC_setBurstModeConfig (uint32_t *base*, **ADC_Trigger** *trigger*, uint16_t *burstSize*) [inline], [static]

Set SOC burst mode.

Parameters

base	is the base address of the ADC.
trigger	the source that will cause the burst conversion sequence.
burstSize	is the number of SOCs converted during a burst sequence.

This function configures the burst trigger and burstSize of an ADC module. Burst mode allows a single trigger to walk through the round-robin SOCs one or more at a time. When burst mode is enabled, the trigger selected by the ADC_setupSOC() API will no longer have an effect on the SOCs in round-robin mode. Instead, the source specified through the *trigger* parameter will cause a burst of *burstSize* conversions to occur.

The *trigger* parameter takes the same values as the ADC_setupSOC() API The *burstSize* parameter should be a value between 1 and 16 inclusive.

Returns

None.

5.2.3.14 static void ADC_enableBurstMode (uint32_t *base*) [inline], [static]

Enables SOC burst mode.

Parameters

base	is the base address of the ADC.

This function enables SOC burst mode operation of the ADC. Burst mode allows a single trigger to walk through the round-robin SOCs one or more at a time. When burst mode is enabled, the trigger selected by the ADC_setupSOC() API will no longer have an effect on the SOCs in round-robin mode. Use ADC_setBurstMode() to configure the burst trigger and size.

Returns

None.

5.2.3.15 static void ADC_disableBurstMode (uint32_t *base*) [inline], [static]

Disables SOC burst mode.

Parameters

base	is the base address of the ADC.

This function disables SOC burst mode operation of the ADC. SOCs in round-robin mode will be triggered by the trigger configured using the ADC_setupSOC() API.

Returns

None.

5.2.3.16 static void ADC_setSOCPriority (uint32_t *base*, **ADC_PriorityMode** *priMode*) [inline], [static]

Sets the priority mode of the SOCs.

Parameters

base	is the base address of the ADC.
priMode	is the priority mode of the SOCs.

This function sets the priority mode of the SOCs. There are three main modes that can be passed in the *priMode* parameter

- All SOCs are in round-robin mode. This means no SOC has an inherent higher priority over another. This is selected by passing in the value **ADC_PRI_ALL_ROUND_ROBIN**.
- All priorities are in high priority mode. This means that the priority of the SOC is determined by its SOC number. This option is selected by passing in the value **ADC_PRI_ALL_HIPRI**.
- A range of SOCs are assigned high priority, with all others in round robin mode. High priority mode means that an SOC with high priority will interrupt the round robin wheel and insert itself as the next conversion. Passing in the value **ADC_PRI_SOC0_HIPRI** will make SOC0 highest priority, **ADC_PRI_THRU_SOC1_HIPRI** will put SOC0 and SOC 1 in high priority, and so on up to **ADC_PRI_THRU_SOC14_HIPRI** where SOCs 0 through 14 are in high priority.

Returns

None.

5.2.3.17 static void ADC_setupPPB (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, **ADC_SOCNumber** *socNumber*) `[inline]`, `[static]`

Configures a post-processing block (PPB) in the ADC.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
socNumber	is the number of the start-of-conversion.

This function associates a post-processing block with a SOC.

The *ppbNumber* is a value **ADC_PPB_NUMBERX** where X is a value from 1 to 4 inclusive that identifies a PPB to be configured. The *socNumber* number is a value **ADC_SOC_NUMBERX** where X is a number from 0 to 15 specifying which SOC is to be configured on the ADC module specified by *base*.

Note

You can have more that one PPB associated with the same SOC, but a PPB can only be configured to correspond to one SOC at a time. Also note that when you have multiple PPBs for the same SOC, the calibration offset that actually gets applied will be that of the PPB with the highest number. Since SOC0 is the default for all PPBs, look out for unintentional overwriting of a lower numbered PPB's offset.

Returns

None.

5.2.3.18 static void ADC_enablePPBEvent (uint32_t *base,* **ADC_PPBNumber** *ppbNumber,* uint16_t *evtFlags*) [inline], [static]

Enables individual ADC PPB event sources.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
evtFlags	is a bit mask of the event sources to be enabled.

This function enables the indicated ADC PPB event sources. This will allow the specified events to propagate through the X-BAR to a pin or to an ePWM module. The *evtFlags* parameter can be any of the **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, or **ADC_EVT_ZERO** values.

Returns

None.

5.2.3.19 static void ADC_disablePPBEvent (uint32_t *base,* **ADC_PPBNumber** *ppbNumber,* uint16_t *evtFlags*) [inline], [static]

Disables individual ADC PPB event sources.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
evtFlags	is a bit mask of the event sources to be enabled.

This function disables the indicated ADC PPB event sources. This will stop the specified events from propagating through the X-BAR to other modules. The *evtFlags* parameter can be any of the **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, or **ADC_EVT_ZERO** values.

Returns
None.

5.2.3.20 static void ADC_enablePPBEventInterrupt (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, uint16_t *intFlags*) [inline], [static]

Enables individual ADC PPB event interrupt sources.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
intFlags	is a bit mask of the interrupt sources to be enabled.

This function enables the indicated ADC PPB interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor. The *intFlags* parameter can be any of the **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, or **ADC_EVT_ZERO** values.

Returns
None.

5.2.3.21 static void ADC_disablePPBEventInterrupt (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, uint16_t *intFlags*) [inline], [static]

Disables individual ADC PPB event interrupt sources.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
intFlags	is a bit mask of the interrupt source to be disabled.

This function disables the indicated ADC PPB interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor. The *intFlags* parameter can be any of the **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, or **ADC_EVT_ZERO** values.

Returns
None.

5.2.3.22 static uint16_t ADC_getPPBEventStatus (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*) [inline], [static]

Gets the current ADC event status.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.

This function returns the event status for the analog-to-digital converter.

Returns

Returns the current event status, enumerated as a bit field of **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, and **ADC_EVT_ZERO**.

5.2.3.23 static void ADC_clearPPBEventStatus (uint32_t *base,* **ADC_PPBNumber** *ppbNumber,* uint16_t *evtFlags*) `[inline]`, `[static]`

Clears ADC event flags.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
evtFlags	is a bit mask of the event source to be cleared.

This function clears the indicated ADC PPB event flags. After an event occurs this function must be called to allow additional events to be produced. The *evtFlags* parameter can be any of the **ADC_EVT_TRIPHI**, **ADC_EVT_TRIPLO**, or **ADC_EVT_ZERO** values.

Returns

None.

5.2.3.24 static int32_t ADC_readPPBResult (uint32_t *resultBase,* **ADC_PPBNumber** *ppbNumber*) `[inline]`, `[static]`

Reads the processed conversion result from the PPB.

Parameters

resultBase	is the base address of the ADC results.
ppbNumber	is the number of the post-processing block.

This function returns the processed conversion result that corresponds to the base address passed into *resultBase* and the PPB passed into *ppbNumber*.

Note

Take care that you are using a base address for the result registers (ADCxRESULT_BASE) and not a base address for the control registers.

Returns

Returns the signed 32-bit conversion result.

5.2.3.25 static uint16_t ADC_getPPBDelayTimeStamp (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*) [inline], [static]

Reads sample delay time stamp from a PPB.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.

This function returns the sample delay time stamp. This delay is the number of system clock cycles between the SOC being triggered and when it began converting.

Returns

Returns the delay time stamp.

5.2.3.26 static void ADC_setPPBCalibrationOffset (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, int16_t *offset*) [inline], [static]

Sets the post processing block offset correction.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
offset	is the 10-bit signed value subtracted from ADC the output.

This function sets the PPB offset correction value. This value can be used to digitally remove any system-level offset inherent in the ADCIN circuit before it is stored in the appropriate result register. The *offset* parameter is **subtracted** from the ADC output and is a signed value from -512 to 511 inclusive. For example, when *offset* = 1, ADCRESULT = ADC output - 1. When *offset* = -512, ADCRESULT = ADC output - (-512) or ADC output + 512.

Passing a zero in to the *offset* parameter will effectively disable the calculation, allowing the raw ADC result to be passed unchanged into the result register.

Note

If multiple PPBs are applied to the same SOC, the offset that will be applied will be that of the PPB with the highest number.

Returns

None

5.2.3.27 static void ADC_setPPBReferenceOffset (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, uint16_t *offset*) [inline], [static]

Sets the post processing block reference offset.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
offset	is the 16-bit unsigned value subtracted from ADC the output.

This function sets the PPB reference offset value. This can be used to either calculate the feedback error or convert a unipolar signal to bipolar by subtracting a reference value. The result will be stored in the appropriate PPB result register which can be read using ADC_readPPBResult().

Passing a zero in to the *offset* parameter will effectively disable the calculation and will pass the ADC result to the PPB result register unchanged.

Note

If in 12-bit mode, you may only pass a 12-bit value into the *offset* parameter.

Returns

None

5.2.3.28 static void ADC_enablePPBTwosComplement (uint32_t *base,* **ADC_PPBNumber** *ppbNumber*) [inline], [static]

Enables two's complement capability in the PPB.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.

This function enables two's complement in the post-processing block specified by the *ppbNumber* parameter. When enabled, a two's complement will be performed on the output of the offset subtraction before it is stored in the appropriate PPB result register. In other words, the PPB result will be the reference offset value minus the the ADC result value (ADCPPBxRESULT = ADCSOCxOFFREF - ADCRESULTx).

Returns

None

5.2.3.29 static void ADC_disablePPBTwosComplement (uint32_t *base,* **ADC_PPBNumber** *ppbNumber*) [inline], [static]

Disables two's complement capability in the PPB.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.

This function disables two's complement in the post-processing block specified by the *ppbNumber* parameter. When disabled, a two's complement will **NOT** be performed on the output of the offset subtraction before it is stored in the appropriate PPB result register. In other words, the PPB result will be the ADC result value minus the reference offset value (ADCPPBxRESULT = ADCRESULTx - ADCSOCxOFFREF).

Returns

None

5.2.3.30 static void ADC_enableInterrupt (uint32_t *base,* **ADC_IntNumber** *adcIntNum*) [inline], [static]

Enables an ADC interrupt source.

Parameters

base	is the base address of the ADC module.
adcIntNum	is interrupt number within the ADC wrapper.

This function enables the indicated ADC interrupt source. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

adcIntNum can take the value **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, **or** ADC_INT_NUMBER4 to express which of the four interrupts of the ADC module should be enabled.

Returns
None.

5.2.3.31 static void ADC_disableInterrupt (uint32_t *base*, **ADC_IntNumber** *adcIntNum*) `[inline]`, `[static]`

Disables an ADC interrupt source.

Parameters

base	is the base address of the ADC module.
adcIntNum	is interrupt number within the ADC wrapper.

This function disables the indicated ADC interrupt source. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

adcIntNum can take the value **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, or **ADC_INT_NUMBER4** to express which of the four interrupts of the ADC module should be disabled.

Returns
None.

5.2.3.32 static void ADC_setInterruptSource (uint32_t *base*, **ADC_IntNumber** *adcIntNum*, **ADC_SOCNumber** *socNumber*) `[inline]`, `[static]`

Sets the source EOC for an analog-to-digital converter interrupt.

Parameters

base	is the base address of the ADC module.
adcIntNum	is interrupt number within the ADC wrapper.
socNumber	is the number of the start-of-conversion.

This function sets which conversion is the source of an ADC interrupt.

The *socNumber* number is a value **ADC_SOC_NUMBERX** where X is a number from 0 to 15 specifying which EOC is to be configured on the ADC module specified by *base*.

adcIntNum can take the value **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, **or** ADC_INT_NUMBER4 to express which of the four interrupts of the ADC module is being configured.

Returns
None.

5.2.3.33 static void ADC_enableContinuousMode (uint32_t *base*, **ADC_IntNumber** *adcIntNum*) [inline], [static]

Enables continuous mode for an ADC interrupt.

Parameters

base	is the base address of the ADC.
adcIntNum	is interrupt number within the ADC wrapper.

This function enables continuous mode for the ADC interrupt passed into *adcIntNum*. This means that pulses will be generated for the specified ADC interrupt whenever an EOC pulse is generated irrespective of whether or not the flag bit is set.

adcIntNum can take the value **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, **or** ADC_INT_NUMBER4 to express which of the four interrupts of the ADC module is being configured.

Returns
None.

5.2.3.34 static void ADC_disableContinuousMode (uint32_t *base*, **ADC_IntNumber** *adcIntNum*) [inline], [static]

Disables continuous mode for an ADC interrupt.

Parameters

base	is the base address of the ADC.
adcIntNum	is interrupt number within the ADC wrapper.

This function disables continuous mode for the ADC interrupt passed into *adcIntNum*. This means that pulses will not be generated for the specified ADC interrupt until the corresponding interrupt flag for the previous interrupt occurrence has been cleared using ADC_clearInterruptStatus().

adcIntNum can take the value **ADC_INT_NUMBER1**, **ADC_INT_NUMBER2**, **ADC_INT_NUMBER3**, **or** ADC_INT_NUMBER4 to express which of the four interrupts of the ADC module is being configured.

Returns
None.

5.2.3.35 static int16_t ADC_getTemperatureC (uint16_t *tempResult*, float32_t *vref*) [inline], [static]

Converts temperature from sensor reading to degrees C

Parameters

tempResult	is the raw ADC A conversion result from the temp sensor.
vref	is the reference voltage being used (for example 3.3 for 3.3V).

This function converts temperature from temp sensor reading to degrees C. Temp sensor values in production test are derived with 2.5V reference. The **vref** argument in the function is used to scale the temp sensor reading accordingly if temp sensor value is read at a different VREF setting.

Note

Only external reference mode is supported for the temperature sensor. This function does not set the reference mode. Reference mode can be set using ADC_setVREF(). using ADC_setVREF().

Returns

Returns the temperature sensor reading converted to degrees C.

5.2.3.36 static int16_t ADC_getTemperatureK (uint16_t *tempResult,* float32_t *vref*) [inline], [static]

Converts temperature from sensor reading to degrees K

Parameters

tempResult	is the raw ADC A conversion result from the temp sensor.
vref	is the reference voltage being used (for example 3.3 for 3.3V).

This function converts temperature from temp sensor reading to degrees K. Temp sensor values in production test are derived with 2.5V reference. The **vref** argument in the function is used to scale the temp sensor reading accordingly if temp sensor value is read at a different VREF setting.

Note

Only external reference mode is supported for the temperature sensor. This function does not set the reference mode. Reference mode can be set using ADC_setVREF(). using ADC_setVREF().

Returns

Returns the temperature sensor reading converted to degrees K.

5.2.3.37 void ADC_setVREF (uint32_t *base,* **ADC_ReferenceMode** *refMode,* **ADC_ReferenceVoltage** *refVoltage*)

Configures the ADC module's reference mode and offset trim

Parameters

base	is the base address of the ADC module.
refMode	is the reference mode being used (**ADC_REFERENCE_INTERNAL** or **ADC_REFERENCE_EXTERNAL**).
refVoltage	is the reference voltage being used (**ADC_REFERENCE_2_5V** or **ADC_REFERENCE_3_3V**). This is ignored when the reference mode is external.

This function configures the ADC module's reference mode and loads the corresponding offset trims.

Note

When the *refMode* parameter is **ADC_REFERENCE_EXTERNAL**, the value of the *refVoltage* parameter has no effect on the operation of the ADC.

Returns

None.

5.2.3.38 void ADC_setPPBTripLimits (uint32_t *base*, **ADC_PPBNumber** *ppbNumber*, int32_t *tripHiLimit*, int32_t *tripLoLimit*)

Sets the windowed trip limits for a PPB.

Parameters

base	is the base address of the ADC module.
ppbNumber	is the number of the post-processing block.
tripHiLimit	is the value is the digital comparator trip high limit.
tripLoLimit	is the value is the digital comparator trip low limit.

This function sets the windowed trip limits for a PPB. These values set the digital comparator so that when one of the values is exceeded, either a high or low trip event will occur.

The *ppbNumber* is a value **ADC_PPB_NUMBERX** where X is a value from 1 to 4 inclusive that identifies a PPB to be configured.

If using 16-bit mode, you may pass a 17-bit number into the *tripHiLimit* and *tripLoLimit* parameters where the 17th bit is the sign bit (that is a value from -65536 and 65535). In 12-bit mode, only bits 12:0 will be compared against bits 12:0 of the PPB result.

Returns

None.

6 ASysCtl Module

Introduction 37
API Functions 37

6.1 ASysCtl Introduction

The ASysCtl or Analog System Control driver provides functions to enable, disable and lock the temperature sensor on the device. It will also provide additional functionality if available for that device.

6.2 API Functions

Macros

- #define ASYSCTL_VREFHIA
- #define ASYSCTL_VREFHIB
- #define ASYSCTL_VREFHIC
- #define ASYSCTL_CMPHNMUX_SELECT_1
- #define ASYSCTL_CMPHNMUX_SELECT_2
- #define ASYSCTL_CMPHNMUX_SELECT_3
- #define ASYSCTL_CMPHNMUX_SELECT_4
- #define ASYSCTL_CMPHNMUX_SELECT_5
- #define ASYSCTL_CMPHNMUX_SELECT_6
- #define ASYSCTL_CMPHNMUX_SELECT_7
- #define ASYSCTL_CMPLNMUX_SELECT_1
- #define ASYSCTL_CMPLNMUX_SELECT_2
- #define ASYSCTL_CMPLNMUX_SELECT_3
- #define ASYSCTL_CMPLNMUX_SELECT_4
- #define ASYSCTL_CMPLNMUX_SELECT_5
- #define ASYSCTL_CMPLNMUX_SELECT_6
- #define ASYSCTL_CMPLNMUX_SELECT_7

Enumerations

- enum ASysCtl_CMPHPMuxSelect {
ASYSCTL_CMPHPMUX_SELECT_1, ASYSCTL_CMPHPMUX_SELECT_2,
ASYSCTL_CMPHPMUX_SELECT_3, ASYSCTL_CMPHPMUX_SELECT_4,
ASYSCTL_CMPHPMUX_SELECT_5, ASYSCTL_CMPHPMUX_SELECT_6,
ASYSCTL_CMPHPMUX_SELECT_7 }
- enum ASysCtl_CMPLPMuxSelect {
ASYSCTL_CMPLPMUX_SELECT_1, ASYSCTL_CMPLPMUX_SELECT_2,
ASYSCTL_CMPLPMUX_SELECT_3, ASYSCTL_CMPLPMUX_SELECT_4,
ASYSCTL_CMPLPMUX_SELECT_5, ASYSCTL_CMPLPMUX_SELECT_6,
ASYSCTL_CMPLPMUX_SELECT_7 }

Functions

- static void ASysCtl_enableTemperatureSensor (void)
- static void ASysCtl_disableTemperatureSensor (void)
- static void ASysCtl_setAnalogReferenceInternal (uint16_t reference)
- static void ASysCtl_setAnalogReferenceExternal (uint16_t reference)
- static void ASysCtl_setAnalogReference2P5 (uint16_t reference)
- static void ASysCtl_setAnalogReference1P65 (uint16_t reference)
- static void ASysCtl_enableDCDC (void)
- static void ASysCtl_disableDCDC (void)
- static bool ASysCtl_getInductorFaultStatus (void)
- static bool ASysCtl_getSwitchSequenceStatus (void)
- static void ASysCtl_selectCMPHNMux (uint16_t select)
- static void ASysCtl_selectCMPLNMux (uint16_t select)
- static void ASysCtl_selectCMPHPMux (ASysCtl_CMPHPMuxSelect select, uint32_t value)
- static void ASysCtl_selectCMPLPMux (ASysCtl_CMPLPMuxSelect select, uint32_t value)
- static void ASysCtl_lockTemperatureSensor (void)
- static void ASysCtl_lockANAREF (void)
- static void ASysCtl_lockVMON (void)
- static void ASysCtl_lockDCDC (void)
- static void ASysCtl_lockPGAADCINMux (void)
- static void ASysCtl_lockCMPHPMux (void)
- static void ASysCtl_lockCMPLPMux (void)
- static void ASysCtl_lockCMPHNMux (void)
- static void ASysCtl_lockCMPLNMux (void)
- static void ASysCtl_lockVREG (void)

6.2.1 Detailed Description

The code for this module is contained in `driverlib/asysctl.c`, with `driverlib/asysctl.h` containing the API declarations for use by applications.

6.2.2 Enumeration Type Documentation

6.2.2.1 enum **ASysCtl_CMPHPMuxSelect**

ASysCtl_CMPHPMuxSelect used for function ASysCtl_selectCMPHPMux().

Enumerator

ASYSCTL_CMPHPMUX_SELECT_1 CMPHPMUX select 1.
ASYSCTL_CMPHPMUX_SELECT_2 CMPHPMUX select 2.
ASYSCTL_CMPHPMUX_SELECT_3 CMPHPMUX select 3.
ASYSCTL_CMPHPMUX_SELECT_4 CMPHPMUX select 4.
ASYSCTL_CMPHPMUX_SELECT_5 CMPHPMUX select 5.
ASYSCTL_CMPHPMUX_SELECT_6 CMPHPMUX select 6.
ASYSCTL_CMPHPMUX_SELECT_7 CMPHPMUX select 7.

6.2.2.2 enum **ASysCtl_CMPLPMuxSelect**

ASysCtl_CMPLPMuxSelect used for function ASysCtl_selectCMPLPMux().

Enumerator

ASYSCTL_CMPLPMUX_SELECT_1 CMPLPMUX select 1.
ASYSCTL_CMPLPMUX_SELECT_2 CMPLPMUX select 2.
ASYSCTL_CMPLPMUX_SELECT_3 CMPLPMUX select 3.
ASYSCTL_CMPLPMUX_SELECT_4 CMPLPMUX select 4.
ASYSCTL_CMPLPMUX_SELECT_5 CMPLPMUX select 5.
ASYSCTL_CMPLPMUX_SELECT_6 CMPLPMUX select 6.
ASYSCTL_CMPLPMUX_SELECT_7 CMPLPMUX select 7.

6.2.3 Function Documentation

6.2.3.1 static void ASysCtl_enableTemperatureSensor (void) `[inline]`, `[static]`

Enable temperature sensor.

This function enables the temperature sensor output to the ADC.

Returns

None.

6.2.3.2 static void ASysCtl_disableTemperatureSensor (void) `[inline]`, `[static]`

Disable temperature sensor.

This function disables the temperature sensor output to the ADC.

Returns

None.

6.2.3.3 static void ASysCtl_setAnalogReferenceInternal (uint16_t *reference*) `[inline]`, `[static]`

Set the analog voltage reference selection to internal.

Parameters

reference	is the analog reference.

The parameter *reference* can be a combination of the following values:

- **ASYSCTL_VREFHIA**
- **ASYSCTL_VREFHIB**
- **ASYSCTL_VREFHIC**

Returns
None.

6.2.3.4 static void ASysCtl_setAnalogReferenceExternal (uint16_t *reference*) [inline], [static]

Set the analog voltage reference selection to external.

Parameters

reference	is the analog reference.

The parameter *reference* can be a combination of the following values:

- **ASYSCTL_VREFHIA**
- **ASYSCTL_VREFHIB**
- **ASYSCTL_VREFHIC**

Returns
None.

6.2.3.5 static void ASysCtl_setAnalogReference2P5 (uint16_t *reference*) [inline], [static]

Set the external analog voltage reference selection to 2.5V.

Parameters

reference	is the analog reference.

The parameter *reference* can be a combination of the following values:

- **ASYSCTL_VREFHIA**
- **ASYSCTL_VREFHIB**
- **ASYSCTL_VREFHIC**

Returns
None.

6.2.3.6 static void ASysCtl_setAnalogReference1P65 (uint16_t *reference*) [inline], [static]

Set the external analog voltage reference selection to 1.65V.

Parameters

reference	is the analog reference.

The parameter *reference* can be a combination of the following values:

- **ASYSCTL_VREFHIA**
- **ASYSCTL_VREFHIB**
- **ASYSCTL_VREFHIC**

Returns
None.

6.2.3.7 static void ASysCtl_enableDCDC (void) [inline], [static]

Enable DC-DC.

Returns
None.

6.2.3.8 static void ASysCtl_disableDCDC (void) [inline], [static]

Disable DC-DC.

Returns
None.

6.2.3.9 static bool ASysCtl_getInductorFaultStatus (void) [inline], [static]

Gets the inductor status.

This function returns the inductor status.

Returns
Return value **true** indicates that the external inductor connected to DC-DC is functional. Return value of **false** indicates it is faulty or not connected.

6.2.3.10 static bool ASysCtl_getSwitchSequenceStatus (void) [inline], [static]

Gets the Switch Sequence Status.

This function returns the Switch Sequence Status.

Returns
Return value **false** indicates that the switch to DC-DC is not complete. Return value of **true** indicates it is complete.

6.2.3.11 static void ASysCtl_selectCMPHNMux (uint16_t *select*) [inline], [static]

Select the value for CMPHNMXSEL.

Parameters

select	is a combination of CMPHNMXSEL values.

The parameter *select* can be a bitwise OR of the below values:

- **ASYSCTL_CMPHNMUX_SELECT_1**
- **ASYSCTL_CMPHNMUX_SELECT_2**
- **ASYSCTL_CMPHNMUX_SELECT_3**
- **ASYSCTL_CMPHNMUX_SELECT_4**
- **ASYSCTL_CMPHNMUX_SELECT_5**
- **ASYSCTL_CMPHNMUX_SELECT_6**
- **ASYSCTL_CMPHNMUX_SELECT_7**

Returns
None.

6.2.3.12 static void ASysCtl_selectCMPLNMux (uint16_t *select*) [inline], [static]

Select the value for CMPLNMXSEL.

Parameters

select	is a combination of CMPLNMXSEL values.

The parameter *select* can be the bitwise OR of the below values:

- **ASYSCTL_CMPLNMUX_SELECT_1**
- **ASYSCTL_CMPLNMUX_SELECT_2**
- **ASYSCTL_CMPLNMUX_SELECT_3**
- **ASYSCTL_CMPLNMUX_SELECT_4**
- **ASYSCTL_CMPLNMUX_SELECT_5**
- **ASYSCTL_CMPLNMUX_SELECT_6**
- **ASYSCTL_CMPLNMUX_SELECT_7**

Returns
None.

6.2.3.13 static void ASysCtl_selectCMPHPMux (**ASysCtl_CMPHPMuxSelect** *select,* uint32_t *value*) [inline], [static]

Select the value for CMPHPMXSEL.

Parameters

select	is of type ASysCtl_CMPHPMuxSelect.
value	is 0, 1, 2, 3, or 4.

This function is used to write a value to one mux select at a time. The parameter *select* can be one of the following values:

- **ASYSCTL_CMPHPMUX_SELECT_1**
- **ASYSCTL_CMPHPMUX_SELECT_2**
- **ASYSCTL_CMPHPMUX_SELECT_3**
- **ASYSCTL_CMPHPMUX_SELECT_4**
- **ASYSCTL_CMPHPMUX_SELECT_5**
- **ASYSCTL_CMPHPMUX_SELECT_6**
- **ASYSCTL_CMPHPMUX_SELECT_7**

Returns
None.

6.2.3.14 static void ASysCtl_selectCMPLPMux (**ASysCtl_CMPLPMuxSelect** *select*, uint32_t *value*) [inline], [static]

Select the value for CMPLPMXSEL.

Parameters

select	is of type ASysCtl_CMPLPMuxSelect.
value	is 0, 1, 2, 3, or 4.

This function is used to write a value to one mux select at a time. The parameter *select* can be one of the following values:

- **ASYSCTL_CMPLPMUX_SELECT_1**
- **ASYSCTL_CMPLPMUX_SELECT_2**
- **ASYSCTL_CMPLPMUX_SELECT_3**
- **ASYSCTL_CMPLPMUX_SELECT_4**
- **ASYSCTL_CMPLPMUX_SELECT_5**
- **ASYSCTL_CMPLPMUX_SELECT_6**
- **ASYSCTL_CMPLPMUX_SELECT_7**

Returns
None.

6.2.3.15 static void ASysCtl_lockTemperatureSensor (void) [inline], [static]

Locks the temperature sensor control register.

Returns
None.

6.2.3.16 static void ASysCtl_lockANAREF (void) [inline], [static]

Locks the analog reference control register.

Returns
None.

6.2.3.17 static void ASysCtl_lockVMON (void) [inline], [static]

Locks the voltage monitor control register.

Returns
None.

6.2.3.18 static void ASysCtl_lockDCDC (void) [inline], [static]

Locks the DCDC control register.

Returns
None.

6.2.3.19 static void ASysCtl_lockPGAADCINMux (void) [inline], [static]

Locks the ADCIN control register.

Returns
None.

6.2.3.20 static void ASysCtl_lockCMPHPMux (void) [inline], [static]

Locks the CMPHPMXSEL control register.

Returns
None.

6.2.3.21 static void ASysCtl_lockCMPLPMux (void) [inline], [static]

Locks the CMPLPMXSEL control register.

Returns
None.

6.2.3.22 static void ASysCtl_lockCMPHNMux (void) [inline], [static]

Locks the CMPHNMXSEL control register.

Returns
None.

6.2.3.23 static void ASysCtl_lockCMPLNMux (void) [inline], [static]

Locks the CMPLNMXSEL control register.

Returns
None.

6.2.3.24 static void ASysCtl_lockVREG (void) [inline], [static]

Locks the VREG control register.

Returns
None.

7 CAN Module

Introduction . . . 46
API Functions . . . 46

7.1 CAN Introduction

The controller area network (CAN) API provides a set of functions for configuring and using the CAN module, a serial communications protocol. Functions are provided to setup and configure the module operating options, setup the different types of message objects, send and read messages, and setup and handle interrupts and events.

7.2 API Functions

Enumerations

- enum CAN_MsgFrameType { CAN_MSG_FRAME_STD, CAN_MSG_FRAME_EXT }
- enum CAN_MsgObjType { CAN_MSG_OBJ_TYPE_TX, CAN_MSG_OBJ_TYPE_TX_REMOTE, CAN_MSG_OBJ_TYPE_RX, CAN_MSG_OBJ_TYPE_RXTX_REMOTE }
- enum CAN_ClockSource { CAN_CLOCK_SOURCE_SYS, CAN_CLOCK_SOURCE_XTAL, CAN_CLOCK_SOURCE_AUX }

Functions

- static void CAN_initRAM (uint32_t base)
- static void CAN_selectClockSource (uint32_t base, CAN_ClockSource source)
- static void CAN_startModule (uint32_t base)
- static void CAN_enableController (uint32_t base)
- static void CAN_disableController (uint32_t base)
- static void CAN_enableTestMode (uint32_t base, uint16_t mode)
- static void CAN_disableTestMode (uint32_t base)
- static uint32_t CAN_getBitTiming (uint32_t base)
- static void CAN_enableMemoryAccessMode (uint32_t base)
- static void CAN_disableMemoryAccessMode (uint32_t base)
- static void CAN_setInterruptionDebugMode (uint32_t base, bool enable)
- static void CAN_enableDMARequests (uint32_t base)
- static void CAN_disableDMARequests (uint32_t base)
- static void CAN_disableAutoBusOn (uint32_t base)
- static void CAN_enableAutoBusOn (uint32_t base)
- static void CAN_setAutoBusOnTime (uint32_t base, uint32_t time)
- static void CAN_enableInterrupt (uint32_t base, uint32_t intFlags)
- static void CAN_disableInterrupt (uint32_t base, uint32_t intFlags)
- static uint32_t CAN_getInterruptMux (uint32_t base)
- static void CAN_setInterruptMux (uint32_t base, uint32_t mux)
- static void CAN_enableRetry (uint32_t base)
- static void CAN_disableRetry (uint32_t base)
- static bool CAN_isRetryEnabled (uint32_t base)

- static bool CAN_getErrorCount (uint32_t base, uint32_t ∗rxCount, uint32_t ∗txCount)
- static uint16_t CAN_getStatus (uint32_t base)
- static uint32_t CAN_getTxRequests (uint32_t base)
- static uint32_t CAN_getNewDataFlags (uint32_t base)
- static uint32_t CAN_getValidMessageObjects (uint32_t base)
- static uint32_t CAN_getInterruptCause (uint32_t base)
- static uint32_t CAN_getInterruptMessageSource (uint32_t base)
- static void CAN_enableGlobalInterrupt (uint32_t base, uint16_t intFlags)
- static void CAN_disableGlobalInterrupt (uint32_t base, uint16_t intFlags)
- static void CAN_clearGlobalInterruptStatus (uint32_t base, uint16_t intFlags)
- static bool CAN_getGlobalInterruptStatus (uint32_t base, uint16_t intFlags)
- void CAN_initModule (uint32_t base)
- void CAN_setBitRate (uint32_t base, uint32_t clock, uint32_t bitRate, uint16_t bitTime)
- void CAN_setBitTiming (uint32_t base, uint16_t prescaler, uint16_t prescalerExtension, uint16_t tSeg1, uint16_t tSeg2, uint16_t sjw)
- void CAN_clearInterruptStatus (uint32_t base, uint32_t intClr)
- void CAN_setupMessageObject (uint32_t base, uint32_t objID, uint32_t msgID, CAN_MsgFrameType frame, CAN_MsgObjType msgType, uint32_t msgIDMask, uint32_t flags, uint16_t msgLen)
- void CAN_sendMessage (uint32_t base, uint32_t objID, uint16_t msgLen, const uint16_t ∗msgData)
- bool CAN_readMessage (uint32_t base, uint32_t objID, uint16_t ∗msgData)
- void CAN_transferMessage (uint32_t base, uint16_t interface, uint32_t objID, bool direction, bool dmaRequest)
- void CAN_clearMessage (uint32_t base, uint32_t objID)

7.2.1 Detailed Description

The following describes important details and recommendations when using the CAN API.

Once system control enables the CAN module, **CAN_initModule()** needs to be called with the desired CAN module base to put the controller in the init state, initialize the message RAM, and enable access to the configuration registers. Next, use **CAN_setBitRate()** to set the CAN bit timing values for the bit rate and timing parameters. For tighter timing requirements, use **CAN_setBitTiming()** instead.

To setup any of the types of message objects, use **CAN_setupMessageObject()**.

Once all of the module configurations are setup, **CAN_startModule()** starts the CAN module's operations and disables access to the configuration registers.

If the application needs to disable message processing on the CAN controller, use **CAN_disableController()** to disable the message processing. Message processing can be re-enabled using **CAN_enableController()**.

The code for this module is contained in `driverlib/can.c`, with `driverlib/can.h` containing the API declarations for use by applications.

7.2.2 Enumeration Type Documentation

7.2.2.1 enum **CAN_MsgFrameType**

This data type is used to identify the interrupt status register. This is used when calling the CAN_setupMessageObject() function.

Enumerator

CAN_MSG_FRAME_STD Set the message ID frame to standard.

CAN_MSG_FRAME_EXT Set the message ID frame to extended.

7.2.2.2 enum **CAN_MsgObjType**

This definition is used to determine the type of message object that will be set up via a call to the CAN_setupMessageObject() API.

Enumerator

CAN_MSG_OBJ_TYPE_TX Transmit message object.

CAN_MSG_OBJ_TYPE_TX_REMOTE Transmit remote request message object.

CAN_MSG_OBJ_TYPE_RX Receive message object.

CAN_MSG_OBJ_TYPE_RXTX_REMOTE Remote frame receive remote, with auto-transmit message object.

7.2.2.3 enum **CAN_ClockSource**

This definition is used to determine the clock source that will be set up via a call to the CAN_selectClockSource() API.

Enumerator

CAN_CLOCK_SOURCE_SYS Peripheral System Clock Source.

CAN_CLOCK_SOURCE_XTAL External Oscillator Clock Source.

CAN_CLOCK_SOURCE_AUX Auxiliary Clock Input Source.

7.2.3 Function Documentation

7.2.3.1 static void CAN_initRAM (uint32_t *base*) `[inline]`, `[static]`

Initializes the CAN controller's RAM.

Parameters

base	is the base address of the CAN controller.

Performs the initialization of the RAM used for the CAN message objects.

Returns

None.

Referenced by CAN_initModule().

7.2.3.2 static void CAN_selectClockSource (uint32_t *base*, **CAN_ClockSource** *source*) `[inline]`, `[static]`

Select CAN Clock Source

Parameters

base	is the base address of the CAN controller.
source	is the clock source to use for the CAN controller.

This function selects the specified clock source for the CAN controller.

The *source* parameter can be any one of the following:

- **CAN_CLOCK_SOURCE_SYS** - Peripheral System Clock
- **CAN_CLOCK_SOURCE_XTAL** - External Oscillator
- **CAN_CLOCK_SOURCE_AUX** - Auxiliary Clock Input from GPIO

Returns
None.

7.2.3.3 static void CAN_startModule (uint32_t *base*) [inline], [static]

Starts the CAN Module's Operations

Parameters

base	is the base address of the CAN controller.

This function starts the CAN module's operations after initialization, which includes the CAN protocol controller state machine of the CAN core and the message handler state machine to begin controlling the CAN's internal data flow.

Returns
None.

7.2.3.4 static void CAN_enableController (uint32_t *base*) [inline], [static]

Enables the CAN controller.

Parameters

base	is the base address of the CAN controller to enable.

Enables the CAN controller for message processing. Once enabled, the controller will automatically transmit any pending frames, and process any received frames. The controller can be stopped by calling CAN_disableController().

Returns
None.

7.2.3.5 static void CAN_disableController (uint32_t *base*) [inline], [static]

Disables the CAN controller.

Parameters

base	is the base address of the CAN controller to disable.

Disables the CAN controller for message processing. When disabled, the controller will no longer automatically process data on the CAN bus. The controller can be restarted by calling CAN_enableController(). The state of the CAN controller and the message objects in the controller are left as they were before this call was made.

Returns

None.

7.2.3.6 static void CAN_enableTestMode (uint32_t *base,* uint16_t *mode*) [inline], [static]

Enables the test modes of the CAN controller.

Parameters

base	is the base address of the CAN controller.
mode	are the the test modes to enable.

Enables test modes within the controller. The following valid options for *mode* can be OR'ed together:

- **CAN_TEST_SILENT** - Silent Mode
- **CAN_TEST_LBACK** - Loopback Mode
- **CAN_TEST_EXL** - External Loopback Mode

Note

Loopback mode and external loopback mode **can not** be enabled at the same time.

Returns

None.

7.2.3.7 static void CAN_disableTestMode (uint32_t *base*) [inline], [static]

Disables the test modes of the CAN controller.

Parameters

base	is the base address of the CAN controller.

Disables test modes within the controller and clears the test bits.

Returns

None.

7.2.3.8 static uint32_t CAN_getBitTiming (uint32_t *base*) [inline], [static]

Get the current settings for the CAN controller bit timing.

Parameters

base	is the base address of the CAN controller.

This function reads the current configuration of the CAN controller bit clock timing.

Returns

Returns the value of the bit timing register.

7.2.3.9 static void CAN_enableMemoryAccessMode (uint32_t *base*) [inline], [static]

Enables direct access to the RAM.

Parameters

base	is the base address of the CAN controller.

Enables direct access to the RAM while in Test mode.

Note

Test Mode must first be enabled to use this function.

Returns

None.

7.2.3.10 static void CAN_disableMemoryAccessMode (uint32_t *base*) [inline], [static]

Disables direct access to the RAM.

Parameters

base	is the base address of the CAN controller.

Disables direct access to the RAM while in Test mode.

Returns

None.

7.2.3.11 static void CAN_setInterruptionDebugMode (uint32_t *base,* bool *enable*) [inline], [static]

Sets the interruption debug mode of the CAN controller.

Parameters

base	is the base address of the CAN controller.

enable	is a flag to enable or disable the interruption debug mode.

This function sets the interruption debug mode of the CAN controller. When the *enable* parameter is **true**, CAN will be configured to interrupt any transmission or reception and enter debug mode immediately after it is requested. When **false**, CAN will wait for a started transmission or reception to be completed before entering debug mode.

Returns
None.

7.2.3.12 static void CAN_enableDMARequests (uint32_t *base*) `[inline]`, `[static]`

Enables DMA Requests from the CAN controller.

Parameters

base	is the base address of the CAN controller to enable.

Enables the CAN controller DMA request lines for each of the 3 interface register sets. To actually assert the request line, the DMA Active bit must be set in the corresponding interface CMD register.

Returns
None.

7.2.3.13 static void CAN_disableDMARequests (uint32_t *base*) `[inline]`, `[static]`

Disables DMA Requests from the CAN controller.

Parameters

base	is the base address of the CAN controller to enable.

Disables the CAN controller DMA request lines for each of the 3 interface register sets.

Returns
None.

7.2.3.14 static void CAN_disableAutoBusOn (uint32_t *base*) `[inline]`, `[static]`

Disables Auto-Bus-On.

Parameters

base	is the base address of the CAN controller.

Disables the Auto-Bus-On feature of the CAN controller.

Returns
None.

7.2.3.15 static void CAN_enableAutoBusOn (uint32_t *base*) [inline], [static]

Enables Auto-Bus-On.

Parameters

base	is the base address of the CAN controller.

Enables the Auto-Bus-On feature of the CAN controller. Be sure to also configure the Auto-Bus-On time using the CAN_setAutoBusOnTime function.

Returns

None.

7.2.3.16 static void CAN_setAutoBusOnTime (uint32_t *base*, uint32_t *time*) `[inline]`, `[static]`

Sets the time before a Bus-Off recovery sequence is started.

Parameters

base	is the base address of the CAN controller.
time	is number of clock cycles before a Bus-Off recovery sequence is started.

This function sets the number of clock cycles before a Bus-Off recovery sequence is started by clearing the Init bit.

Note

To enable this functionality, use CAN_enableAutoBusOn().

Returns

None.

7.2.3.17 static void CAN_enableInterrupt (uint32_t *base*, uint32_t *intFlags*) `[inline]`, `[static]`

Enables individual CAN controller interrupt sources.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be enabled.

Enables specific interrupt sources of the CAN controller. Only enabled sources will cause a processor interrupt.

The *intFlags* parameter is the logical OR of any of the following:

- **CAN_INT_ERROR** - a controller error condition has occurred
- **CAN_INT_STATUS** - a message transfer has completed, or a bus error has been detected
- **CAN_INT_IE0** - allow CAN controller to generate interrupts on interrupt line 0
- **CAN_INT_IE1** - allow CAN controller to generate interrupts on interrupt line 1

Returns

None.

7.2.3.18 static void CAN_disableInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Disables individual CAN controller interrupt sources.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be disabled.

Disables the specified CAN controller interrupt sources. Only enabled interrupt sources can cause a processor interrupt.

The *intFlags* parameter has the same definition as in the CAN_enableInterrupt() function.

Returns

None.

7.2.3.19 static uint32_t CAN_getInterruptMux (uint32_t *base*) [inline], [static]

Get the CAN controller Interrupt Line set for each mailbox

Parameters

base	is the base address of the CAN controller.

Gets which interrupt line each message object should assert when an interrupt occurs. Bit 0 corresponds to message object 32 and then bits 1 to 31 correspond to message object 1 through 31 respectively. Bits that are asserted indicate the message object should generate an interrupt on interrupt line 1, while bits that are not asserted indicate the message object should generate an interrupt on line 0.

Returns

Returns the value of the interrupt muxing register.

7.2.3.20 static void CAN_setInterruptMux (uint32_t *base,* uint32_t *mux*) [inline], [static]

Set the CAN controller Interrupt Line for each mailbox

Parameters

base	is the base address of the CAN controller.
mux	bit packed representation of which message objects should generate an interrupt on a given interrupt line.

Selects which interrupt line each message object should assert when an interrupt occurs. Bit 0 corresponds to message object 32 and then bits 1 to 31 correspond to message object 1 through 31 respectively. Bits that are asserted indicate the message object should generate an interrupt on interrupt line 1, while bits that are not asserted indicate the message object should generate an interrupt on line 0.

Returns

None.

7.2.3.21 static void CAN_enableRetry (uint32_t *base*) [inline], [static]

Enables the CAN controller automatic retransmission behavior.

Parameters

base	is the base address of the CAN controller.

Enables the automatic retransmission of messages with detected errors.

Returns

None.

7.2.3.22 static void CAN_disableRetry (uint32_t *base*) [inline], [static]

Disables the CAN controller automatic retransmission behavior.

Parameters

base	is the base address of the CAN controller.

Disables the automatic retransmission of messages with detected errors.

Returns

None.

7.2.3.23 static bool CAN_isRetryEnabled (uint32_t *base*) [inline], [static]

Returns the current setting for automatic retransmission.

Parameters

base	is the base address of the CAN controller.

Reads the current setting for the automatic retransmission in the CAN controller and returns it to the caller.

Returns

Returns **true** if automatic retransmission is enabled, **false** otherwise.

7.2.3.24 static bool CAN_getErrorCount (uint32_t *base,* uint32_t * *rxCount,* uint32_t * *txCount*) [inline], [static]

Reads the CAN controller error counter register.

Parameters

base	is the base address of the CAN controller.
rxCount	is a pointer to storage for the receive error counter.
txCount	is a pointer to storage for the transmit error counter.

Reads the error counter register and returns the transmit and receive error counts to the caller along with a flag indicating if the controller receive counter has reached the error passive limit. The values of the receive and transmit error counters are returned through the pointers provided as parameters.

After this call, *rxCount* will hold the current receive error count and *txCount* will hold the current transmit error count.

Returns

Returns **true** if the receive error count has reached the error passive limit, and **false** if the error count is below the error passive limit.

7.2.3.25 static uint16_t CAN_getStatus (uint32_t *base*) [inline], [static]

Reads the CAN controller error and status register.

Parameters

base	is the base address of the CAN controller.

Reads the error and status register of the CAN controller.

Returns

Returns the value of the register.

7.2.3.26 static uint32_t CAN_getTxRequests (uint32_t *base*) [inline], [static]

Reads the CAN controller TX request register.

Parameters

base	is the base address of the CAN controller.

Reads the TX request register of the CAN controller.

Returns

Returns the value of the register.

7.2.3.27 static uint32_t CAN_getNewDataFlags (uint32_t *base*) [inline], [static]

Reads the CAN controller new data status register.

Parameters

base	is the base address of the CAN controller.

Reads the new data status register of the CAN controller for all message objects.

Returns

Returns the value of the register.

7.2.3.28 static uint32_t CAN_getValidMessageObjects (uint32_t *base*) [inline], [static]

Reads the CAN controller valid message object register.

Parameters

base	is the base address of the CAN controller.

Reads the valid message object register of the CAN controller.

Returns

Returns the value of the register.

7.2.3.29 static uint32_t CAN_getInterruptCause (uint32_t *base*) `[inline]`, `[static]`

Get the CAN controller interrupt cause.

Parameters

base	is the base address of the CAN controller.

This function returns the value of the interrupt register that indicates the cause of the interrupt.

Returns

Returns the value of the interrupt register.

7.2.3.30 static uint32_t CAN_getInterruptMessageSource (uint32_t *base*) `[inline]`, `[static]`

Get the CAN controller pending interrupt message source.

Parameters

base	is the base address of the CAN controller.

Returns the value of the pending interrupts register that indicates which messages are the source of pending interrupts.

Returns

Returns the value of the pending interrupts register.

7.2.3.31 static void CAN_enableGlobalInterrupt (uint32_t *base*, uint16_t *intFlags*) `[inline]`, `[static]`

CAN Global interrupt Enable function.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be enabled.

Enables specific CAN interrupt in the global interrupt enable register

The *intFlags* parameter is the logical OR of any of the following:

- **CAN_GLOBAL_INT_CANINT0** - Global Interrupt Enable bit for CAN INT0
- **CAN_GLOBAL_INT_CANINT1** - Global Interrupt Enable bit for CAN INT1

Returns
None.

7.2.3.32 static void CAN_disableGlobalInterrupt (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

CAN Global interrupt Disable function.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be disabled.

Disables the specific CAN interrupt in the global interrupt enable register

The *intFlags* parameter is the logical OR of any of the following:

- **CAN_GLOBAL_INT_CANINT0** - Global Interrupt bit for CAN INT0
- **CAN_GLOBAL_INT_CANINT1** - Global Interrupt bit for CAN INT1

Returns
None.

7.2.3.33 static void CAN_clearGlobalInterruptStatus (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

CAN Global interrupt Clear function.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be cleared.

Clear the specific CAN interrupt bit in the global interrupt flag register.

The *intFlags* parameter is the logical OR of any of the following:

- **CAN_GLOBAL_INT_CANINT0** - Global Interrupt bit for CAN INT0
- **CAN_GLOBAL_INT_CANINT1** - Global Interrupt bit for CAN INT1

Returns
None.

7.2.3.34 static bool CAN_getGlobalInterruptStatus (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

Get the CAN Global Interrupt status.

Parameters

base	is the base address of the CAN controller.
intFlags	is the bit mask of the interrupt sources to be enabled.

Check if any interrupt bit is set in the global interrupt flag register.

The *intFlags* parameter is the logical OR of any of the following:

- **CAN_GLOBAL_INT_CANINT0** - Global Interrupt bit for CAN INT0
- **CAN_GLOBAL_INT_CANINT1** - Global Interrupt bit for CAN INT1

Returns

True if any of the requested interrupt bits are set. False, if none of the requested bits are set.

7.2.3.35 void CAN_initModule (uint32_t *base*)

Initializes the CAN controller

Parameters

base	is the base address of the CAN controller.

This function initializes the message RAM, which also clears all the message objects, and places the CAN controller in an init state. Write access to the configuration registers is available as a result, allowing the bit timing and message objects to be setup.

Note

To exit the initialization mode and start the CAN module, use the CAN_startModule() function.

Returns

None.

References CAN_initRAM(), and SysCtl_delay().

7.2.3.36 void CAN_setBitRate (uint32_t *base,* uint32_t *clock,* uint32_t *bitRate,* uint16_t *bitTime*)

Sets the CAN Bit Timing based on requested Bit Rate.

Parameters

base	is the base address of the CAN controller.
clock	is the CAN module clock frequency before the bit rate prescaler (Hertz)
bitRate	is the desired bit rate (bits/sec)
bitTime	is the number of time quanta per bit required for desired bit time (Tq) and must be in the range from 8 to 25

This function sets the CAN bit timing values for the bit rate passed in the *bitRate* and *bitTime* parameters based on the *clock* parameter. The CAN bit clock is calculated to be an average timing value that should work for most systems. If tighter timing requirements are needed, then the CAN_setBitTiming() function is available for full customization of all of the CAN bit timing values.

Returns
None.

References CAN_setBitTiming().

7.2.3.37 void CAN_setBitTiming (uint32_t *base,* uint16_t *prescaler,* uint16_t *prescalerExtension,* uint16_t *tSeg1,* uint16_t *tSeg2,* uint16_t *sjw*)

Manually set the CAN controller bit timing.

Parameters

base	is the base address of the CAN controller.
prescaler	is the baud rate prescaler
prescalerExtension	is the baud rate prescaler extension
tSeg1	is the time segment 1
tSeg2	is the time segment 2
sjw	is the synchronization jump width

This function sets the various timing parameters for the CAN bus bit timing: baud rate prescaler, prescaler extension, time segment 1, time segment 2, and the Synchronization Jump Width.

Returns
None.

Referenced by CAN_setBitRate().

7.2.3.38 void CAN_clearInterruptStatus (uint32_t *base,* uint32_t *intClr*)

Clears a CAN interrupt source.

Parameters

base	is the base address of the CAN controller.
intClr	is a value indicating which interrupt source to clear.

This function can be used to clear a specific interrupt source. The *intClr* parameter should be either a number from 1 to 32 to clear a specific message object interrupt or can be the following:

- **CAN_INT_INT0ID_STATUS** - Clears a status interrupt.

It is not necessary to use this function to clear an interrupt. This should only be used if the application wants to clear an interrupt source without taking the normal interrupt action.

Returns
None.

7.2.3.39 void CAN_setupMessageObject (uint32_t *base,* uint32_t *objID,* uint32_t *msgID,* **CAN_MsgFrameType** *frame,* **CAN_MsgObjType** *msgType,* uint32_t *msgIDMask,* uint32_t *flags,* uint16_t *msgLen*)

Setup a Message Object

Parameters

base	is the base address of the CAN controller.
objID	is the message object number to configure (1-32).
msgID	is the CAN message identifier used for the 11 or 29 bit identifiers
frame	is the CAN ID frame type
msgType	is the message object type
msgIDMask	is the CAN message identifier mask used when identifier filtering is enabled
flags	is the various flags and settings to be set for the message object
msgLen	is the number of bytes of data in the message object (0-8)

This function sets the various values required for a message object.

The *frame* parameter can be one of the following values:

- **CAN_MSG_FRAME_STD** - Standard 11 bit identifier
- **CAN_MSG_FRAME_EXT** - Extended 29 bit identifier

The *msgType* parameter can be one of the following values:

- **CAN_MSG_OBJ_TYPE_TX** - Transmit Message
- **CAN_MSG_OBJ_TYPE_TX_REMOTE** - Transmit Remote Message
- **CAN_MSG_OBJ_TYPE_RX** - Receive Message
- **CAN_MSG_OBJ_TYPE_RXTX_REMOTE** - Receive Remote message with auto-transmit

The *flags* parameter can be set as **CAN_MSG_OBJ_NO_FLAGS** if no flags are required or the parameter can be a logical OR of any of the following values:

- **CAN_MSG_OBJ_TX_INT_ENABLE** - Enable Transmit Interrupts
- **CAN_MSG_OBJ_RX_INT_ENABLE** - Enable Receive Interrupts
- **CAN_MSG_OBJ_USE_ID_FILTER** - Use filtering based on the Message ID
- **CAN_MSG_OBJ_USE_EXT_FILTER** - Use filtering based on the Extended Message ID
- **CAN_MSG_OBJ_USE_DIR_FILTER** - Use filtering based on the direction of the transfer
- **CAN_MSG_OBJ_FIFO** - Message object is part of a FIFO structure and isn't the final message object in FIFO

Returns

None.

References CAN_MSG_FRAME_EXT, CAN_MSG_OBJ_TYPE_RXTX_REMOTE, and CAN_MSG_OBJ_TYPE_TX.

7.2.3.40 void CAN_sendMessage (uint32_t *base,* uint32_t *objID,* uint16_t *msgLen,* const uint16_t ∗ *msgData*)

Sends a Message Object

Parameters

base	is the base address of the CAN controller.
objID	is the object number to configure (1-32).
msgLen	is the number of bytes of data in the message object (0-8)
msgData	is a pointer to the message object's data

This function is used to transmit a message object and the message data, if applicable.

Note

The message object requested by the *objID* must first be setup using the CAN_setupMessageObject() function.

Returns

None.

7.2.3.41 bool CAN_readMessage (uint32_t *base,* uint32_t *objID,* uint16_t * *msgData*)

Reads the data in a Message Object

Parameters

base	is the base address of the CAN controller.
objID	is the object number to read (1-32).
msgData	is a pointer to the array to store the message data

This function is used to read the data contents of the specified message object in the CAN controller. The data returned is stored in the *msgData* parameter.

Note

1. The message object requested by the *objID* must first be setup using the CAN_setupMessageObject() function.
2. If the DLC of the received message is larger than the *msgData* buffer provided, then it is possible for a buffer overflow to occur.

Returns

Returns **true** if new data was retrieved, else returns **false** to indicate no new data was retrieved.

7.2.3.42 void CAN_transferMessage (uint32_t *base,* uint16_t *interface,* uint32_t *objID,* bool *direction,* bool *dmaRequest*)

Transfers a CAN message between the IF registers and Message RAM.

Parameters

base	is the base address of the CAN controller.

interface	is the interface to use for the transfer. Valid value are 1 or 2.
objID	is the object number to transfer (1-32).
direction	is the of the transfer. False is Message RAM to IF, True is IF to Message RAM.
dmaRequest	asserts the DMA request line after a transfer if set to True.

This function transfers the contents of the interface registers to message RAM or vice versa depending on the value passed to direction. This function is designed to be used with DMA transfers.

Returns
None.

7.2.3.43 void CAN_clearMessage (uint32_t *base,* uint32_t *objID*)

Clears a message object so that it is no longer used.

Parameters

base	is the base address of the CAN controller.
objID	is the message object number to disable (1-32).

This function frees the specified message object from use. Once a message object has been cleared, it will no longer automatically send or receive messages, or generate interrupts.

Returns
None.

8 CLA Module

Introduction . . . 67
API Functions . . . 67

8.1 CLA Introduction

The Control Law Accelerator (CLA) API provides a set of functions to configure the CLA. The CLA is an independent accelerator with its own buses, ALU and register set. It does share memory, both program and data, with the main processor; it comes out of a power reset with no memory assets and therefore the C28x must configure how the CLA runs, which memory spaces it uses, and when code must run.

The primary use of the CLA is to implement small, fast control loops that run periodically, responding to specific trigger sources like the PWM or an ADC conversion in a deterministic (fixed and low latency) fashion.

8.2 API Functions

Macros

- #define CLA_TASKFLAG_1
- #define CLA_TASKFLAG_2
- #define CLA_TASKFLAG_3
- #define CLA_TASKFLAG_4
- #define CLA_TASKFLAG_5
- #define CLA_TASKFLAG_6
- #define CLA_TASKFLAG_7
- #define CLA_TASKFLAG_8
- #define CLA_TASKFLAG_ALL

Enumerations

- enum CLA_BGTaskStatus { CLA_BGSTS_RUNNING, CLA_BGSTS_CANNOT_INTERRUPT, CLA_BGSTS_OVERFLOW }
- enum CLA_TaskNumber { CLA_TASK_1, CLA_TASK_2, CLA_TASK_3, CLA_TASK_4, CLA_TASK_5, CLA_TASK_6, CLA_TASK_7, CLA_TASK_8 }
- enum CLA_MVECTNumber { CLA_MVECT_1, CLA_MVECT_2, CLA_MVECT_3, CLA_MVECT_4, CLA_MVECT_5, CLA_MVECT_6, CLA_MVECT_7, CLA_MVECT_8 }
- enum CLA_Trigger { CLA_TRIGGER_SOFTWARE, CLA_TRIGGER_ADCA1, CLA_TRIGGER_ADCA2, CLA_TRIGGER_ADCA3, CLA_TRIGGER_ADCA4, CLA_TRIGGER_ADCAEVT, CLA_TRIGGER_ADCB1, CLA_TRIGGER_ADCB2, CLA_TRIGGER_ADCB3, CLA_TRIGGER_ADCB4, CLA_TRIGGER_ADCBEVT,

CLA_TRIGGER_ADCC1,
CLA_TRIGGER_ADCC2, CLA_TRIGGER_ADCC3, CLA_TRIGGER_ADCC4,
CLA_TRIGGER_ADCCEVT,
CLA_TRIGGER_XINT1, CLA_TRIGGER_XINT2, CLA_TRIGGER_XINT3,
CLA_TRIGGER_XINT4,
CLA_TRIGGER_XINT5, CLA_TRIGGER_EPWM1INT, CLA_TRIGGER_EPWM2INT,
CLA_TRIGGER_EPWM3INT,
CLA_TRIGGER_EPWM4INT, CLA_TRIGGER_EPWM5INT, CLA_TRIGGER_EPWM6INT,
CLA_TRIGGER_EPWM7INT,
CLA_TRIGGER_EPWM8INT, CLA_TRIGGER_TINT0, CLA_TRIGGER_TINT1,
CLA_TRIGGER_TINT2,
CLA_TRIGGER_ECAP1INT, CLA_TRIGGER_ECAP2INT, CLA_TRIGGER_ECAP3INT,
CLA_TRIGGER_ECAP4INT,
CLA_TRIGGER_ECAP5INT, CLA_TRIGGER_ECAP6INT, CLA_TRIGGER_ECAP7INT,
CLA_TRIGGER_EQEP1INT,
CLA_TRIGGER_EQEP2INT, CLA_TRIGGER_ECAP6INT2, CLA_TRIGGER_ECAP7INT2,
CLA_TRIGGER_SDFM1INT,
CLA_TRIGGER_SDFM1DRINT1, CLA_TRIGGER_SDFM1DRINT2,
CLA_TRIGGER_SDFM1DRINT3, CLA_TRIGGER_SDFM1DRINT4,
CLA_TRIGGER_PMBUSAINT, CLA_TRIGGER_SPITXAINT, CLA_TRIGGER_SPIRXAINT,
CLA_TRIGGER_SPITXBINT,
CLA_TRIGGER_SPIRXBINT, CLA_TRIGGER_LINAINT1, CLA_TRIGGER_LINAINT0,
CLA_TRIGGER_CLA1PROMCRC,
CLA_TRIGGER_FSITXAINT1, CLA_TRIGGER_FSITXAINT2,
CLA_TRIGGER_FSIRXAINT1, CLA_TRIGGER_FSIRXAINT2 }

Functions

- static void CLA_mapTaskVector (uint32_t base, CLA_MVECTNumber claIntVect, uint16_t claTaskAddr)
- static void CLA_performHardReset (uint32_t base)
- static void CLA_performSoftReset (uint32_t base)
- static void CLA_enableIACK (uint32_t base)
- static void CLA_disableIACK (uint32_t base)
- static bool CLA_getPendingTaskFlag (uint32_t base, CLA_TaskNumber taskNumber)
- static uint16_t CLA_getAllPendingTaskFlags (uint32_t base)
- static bool CLA_getTaskOverflowFlag (uint32_t base, CLA_TaskNumber taskNumber)
- static uint16_t CLA_getAllTaskOverflowFlags (uint32_t base)
- static void CLA_clearTaskFlags (uint32_t base, uint16_t taskFlags)
- static void CLA_forceTasks (uint32_t base, uint16_t taskFlags)
- static void CLA_enableTasks (uint32_t base, uint16_t taskFlags)
- static void CLA_disableTasks (uint32_t base, uint16_t taskFlags)
- static bool CLA_getTaskRunStatus (uint32_t base, CLA_TaskNumber taskNumber)
- static uint16_t CLA_getAllTaskRunStatus (uint32_t base)
- static uint16_t CLA_getBackgroundActiveVector (uint32_t base)
- static void CLA_enableBackgroundTask (uint32_t base)
- static void CLA_disableBackgroundTask (uint32_t base)
- static void CLA_startBackgroundTask (uint32_t base)
- static void CLA_enableHardwareTrigger (uint32_t base)
- static void CLA_disableHardwareTrigger (uint32_t base)
- static void CLA_mapBackgroundTaskVector (uint32_t base, uint16_t claTaskAddr)
- static bool CLA_getBackgroundTaskStatus (uint32_t base, CLA_BGTaskStatus stsFlag)
- static void CLA_enableSoftwareInterrupt (uint32_t base, uint16_t taskFlags)
- static void CLA_disableSoftwareInterrupt (uint32_t base, uint16_t taskFlags)

- static void CLA_forceSoftwareInterrupt (uint32_t base, uint16_t taskFlags)
- void CLA_setTriggerSource (CLA_TaskNumber taskNumber, CLA_Trigger trigger)

8.2.1 Detailed Description

The next few paragraphs describe configuration options that are accessible via the main processor (the C28x).

The CLA code is broken up into a main background task and a set of 7 tasks, each of which requires a trigger source either from a hardware peripheral or software. Each task begins at an address that is given by its vector register. The vector for the background task can be configured using the **CLA_mapBackgroundTaskVector()**, and the task's vector is set using **CLA_mapTaskVector()**. The trigger source for all the tasks can be set with **CLA_setTriggerSource()**. If using a software trigger, the user must first enable the feature with **CLA_enableIACK()**, and then trigger the task with the assembly instruction,

```
__asm(" IACK #<Task>");
```

Task refers to the task to trigger; it is one less than the actual task. For example, if attempting to trigger task 1 you would issue,

```
__asm(" IACK #0");
```

A task will only start to execute if it is globally enabled. This is done through **CLA_enableTasks()**. Once enabled, a task will respond to a peripheral trigger (if configured to do so), a software force (with the IACK instruction), or through **CLA_forceTasks()**.

In this type of CLA, a background task is always running. It is enabled using **CLA_enableBackgroundTask()** and subsequently kicked off by **CLA_startBackgroundTask()**, or through a peripheral trigger (it takes the same trigger as task 8 on older CLAs). The user may enable the background task peripheral trigger feature using **CLA_enableHardwareTrigger()**.

The tasks (1 to 7) have a fixed priority, with 1 being the highest and 7 the lowest. They will interrupt the background task, when triggered, in priority order. The user may query the status of all tasks with **CLA_getAllTaskRunStatus()** or a particular task with **CLA_getTaskRunStatus()** to determine if its pending, running or idle.

Each task (1 through 7) can issue an interrupt to the main CPU after it completes execution. This is configured through the PIE module, and registering the handler (ISR) for each end-of-task interrupt with **CLA_registerEndOfTaskInterrupt()**.

The CLA can undergo a soft reset with **CLA_performSoftReset()** or emulate a power cycle or hard reset with **CLA_performHardReset()**.

The CLA can access and configure a few configuration registers (the C28x can read but not alter these registers). A task can force another's end-of-task interrupt to the main CPU by enabling that task's software interrupt using **CLA_enableSoftwareInterrupt()** and subsequently forcing it using **CLA_forceSoftwareInterrupt()**. Its important to keep in mind that enabling a software interrupt for a given task disables its ability to generate an interrupt to the main CPU once it completes execution.

The code for this module is contained in `driverlib/cla.c`, with `driverlib/cla.h` containing the API declarations for use by applications.

8.2.2 Enumeration Type Documentation

8.2.2.1 enum **CLA_BGTaskStatus**

Values that can be passed to CLA_getBackgroundTaskStatus() as the *stsFlag* parameter.

Enumerator

CLA_BGSTS_RUNNING Run status.

CLA_BGSTS_CANNOT_INTERRUPT Can BG task be interrupted?

CLA_BGSTS_OVERFLOW BG task hardware trigger overflow - if a second trigger occurs while the BG is already running, the overflow is set

8.2.2.2 enum **CLA_TaskNumber**

Enumerator

CLA_TASK_1 CLA Task 1.

CLA_TASK_2 CLA Task 2.

CLA_TASK_3 CLA Task 3.

CLA_TASK_4 CLA Task 4.

CLA_TASK_5 CLA Task 5.

CLA_TASK_6 CLA Task 6.

CLA_TASK_7 CLA Task 7.

CLA_TASK_8 CLA Task 8.

8.2.2.3 enum **CLA_MVECTNumber**

Values that can be passed to CLA_mapTaskVector() as the *claIntVect* parameter.

Enumerator

CLA_MVECT_1 Task Interrupt Vector 1.

CLA_MVECT_2 Task Interrupt Vector 2.

CLA_MVECT_3 Task Interrupt Vector 3.

CLA_MVECT_4 Task Interrupt Vector 4.

CLA_MVECT_5 Task Interrupt Vector 5.

CLA_MVECT_6 Task Interrupt Vector 6.

CLA_MVECT_7 Task Interrupt Vector 7.

CLA_MVECT_8 Task Interrupt Vector 8.

8.2.2.4 enum **CLA_Trigger**

Values that can be passed to CLA_setTriggerSource() as the *trigger* parameter.

Enumerator

CLA_TRIGGER_SOFTWARE CLA Task Trigger Source is Software.

CLA_TRIGGER_ADCA1 CLA Task Trigger Source is ADCA1.

CLA_TRIGGER_ADCA2 CLA Task Trigger Source is ADCA2.

CLA_TRIGGER_ADCA3 CLA Task Trigger Source is ADCA3.
CLA_TRIGGER_ADCA4 CLA Task Trigger Source is ADCA4.
CLA_TRIGGER_ADCAEVT CLA Task Trigger Source is ADCAEVT.
CLA_TRIGGER_ADCB1 CLA Task Trigger Source is ADCB1.
CLA_TRIGGER_ADCB2 CLA Task Trigger Source is ADCB2.
CLA_TRIGGER_ADCB3 CLA Task Trigger Source is ADCB3.
CLA_TRIGGER_ADCB4 CLA Task Trigger Source is ADCB4.
CLA_TRIGGER_ADCBEVT CLA Task Trigger Source is ADCBEVT.
CLA_TRIGGER_ADCC1 CLA Task Trigger Source is ADCC1.
CLA_TRIGGER_ADCC2 CLA Task Trigger Source is ADCC2.
CLA_TRIGGER_ADCC3 CLA Task Trigger Source is ADCC3.
CLA_TRIGGER_ADCC4 CLA Task Trigger Source is ADCC4.
CLA_TRIGGER_ADCCEVT CLA Task Trigger Source is ADCCEVT.
CLA_TRIGGER_XINT1 CLA Task Trigger Source is XINT1.
CLA_TRIGGER_XINT2 CLA Task Trigger Source is XINT2.
CLA_TRIGGER_XINT3 CLA Task Trigger Source is XINT3.
CLA_TRIGGER_XINT4 CLA Task Trigger Source is XINT4.
CLA_TRIGGER_XINT5 CLA Task Trigger Source is XINT5.
CLA_TRIGGER_EPWM1INT CLA Task Trigger Source is EPWM1INT.
CLA_TRIGGER_EPWM2INT CLA Task Trigger Source is EPWM2INT.
CLA_TRIGGER_EPWM3INT CLA Task Trigger Source is EPWM3INT.
CLA_TRIGGER_EPWM4INT CLA Task Trigger Source is EPWM4INT.
CLA_TRIGGER_EPWM5INT CLA Task Trigger Source is EPWM5INT.
CLA_TRIGGER_EPWM6INT CLA Task Trigger Source is EPWM6INT.
CLA_TRIGGER_EPWM7INT CLA Task Trigger Source is EPWM7INT.
CLA_TRIGGER_EPWM8INT CLA Task Trigger Source is EPWM8INT.
CLA_TRIGGER_TINT0 CLA Task Trigger Source is TINT0.
CLA_TRIGGER_TINT1 CLA Task Trigger Source is TINT1.
CLA_TRIGGER_TINT2 CLA Task Trigger Source is TINT2.
CLA_TRIGGER_ECAP1INT CLA Task Trigger Source is ECAP1INT.
CLA_TRIGGER_ECAP2INT CLA Task Trigger Source is ECAP2INT.
CLA_TRIGGER_ECAP3INT CLA Task Trigger Source is ECAP3INT.
CLA_TRIGGER_ECAP4INT CLA Task Trigger Source is ECAP4INT.
CLA_TRIGGER_ECAP5INT CLA Task Trigger Source is ECAP5INT.
CLA_TRIGGER_ECAP6INT CLA Task Trigger Source is ECAP6INT.
CLA_TRIGGER_ECAP7INT CLA Task Trigger Source is ECAP7INT.
CLA_TRIGGER_EQEP1INT CLA Task Trigger Source is EQEP1INT.
CLA_TRIGGER_EQEP2INT CLA Task Trigger Source is EQEP2INT.
CLA_TRIGGER_ECAP6INT2 CLA Task Trigger Source is ECAP6INT2.
CLA_TRIGGER_ECAP7INT2 CLA Task Trigger Source is ECAP7INT2.
CLA_TRIGGER_SDFM1INT CLA Task Trigger Source is SDFM1INT.
CLA_TRIGGER_SDFM1DRINT1 CLA Task Trigger Source is SDFM1DRINT1.
CLA_TRIGGER_SDFM1DRINT2 CLA Task Trigger Source is SDFM1DRINT2.
CLA_TRIGGER_SDFM1DRINT3 CLA Task Trigger Source is SDFM1DRINT3.
CLA_TRIGGER_SDFM1DRINT4 CLA Task Trigger Source is SDFM1DRINT4.
CLA_TRIGGER_PMBUSAINT CLA Task Trigger Source is PMBUSAINT.

CLA_TRIGGER_SPITXAINT CLA Task Trigger Source is SPITXAINT.
CLA_TRIGGER_SPIRXAINT CLA Task Trigger Source is SPIRXAINT.
CLA_TRIGGER_SPITXBINT CLA Task Trigger Source is SPITXBINT.
CLA_TRIGGER_SPIRXBINT CLA Task Trigger Source is SPIRXBINT.
CLA_TRIGGER_LINAINT1 CLA Task Trigger Source is LINAINT1.
CLA_TRIGGER_LINAINT0 CLA Task Trigger Source is LINAINT0.
CLA_TRIGGER_CLA1PROMCRC CLA Task Trigger Source is CLA1PROMCRC.
CLA_TRIGGER_FSITXAINT1 CLA Task Trigger Source is FSITXAINT1.
CLA_TRIGGER_FSITXAINT2 CLA Task Trigger Source is FSITXAINT2.
CLA_TRIGGER_FSIRXAINT1 CLA Task Trigger Source is FSIRXAINT1.
CLA_TRIGGER_FSIRXAINT2 CLA Task Trigger Source is FSIRXAINT2.

8.2.3 Function Documentation

8.2.3.1 static void CLA_mapTaskVector (uint32_t *base*, **CLA_MVECTNumber** *claIntVect*, uint16_t *claTaskAddr*) `[inline]`, `[static]`

Map CLA Task Interrupt Vector

Parameters

base	is the base address of the CLA controller.
claIntVect	is CLA interrupt vector (MVECT1 to MVECT8) the value of claIntVect can be any of the following: ■ **CLA_MVECT_1** - Task Interrupt Vector 1 ■ **CLA_MVECT_2** - Task Interrupt Vector 2 ■ **CLA_MVECT_3** - Task Interrupt Vector 3 ■ **CLA_MVECT_4** - Task Interrupt Vector 4 ■ **CLA_MVECT_5** - Task Interrupt Vector 5 ■ **CLA_MVECT_6** - Task Interrupt Vector 6 ■ **CLA_MVECT_7** - Task Interrupt Vector 7 ■ **CLA_MVECT_8** - Task Interrupt Vector 8

claTaskAddr	is the start address of the code for task

Each CLA Task (1 to 8) has its own MVECTx register. When a task is triggered, the CLA loads the MVECTx register of the task in question to the MPC (CLA program counter) and begins execution from that point. The CLA has a 16-bit address bus, and can therefore, access the lower 64 KW space. The MVECTx registers take an address anywhere in this space.

Returns
None.

8.2.3.2 static void CLA_performHardReset (uint32_t *base*) [inline], [static]

Hard Reset

Parameters

base	is the base address of the CLA controller.

This function will cause a hard reset of the CLA and set all CLA registers to their default state.

Returns
None.

8.2.3.3 static void CLA_performSoftReset (uint32_t *base*) [inline], [static]

Soft Reset

Parameters

base	is the base address of the CLA controller.

This function will cause a soft reset of the CLA. This will stop the current task, clear the MIRUN flag and clear all bits in the MIER register.

Returns
None.

8.2.3.4 static void CLA_enableIACK (uint32_t *base*) [inline], [static]

IACK enable

Parameters

base	is the base address of the CLA controller.

This function enables the main CPU to use the IACK #16bit instruction to set MIFR bits in the same manner as writing to the MIFRC register.

Returns
None.

8.2.3.5 static void CLA_disableIACK (uint32_t *base*) [inline], [static]

IACK disable

Parameters

base	is the base address of the CLA controller.

This function disables the main CPU to use the IACK #16bit instruction to set MIFR bits in the same manner as writing to the MIFRC register.

Returns
None.

8.2.3.6 static bool CLA_getPendingTaskFlag (uint32_t *base,* **CLA_TaskNumber** *taskNumber*) [inline], [static]

Query task N to see if it is flagged and pending execution

Parameters

base	is the base address of the CLA controller.
taskNumber	is the number of the task CLA_TASK_N where N is a number from 1 to 8. Do not use CLA_TASKFLAG_ALL.

This function gets the status of each bit in the interrupt flag register corresponds to a CLA task. The corresponding bit is automatically set when the task is triggered (either from a peripheral, through software, or through the MIFRC register). The bit gets cleared when the CLA starts to execute the flagged task.

Returns
True if the queried task has been triggered but pending execution.

8.2.3.7 static uint16_t CLA_getAllPendingTaskFlags (uint32_t *base*) [inline], [static]

Get status of All Task Interrupt Flag

Parameters

base	is the base address of the CLA controller.

This function gets the value of the interrupt flag register (MIFR)

Returns
the value of Interrupt Flag Register (MIFR)

8.2.3.8 static bool CLA_getTaskOverflowFlag (uint32_t *base,* **CLA_TaskNumber** *taskNumber*) [inline], [static]

Get status of Task n Interrupt Overflow Flag

Parameters

base	is the base address of the CLA controller.
taskNumber	is the number of the task CLA_TASK_N where N is a number from 1 to 8. Do not use CLA_TASKFLAG_ALL.

This function gets the status of each bit in the overflow flag register corresponds to a CLA task, This bit is set when an interrupt overflow event has occurred for the specific task.

Returns
True if any of task interrupt overflow has occurred.

8.2.3.9 static uint16_t CLA_getAllTaskOverflowFlags (uint32_t *base*) `[inline]`, `[static]`

Get status of All Task Interrupt Overflow Flag

Parameters

base	is the base address of the CLA controller.

This function gets the value of the Interrupt Overflow Flag Register

Returns
the value of Interrupt Overflow Flag Register(MIOVF)

8.2.3.10 static void CLA_clearTaskFlags (uint32_t *base,* uint16_t *taskFlags*) `[inline]`, `[static]`

Clear the task interrupt flag

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks' flags to be cleared CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to clear all flags.

This function is used to manually clear bits in the interrupt flag (MIFR) register

Returns
None.

8.2.3.11 static void CLA_forceTasks (uint32_t *base,* uint16_t *taskFlags*) `[inline]`, `[static]`

Force a CLA Task

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks' flags to be forced CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to force all tasks.

This function forces a task through software.

Returns
None.

8.2.3.12 static void CLA_enableTasks (uint32_t *base,* uint16_t *taskFlags*) `[inline]`, `[static]`

Enable CLA task(s)

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks' flags to be enabled CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to enable all tasks

This function allows an incoming interrupt or main CPU software to start the corresponding CLA task.

Returns
None.

8.2.3.13 static void CLA_disableTasks (uint32_t *base,* uint16_t *taskFlags*) `[inline]`, `[static]`

Disable CLA task interrupt

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks' flags to be disabled CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to disable all tasks

This function disables CLA task interrupt by setting the MIER register bit to 0, while the corresponding task is executing this will have no effect on the task. The task will continue to run until it hits the MSTOP instruction.

Returns
None.

8.2.3.14 static bool CLA_getTaskRunStatus (uint32_t *base,* **CLA_TaskNumber** *taskNumber*) `[inline]`, `[static]`

Get the value of a task run status

Parameters

base	is the base address of the CLA controller.
taskNumber	is the number of the task CLA_TASK_N where N is a number from 1 to 8. Do not use CLA_TASKFLAG_ALL.

This function gets the status of each bit in the Interrupt Run Status Register which indicates whether the task is currently executing

Returns

True if the task is executing.

8.2.3.15 static uint16_t CLA_getAllTaskRunStatus (uint32_t *base*) [inline], [static]

Get the value of all task run status

Parameters

base	is the base address of the CLA controller.

This function indicates which task is currently executing.

Returns

the value of Interrupt Run Status Register (MIRUN)

8.2.3.16 static uint16_t CLA_getBackgroundActiveVector (uint32_t *base*) [inline], [static]

Get the value of Active register for MVECTBGRNDACTIVE

Parameters

base	is the base address of the CLA controller.

This function gives the current interrupted MPC value of the background task.

Returns

the value of Active register for the Background Task Vector

8.2.3.17 static void CLA_enableBackgroundTask (uint32_t *base*) [inline], [static]

Enable the background task

Parameters

base	is the base address of the CLA controller.

This function enables the background task

Returns

None.

8.2.3.18 static void CLA_disableBackgroundTask (uint32_t *base*) [inline], [static]

Disable background task

Parameters

base	is the base address of the CLA controller.

This function disables the background task

Returns
None.

8.2.3.19 static void CLA_startBackgroundTask (uint32_t *base*) [inline], [static]

Start background task

Parameters

base	is the base address of the CLA controller.

This function will start the background task, provided there are no other pending tasks.

Returns
None.

8.2.3.20 static void CLA_enableHardwareTrigger (uint32_t *base*) [inline], [static]

Enable background task hardware trigger

Parameters

base	is the base address of the CLA controller.

This function enables hardware trigger for background task

Note
Trigger source for the background task will be MPERINT8.1.

Returns
None.

8.2.3.21 static void CLA_disableHardwareTrigger (uint32_t *base*) [inline], [static]

Disable background task hardware trigger

Parameters

base	is the base address of the CLA controller.

This function disables hardware trigger for background task

Returns
None.

8.2.3.22 static void CLA_mapBackgroundTaskVector (uint32_t *base,* uint16_t *claTaskAddr*) [inline], [static]

Map background task vector

Parameters

base	is the base address of the CLA controller.
claTaskAddr	is the start address of the code for task

This function specifies the start address for the background task

Returns

None.

8.2.3.23 static bool CLA_getBackgroundTaskStatus (uint32_t *base*, **CLA_BGTaskStatus** *stsFlag*) [inline], [static]

Get Status register for the back ground task.

Parameters

base	is the base address of the CLA controller.
stsFlag	is status item to be returned.

The value of *stsFlag* can be any of the following:

- **CLA_BGSTS_RUNNING**
- **CLA_BGSTS_CANNOT_INTERRUPT**
- **CLA_BGSTS_OVERFLOW**

This function gets the status of background task

Returns

Based on the value of *stsFlag*, the function will return:

- **CLA_BGSTS_RUNNING** - The function will return **true** if the background task is running.
- **CLA_BGSTS_CANNOT_INTERRUPT** - The function will return **true** if the background task will not be interrupted (when MSETC BGINTM is executed).
- **CLA_BGSTS_OVERFLOW** - This function will return **true** if an enabled hardware trigger occurred while _MCTLBGRND.BGSTART is set.

References CLA_BGSTS_CANNOT_INTERRUPT, CLA_BGSTS_OVERFLOW, and CLA_BGSTS_RUNNING.

8.2.3.24 static void CLA_enableSoftwareInterrupt (uint32_t *base*, uint16_t *taskFlags*) [inline], [static]

Enable the Software Interrupt for a given CLA Task

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks for which software interrupts are to be enabled, CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to enable software interrupts of all tasks

This function enables the Software Interrupt for a single, or set of, CLA task(s). It does this by writing a 1 to the task's bit in the CLA1SOFTINTEN register. By setting a task's SOFTINT bit, you disable its ability to generate an end-of-task interrupt For example, if we enable Task 2's SOFTINT bit, we disable its ability to generate an end-of-task interrupt, but now any running CLA task has the ability to force task 2's interrupt (through the CLA1INTFRC register) to the main CPU. This interrupt will be handled by the End-of-Task 2 interrupt handler even though the interrupt was not caused by Task 2 running to completion. This allows programmers to generate interrupts while a control task is running.

Note

1. The CLA1SOFTINTEN and CLA1INTFRC are only writable from the CLA.
2. Enabling a given task's software interrupt enable bit disables that task's ability to generate an End-of-Task interrupt to the main CPU, however, should another task force its interrupt (through the CLA1INTFRC register), it will be handled by that task's End-of-Task Interrupt Handler.

Returns

None.

8.2.3.25 static void CLA_disableSoftwareInterrupt (uint32_t *base,* uint16_t *taskFlags*) [inline], [static]

Disable the Software Interrupt for a given CLA Task

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the tasks for which software interrupts are to be disabled, CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to disable software interrupts of all tasks

This function disables the Software Interrupt for a single, or set of, CLA task(s). It does this by writing a 0 to the task's bit in the CLA1SOFTINTEN register.

Note

1. The CLA1SOFTINTEN and CLA1INTFRC are only writable from the CLA.
2. Disabling a given task's software interrupt ability allows that task to generate an End-of-Task interrupt to the main CPU.

Returns

None.

8.2.3.26 static void CLA_forceSoftwareInterrupt (uint32_t *base,* uint16_t *taskFlags*) [inline], [static]

Force a particular Task's Software Interrupt

Parameters

base	is the base address of the CLA controller.
taskFlags	is the bitwise OR of the task's whose software interrupts are to be forced, CLA_TASKFLAG_N where N is the task number from 1 to 8, or CLA_TASKFLAG_ALL to force software interrupts for all tasks

This function forces the Software Interrupt for a single, or set of, CLA task(s). It does this by writing a 1 to the task's bit in the CLA1INTFRC register. For example, if we enable Task 2's SOFTINT bit, we disable its ability to generate an end-of-task interrupt, but now any running CLA task has the ability to force task 2's interrupt (through the CLA1INTFRC register) to the main CPU. This interrupt will be handled by the End-of-Task 2 interrupt handler even though the interrupt was not caused by Task 2 running to completion. This allows programmers to generate interrupts while a control task is running.

Note

1. The CLA1SOFTINTEN and CLA1INTFRC are only writable from the CLA.
2. Enabling a given task's software interrupt enable bit disables that task's ability to generate an End-of-Task interrupt to the main CPU, however, should another task force its interrupt (through the CLA1INTFRC register), it will be handled by that task's End-of-Task Interrupt Handler.
3. This function will set the INTFRC bit for a task, but does not check that its SOFTINT bit is set. It falls to the user to ensure that software interrupt for a given task is enabled before it can be forced.

Returns

None.

8.2.3.27 void CLA_setTriggerSource (**CLA_TaskNumber** *taskNumber,* **CLA_Trigger** *trigger*)

Configures CLA task triggers.

Parameters

taskNumber	is the number of the task CLA_TASK_N where N is a number from 1 to 8.
trigger	is the trigger source to be assigned to the selected task.

This function configures the trigger source of a CLA task. The *taskNumber* parameter indicates which task is being configured, and the *trigger* parameter is the interrupt source from a specific peripheral interrupt (or software) that will trigger the task.

Returns

None.

References CLA_TASK_4.

9 CLAPROMCRC Module

Introduction . . . 86
API Functions . . . 86

9.1 CLAPROMCRC Introduction

The CLAPROMCRC is a non-intrusive CRC engine for the CLA Program ROM. It calculates the CRC-32 value of a configurable block of memory of the CLA Program ROM and can be used to validate the contents of the CLA Program ROM. The CLAPROMCRC driver provides functions to use the CLAPROMCRC and can be used from either the CLA or the C28x. The driver can be used to configure the start address of the CRC and the block size of the CRC in 1kB increments. It also provides functions to start, halt and check the status and result of the CLAPROMCRC calculation.

9.2 API Functions

Enumerations

- enum CLAPROMCRC_EmulationMode { CLAPROMCRC_MODE_SOFT, CLAPROMCRC_MODE_FREE }
- enum CLAPROMCRC_IntFlag { CLAPROMCRC_INT_FLG, CLAPROMCRC_CRCDONE_FLG }

Functions

- static void CLAPROMCRC_setEmulationMode (uint32_t base, CLAPROMCRC_EmulationMode emulationMode)
- static void CLAPROMCRC_start (uint32_t base)
- static void CLAPROMCRC_halt (uint32_t base)
- static void CLAPROMCRC_resume (uint32_t base)
- static void CLAPROMCRC_setBlockSize (uint32_t base, uint16_t blockSize)
- static void CLAPROMCRC_setStartAddress (uint32_t base, uint32_t startAddress)
- static void CLAPROMCRC_setSeed (uint32_t base, uint32_t seed)
- static uint16_t CLAPROMCRC_getCurrentAddress (uint32_t base)
- static bool CLAPROMCRC_checkStatus (uint32_t base)
- static bool CLAPROMCRC_getRunStatus (uint32_t base)
- static uint32_t CLAPROMCRC_getResult (uint32_t base)
- static void CLAPROMCRC_setGoldenCRC (uint32_t base, uint32_t goldenCRC)
- static void CLAPROMCRC_disableDoneInterrupt (uint32_t base)
- static void CLAPROMCRC_enableDoneInterrupt (uint32_t base)
- static bool CLAPROMCRC_getInterruptStatus (uint32_t base, CLAPROMCRC_IntFlag intFlag)
- static void CLAPROMCRC_clearInterruptFlag (uint32_t base, CLAPROMCRC_IntFlag intFlag)
- static void CLAPROMCRC_forceDoneFlag (uint32_t base)

9.2.1 Detailed Description

The code for this module is contained in `driverlib/clapromcrc.c`, with `driverlib/clapromcrc.h` containing the API declarations for use by applications.

9.2.2 Enumeration Type Documentation

9.2.2.1 enum **CLAPROMCRC_EmulationMode**

Values that can be passed to CLAPROMCRC_setEmulationMode().

Enumerator

CLAPROMCRC_MODE_SOFT Soft Mode.
CLAPROMCRC_MODE_FREE Free Mode.

9.2.2.2 enum **CLAPROMCRC_IntFlag**

Values that can be passed to CLAPROMCRC_getInterruptStatus() and CLAPROMCRC_clearInterruptFlag().

Enumerator

CLAPROMCRC_INT_FLG Global Interrupt Flag.
CLAPROMCRC_CRCDONE_FLG CRCDONE Interrupt Flag.

9.2.3 Function Documentation

9.2.3.1 static void CLAPROMCRC_setEmulationMode (uint32_t *base*, **CLAPROMCRC_EmulationMode** *emulationMode*) `[inline]`, `[static]`

Sets the Emulation Mode.

Parameters

base	is the base address of the CLAPROMCRC module.
emulationMode	is soft mode or free mode. It can take values CLAPROMCRC_MODE_SOFT or CLAPROMCRC_MODE_FREE.

This function sets the emulation mode which controls the behaviour of the CRC32 calculation during emulation. CLAPROMCRC_MODE_SOFT mode will stop the CLAPROMCRC module on CLA debug suspend. CLAPROMCRC_MODE_FREE mode sets the CLAPROMCRC module so that the CRC32 calculation is not affected by debug halt of the CLA.

Returns

None.

9.2.3.2 static void CLAPROMCRC_start (uint32_t *base*) `[inline]`, `[static]`

Starts the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.

This function starts CRC32 calculation.

Returns
None.

9.2.3.3 static void CLAPROMCRC_halt (uint32_t *base*) [inline], [static]

Halts the CRC32 calculations.

Parameters

base	is the base address of the CLAPROMCRC module.

This function halts the CRC32 calculation.

Returns
None.

9.2.3.4 static void CLAPROMCRC_resume (uint32_t *base*) [inline], [static]

Resumes the CRC32 calculations.

Parameters

base	is the base address of the CLAPROMCRC module.

This function resumes the CRC32 calculation.

Returns
None.

9.2.3.5 static void CLAPROMCRC_setBlockSize (uint32_t *base,* uint16_t *blockSize*) [inline], [static]

Sets the Block Size of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.
blockSize	is the number of KB. The maximum value is 128 KB

This function sets the block size for the CRC32 calculation.

Returns
None.

9.2.3.6 static void CLAPROMCRC_setStartAddress (uint32_t *base,* uint32_t *startAddress*) [inline], [static]

Sets the Start Address of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.
startAddress	defines the starting point for the CRC32 calculation. A startAddress corresponding to the CLA memory map is to be used. startAddress has to be a 1KB aligned address. If it is not aligned, then the LSB bits are ignored to get a 1KB aligned address.

This function sets the start address with *startAddress* for the CRC32 calculation.

Returns

None.

9.2.3.7 static void CLAPROMCRC_setSeed (uint32_t *base,* uint32_t *seed*) [inline], [static]

Sets the Seed of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.
seed	is the initial value of the CRC32 calculation.

This function sets the seed with *Seed* for CRC32 calculation.

Returns

None.

9.2.3.8 static uint16_t CLAPROMCRC_getCurrentAddress (uint32_t *base*) [inline], [static]

Gets the Current Address of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.

This function returns the current CLA memory map address of the data fetch unit of the CLAPROMCRC.

Returns

Returns the current address.

9.2.3.9 static bool CLAPROMCRC_checkStatus (uint32_t *base*) [inline], [static]

Check the status of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.

This function returns the status for the CLAPROMCRC. Return value of true means PASS. Return value of false means FAIL. The comparison is enabled after CRC calculation is completed.

Returns

Returns true (PASS) or false (FAIL) as the status of the CRC32 calculation.

9.2.3.10 static bool CLAPROMCRC_getRunStatus (uint32_t *base*) [inline], [static]

Gets the Run Status of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.

This function returns the run status for the CLAPROMCRC with the base address passed in the *base* parameter. Return value of false means IDLE. Return value of true means ACTIVE.

Returns

Returns true (Active) or false (Idle) as the run status of the CRC32 calculation.

9.2.3.11 static uint32_t CLAPROMCRC_getResult (uint32_t *base*) [inline], [static]

Gets the Result of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.

This function returns the result of the CRC32 calculation.

Returns

Returns the result of the CRC32 calculation.

9.2.3.12 static void CLAPROMCRC_setGoldenCRC (uint32_t *base,* uint32_t *goldenCRC*) [inline], [static]

Sets the Golden CRC of the CRC32 calculation.

Parameters

base	is the base address of the CLAPROMCRC module.
goldenCRC	is value which will be compared with CRCRESULT.

This function sets the GOLDENCRC register with *goldenCRC* for the CLAPROMCRC module. The value of GOLDENCRC is compared with CRCRESULT to determine a PASS or FAIL.

Returns

None.

9.2.3.13 static void CLAPROMCRC_disableDoneInterrupt (uint32_t *base*) [inline], [static]

Disables Interrupts the CRC32 calculations.

Parameters

base	is the base address of the CLAPROMCRC module.

This function disables interrupts for the CRC32 calculation.

Returns
None.

9.2.3.14 static void CLAPROMCRC_enableDoneInterrupt (uint32_t *base*) [inline], [static]

Enables Interrupts the CRC32 calculations.

Parameters

base	is the base address of the CLAPROMCRC module.

This function enables interrupts for the CRC32 calculation.

Returns
None.

9.2.3.15 static bool CLAPROMCRC_getInterruptStatus (uint32_t *base*, **CLAPROMCRC_IntFlag** *intFlag*) [inline], [static]

Gets the Interrupt Status of of flag.

Parameters

base	is the base address of the CLAPROMCRC module.
intFlag	is a CLAPROMCRC_IntFlag type and is either CLAPROMCRC_INT_FLG or CLAPROM-CRC_CRCDONE_FLG.

This function returns the interrupt status for *intFlag*. Return value of false means no interrupt generated. Return value of true means interrupt was generated.

Returns
Returns the interrupt status. True means interrupt was generated and false means no interrupt was generated.

9.2.3.16 static void CLAPROMCRC_clearInterruptFlag (uint32_t *base*, **CLAPROMCRC_IntFlag** *intFlag*) [inline], [static]

Clears the Global Interrupt Flag of the CLAPROMCRC.

Parameters

base	is the base address of the CLAPROMCRC module.
intFlag	is either CLAPROMCRC_INT_FLG or CLAPROMCRC_CRCDONE_FLG.

This function clears the interrupt flag for the CLAPROMCRC with the base address passed in the *base* parameter.

Returns
None.

9.2.3.17 static void CLAPROMCRC_forceDoneFlag (uint32_t *base*) [inline], [static]

Force the CRCDONE Interrupt Flag of the CLAPROMCRC.

Parameters

base	is the base address of the CLAPROMCRC module.

This function forces the CRCDONE interrupt flag for the CLAPROMCRC with the base address passed in the *base* parameter.

Returns
None.

10 CMPSS Module

Introduction 94
API Functions 94

10.1 CMPSS Introduction

The comparator subsystem (CMPSS) API provides a set of functions for programming the digital circuits of a pair of analog comparators. Functions are provided to configure each comparator and its corresponding 12-bit DAC and digital filter and to get both the latched and unlatched status of their output. There are also functions to configure the optional ramp generator circuit and to route incoming sync signals from the ePWM module.

The output signals of the CMPSS (referred to as CTRIPH, CTRIPOUTH, CTRIPL, and CTRIPOUTL) may be routed to GPIOs or other internal destinations using the X-BARs. See the X-BAR driver for details.

10.2 API Functions

Functions

- static void CMPSS_enableModule (uint32_t base)
- static void CMPSS_disableModule (uint32_t base)
- static void CMPSS_configHighComparator (uint32_t base, uint16_t config)
- static void CMPSS_configLowComparator (uint32_t base, uint16_t config)
- static void CMPSS_configOutputsHigh (uint32_t base, uint16_t config)
- static void CMPSS_configOutputsLow (uint32_t base, uint16_t config)
- static uint16_t CMPSS_getStatus (uint32_t base)
- static void CMPSS_configDAC (uint32_t base, uint16_t config)
- static void CMPSS_setDACValueHigh (uint32_t base, uint16_t value)
- static void CMPSS_setDACValueLow (uint32_t base, uint16_t value)
- static void CMPSS_initFilterHigh (uint32_t base)
- static void CMPSS_initFilterLow (uint32_t base)
- static uint16_t CMPSS_getDACValueHigh (uint32_t base)
- static uint16_t CMPSS_getDACValueLow (uint32_t base)
- static void CMPSS_clearFilterLatchHigh (uint32_t base)
- static void CMPSS_clearFilterLatchLow (uint32_t base)
- static void CMPSS_setMaxRampValue (uint32_t base, uint16_t value)
- static uint16_t CMPSS_getMaxRampValue (uint32_t base)
- static void CMPSS_setRampDecValue (uint32_t base, uint16_t value)
- static uint16_t CMPSS_getRampDecValue (uint32_t base)
- static void CMPSS_setRampDelayValue (uint32_t base, uint16_t value)
- static uint16_t CMPSS_getRampDelayValue (uint32_t base)
- static void CMPSS_setHysteresis (uint32_t base, uint16_t value)
- static void CMPSS_configBlanking (uint32_t base, uint16_t pwmBlankSrc)
- static void CMPSS_enableBlanking (uint32_t base)
- static void CMPSS_disableBlanking (uint32_t base)
- void CMPSS_configFilterHigh (uint32_t base, uint16_t samplePrescale, uint16_t sampleWindow, uint16_t threshold)

- void CMPSS_configFilterLow (uint32_t base, uint16_t samplePrescale, uint16_t sampleWindow, uint16_t threshold)
- void CMPSS_configLatchOnPWMSYNC (uint32_t base, bool highEnable, bool lowEnable)
- void CMPSS_configRamp (uint32_t base, uint16_t maxRampVal, uint16_t decrementVal, uint16_t delayVal, uint16_t pwmSyncSrc, bool useRampValShdw)

10.2.1 Detailed Description

The two comparators are referred to as the high comparator and the low comparator. Accordingly, many API functions come in pairs with both a "High" and a "Low" version. See the device's Technical Reference Manual for diagrams showing what resources the comparators share and what they contain separately.

The code for this module is contained in `driverlib/cmpss.c`, with `driverlib/cmpss.h` containing the API declarations for use by applications.

10.2.2 Function Documentation

10.2.2.1 static void CMPSS_enableModule (uint32_t *base*) `[inline]`, `[static]`

Enables the CMPSS module.

Parameters

base	is the base address of the CMPSS module.

This function enables the CMPSS module passed into the *base* parameter.

Returns

None.

10.2.2.2 static void CMPSS_disableModule (uint32_t *base*) `[inline]`, `[static]`

Disables the CMPSS module.

Parameters

base	is the base address of the CMPSS module.

This function disables the CMPSS module passed into the *base* parameter.

Returns

None.

10.2.2.3 static void CMPSS_configHighComparator (uint32_t *base,* uint16_t *config*) `[inline]`, `[static]`

Sets the configuration for the high comparator.

Parameters

base	is the base address of the CMPSS module.
config	is the configuration of the high comparator.

This function configures a comparator. The *config* parameter is the result of a logical OR operation between a **CMPSS_INSRC_xxx** value and if desired, **CMPSS_INV_INVERTED** and **CMPSS_OR_ASYNC_OUT_W_FILT** values.

The **CMPSS_INSRC_xxx** term can take on the following values to specify the high comparator negative input source:

- **CMPSS_INSRC_DAC** - The internal DAC.
- **CMPSS_INSRC_PIN** - An external pin.

CMPSS_INV_INVERTED may be ORed into *config* if the comparator output should be inverted.

CMPSS_OR_ASYNC_OUT_W_FILT may be ORed into *config* if the asynchronous comparator output should be fed into an OR gate with the latched digital filter output before it is made available for CTRIPH or CTRIPOUTH.

Returns

None.

10.2.2.4 static void CMPSS_configLowComparator (uint32_t *base,* uint16_t *config*) [inline], [static]

Sets the configuration for the low comparator.

Parameters

base	is the base address of the CMPSS module.
config	is the configuration of the low comparator.

This function configures a comparator. The *config* parameter is the result of a logical OR operation between a **CMPSS_INSRC_xxx** value and if desired, **CMPSS_INV_INVERTED** and **CMPSS_OR_ASYNC_OUT_W_FILT** values.

The **CMPSS_INSRC_xxx** term can take on the following values to specify the low comparator negative input source:

- **CMPSS_INSRC_DAC** - The internal DAC.
- **CMPSS_INSRC_PIN** - An external pin.

CMPSS_INV_INVERTED may be ORed into *config* if the comparator output should be inverted.

CMPSS_OR_ASYNC_OUT_W_FILT may be ORed into *config* if the asynchronous comparator output should be fed into an OR gate with the latched digital filter output before it is made available for CTRIPL or CTRIPOUTL.

Returns

None.

10.2.2.5 static void CMPSS_configOutputsHigh (uint32_t *base,* uint16_t *config*) [inline], [static]

Sets the output signal configuration for the high comparator.

Parameters

base	is the base address of the CMPSS module.
config	is the configuration of the high comparator output signals.

This function configures a comparator's output signals CTRIP and CTRIPOUT. The *config* parameter is the result of a logical OR operation between the **CMPSS_TRIPOUT_xxx** and **CMPSS_TRIP_xxx** values.

The **CMPSS_TRIPOUT_xxx** term can take on the following values to specify which signal drives CTRIPOUTH:

- **CMPSS_TRIPOUT_ASYNC_COMP** - The asynchronous comparator output.
- **CMPSS_TRIPOUT_SYNC_COMP** - The synchronous comparator output.
- **CMPSS_TRIPOUT_FILTER** - The output of the digital filter.
- **CMPSS_TRIPOUT_LATCH** - The latched output of the digital filter.

The **CMPSS_TRIP_xxx** term can take on the following values to specify which signal drives CTRIPH:

- **CMPSS_TRIP_ASYNC_COMP** - The asynchronous comparator output.
- **CMPSS_TRIP_SYNC_COMP** - The synchronous comparator output.
- **CMPSS_TRIP_FILTER** - The output of the digital filter.
- **CMPSS_TRIP_LATCH** - The latched output of the digital filter.

Returns

None.

10.2.2.6 static void CMPSS_configOutputsLow (uint32_t *base*, uint16_t *config*) `[inline]`, `[static]`

Sets the output signal configuration for the low comparator.

Parameters

base	is the base address of the CMPSS module.
config	is the configuration of the low comparator output signals.

This function configures a comparator's output signals CTRIP and CTRIPOUT. The *config* parameter is the result of a logical OR operation between the **CMPSS_TRIPOUT_xxx** and **CMPSS_TRIP_xxx** values.

The **CMPSS_TRIPOUT_xxx** term can take on the following values to specify which signal drives CTRIPOUTL:

- **CMPSS_TRIPOUT_ASYNC_COMP** - The asynchronous comparator output.
- **CMPSS_TRIPOUT_SYNC_COMP** - The synchronous comparator output.
- **CMPSS_TRIPOUT_FILTER** - The output of the digital filter.
- **CMPSS_TRIPOUT_LATCH** - The latched output of the digital filter.

The **CMPSS_TRIP_xxx** term can take on the following values to specify which signal drives CTRIPL:

- **CMPSS_TRIP_ASYNC_COMP** - The asynchronous comparator output.
- **CMPSS_TRIP_SYNC_COMP** - The synchronous comparator output.
- **CMPSS_TRIP_FILTER** - The output of the digital filter.
- **CMPSS_TRIP_LATCH** - The latched output of the digital filter.

Returns

None.

10.2.2.7 static uint16_t CMPSS_getStatus (uint32_t *base*) [inline], [static]

Gets the current comparator status.

Parameters

base	is the base address of the comparator module.

This function returns the current status for the comparator, specifically the digital filter output and latched digital filter output.

Returns

Returns the current interrupt status, enumerated as a bit field of the following values:

- **CMPSS_STS_HI_FILTOUT** - High digital filter output
- **CMPSS_STS_HI_LATCHFILTOUT** - Latched value of high digital filter output
- **CMPSS_STS_LO_FILTOUT** - Low digital filter output
- **CMPSS_STS_LO_LATCHFILTOUT** - Latched value of low digital filter output

10.2.2.8 static void CMPSS_configDAC (uint32_t *base,* uint16_t *config*) [inline], [static]

Sets the configuration for the internal comparator DACs.

Parameters

base	is the base address of the CMPSS module.
config	is the configuration of the internal DAC.

This function configures the comparator's internal DAC. The *config* parameter is the result of a logical OR operation between the **CMPSS_DACVAL_xxx**, **CMPSS_DACREF_xxx**, and **CMPSS_DACSRC_xxx**.

The **CMPSS_DACVAL_xxx** term can take on the following values to specify when the DAC value is loaded from its shadow register:

- **CMPSS_DACVAL_SYSCLK** - Value register updated on system clock.
- **CMPSS_DACVAL_PWMSYNC** - Value register updated on PWM sync.

The **CMPSS_DACREF_xxx** term can take on the following values to specify which voltage supply is used as reference for the DACs:

- **CMPSS_DACREF_VDDA** - VDDA is the voltage reference for the DAC.
- **CMPSS_DACREF_VDAC** - VDAC is the voltage reference for the DAC.

The **CMPSS_DACSRC_xxx** term can take on the following values to specify the DAC value source for the high comparator's internal DAC:

- **CMPSS_DACSRC_SHDW** - The user-programmed DACVALS register.
- **CMPSS_DACSRC_RAMP** - The ramp generator RAMPSTS register

Note

The **CMPSS_DACVAL_xxx** and **CMPSS_DACREF_xxx** terms apply to both the high and low comparators. **CMPSS_DACSRC_xxx** will only affect the high comparator's internal DAC.

Returns

None.

10.2.2.9 static void CMPSS_setDACValueHigh (uint32_t *base*, uint16_t *value*) [inline], [static]

Sets the value of the internal DAC of the high comparator.

Parameters

base	is the base address of the comparator module.
value	is the value actively driven by the DAC.

This function sets the 12-bit value driven by the internal DAC of the high comparator. This function will load the value into the shadow register from which the actual DAC value register will be loaded. To configure which event causes this shadow load to take place, use CMPSS_configDAC().

Returns

None.

10.2.2.10 static void CMPSS_setDACValueLow (uint32_t *base*, uint16_t *value*) [inline], [static]

Sets the value of the internal DAC of the low comparator.

Parameters

base	is the base address of the comparator module.
value	is the value actively driven by the DAC.

This function sets the 12-bit value driven by the internal DAC of the low comparator. This function will load the value into the shadow register from which the actual DAC value register will be loaded. To configure which event causes this shadow load to take place, use CMPSS_configDAC().

Returns

None.

10.2.2.11 static void CMPSS_initFilterHigh (uint32_t *base*) [inline], [static]

Initializes the digital filter of the high comparator.

Parameters

base	is the base address of the comparator module.

This function initializes all the samples in the high comparator digital filter to the filter input value.

Note

See CMPSS_configFilterHigh() for the proper initialization sequence to avoid glitches.

Returns

None.

10.2.2.12 static void CMPSS_initFilterLow (uint32_t *base*) [inline], [static]

Initializes the digital filter of the low comparator.

Parameters

base	is the base address of the comparator module.

This function initializes all the samples in the low comparator digital filter to the filter input value.

Note

See CMPSS_configFilterLow() for the proper initialization sequence to avoid glitches.

Returns

None.

10.2.2.13 static uint16_t CMPSS_getDACValueHigh (uint32_t *base*) [inline], [static]

Gets the value of the internal DAC of the high comparator.

Parameters

base	is the base address of the comparator module.

This function gets the value of the internal DAC of the high comparator. The value is read from the *active* register–not the shadow register to which CMPSS_setDACValueHigh() writes.

Returns

Returns the value driven by the internal DAC of the high comparator.

10.2.2.14 static uint16_t CMPSS_getDACValueLow (uint32_t *base*) [inline], [static]

Gets the value of the internal DAC of the low comparator.

Parameters

base	is the base address of the comparator module.

This function gets the value of the internal DAC of the low comparator. The value is read from the *active* register–not the shadow register to which CMPSS_setDACValueLow() writes.

Returns

Returns the value driven by the internal DAC of the low comparator.

10.2.2.15 static void CMPSS_clearFilterLatchHigh (uint32_t *base*) [inline], [static]

Causes a software reset of the high comparator digital filter output latch.

Parameters

base	is the base address of the comparator module.

This function causes a software reset of the high comparator digital filter output latch. It will generate a single pulse of the latch reset signal.

Returns

None.

10.2.2.16 static void CMPSS_clearFilterLatchLow (uint32_t *base*) [inline], [static]

Causes a software reset of the low comparator digital filter output latch.

Parameters

base	is the base address of the comparator module.

This function causes a software reset of the low comparator digital filter output latch. It will generate a single pulse of the latch reset signal.

Returns

None.

10.2.2.17 static void CMPSS_setMaxRampValue (uint32_t *base,* uint16_t *value*) [inline], [static]

Sets the ramp generator maximum reference value.

Parameters

base	is the base address of the comparator module.

value	the ramp maximum reference value.

This function sets the ramp maximum reference value that will be loaded into the ramp generator.

Returns

None.

10.2.2.18 static uint16_t CMPSS_getMaxRampValue (uint32_t *base*) [inline], [static]

Gets the ramp generator maximum reference value.

Parameters

base	is the base address of the comparator module.

Returns

Returns the latched ramp maximum reference value that will be loaded into the ramp generator.

10.2.2.19 static void CMPSS_setRampDecValue (uint32_t *base,* uint16_t *value*) [inline], [static]

Sets the ramp generator decrement value.

Parameters

base	is the base address of the comparator module.
value	is the ramp decrement value.

This function sets the value that is subtracted from the ramp value on every system clock cycle.

Returns

None.

10.2.2.20 static uint16_t CMPSS_getRampDecValue (uint32_t *base*) [inline], [static]

Gets the ramp generator decrement value.

Parameters

base	is the base address of the comparator module.

Returns

Returns the latched ramp decrement value that is subtracted from the ramp value on every system clock cycle.

10.2.2.21 static void CMPSS_setRampDelayValue (uint32_t *base,* uint16_t *value*) [inline], [static]

Sets the ramp generator delay value.

Parameters

base	is the base address of the comparator module.
value	is the 13-bit ramp delay value.

This function sets the value that configures the number of system clock cycles to delay the start of the ramp generator decrementer after a PWMSYNC event is received. Delay value can be no greater than 8191.

Returns

None.

10.2.2.22 static uint16_t CMPSS_getRampDelayValue (uint32_t *base*) [inline], [static]

Gets the ramp generator delay value.

Parameters

base	is the base address of the comparator module.

Returns

Returns the latched ramp delay value that is subtracted from the ramp value on every system clock cycle.

10.2.2.23 static void CMPSS_setHysteresis (uint32_t *base,* uint16_t *value*) [inline], [static]

Sets the comparator hysteresis settings.

Parameters

base	is the base address of the comparator module.
value	is the amount of hysteresis on the comparator inputs.

This function sets the amount of hysteresis on the comparator inputs. The *value* parameter indicates the amount of hysteresis desired. Passing in 0 results in none, passing in 1 results in typical hysteresis, passing in 2 results in 2x of typical hysteresis, and so on where *value* x of typical hysteresis is the amount configured.

Returns

None.

10.2.2.24 static void CMPSS_configBlanking (uint32_t *base,* uint16_t *pwmBlankSrc*) [inline], [static]

Sets the ePWM module blanking signal that holds trip in reset.

Parameters

base	is the base address of the comparator module.
pwmBlankSrc	is the number of the PWMBLANK source.

This function configures which PWMBLANK signal from the ePWM module will hold trip in reset when blanking is enabled.

The number of the PWMBLANK signal to be used to reset the ramp generator should be specified by passing it into the *pwmBlankSrc* parameter. For instance, passing a 2 into *pwmBlankSrc* will select PWMBLANK2.

Returns
None.

10.2.2.25 static void CMPSS_enableBlanking (uint32_t *base*) [inline], [static]

Enables an ePWM blanking signal to hold trip in reset.

Parameters

base	is the base address of the comparator module.

This function enables a selected ePWM blanking signal to hold trip in reset.

Returns
None.

10.2.2.26 static void CMPSS_disableBlanking (uint32_t *base*) [inline], [static]

Disables an ePWM blanking signal from holding trip in reset.

Parameters

base	is the base address of the comparator module.

This function disables a selected ePWM blanking signal from holding trip in reset.

Returns
None.

10.2.2.27 void CMPSS_configFilterHigh (uint32_t *base,* uint16_t *samplePrescale,* uint16_t *sampleWindow,* uint16_t *threshold*)

Configures the digital filter of the high comparator.

Parameters

base	is the base address of the comparator module.

samplePrescale	is the number of system clock cycles between samples.
sampleWindow	is the number of FIFO samples to monitor.
threshold	is the majority threshold of samples to change state.

This function configures the operation of the digital filter of the high comparator.

The *samplePrescale* parameter specifies the number of system clock cycles between samples. It is a 10-bit value so a number higher than 1023 should not be passed as this parameter.

The *sampleWindow* parameter configures the size of the window of FIFO samples taken from the input that will be monitored to determine when to change the filter output. This sample window may be no larger than 32 samples.

The filter output resolves to the majority value of the sample window where majority is defined by the value passed into the *threshold* parameter. For proper operation, the value of *threshold* must be greater than sampleWindow / 2.

To ensure proper operation of the filter, the following is the recommended function call sequence for initialization:

1. Configure and enable the comparator using CMPSS_configHighComparator() and CMPSS_enableModule()
2. Configure the digital filter using CMPSS_configFilterHigh()
3. Initialize the sample values using CMPSS_initFilterHigh()
4. Configure the module output signals CTRIP and CTRIPOUT using CMPSS_configOutputsHigh()

Returns

None.

10.2.2.28 void CMPSS_configFilterLow (uint32_t *base,* uint16_t *samplePrescale,* uint16_t *sampleWindow,* uint16_t *threshold*)

Configures the digital filter of the low comparator.

Parameters

base	is the base address of the comparator module.
samplePrescale	is the number of system clock cycles between samples.
sampleWindow	is the number of FIFO samples to monitor.
threshold	is the majority threshold of samples to change state.

This function configures the operation of the digital filter of the low comparator.

The *samplePrescale* parameter specifies the number of system clock cycles between samples. It is a 10-bit value so a number higher than 1023 should not be passed as this parameter.

The *sampleWindow* parameter configures the size of the window of FIFO samples taken from the input that will be monitored to determine when to change the filter output. This sample window may be no larger than 32 samples.

The filter output resolves to the majority value of the sample window where majority is defined by the value passed into the *threshold* parameter. For proper operation, the value of *threshold* must be greater than sampleWindow / 2.

To ensure proper operation of the filter, the following is the recommended function call sequence for initialization:

1. Configure and enable the comparator using CMPSS_configLowComparator() and CMPSS_enableModule()
2. Configure the digital filter using CMPSS_configFilterLow()
3. Initialize the sample values using CMPSS_initFilterLow()
4. Configure the module output signals CTRIP and CTRIPOUT using CMPSS_configOutputsLow()

Returns

None.

10.2.2.29 void CMPSS_configLatchOnPWMSYNC (uint32_t *base,* bool *highEnable,* bool *lowEnable*)

Configures whether or not the digital filter latches are reset by PWMSYNC

Parameters

base	is the base address of the comparator module.
highEnable	indicates filter latch settings in the high comparator.
lowEnable	indicates filter latch settings in the low comparator.

This function configures whether or not the digital filter latches in both the high and low comparators should be reset by PWMSYNC. If the *highEnable* parameter is **true**, the PWMSYNC will be allowed to reset the high comparator's digital filter latch. If it is false, the ability of the PWMSYNC to reset the latch will be disabled. The *lowEnable* parameter has the same effect on the low comparator's digital filter latch.

Returns

None.

10.2.2.30 void CMPSS_configRamp (uint32_t *base,* uint16_t *maxRampVal,* uint16_t *decrementVal,* uint16_t *delayVal,* uint16_t *pwmSyncSrc,* bool *useRampValShdw*)

Configures the comparator subsystem's ramp generator.

Parameters

base	is the base address of the comparator module.
maxRampVal	is the ramp maximum reference value.
decrementVal	value is the ramp decrement value.
delayVal	is the ramp delay value.

pwmSyncSrc	is the number of the PWMSYNC source.
useRampValShdw	indicates if the max ramp shadow should be used.

This function configures many of the main settings of the comparator subsystem's ramp generator. The *maxRampVal* parameter should be passed the ramp maximum reference value that will be loaded into the ramp generator. The *decrementVal* parameter should be passed the decrement value that will be subtracted from the ramp generator on each system clock cycle. The *delayVal* parameter should be passed the 13-bit number of system clock cycles the ramp generator should delay before beginning to decrement the ramp generator after a PWMSYNC signal is received.

These three values may be be set individually using the CMPSS_setMaxRampValue(), CMPSS_setRampDecValue(), and CMPSS_setRampDelayValue() APIs.

The number of the PWMSYNC signal to be used to reset the ramp generator should be specified by passing it into the *pwmSyncSrc* parameter. For instance, passing a 2 into *pwmSyncSrc* will select PWMSYNC2.

To indicate whether the ramp generator should reset with the value from the ramp max reference value shadow register or with the latched ramp max reference value, use the *useRampValShdw* parameter. Passing it **true** will result in the latched value being bypassed. The ramp generator will be loaded right from the shadow register. A value of **false** will load the ramp generator from the latched value.

Returns
None.

11 CPU Timer

Introduction 110
API Functions 110

11.1 CPU Timer Introduction

The CPU timer API provides a set of functions for configuring and using the CPU Timer module. Functions are provided to setup and configure the timer module operating conditions along with functions to get the status of the module and to clear overflow flag.

11.2 API Functions

Enumerations

- enum CPUTimer_EmulationMode {
  CPUTIMER_EMULATIONMODE_STOPAFTERNEXTDECREMENT,
  CPUTIMER_EMULATIONMODE_STOPATZERO,
  CPUTIMER_EMULATIONMODE_RUNFREE }
- enum CPUTimer_ClockSource {
  CPUTIMER_CLOCK_SOURCE_SYS, CPUTIMER_CLOCK_SOURCE_INTOSC1,
  CPUTIMER_CLOCK_SOURCE_INTOSC2, CPUTIMER_CLOCK_SOURCE_XTAL,
  CPUTIMER_CLOCK_SOURCE_AUX }
- enum CPUTimer_Prescaler {
  CPUTIMER_CLOCK_PRESCALER_1, CPUTIMER_CLOCK_PRESCALER_2,
  CPUTIMER_CLOCK_PRESCALER_4, CPUTIMER_CLOCK_PRESCALER_8,
  CPUTIMER_CLOCK_PRESCALER_16 }

Functions

- static void CPUTimer_clearOverflowFlag (uint32_t base)
- static void CPUTimer_disableInterrupt (uint32_t base)
- static void CPUTimer_enableInterrupt (uint32_t base)
- static void CPUTimer_reloadTimerCounter (uint32_t base)
- static void CPUTimer_stopTimer (uint32_t base)
- static void CPUTimer_resumeTimer (uint32_t base)
- static void CPUTimer_startTimer (uint32_t base)
- static void CPUTimer_setPeriod (uint32_t base, uint32_t periodCount)
- static uint32_t CPUTimer_getTimerCount (uint32_t base)
- static void CPUTimer_setPreScaler (uint32_t base, uint16_t prescaler)
- static bool CPUTimer_getTimerOverflowStatus (uint32_t base)
- static void CPUTimer_selectClockSource (uint32_t base, CPUTimer_ClockSource source, CPUTimer_Prescaler prescaler)
- void CPUTimer_setEmulationMode (uint32_t base, CPUTimer_EmulationMode mode)

11.2.1 Detailed Description

The code for this module is contained in `driverlib/cputimer.c`, with `driverlib/cputimer.h` containing the API declarations for use by applications.

11.2.2 Enumeration Type Documentation

11.2.2.1 enum **CPUTimer_EmulationMode**

Values that can be passed to CPUTimer_setEmulationMode() as the *mode* parameter.

Enumerator

CPUTIMER_EMULATIONMODE_STOPAFTERNEXTDECREMENT Denotes that the timer will stop after the next decrement.

CPUTIMER_EMULATIONMODE_STOPATZERO Denotes that the timer will stop when it reaches zero.

CPUTIMER_EMULATIONMODE_RUNFREE Denotes that the timer will run free.

11.2.2.2 enum **CPUTimer_ClockSource**

The following are values that can be passed to CPUTimer_selectClockSource() as the *source* parameter.

Enumerator

CPUTIMER_CLOCK_SOURCE_SYS System Clock Source.

CPUTIMER_CLOCK_SOURCE_INTOSC1 Internal Oscillator 1 Clock Source.

CPUTIMER_CLOCK_SOURCE_INTOSC2 Internal Oscillator 2 Clock Source.

CPUTIMER_CLOCK_SOURCE_XTAL External Clock Source.

CPUTIMER_CLOCK_SOURCE_AUX Auxiliary PLL Clock Source.

11.2.2.3 enum **CPUTimer_Prescaler**

The following are values that can be passed to CPUTimer_selectClockSource() as the *prescaler* parameter.

Enumerator

CPUTIMER_CLOCK_PRESCALER_1 Prescaler value of / 1.

CPUTIMER_CLOCK_PRESCALER_2 Prescaler value of / 2.

CPUTIMER_CLOCK_PRESCALER_4 Prescaler value of / 4.

CPUTIMER_CLOCK_PRESCALER_8 Prescaler value of / 8.

CPUTIMER_CLOCK_PRESCALER_16 Prescaler value of / 16.

11.2.3 Function Documentation

11.2.3.1 static void CPUTimer_clearOverflowFlag (uint32_t *base*) `[inline]`, `[static]`

Clears CPU timer overflow flag.

Parameters

base	is the base address of the timer module.

This function clears the CPU timer overflow flag.

Returns
None.

11.2.3.2 static void CPUTimer_disableInterrupt (uint32_t *base*) [inline], [static]

Disables CPU timer interrupt.

Parameters

base	is the base address of the timer module.

This function disables the CPU timer interrupt.

Returns
None.

11.2.3.3 static void CPUTimer_enableInterrupt (uint32_t *base*) [inline], [static]

Enables CPU timer interrupt.

Parameters

base	is the base address of the timer module.

This function enables the CPU timer interrupt.

Returns
None.

11.2.3.4 static void CPUTimer_reloadTimerCounter (uint32_t *base*) [inline], [static]

Reloads CPU timer counter.

Parameters

base	is the base address of the timer module.

This function reloads the CPU timer counter with the values contained in the CPU timer period register.

Returns
None.

11.2.3.5 static void CPUTimer_stopTimer (uint32_t *base*) [inline], [static]

Stops CPU timer.

Parameters

base	is the base address of the timer module.

This function stops the CPU timer.

Returns
None.

11.2.3.6 static void CPUTimer_resumeTimer (uint32_t *base*) `[inline]`, `[static]`

Starts(restarts) CPU timer.

Parameters

base	is the base address of the timer module.

This function starts (restarts) the CPU timer.

Note: This function doesn't reset the timer counter.

Returns
None.

11.2.3.7 static void CPUTimer_startTimer (uint32_t *base*) `[inline]`, `[static]`

Starts(restarts) CPU timer.

Parameters

base	is the base address of the timer module.

This function starts (restarts) the CPU timer.

Note: This function reloads the timer counter.

Returns
None.

11.2.3.8 static void CPUTimer_setPeriod (uint32_t *base,* uint32_t *periodCount*) `[inline]`, `[static]`

Sets CPU timer period.

Parameters

base	is the base address of the timer module.
periodCount	is the CPU timer period count.

This function sets the CPU timer period count.

Returns
None.

11.2.3.9 static uint32_t CPUTimer_getTimerCount (uint32_t *base*) `[inline]`, `[static]`

Returns the current CPU timer counter value.

Parameters

base	is the base address of the timer module.

This function returns the current CPU timer counter value.

Returns

Returns the current CPU timer count value.

11.2.3.10 static void CPUTimer_setPreScaler (uint32_t *base*, uint16_t *prescaler*) [inline], [static]

Set CPU timer pre-scaler value.

Parameters

base	is the base address of the timer module.
prescaler	is the CPU timer pre-scaler value.

This function sets the pre-scaler value for the CPU timer. For every value of (prescaler + 1), the CPU timer counter decrements by 1.

Returns

None.

11.2.3.11 static bool CPUTimer_getTimerOverflowStatus (uint32_t *base*) [inline], [static]

Return the CPU timer overflow status.

Parameters

base	is the base address of the timer module.

This function returns the CPU timer overflow status.

Returns

Returns true if the CPU timer has overflowed, false if not.

11.2.3.12 static void CPUTimer_selectClockSource (uint32_t *base*, **CPUTimer_ClockSource** *source*, **CPUTimer_Prescaler** *prescaler*) [inline], [static]

Select CPU Timer 2 Clock Source and Prescaler

Parameters

base	is the base address of the timer module.

source	is the clock source to use for CPU Timer 2
prescaler	is the value that configures the selected clock source relative to the system clock

This function selects the specified clock source and prescaler value for the CPU timer (CPU timer 2 only).

The *source* parameter can be any one of the following:

- **CPUTIMER_CLOCK_SOURCE_SYS** - System Clock
- **CPUTIMER_CLOCK_SOURCE_INTOSC1** - Internal Oscillator 1 Clock
- **CPUTIMER_CLOCK_SOURCE_INTOSC2** - Internal Oscillator 2 Clock
- **CPUTIMER_CLOCK_SOURCE_XTAL** - External Clock
- **CPUTIMER_CLOCK_SOURCE_AUX** - Auxiliary PLL Clock

The *prescaler* parameter can be any one of the following:

- **CPUTIMER_CLOCK_PRESCALER_1** - Prescaler value of / 1
- **CPUTIMER_CLOCK_PRESCALER_2** - Prescaler value of / 2
- **CPUTIMER_CLOCK_PRESCALER_4** - Prescaler value of / 4
- **CPUTIMER_CLOCK_PRESCALER_8** - Prescaler value of / 8
- **CPUTIMER_CLOCK_PRESCALER_16** - Prescaler value of / 16

Returns
None.

11.2.3.13 void CPUTimer_setEmulationMode (uint32_t *base*, **CPUTimer_EmulationMode** *mode*)

Sets Emulation mode for CPU timer.

Parameters

base	is the base address of the timer module.
mode	is the emulation mode of the timer.

This function sets the behaviour of CPU timer during emulation. Valid values mode are: CPUTIMER_EMULATIONMODE_STOPAFTERNEXTDECREMENT, CPUTIMER_EMULATIONMODE_STOPATZERO and CPUTIMER_EMULATIONMODE_RUNFREE.

Returns
None.

12 DAC Module

Introduction 118
API Functions 118

12.1 DAC Introduction

The buffered digital to analog converter (DAC) API provides a set of functions for programming the digital circuits of the DAC. Functions are provided to set the reference voltage, the synchronization mode, the internal 12-bit DAC value, and set the state of the DAC output.

12.2 API Functions

Macros

- #define DAC_REG_BYTE_MASK
- #define DAC_LOCK_KEY

Enumerations

- enum DAC_ReferenceVoltage { DAC_REF_VDAC, DAC_REF_ADC_VREFHI }
- enum DAC_GainMode { DAC_GAIN_ONE, DAC_GAIN_TWO }
- enum DAC_LoadMode { DAC_LOAD_SYSCLK, DAC_LOAD_PWMSYNC }

Functions

- static uint16_t DAC_getRevision (uint32_t base)
- static void DAC_setReferenceVoltage (uint32_t base, DAC_ReferenceVoltage source)
- static void DAC_setGainMode (uint32_t base, DAC_GainMode mode)
- static void DAC_setLoadMode (uint32_t base, DAC_LoadMode mode)
- static void DAC_setPWMSyncSignal (uint32_t base, uint16_t signal)
- static uint16_t DAC_getActiveValue (uint32_t base)
- static void DAC_setShadowValue (uint32_t base, uint16_t value)
- static uint16_t DAC_getShadowValue (uint32_t base)
- static void DAC_enableOutput (uint32_t base)
- static void DAC_disableOutput (uint32_t base)
- static void DAC_setOffsetTrim (uint32_t base, int16_t offset)
- static int16_t DAC_getOffsetTrim (uint32_t base)
- static void DAC_lockRegister (uint32_t base, uint16_t reg)
- static bool DAC_isRegisterLocked (uint32_t base, uint16_t reg)
- void DAC_tuneOffsetTrim (uint32_t base, float32_t referenceVoltage)

12.2.1 Detailed Description

The code for this module is contained in `driverlib/dac.c`, with `driverlib/dac.h` containing the API declarations for use by applications.

12.2.2 Enumeration Type Documentation

12.2.2.1 enum **DAC_ReferenceVoltage**

Values that can be passed to DAC_setReferenceVoltage() as the *source* parameter.

Enumerator

DAC_REF_VDAC VDAC reference voltage.

DAC_REF_ADC_VREFHI ADC VREFHI reference voltage.

12.2.2.2 enum **DAC_GainMode**

Values that can be passed to DAC_setGainMode() as the *mode* parameter.

Enumerator

DAC_GAIN_ONE Gain set to 1.

DAC_GAIN_TWO Gain set to 2.

12.2.2.3 enum **DAC_LoadMode**

Values that can be passed to DAC_setLoadMode() as the *mode* parameter.

Enumerator

DAC_LOAD_SYSCLK Load on next SYSCLK.

DAC_LOAD_PWMSYNC Load on next PWMSYNC specified by SYNCSEL.

12.2.3 Function Documentation

12.2.3.1 static uint16_t DAC_getRevision (uint32_t *base*) `[inline]`, `[static]`

Get the DAC Revision value

Parameters

base	is the DAC module base address

This function gets the DAC revision value.

Returns

Returns the DAC revision value.

12.2.3.2 static void DAC_setReferenceVoltage (uint32_t *base*, **DAC_ReferenceVoltage** *source*) `[inline]`, `[static]`

Sets the DAC Reference Voltage

Parameters

base	is the DAC module base address
source	is the selected reference voltage

This function sets the DAC reference voltage.

The *source* parameter can have one of two values:

- **DAC_REF_VDAC** - The VDAC reference voltage
- **DAC_REF_ADC_VREFHI** - The ADC VREFHI reference voltage

Returns

None.

12.2.3.3 static void DAC_setGainMode (uint32_t *base*, **DAC_GainMode** *mode*) [inline], [static]

Sets the DAC Gain Mode

Parameters

base	is the DAC module base address
mode	is the selected gain mode

This function sets the DAC gain mode for the buffered output.

The *mode* parameter can have one of two values:

- **DAC_GAIN_ONE** - Gain is set to 1
- **DAC_GAIN_TWO** - Gain is set to 2

Note

This value is only used when *DAC_REF_ADC_VREFHI* is set using DAC_setReferenceVoltage() and internal ADC reference mode is selected.

Returns

None.

12.2.3.4 static void DAC_setLoadMode (uint32_t *base*, **DAC_LoadMode** *mode*) [inline], [static]

Sets the DAC Load Mode

Parameters

base	is the DAC module base address
mode	is the selected load mode

This function sets the DAC load mode.

The *mode* parameter can have one of two values:

- **DAC_LOAD_SYSCLK** - Load on next SYSCLK

- **DAC_LOAD_PWMSYNC** - Load on next PWMSYNC specified by SYNCSEL

Returns
None.

12.2.3.5 static void DAC_setPWMSyncSignal (uint32_t *base,* uint16_t *signal*) [inline], [static]

Sets the DAC PWMSYNC Signal

Parameters

base	is the DAC module base address
signal	is the selected PWM signal

This function sets the DAC PWMSYNC signal.

The *signal* parameter must be set to a number that represents the PWM signal that will be set. For instance, passing 2 into *signal* will select PWM sync signal 2.

Returns
None.

12.2.3.6 static uint16_t DAC_getActiveValue (uint32_t *base*) [inline], [static]

Get the DAC Active Output Value

Parameters

base	is the DAC module base address

This function gets the DAC active output value.

Returns
Returns the DAC active output value.

12.2.3.7 static void DAC_setShadowValue (uint32_t *base,* uint16_t *value*) [inline], [static]

Set the DAC Shadow Output Value

Parameters

base	is the DAC module base address
value	is the 12-bit code to be loaded into the active value register

This function sets the DAC shadow output value.

Returns
None.

12.2.3.8 static uint16_t DAC_getShadowValue (uint32_t *base*) [inline], [static]

Get the DAC Shadow Output Value

Parameters

base	is the DAC module base address

This function gets the DAC shadow output value.

Returns

Returns the DAC shadow output value.

12.2.3.9 static void DAC_enableOutput (uint32_t *base*) [inline], [static]

Enable the DAC Output

Parameters

base	is the DAC module base address

This function enables the DAC output.

Note

A delay is required after enabling the DAC. Further details regarding the exact delay time length can be found in the device datasheet.

Returns

None.

12.2.3.10 static void DAC_disableOutput (uint32_t *base*) [inline], [static]

Disable the DAC Output

Parameters

base	is the DAC module base address

This function disables the DAC output.

Returns

None.

12.2.3.11 static void DAC_setOffsetTrim (uint32_t *base,* int16_t *offset*) [inline], [static]

Set DAC Offset Trim

Parameters

base	is the DAC module base address
offset	is the specified value for the offset trim

This function sets the DAC offset trim. The *offset* value should be a signed number in the range of -128 to 127.

Note

The offset should not be modified unless specifically indicated by TI Errata or other documentation. Modifying the offset value could cause this module to operate outside of the datasheet specifications.

Returns

None.

12.2.3.12 static int16_t DAC_getOffsetTrim (uint32_t *base*) [inline], [static]

Get DAC Offset Trim

Parameters

base	is the DAC module base address

This function gets the DAC offset trim value.

Returns

None.

References DAC_REG_BYTE_MASK.

12.2.3.13 static void DAC_lockRegister (uint32_t *base*, uint16_t *reg*) [inline], [static]

Lock write-access to DAC Register

Parameters

base	is the DAC module base address
reg	is the selected DAC registers

This function locks the write-access to the specified DAC register. Only a system reset can unlock the register once locked.

The *reg* parameter can be an ORed combination of any of the following values:

- **DAC_LOCK_CONTROL** - Lock the DAC control register
- **DAC_LOCK_SHADOW** - Lock the DAC shadow value register
- **DAC_LOCK_OUTPUT** - Lock the DAC output enable/disable register

Returns

None.

References DAC_LOCK_KEY.

12.2.3.14 static bool DAC_isRegisterLocked (uint32_t *base*, uint16_t *reg*) [inline], [static]

Check if DAC Register is locked

Parameters

base	is the DAC module base address
reg	is the selected DAC register locks to check

This function checks if write-access has been locked on the specified DAC register.

The *reg* parameter can be an ORed combination of any of the following values:

- **DAC_LOCK_CONTROL** - Lock the DAC control register
- **DAC_LOCK_SHADOW** - Lock the DAC shadow value register
- **DAC_LOCK_OUTPUT** - Lock the DAC output enable/disable register

Returns

Returns **true** if any of the registers specified are locked, and **false** if all specified registers aren't locked.

12.2.3.15 void DAC_tuneOffsetTrim (uint32_t *base,* float32_t *referenceVoltage*)

Tune DAC Offset Trim

Parameters

base	is the DAC module base address
referenceVoltage	is the reference voltage the DAC module is operating at.

This function adjusts/tunes the DAC offset trim. The *referenceVoltage* value should be a floating point number in the range specified in the device data manual.

Note

Use this function to adjust the DAC offset trim if operating at a reference voltage other than 2.5v. Since this function modifies the DAC offset trim register, it should only be called once after Device_cal. If it is called multiple times after Device_cal, the offset value scaled would be the wrong value.

Returns

None.

References DAC_REG_BYTE_MASK.

13 DCC Module

Introduction 127
API Functions 127

13.1 DCC Introduction

The dual clock comparator (DCC) API provides a set of functions for configuring and using the DCC module. The functions provided allow for setting up the module including the operating modes, configuring the clock sources, enabling interrupt signals, and reading the various clock counters.

13.2 API Functions

Enumerations

- enum DCC_SingleShotMode { DCC_MODE_COUNTER_ZERO, DCC_MODE_COUNTER_ONE }
- enum DCC_RevisionNumber { DCC_REVISION_MINOR, DCC_REVISION_CUSTOM, DCC_REVISION_MAJOR, DCC_REVISION_DESIGN, DCC_REVISION_FUNCTIONAL, DCC_REVISION_SCHEME }
- enum DCC_Count1ClockSource { DCC_COUNT1SRC_PLL, DCC_COUNT1SRC_INTOSC1, DCC_COUNT1SRC_INTOSC2, DCC_COUNT1SRC_PUMOSC, DCC_COUNT1SRC_DCDC, DCC_COUNT1SRC_SYSCLK, DCC_COUNT1SRC_FOSCLK, DCC_COUNT1SRC_ODPOSC, DCC_COUNT1SRC_CROSSBAR, DCC_COUNT1SRC_AUXCLK, DCC_COUNT1SRC_ETPWM, DCC_COUNT1SRC_LSPCLK, DCC_COUNT1SRC_ADCCLK, DCC_COUNT1SRC_WDCLK, DCC_COUNT1SRC_CANX }
- enum DCC_Count0ClockSource { DCC_COUNT0SRC_XTAL, DCC_COUNT0SRC_INTOSC1, DCC_COUNT0SRC_INTOSC2, DCC_COUNT0SRC_TCK, DCC_COUNT0SRC_AUXCLK, DCC_COUNT0SRC_XBAR }

Functions

- static void DCC_enableModule (uint32_t base)
- static void DCC_disableModule (uint32_t base)
- static void DCC_enableErrorSignal (uint32_t base)
- static void DCC_enableDoneSignal (uint32_t base)
- static void DCC_disableErrorSignal (uint32_t base)
- static void DCC_disableDoneSignal (uint32_t base)
- static void DCC_enableSingleShotMode (uint32_t base, DCC_SingleShotMode mode)
- static void DCC_disableSingleShotMode (uint32_t base)
- static bool DCC_getErrorStatus (uint32_t base)
- static bool DCC_getSingleShotStatus (uint32_t base)

- static void DCC_clearErrorFlag (uint32_t base)
- static void DCC_clearDoneFlag (uint32_t base)
- static uint32_t DCC_getCounter0Value (uint32_t base)
- static uint16_t DCC_getValidCounter0Value (uint32_t base)
- static uint32_t DCC_getCounter1Value (uint32_t base)
- static void DCC_setCounter1ClkSource (uint32_t base, DCC_Count1ClockSource source)
- static void DCC_setCounter0ClkSource (uint32_t base, DCC_Count0ClockSource source)
- static uint16_t DCC_getCounter1ClkSource (uint32_t base)
- static uint16_t DCC_getCounter0ClkSource (uint32_t base)
- static void DCC_setCounterSeeds (uint32_t base, uint32_t counter0, uint32_t validCounter0, uint32_t counter1)
- uint16_t DCC_getRevisionNumber (uint32_t base, DCC_RevisionNumber identifier)

13.2.1 Detailed Description

The code for this module is contained in `driverlib/dcc.c`, with `driverlib/dcc.h` containing the API declarations for use by applications.

13.2.2 Enumeration Type Documentation

13.2.2.1 enum **DCC_SingleShotMode**

The following are defines for the mode parameter of the DCC_enableSingleShotMode() function.

Enumerator

DCC_MODE_COUNTER_ZERO Use to stop counting when counter0 and valid0 both reach zero.

DCC_MODE_COUNTER_ONE Use to stop counting when counter1 reaches zero.

13.2.2.2 enum **DCC_RevisionNumber**

The following are defines for the identifier parameter of the DCC_getRevisionNumber() function.

Enumerator

DCC_REVISION_MINOR The module minor revision number.

DCC_REVISION_CUSTOM The custom module revision number.

DCC_REVISION_MAJOR The module major revision number.

DCC_REVISION_DESIGN The module design release number.

DCC_REVISION_FUNCTIONAL The module functional release number.

DCC_REVISION_SCHEME The scheme of the module.

13.2.2.3 enum **DCC_Count1ClockSource**

The following are defines for the source parameter of the DCC_setCounter1ClkSource() function.

Enumerator

DCC_COUNT1SRC_PLL PLL021SSP Clock Out Source.
DCC_COUNT1SRC_INTOSC1 Internal Oscillator 1 Clock Source.
DCC_COUNT1SRC_INTOSC2 Internal Oscillator 2 Clock Source.
DCC_COUNT1SRC_PUMOSC PUMOSC Clock Source.
DCC_COUNT1SRC_DCDC DCDC Clock Source.
DCC_COUNT1SRC_SYSCLK System Clock Source.
DCC_COUNT1SRC_FOSCLK FOS Clock Source.
DCC_COUNT1SRC_ODPOSC ODP Oscillator Clock Source.
DCC_COUNT1SRC_CROSSBAR Input Crossbar Clock Source.
DCC_COUNT1SRC_AUXCLK AUX Clock Source.
DCC_COUNT1SRC_ETPWM ETPWM Clock Source.
DCC_COUNT1SRC_LSPCLK LSP Clock Source.
DCC_COUNT1SRC_ADCCLK ADC Clock Source.
DCC_COUNT1SRC_WDCLK Watch Dog Clock Source.
DCC_COUNT1SRC_CANX CANxBIT Clock Source.

13.2.2.4 enum **DCC_Count0ClockSource**

The following are defines for the source parameter of the DCC_setCounter0ClkSource() function.

Enumerator

DCC_COUNT0SRC_XTAL Accurate Clock Source.
DCC_COUNT0SRC_INTOSC1 Internal Oscillator 1 Clock Source.
DCC_COUNT0SRC_INTOSC2 Internal Oscillator 2 Clock Source.
DCC_COUNT0SRC_TCK Preliminary Clock Source.
DCC_COUNT0SRC_AUXCLK AUX Clock Source.
DCC_COUNT0SRC_XBAR Input XBAR Clock Source.

13.2.3 Function Documentation

13.2.3.1 static void DCC_enableModule (uint32_t *base*) `[inline]`, `[static]`

Enables the DCC module.

Parameters

base	is the DCC module base address

This function starts the DCC counter operation.

Returns

None.

Referenced by SysCtl_isPLLValid().

13.2.3.2 static void DCC_disableModule (uint32_t *base*) [inline], [static]

Disable the DCC module.

Parameters

base	is the DCC module base address

This function stops the DCC counter operation.

Returns
None.

Referenced by SysCtl_isPLLValid().

13.2.3.3 static void DCC_enableErrorSignal (uint32_t *base*) [inline], [static]

Enable DCC Error Signal

Parameters

base	is the DCC module base address

This function enables the error signal interrupt.

Returns
None.

Referenced by SysCtl_isPLLValid().

13.2.3.4 static void DCC_enableDoneSignal (uint32_t *base*) [inline], [static]

Enable DCC Done Signal

Parameters

base	is the DCC module base address

This function enables the done signal interrupt.

Returns
None.

Referenced by SysCtl_isPLLValid().

13.2.3.5 static void DCC_disableErrorSignal (uint32_t *base*) [inline], [static]

Disable DCC Error Signal

Parameters

base	is the DCC module base address

This function disables the error signal interrupt.

Returns
None.

Referenced by SysCtl_isPLLValid().

13.2.3.6 static void DCC_disableDoneSignal (uint32_t *base*) [inline], [static]

Disable DCC Done Signal

Parameters

base	is the DCC module base address

This function disables the done signal interrupt.

Returns

None.

13.2.3.7 static void DCC_enableSingleShotMode (uint32_t *base*, **DCC_SingleShotMode** *mode*) [inline], [static]

Enable DCC Single-Shot Mode

Parameters

base	is the DCC module base address
mode	is the selected Single-Shot operation mode

This function enables the single-shot mode and sets the operation mode.

The *mode* parameter can have one of two values:

- **DCC_MODE_COUNTER_ZERO** - Stops counting when counter0 and valid0 both reach zero
- **DCC_MODE_COUNTER_ONE** - Stops counting when counter1 reaches zero

Returns

None.

References DCC_MODE_COUNTER_ONE, and DCC_MODE_COUNTER_ZERO.

Referenced by SysCtl_isPLLValid().

13.2.3.8 static void DCC_disableSingleShotMode (uint32_t *base*) [inline], [static]

Disable DCC Single-Shot Mode

Parameters

base	is the DCC module base address

This function disables the DCC Single-Shot operation mode

Returns

None.

13.2.3.9 static bool DCC_getErrorStatus (uint32_t *base*) [inline], [static]

Get Error Flag Status

Parameters

base	is the DCC module base address

This function gets the error flag status.

Returns

Returns **true** if an error has occurred, **false** if no errors have occurred.

13.2.3.10 static bool DCC_getSingleShotStatus (uint32_t *base*) [inline], [static]

Get Single-Shot Done Flag Status

Parameters

base	is the DCC module base address

This function gets the single-shot done flag status.

Returns

Returns **true** if single-shot mode has completed, **false** if single-shot mode has not completed.

13.2.3.11 static void DCC_clearErrorFlag (uint32_t *base*) [inline], [static]

Clear Error Status Flag

Parameters

base	is the DCC module base address

This function clears the DCC error status flag.

Returns

None.

13.2.3.12 static void DCC_clearDoneFlag (uint32_t *base*) [inline], [static]

Clear Single-Shot Done Status Flag

Parameters

base	is the DCC module base address

This function clears the DCC single-shot done status flag.

Returns

None.

13.2.3.13 static uint32_t DCC_getCounter0Value (uint32_t *base*) [inline], [static]

Get Current Value of Counter 0

Parameters

base	is the DCC module base address

This function gets current value of counter 0.

Note

Reads of the counter value may not be exact since the read operation is synchronized to the vbus clock.

Returns

Returns the current value of counter 0.

13.2.3.14 static uint16_t DCC_getValidCounter0Value (uint32_t *base*) [inline], [static]

Get Current Value of the Valid Duration Counter for Counter 0

Parameters

base	is the DCC module base address

This function gets current value of the valid duration counter for counter 0.

Note

Reads of the counter value may not be exact since the read operation is synchronized to the vbus clock.

Returns

Returns the current value of the valid duration counter.

13.2.3.15 static uint32_t DCC_getCounter1Value (uint32_t *base*) [inline], [static]

Get Current Value of Counter 1

Parameters

base	is the DCC module base address

This function gets current value of counter 1.

Note

Reads of the counter value may not be exact since the read operation is synchronized to the vbus clock.

Returns

Returns the current value of counter 1.

13.2.3.16 static void DCC_setCounter1ClkSource (uint32_t *base*, **DCC_Count1ClockSource** *source*) [inline], [static]

Set Counter 1 Clock Source

Parameters

base	is the DCC module base address
source	is the selected clock source for counter 1

This function sets the counter 1 clock source.

The *source* parameter can have one of fifteen values:

- **DCC_COUNT1SRC_PLL** - PLL021SSP Clock Out Source
- **DCC_COUNT1SRC_INTOSC1** - Internal Oscillator 1 Clock Source
- **DCC_COUNT1SRC_INTOSC2** - Internal Oscillator 2 Clock Source
- **DCC_COUNT1SRC_PUMOSC** - PUMOSC Clock Source
- **DCC_COUNT1SRC_DCDC** - DCDC Clock Source
- **DCC_COUNT1SRC_SYSCLK** - System Clock Source
- **DCC_COUNT1SRC_FOSCLK** - FOS Clock Source
- **DCC_COUNT1SRC_ODPOSC** - ODP Oscillator Clock Source
- **DCC_COUNT1SRC_CROSSBAR** - Input Crossbar Clock Source
- **DCC_COUNT1SRC_AUXCLK** - AUX Clock Source
- **DCC_COUNT1SRC_ETPWM** - ETPWM Clock Source
- **DCC_COUNT1SRC_LSPCLK** - LSP Clock Source
- **DCC_COUNT1SRC_ADCCLK** - ADC Clock Source
- **DCC_COUNT1SRC_WDCLK** - Watch Dog Clock Source
- **DCC_COUNT1SRC_CANX** - CANxBIT Clock Source

Returns

None.

Referenced by SysCtl_isPLLValid().

13.2.3.17 static void DCC_setCounter0ClkSource (uint32_t *base*, **DCC_Count0ClockSource** *source*) [inline], [static]

Set Counter 0 Clock Source

Parameters

base	is the DCC module base address
source	is the selected clock source for counter 0

This function sets the counter 0 clock source.

The *source* parameter can have one of six values:

- **DCC_COUNT0SRC_XTAL** - Accurate Clock Source
- **DCC_COUNT0SRC_INTOSC1** - Internal Oscillator 1 Clock Source
- **DCC_COUNT0SRC_INTOSC2** - Internal Oscillator 2 Clock Source
- **DCC_COUNT0SRC_TCK** - Preliminary Clock Source
- **DCC_COUNT0SRC_AUXCLK** - AUX Clock Source
- **DCC_COUNT0SRC_XBAR** - Input XBAR Clock Source

Returns
None.

Referenced by SysCtl_isPLLValid().

13.2.3.18 static uint16_t DCC_getCounter1ClkSource (uint32_t *base*) [inline], [static]

Get Counter 1 Clock Source

Parameters

base	is the DCC module base address

This function gets the counter 1 clock source.

Returns
Returns one of the following enumerated source values:

- **DCC_COUNT1SRC_PLL** - PLL021SSP Clock Out Source
- **DCC_COUNT1SRC_INTOSC1** - Internal Oscillator 1 Clock Source
- **DCC_COUNT1SRC_INTOSC2** - Internal Oscillator 2 Clock Source
- **DCC_COUNT1SRC_PUMOSC** - PUMOSC Clock Source
- **DCC_COUNT1SRC_DCDC** - DCDC Clock Source
- **DCC_COUNT1SRC_SYSCLK** - System Clock Source
- **DCC_COUNT1SRC_FOSCLK** - FOS Clock Source
- **DCC_COUNT1SRC_ODPOSC** - ODP Oscillator Clock Source
- **DCC_COUNT1SRC_CROSSBAR** - Input Crossbar Clock Source
- **DCC_COUNT1SRC_AUXCLK** - AUX Clock Source
- **DCC_COUNT1SRC_ETPWM** - ETPWM Clock Source
- **DCC_COUNT1SRC_LSPCLK** - LSP Clock Source
- **DCC_COUNT1SRC_ADCCLK** - ADC Clock Source
- **DCC_COUNT1SRC_WDCLK** - Watch Dog Clock Source
- **DCC_COUNT1SRC_CANX** - CANxBIT Clock Source

13.2.3.19 static uint16_t DCC_getCounter0ClkSource (uint32_t *base*) [inline], [static]

Get Counter 0 Clock Source

Parameters

base	is the DCC module base address

This function gets the counter 0 clock source.

Returns
Returns one of the following enumerated source values:

- **DCC_COUNT0SRC_XTAL** - Accurate Clock Source
- **DCC_COUNT0SRC_INTOSC1** - Internal Oscillator 1 Clock Source
- **DCC_COUNT0SRC_INTOSC2** - Internal Oscillator 2 Clock Source

- **DCC_COUNT0SRC_TCK** - Preliminary Clock Source
- **DCC_COUNT0SRC_AUXCLK** - AUX Clock Source
- **DCC_COUNT0SRC_XBAR** - Input XBAR Clock Source

13.2.3.20 static void DCC_setCounterSeeds (uint32_t *base,* uint32_t *counter0,* uint32_t *validCounter0,* uint32_t *counter1*) [inline], [static]

Set the seed values

Parameters

base	is the DCC module base address
counter0	sets the seed value that gets loaded into Counter 0
validCounter0	sets the seed value that gets loaded into the valid duration counter for Counter 0
counter1	sets the seed value that gets loaded into Counter 1

This function sets the seed values for Counter 0, Valid Duration Counter 0, and Counter 1.

Note

1. Operating DCC with '0' set as the seed value for Counter 0, Valid Duration Counter 0, and/or Counter 1 will result in undefined operation.
2. The Valid Duration Counter 0 is designed to be at least four cycles wide and shouldn't be programmed with a value less than '4'.

Returns

None.

Referenced by SysCtl_isPLLValid().

13.2.3.21 uint16_t DCC_getRevisionNumber (uint32_t *base,* **DCC_RevisionNumber** *identifier*)

Get DCC Version Number

Parameters

base	is the DCC module base address
identifier	is the selected revision number identifier

This function gets the specific version number.

The *identifier* parameter can have one of six values:

- **DCC_REVISION_MINOR** - The minor revision number
- **DCC_REVISION_CUSTOM** - The custom module number
- **DCC_REVISION_MAJOR** - The major revision number
- **DCC_REVISION_DESIGN** - The design release number
- **DCC_REVISION_FUNCTIONAL** - The functional release number
- **DCC_REVISION_SCHEME** - The scheme of the module

Returns

Specified revision number

References DCC_REVISION_CUSTOM, DCC_REVISION_DESIGN, DCC_REVISION_FUNCTIONAL, DCC_REVISION_MAJOR, and DCC_REVISION_MINOR.

14 DCSM Module

Introduction 140
API Functions 140

14.1 DCSM Introduction

The DCSM driver accesses the DCSM COMMON registers. In order to configure the Dual Code Security Module, the user must program the Linkpointer in DCSM OTP as well as the security configuration registers of the Zone Select Blocks in DCSM OTP. The DCSM driver provides functions which secure and unsecure each zone and return the ownership, security status, EXEONLY status of specific RAM modules or Flash sectors. Included are two functions which can claim and release the Flash pump to operate on a specific zone.

14.2 API Functions

Data Structures

- struct DCSM_CSMPasswordKey

Macros

- #define DCSM_O_Z1_CSMPSWD0
- #define DCSM_O_Z1_CSMPSWD1
- #define DCSM_O_Z1_CSMPSWD2
- #define DCSM_O_Z1_CSMPSWD3
- #define DCSM_O_Z2_CSMPSWD0
- #define DCSM_O_Z2_CSMPSWD1
- #define DCSM_O_Z2_CSMPSWD2
- #define DCSM_O_Z2_CSMPSWD3
- #define FLSEM_KEY
- #define DCSM_ALLZERO
- #define DCSM_ALLONE
- #define DCSM_UNSECURE
- #define DCSM_ARMED
- #define DCSM_FLSEM_ALLACCESS_1
- #define DCSM_FLSEM_Z1ACCESS
- #define DCSM_FLSEM_Z2ACCESS
- #define DCSM_FLSEM_ALLACCESS_2

Enumerations

- enum DCSM_Bank { DCSM_BANK0, DCSM_BANK1 }
- enum DCSM_MemoryStatus { DCSM_MEMORY_INACCESSIBLE, DCSM_MEMORY_ZONE1, DCSM_MEMORY_ZONE2, DCSM_MEMORY_FULL_ACCESS }
- enum DCSM_SemaphoreZone { DCSM_FLSEM_ZONE1, DCSM_FLSEM_ZONE2 }

- enum DCSM_SecurityStatus { DCSM_STATUS_SECURE, DCSM_STATUS_UNSECURE, DCSM_STATUS_LOCKED, DCSM_STATUS_BLOCKED }
- enum DCSM_EXEOnlyStatus { DCSM_PROTECTED, DCSM_UNPROTECTED, DCSM_INCORRECT_ZONE }
- enum DCSM_RAMModule {
  DCSM_RAMLS0, DCSM_RAMLS1, DCSM_RAMLS2, DCSM_RAMLS3,
  DCSM_RAMLS4, DCSM_RAMLS5, DCSM_RAMLS6, DCSM_RAMLS7 }
- enum DCSM_Sector {
  DCSM_BANK0_SECTOR0, DCSM_BANK0_SECTOR1, DCSM_BANK0_SECTOR2,
  DCSM_BANK0_SECTOR3,
  DCSM_BANK0_SECTOR4, DCSM_BANK0_SECTOR5, DCSM_BANK0_SECTOR6,
  DCSM_BANK0_SECTOR7,
  DCSM_BANK0_SECTOR8, DCSM_BANK0_SECTOR9, DCSM_BANK0_SECTOR10,
  DCSM_BANK0_SECTOR11,
  DCSM_BANK0_SECTOR12, DCSM_BANK0_SECTOR13, DCSM_BANK0_SECTOR14,
  DCSM_BANK0_SECTOR15,
  DCSM_BANK1_SECTOR0, DCSM_BANK1_SECTOR1, DCSM_BANK1_SECTOR2,
  DCSM_BANK1_SECTOR3,
  DCSM_BANK1_SECTOR4, DCSM_BANK1_SECTOR5, DCSM_BANK1_SECTOR6,
  DCSM_BANK1_SECTOR7,
  DCSM_BANK1_SECTOR8, DCSM_BANK1_SECTOR9, DCSM_BANK1_SECTOR10,
  DCSM_BANK1_SECTOR11,
  DCSM_BANK1_SECTOR12, DCSM_BANK1_SECTOR13, DCSM_BANK1_SECTOR14,
  DCSM_BANK1_SECTOR15 }

Functions

- static void DCSM_secureZone1 (void)
- static void DCSM_secureZone2 (void)
- static DCSM_SecurityStatus DCSM_getZone1CSMSecurityStatus (void)
- static DCSM_SecurityStatus DCSM_getZone2CSMSecurityStatus (void)
- static uint16_t DCSM_getZone1ControlStatus (void)
- static uint16_t DCSM_getZone2ControlStatus (void)
- static DCSM_MemoryStatus DCSM_getRAMZone (DCSM_RAMModule module)
- static DCSM_MemoryStatus DCSM_getFlashSectorZone (DCSM_Sector sector)
- static uint32_t DCSM_getZone1LinkPointerError (DCSM_Bank bank)
- static uint32_t DCSM_getZone2LinkPointerError (DCSM_Bank bank)
- static bool DCSM_getFlashErrorStatus (void)
- static void DCSM_clearFlashErrorStatus (void)
- static void DCSM_forceFlashErrorStatus (void)
- void DCSM_unlockZone1CSM (const DCSM_CSMPasswordKey ∗const psCMDKey)
- void DCSM_unlockZone2CSM (const DCSM_CSMPasswordKey ∗const psCMDKey)
- DCSM_EXEOnlyStatus DCSM_getZone1FlashEXEStatus (DCSM_Sector sector)
- DCSM_EXEOnlyStatus DCSM_getZone1RAMEXEStatus (DCSM_RAMModule module)
- DCSM_EXEOnlyStatus DCSM_getZone2FlashEXEStatus (DCSM_Sector sector)
- DCSM_EXEOnlyStatus DCSM_getZone2RAMEXEStatus (DCSM_RAMModule module)
- bool DCSM_claimZoneSemaphore (DCSM_SemaphoreZone zone)
- bool DCSM_releaseZoneSemaphore (void)

14.2.1 Detailed Description

The code for this module is contained in `driverlib/dcsm.c`, with `driverlib/dcsm.h` containing the API declarations for use by applications.

14.2.2 Enumeration Type Documentation

14.2.2.1 enum **DCSM_Bank**

Values to distinguish which bank. These values can be passed to DCSM_getZone1FlashEXEStatus(), DCSM_getZone2FlashEXEStatus(), DCSM_getFlashSectorZone(), DCSM_getZone1LinkPointerError(), DCSM_getZone2LinkPointerError().

Enumerator

DCSM_BANK0 Bank 0.

DCSM_BANK1 Bank 1.

14.2.2.2 enum **DCSM_MemoryStatus**

Values to distinguish the status of RAM or FLASH sectors. These values describe which zone the memory location belongs too. These values can be returned from DCSM_getRAMZone(), DCSM_getFlashSectorZone().

Enumerator

DCSM_MEMORY_INACCESSIBLE Inaccessible.

DCSM_MEMORY_ZONE1 Zone 1.

DCSM_MEMORY_ZONE2 Zone 2.

DCSM_MEMORY_FULL_ACCESS Full access.

14.2.2.3 enum **DCSM_SemaphoreZone**

Values to pass to DCSM_claimZoneSemaphore(). These values are used to describe the zone that can write to Flash Wrapper registers.

Enumerator

DCSM_FLSEM_ZONE1 Flash semaphore Zone 1.

DCSM_FLSEM_ZONE2 Flash semaphore Zone 2.

14.2.2.4 enum **DCSM_SecurityStatus**

Values to distinguish the security status of the zones. These values can be returned from DCSM_getZone1CSMSecurityStatus(), DCSM_getZone2CSMSecurityStatus().

Enumerator

DCSM_STATUS_SECURE Secure.

DCSM_STATUS_UNSECURE Unsecure.

DCSM_STATUS_LOCKED Locked.

DCSM_STATUS_BLOCKED Blocked.

14.2.2.5 enum **DCSM_EXEOnlyStatus**

Values to decribe the EXEONLY Status. These values are returned from to DCSM_getZone1RAMEXEStatus(), DCSM_getZone2RAMEXEStatus(), DCSM_getZone1FlashEXEStatus(), DCSM_getZone2FlashEXEStatus().

Enumerator

DCSM_PROTECTED Protected.

DCSM_UNPROTECTED Unprotected.

DCSM_INCORRECT_ZONE Incorrect Zone.

14.2.2.6 enum **DCSM_RAMModule**

Values to distinguish RAM Module. These values can be passed to DCSM_getZone1RAMEXEStatus() DCSM_getZone2RAMEXEStatus(), DCSM_getRAMZone().

Enumerator

DCSM_RAMLS0 RAMLS0.

DCSM_RAMLS1 RAMLS1.

DCSM_RAMLS2 RAMLS2.

DCSM_RAMLS3 RAMLS3.

DCSM_RAMLS4 RAMLS4.

DCSM_RAMLS5 RAMLS5.

DCSM_RAMLS6 RAMLS6.

DCSM_RAMLS7 RAMLS7.

14.2.2.7 enum **DCSM_Sector**

Values to distinguish Flash Sector. These values can be passed to DCSM_getZone1FlashEXEStatus() DCSM_getZone2FlashEXEStatus(), DCSM_getFlashSectorZone().

Enumerator

DCSM_BANK0_SECTOR0 Bank 0 - Sector 0.

DCSM_BANK0_SECTOR1 Bank 0 - Sector 1.

DCSM_BANK0_SECTOR2 Bank 0 - Sector 2.

DCSM_BANK0_SECTOR3 Bank 0 - Sector 3.

DCSM_BANK0_SECTOR4 Bank 0 - Sector 4.

DCSM_BANK0_SECTOR5 Bank 0 - Sector 5.

DCSM_BANK0_SECTOR6 Bank 0 - Sector 6.

DCSM_BANK0_SECTOR7 Bank 0 - Sector 7.

DCSM_BANK0_SECTOR8 Bank 0 - Sector 8.

DCSM_BANK0_SECTOR9 Bank 0 - Sector 9.

DCSM_BANK0_SECTOR10 Bank 0 - Sector 10.

DCSM_BANK0_SECTOR11 Bank 0 - Sector 11.

DCSM_BANK0_SECTOR12 Bank 0 - Sector 12.

DCSM_BANK0_SECTOR13 Bank 0 - Sector 13.
DCSM_BANK0_SECTOR14 Bank 0 - Sector 14.
DCSM_BANK0_SECTOR15 Bank 0 - Sector 15.
DCSM_BANK1_SECTOR0 Bank 1 - Sector 0.
DCSM_BANK1_SECTOR1 Bank 1 - Sector 1.
DCSM_BANK1_SECTOR2 Bank 1 - Sector 2.
DCSM_BANK1_SECTOR3 Bank 1 - Sector 3.
DCSM_BANK1_SECTOR4 Bank 1 - Sector 4.
DCSM_BANK1_SECTOR5 Bank 1 - Sector 5.
DCSM_BANK1_SECTOR6 Bank 1 - Sector 6.
DCSM_BANK1_SECTOR7 Bank 1 - Sector 7.
DCSM_BANK1_SECTOR8 Bank 1 - Sector 8.
DCSM_BANK1_SECTOR9 Bank 1 - Sector 9.
DCSM_BANK1_SECTOR10 Bank 1 - Sector 10.
DCSM_BANK1_SECTOR11 Bank 1 - Sector 11.
DCSM_BANK1_SECTOR12 Bank 1 - Sector 12.
DCSM_BANK1_SECTOR13 Bank 1 - Sector 13.
DCSM_BANK1_SECTOR14 Bank 1 - Sector 14.
DCSM_BANK1_SECTOR15 Bank 1 - Sector 15.

14.2.3 Function Documentation

14.2.3.1 static void DCSM_secureZone1 (void) `[inline]`, `[static]`

Secures zone 1 by setting the FORCESEC bit of Z1_CR register

This function resets the state of the zone. If the zone is unlocked, it will lock (secure) the zone and also reset all the bits in the Control Register.

Returns
None.

14.2.3.2 static void DCSM_secureZone2 (void) `[inline]`, `[static]`

Secures zone 2 by setting the FORCESEC bit of Z2_CR register

This function resets the state of the zone. If the zone is unlocked, it will lock (secure) the zone and also reset all the bits in the Control Register.

Returns
None.

14.2.3.3 static **DCSM_SecurityStatus** DCSM_getZone1CSMSecurityStatus (void) `[inline]`, `[static]`

Returns the CSM security status of zone 1

This function returns the security status of zone 1 CSM

Returns

Returns security status as an enumerated type DCSM_SecurityStatus.

References DCSM_STATUS_BLOCKED, DCSM_STATUS_LOCKED, DCSM_STATUS_SECURE, and DCSM_STATUS_UNSECURE.

14.2.3.4 static **DCSM_SecurityStatus** DCSM_getZone2CSMSecurityStatus (void) `[inline]`, `[static]`

Returns the CSM security status of zone 2

This function returns the security status of zone 2 CSM

Returns

Returns security status as an enumerated type DCSM_SecurityStatus.

References DCSM_STATUS_BLOCKED, DCSM_STATUS_LOCKED, DCSM_STATUS_SECURE, and DCSM_STATUS_UNSECURE.

14.2.3.5 static uint16_t DCSM_getZone1ControlStatus (void) `[inline]`, `[static]`

Returns the Control Status of zone 1

This function returns the Control Status of zone 1 CSM

Returns

Returns the contents of the Control Register which can be used with provided defines.

14.2.3.6 static uint16_t DCSM_getZone2ControlStatus (void) `[inline]`, `[static]`

Returns the Control Status of zone 2

This function returns the Control Status of zone 2 CSM

Returns

Returns the contents of the Control Register which can be used with the provided defines.

14.2.3.7 static **DCSM_MemoryStatus** DCSM_getRAMZone (**DCSM_RAMModule** *module*) `[inline]`, `[static]`

Returns the security zone a RAM section belongs to

Parameters

module	is the RAM module value. Valid values are type DCSM_RAMModule ■ **DCSM_RAMLS0** ■ **DCSM_RAMLS1** ■ **DCSM_RAMLS2** ■ **DCSM_RAMLS3** ■ **DCSM_RAMLS4** ■ **DCSM_RAMLS5** ■ **DCSM_RAMLS6** ■ **DCSM_RAMLS7**

This function returns the security zone a RAM section belongs to.

Returns

Returns DCSM_MEMORY_INACCESSIBLE if the section is inaccessible, DCSM_MEMORY_ZONE1 if the section belongs to zone 1, DCSM_MEMORY_ZONE2 if the section belongs to zone 2 and DCSM_MEMORY_FULL_ACCESS if the section doesn't belong to any zone (or if the section is unsecure).

Referenced by DCSM_getZone1RAMEXEStatus(), and DCSM_getZone2RAMEXEStatus().

14.2.3.8 static **DCSM_MemoryStatus** DCSM_getFlashSectorZone (**DCSM_Sector** *sector*) [inline], [static]

Returns the security zone a flash sector belongs to

Parameters

sector	is the flash sector value. Use DCSM_Sector type.

This function returns the security zone a flash sector belongs to.

Returns

Returns DCSM_MEMORY_INACCESSIBLE if the section is inaccessible , DCSM_MEMORY_ZONE1 if the section belongs to zone 1, DCSM_MEMORY_ZONE2 if the section belongs to zone 2 and DCSM_MEMORY_FULL_ACCESS if the section doesn't belong to any zone (or if the section is unsecure)..

References DCSM_BANK0_SECTOR15.

Referenced by DCSM_getZone1FlashEXEStatus(), and DCSM_getZone2FlashEXEStatus().

14.2.3.9 static uint32_t DCSM_getZone1LinkPointerError (**DCSM_Bank** *bank*) [inline], [static]

Read Zone 1 Link Pointer Error

Parameters

bank	is the DCSM_Bank to operate on.

A non-zero value indicates an error on the bit position that is set to 1.

Returns

Returns the value of the Zone 1 Link Pointer error.

References DCSM_BANK0.

14.2.3.10 static uint32_t DCSM_getZone2LinkPointerError (**DCSM_Bank** *bank*) [inline], [static]

Read Zone 2 Link Pointer Error

Parameters

bank	is the DCSM_Bank to operate on.

A non-zero value indicates an error on the bit position that is set to 1.

Returns

Returns the value of the Zone 2 Link Pointer error.

References DCSM_BANK0.

14.2.3.11 static bool DCSM_getFlashErrorStatus (void) [inline], [static]

Get the status of the security configuration load from USER-OTP or sector error status

Returns

Returns 0 if no error in loading security information from USER-OTP, 1 if an error has occurred in the load from USER-OTP.

14.2.3.12 static void DCSM_clearFlashErrorStatus (void) [inline], [static]

Clear the Flash Error Status bit

Write a '1' to the clear bit to clear the sector error status bit.

Returns

None.

14.2.3.13 static void DCSM_forceFlashErrorStatus (void) [inline], [static]

Set the force Flash Error Status bit

Write a '1' to force bit to set the sector error status bit.

Returns
None.

14.2.3.14 void DCSM_unlockZone1CSM (const **DCSM_CSMPasswordKey** ∗const *psCMDKey*)

Unlocks Zone 1 CSM.

Parameters

psCMDKey	is a pointer to the DCSM_CSMPasswordKey struct that has the CSM password for zone 1.

This function unlocks the CSM password. It first reads the four password locations in the User OTP. If any of the password values is different from 0xFFFFFFFF, it unlocks the device by writing the provided passwords into CSM Key registers

Returns
None.

References DCSM_O_Z1_CSMPSWD0, DCSM_O_Z1_CSMPSWD1, DCSM_O_Z1_CSMPSWD2, and DCSM_O_Z1_CSMPSWD3.

14.2.3.15 void DCSM_unlockZone2CSM (const **DCSM_CSMPasswordKey** ∗const *psCMDKey*)

Unlocks Zone 2 CSM.

Parameters

psCMDKey	is a pointer to the CSMPSWDKEY that has the CSM password for zone 2.

This function unlocks the CSM password. It first reads the four password locations in the User OTP. If any of the password values is different from 0xFFFFFFFF, it unlocks the device by writing the provided passwords into CSM Key registers

Returns
None.

References DCSM_O_Z2_CSMPSWD0, DCSM_O_Z2_CSMPSWD1, DCSM_O_Z2_CSMPSWD2, and DCSM_O_Z2_CSMPSWD3.

14.2.3.16 **DCSM_EXEOnlyStatus** DCSM_getZone1FlashEXEStatus (**DCSM_Sector** *sector*)

Returns the EXE-ONLY status of zone 1 for a flash sector

Parameters

sector	is the flash sector value. Use DCSM_Sector type.

This function takes in a valid sector value and returns the status of EXE ONLY security protection for the sector.

Returns

Returns DCSM_PROTECTED if the sector is EXE-ONLY protected, DCSM_UNPROTECTED if the sector is not EXE-ONLY protected, DCSM_INCORRECT_ZONE if sector does not belong to this zone.

References DCSM_BANK0_SECTOR15, DCSM_getFlashSectorZone(), DCSM_INCORRECT_ZONE, and DCSM_MEMORY_ZONE1.

14.2.3.17 **DCSM_EXEOnlyStatus** DCSM_getZone1RAMEXEStatus (**DCSM_RAMModule** *module*)

Returns the EXE-ONLY status of zone 1 for a RAM module

Parameters

module	is the RAM module value. Valid values are type DCSM_RAMModule ■ **DCSM_RAMLS0** ■ **DCSM_RAMLS1** ■ **DCSM_RAMLS2** ■ **DCSM_RAMLS3** ■ **DCSM_RAMLS4** ■ **DCSM_RAMLS5** ■ **DCSM_RAMLS6** ■ **DCSM_RAMLS7**

This function takes in a valid module value and returns the status of EXE ONLY security protection for that module.

Returns

Returns DCSM_PROTECTED if the module is EXE-ONLY protected, DCSM_UNPROTECTED if the module is not EXE-ONLY protected, DCSM_INCORRECT_ZONE if module does not belong to this zone.

References DCSM_getRAMZone(), DCSM_INCORRECT_ZONE, and DCSM_MEMORY_ZONE1.

14.2.3.18 **DCSM_EXEOnlyStatus** DCSM_getZone2FlashEXEStatus (**DCSM_Sector** *sector*)

Returns the EXE-ONLY status of zone 2 for a flash sector

Parameters

sector	is the flash sector value. Use DCSM_Sector type.

This function takes in a valid sector value and returns the status of EXE ONLY security protection for the sector.

Returns

Returns DCSM_PROTECTED if the sector is EXE-ONLY protected, DCSM_UNPROTECTED if the sector is not EXE-ONLY protected, DCSM_INCORRECT_ZONE if sector does not belong to this zone.

References DCSM_BANK0_SECTOR15, DCSM_getFlashSectorZone(), DCSM_INCORRECT_ZONE, and DCSM_MEMORY_ZONE2.

14.2.3.19 **DCSM_EXEOnlyStatus** DCSM_getZone2RAMEXEStatus (**DCSM_RAMModule** *module*)

Returns the EXE-ONLY status of zone 2 for a RAM module

Parameters

module	is the RAM module value. Valid values are type DCSM_RAMModule ■ **DCSM_RAMLS0** ■ **DCSM_RAMLS1** ■ **DCSM_RAMLS2** ■ **DCSM_RAMLS3** ■ **DCSM_RAMLS4** ■ **DCSM_RAMLS5** ■ **DCSM_RAMLS6** ■ **DCSM_RAMLS7**

This function takes in a valid module value and returns the status of EXE ONLY security protection for that module.

Returns

Returns DCSM_PROTECTED if the module is EXE-ONLY protected, DCSM_UNPROTECTED if the module is not EXE-ONLY protected, DCSM_INCORRECT_ZONE if module does not belong to this zone.

References DCSM_getRAMZone(), DCSM_INCORRECT_ZONE, and DCSM_MEMORY_ZONE2.

14.2.3.20 bool DCSM_claimZoneSemaphore (**DCSM_SemaphoreZone** *zone*)

Claims the zone semaphore which allows access to the Flash Wrapper register for that zone.

Parameters

zone	is the zone which is trying to claim the semaphore which allows access to the Flash Wrapper registers.

Returns

Returns true for a successful semaphore capture, false if it was unable to capture the semaphore.

References FLSEM_KEY.

14.2.3.21 bool DCSM_releaseZoneSemaphore (void)

Releases the zone semaphore.

Returns

Returns true if was successful in releasing the zone semaphore and false if it was unsuccessful in releasing the zone semaphore.

Note

If the calling function is not in the right zone to be able to access this register, it will return a false.

References FLSEM_KEY.

15 DMA Module

Introduction 152
API Functions 152

15.1 DMA Introduction

The direct memory access (DMA) API provides a set of functions to configure transfers of data between peripherals or memory using the device's six-channel DMA module. Functions are provided to configure which event triggers a DMA transfer, to configure the locations, sizes, and behaviors of the transfers, and to set up and handle interrupts.

15.2 API Functions

Enumerations

- enum DMA_InterruptMode { DMA_INT_AT_BEGINNING, DMA_INT_AT_END }
- enum DMA_EmulationMode { DMA_EMULATION_STOP, DMA_EMULATION_FREE_RUN }

Functions

- static void DMA_initController (void)
- static void DMA_setEmulationMode (DMA_EmulationMode mode)
- static void DMA_enableTrigger (uint32_t base)
- static void DMA_disableTrigger (uint32_t base)
- static void DMA_forceTrigger (uint32_t base)
- static void DMA_clearTriggerFlag (uint32_t base)
- static bool DMA_getTriggerFlagStatus (uint32_t base)
- static void DMA_startChannel (uint32_t base)
- static void DMA_stopChannel (uint32_t base)
- static void DMA_enableInterrupt (uint32_t base)
- static void DMA_disableInterrupt (uint32_t base)
- static void DMA_enableOverrunInterrupt (uint32_t base)
- static void DMA_disableOverrunInterrupt (uint32_t base)
- static void DMA_clearErrorFlag (uint32_t base)
- static void DMA_setInterruptMode (uint32_t base, DMA_InterruptMode mode)
- static void DMA_setPriorityMode (bool ch1IsHighPri)
- void DMA_configAddresses (uint32_t base, const void *destAddr, const void *srcAddr)
- void DMA_configBurst (uint32_t base, uint16_t size, int16_t srcStep, int16_t destStep)
- void DMA_configTransfer (uint32_t base, uint32_t transferSize, int16_t srcStep, int16_t destStep)
- void DMA_configWrap (uint32_t base, uint32_t srcWrapSize, int16_t srcStep, uint32_t destWrapSize, int16_t destStep)
- void DMA_configMode (uint32_t base, DMA_Trigger trigger, uint32_t config)

15.2.1 Detailed Description

The DMA API includes functions that configure the module as a whole and functions that configure the individual channels. Functions that fall into the former category are DMA_initController(), DMA_setEmulationMode(), and DMA_setPriorityMode(). The functions that can be configured by channel can easily be identified as they take a base address as their first parameter.

The DMA_configMode() function is used to configure the event that triggers a DMA transfer as well as several other properties of a transfer for the specified channel. Other functions that can be used to control the trigger from within the DMA module are DMA_enableTrigger(), DMA_disableTrigger(), DMA_forceTrigger(), DMA_clearTriggerFlag(), and DMA_getTriggerFlagStatus(). Note that DMA_forceTrigger() is used to trigger a transfer from software.

DMA_configAddresses() is used to write to both the beginning and current address pointer registers. The manner in which these addresses are incremented and decremented as bursts and transfers complete is configured using DMA_configBurst(), DMA_configTransfer(), and DMA_configWrap(). All sizes are in terms of 16-bit words.

DMA_enableInterrupt(), DMA_disableInterrupt(), and DMA_setInterruptMode() configure a channel interrupt that will be generated either at the beginning or the end of a transfer. An additional overrun error interrupt that is ORed into the channel interrupt signal can be configured using DMA_enableOverrunInterrupt(), and DMA_disableOverrunInterrupt(). This error can be cleared using DMA_clearErrorFlag().

When configuration is complete, DMA_startChannel() can be called to start the DMA channel running and it will wait for the first trigger. To halt the operation of the channel DMA_stopChannel() may be used.

The code for this module is contained in `driverlib/dma.c`, with `driverlib/dma.h` containing the API declarations for use by applications.

15.2.2 Enumeration Type Documentation

15.2.2.1 enum **DMA_InterruptMode**

Values that can be passed to DMA_setInterruptMode() as the *mode* parameter.

Enumerator

DMA_INT_AT_BEGINNING DMA interrupt is generated at the beginning of a transfer.

DMA_INT_AT_END DMA interrupt is generated at the end of a transfer.

15.2.2.2 enum **DMA_EmulationMode**

Values that can be passed to DMA_setEmulationMode() as the *mode* parameter.

Enumerator

DMA_EMULATION_STOP Transmission stops after current read-write access is completed.

DMA_EMULATION_FREE_RUN Continue DMA operation regardless of emulation suspend.

15.2.3 Function Documentation

15.2.3.1 static void DMA_initController (void) [inline], [static]

Initializes the DMA controller to a known state.

This function configures does a hard reset of the DMA controller in order to put it into a known state. The function also sets the DMA to run free during an emulation suspend (see the field DEBUGCTRL.FREE for more info).

Returns
None.

15.2.3.2 static void DMA_setEmulationMode (**DMA_EmulationMode** *mode*) [inline], [static]

Sets DMA emulation mode.

Parameters

mode	is the emulation mode to be selected.

This function sets the behavior of the DMA operation when an emulation suspend occurs. The *mode* parameter can be one of the following:

- **DMA_EMULATION_STOP** - DMA runs until the current read-write access is completed.
- **DMA_EMULATION_FREE_RUN** - DMA operation continues regardless of a the suspend.

Returns
None.

References DMA_EMULATION_STOP.

15.2.3.3 static void DMA_enableTrigger (uint32_t *base*) [inline], [static]

Enables peripherals to trigger a DMA transfer.

Parameters

base	is the base address of the DMA channel control registers.

This function enables the selected peripheral trigger to start a DMA transfer on the specified channel.

Returns
None.

15.2.3.4 static void DMA_disableTrigger (uint32_t *base*) [inline], [static]

Disables peripherals from triggering a DMA transfer.

Parameters

base	is the base address of the DMA channel control registers.

This function disables the selected peripheral trigger from starting a DMA transfer on the specified channel. This also disables the use of the software force using the DMA_forceTrigger() API.

Returns

None.

15.2.3.5 static void DMA_forceTrigger (uint32_t *base*) [inline], [static]

Force a peripheral trigger to a DMA channel.

Parameters

base	is the base address of the DMA channel control registers.

This function sets the peripheral trigger flag and if triggering a DMA burst is enabled (see DMA_enableTrigger()), a DMA burst transfer will be forced.

Returns

None.

15.2.3.6 static void DMA_clearTriggerFlag (uint32_t *base*) [inline], [static]

Clears a DMA channel's peripheral trigger flag.

Parameters

base	is the base address of the DMA channel control registers.

This function clears the peripheral trigger flag. Normally, you would use this function when initializing the DMA for the first time. The flag is cleared automatically when the DMA starts the first burst of a transfer.

Returns

None.

15.2.3.7 static bool DMA_getTriggerFlagStatus (uint32_t *base*) [inline], [static]

Gets the status of a DMA channel's peripheral trigger flag.

Parameters

base	is the base address of the DMA channel control registers.

This function returns **true** if a peripheral trigger event has occurred The flag is automatically cleared when the first burst transfer begins, but if needed, it can be cleared using DMA_clearTriggerFlag().

Returns

Returns **true** if a peripheral trigger event has occurred and its flag is set. Returns **false** otherwise.

15.2.3.8 static void DMA_startChannel (uint32_t *base*) `[inline]`, `[static]`

Starts a DMA channel.

Parameters

base	is the base address of the DMA channel control registers.

This function starts the DMA running, typically after you have configured it. It will wait for the first trigger event to start operation. To halt the channel use DMA_stopChannel().

Returns

None.

15.2.3.9 static void DMA_stopChannel (uint32_t *base*) [inline], [static]

Halts a DMA channel.

Parameters

base	is the base address of the DMA channel control registers.

This function halts the DMA at its current state and any current read-write access is completed. To start the channel again use DMA_startChannel().

Returns

None.

15.2.3.10 static void DMA_enableInterrupt (uint32_t *base*) [inline], [static]

Enables a DMA channel interrupt source.

Parameters

base	is the base address of the DMA channel control registers.

This function enables the indicated DMA channel interrupt source.

Returns

None.

15.2.3.11 static void DMA_disableInterrupt (uint32_t *base*) [inline], [static]

Disables a DMA channel interrupt source.

Parameters

base	is the base address of the DMA channel control registers.

This function disables the indicated DMA channel interrupt source.

Returns

None.

15.2.3.12 static void DMA_enableOverrunInterrupt (uint32_t *base*) [inline], [static]

Enables the DMA channel overrun interrupt.

Parameters

base	is the base address of the DMA channel control registers.

This function enables the indicated DMA channel's ability to generate an interrupt upon the detection of an overrun. An overrun is when a peripheral event trigger is received by the DMA before a previous trigger on that channel had been serviced and its flag had been cleared.

Note that this is the same interrupt signal as the interrupt that gets generated at the beginning/end of a transfer. That interrupt must first be enabled using DMA_enableInterrupt() in order for the overrun interrupt to be generated.

Returns
None.

15.2.3.13 static void DMA_disableOverrunInterrupt (uint32_t *base*) `[inline]`, `[static]`

Disables the DMA channel overrun interrupt.

Parameters

base	is the base address of the DMA channel control registers.

This function disables the indicated DMA channel's ability to generate an interrupt upon the detection of an overrun.

Returns
None.

15.2.3.14 static void DMA_clearErrorFlag (uint32_t *base*) `[inline]`, `[static]`

Clears the DMA channel error flags.

Parameters

base	is the base address of the DMA channel control registers.

This function clears both the DMA channel's sync error flag and its overrun error flag.

Returns
None.

15.2.3.15 static void DMA_setInterruptMode (uint32_t *base*, **DMA_InterruptMode** *mode*) `[inline]`, `[static]`

Sets the interrupt generation mode of a DMA channel interrupt.

Parameters

base	is the base address of the DMA channel control registers.
mode	is a flag to indicate the channel interrupt mode.

This function sets the channel interrupt mode. When the *mode* parameter is **DMA_INT_AT_END**, the DMA channel interrupt will be generated at the end of the transfer. If **DMA_INT_AT_BEGINNING**, the interrupt will be generated at the beginning of a new transfer. Generating at the beginning of a new transfer is the default behavior.

Returns
None.

References DMA_INT_AT_END.

15.2.3.16 static void DMA_setPriorityMode (bool *ch1IsHighPri*) [inline], [static]

Sets the DMA channel priority mode.

Parameters

ch1IsHighPri	is a flag to indicate the channel interrupt mode.

This function sets the channel interrupt mode. When the *ch1IsHighPri* parameter is **false**, the DMA channels are serviced in round-robin mode. This is the default behavior.

If **true**, channel 1 will be given higher priority than the other channels. This means that if a channel 1 trigger occurs, the current word transfer on any other channel is completed and channel 1 is serviced for the complete burst count. The lower-priority channel's interrupted transfer will then resume.

Returns
None.

15.2.3.17 void DMA_configAddresses (uint32_t *base,* const void ∗ *destAddr,* const void ∗ *srcAddr*)

Configures the DMA channel

Parameters

base	is the base address of the DMA channel control registers.
∗*destAddr*	is the interrupt source that triggers a DMA transfer.
∗*srcAddr*	is a bit field of several configuration selections.

This function configures the source and destination addresses of a DMA channel. The parameters are pointers to the data to be transferred.

Returns
None.

15.2.3.18 void DMA_configBurst (uint32_t *base,* uint16_t *size,* int16_t *srcStep,* int16_t *destStep*)

Configures the DMA channel's burst settings.

Parameters

base	is the base address of the DMA channel control registers.
size	is the number of words transferred per burst.
srcStep	is the amount to increment or decrement the source address after each word of a burst.
destStep	is the amount to increment or decrement the destination address after each word of a burst.

This function configures the size of each burst and the address step size.

The *size* parameter is the number of words that will be transferred during a single burst. Possible amounts range from 1 word to 32 words.

The *srcStep* and *destStep* parameters specify the address step that should be added to the source and destination addresses after each transferred word of a burst. Only signed values from -4096 to 4095 are valid.

Note

Note that regardless of what data size (configured by DMA_configMode()) is used, parameters are in terms of 16-bits words.

Returns

None.

15.2.3.19 void DMA_configTransfer (uint32_t *base,* uint32_t *transferSize,* int16_t *srcStep,* int16_t *destStep*)

Configures the DMA channel's transfer settings.

Parameters

base	is the base address of the DMA channel control registers.
transferSize	is the number of bursts per transfer.
srcStep	is the amount to increment or decrement the source address after each burst of a transfer unless a wrap occurs.
destStep	is the amount to increment or decrement the destination address after each burst of a transfer unless a wrap occurs.

This function configures the transfer size and the address step that is made after each burst.

The *transferSize* parameter is the number of bursts per transfer. If DMA channel interrupts are enabled, they will occur after this number of bursts have completed. The maximum number of bursts is 65536.

The *srcStep* and *destStep* parameters specify the address step that should be added to the source and destination addresses after each transferred burst of a transfer. Only signed values from -4096 to 4095 are valid. If a wrap occurs, these step values will be ignored. Wrapping is configured with DMA_configWrap().

Note

Note that regardless of what data size (configured by DMA_configMode()) is used, parameters are in terms of 16-bits words.

Returns

None.

15.2.3.20 void DMA_configWrap (uint32_t *base*, uint32_t *srcWrapSize*, int16_t *srcStep*, uint32_t *destWrapSize*, int16_t *destStep*)

Configures the DMA channel's wrap settings.

Parameters

base	is the base address of the DMA channel control registers.
srcWrapSize	is the number of bursts to be transferred before a wrap of the source address occurs.
srcStep	is the amount to increment or decrement the source address after each burst of a transfer unless a wrap occurs.
destWrapSize	is the number of bursts to be transferred before a wrap of the destination address occurs.
destStep	is the amount to increment or decrement the destination address after each burst of a transfer unless a wrap occurs.

This function configures the DMA channel's wrap settings.

The *srcWrapSize* and *destWrapSize* parameters are the number of bursts that are to be transferred before their respective addresses are wrapped. The maximum wrap size is 65536 bursts.

The *srcStep* and *destStep* parameters specify the address step that should be added to the source and destination addresses when the wrap occurs. Only signed values from -4096 to 4095 are valid.

Note

Note that regardless of what data size (configured by DMA_configMode()) is used, parameters are in terms of 16-bits words.

Returns

None.

15.2.3.21 void DMA_configMode (uint32_t *base*, DMA_Trigger *trigger*, uint32_t *config*)

Configures the DMA channel trigger and mode.

Parameters

base	is the base address of the DMA channel control registers.
trigger	is the interrupt source that triggers a DMA transfer.
config	is a bit field of several configuration selections.

This function configures the DMA channel's trigger and mode.

The *trigger* parameter is the interrupt source that will trigger the start of a DMA transfer.

The *config* parameter is the logical OR of the following values:

- **DMA_CFG_ONESHOT_DISABLE** or **DMA_CFG_ONESHOT_ENABLE**. If enabled, the subsequent burst transfers occur without additional event triggers after the first event trigger. If disabled, only one burst transfer is performed per event trigger.

- **DMA_CFG_CONTINUOUS_DISABLE** or **DMA_CFG_CONTINUOUS_ENABLE**. If enabled the DMA reinitializes when the transfer count is zero and waits for the next interrupt event trigger. If disabled, the DMA stops and clears the run status bit.
- **DMA_CFG_SIZE_16BIT** or **DMA_CFG_SIZE_32BIT**. This setting selects whether the databus width is 16 or 32 bits.

Returns

None.

16 ECAP Module

Introduction 165
API Functions 165

16.1 ECAP Introduction

The Enhanced Capture (eCAP) API provides a set of functions for configuring and using the eCAP module. Functions are provided to utilize both the capture and PWM capability of the eCAP module. The APIs allow for the selection and characterization of the input signal to be captured. A provision is also made to provide DMA trigger sources based on the eCAP events. The necessary APIs are also provided for PWM mode of operation.

16.2 API Functions

Macros

- #define ECAP_ISR_SOURCE_CAPTURE_EVENT_1
- #define ECAP_ISR_SOURCE_CAPTURE_EVENT_2
- #define ECAP_ISR_SOURCE_CAPTURE_EVENT_3
- #define ECAP_ISR_SOURCE_CAPTURE_EVENT_4
- #define ECAP_ISR_SOURCE_COUNTER_OVERFLOW
- #define ECAP_ISR_SOURCE_COUNTER_PERIOD
- #define ECAP_ISR_SOURCE_COUNTER_COMPARE

Enumerations

- enum ECAP_EmulationMode { ECAP_EMULATION_STOP, ECAP_EMULATION_RUN_TO_ZERO, ECAP_EMULATION_FREE_RUN }
- enum ECAP_CaptureMode { ECAP_CONTINUOUS_CAPTURE_MODE, ECAP_ONE_SHOT_CAPTURE_MODE }
- enum ECAP_Events { ECAP_EVENT_1, ECAP_EVENT_2, ECAP_EVENT_3, ECAP_EVENT_4 }
- enum ECAP_SyncOutMode { ECAP_SYNC_OUT_SYNCI, ECAP_SYNC_OUT_COUNTER_PRD, ECAP_SYNC_OUT_DISABLED }
- enum ECAP_APWMPolarity { ECAP_APWM_ACTIVE_HIGH, ECAP_APWM_ACTIVE_LOW }
- enum ECAP_EventPolarity { ECAP_EVNT_RISING_EDGE, ECAP_EVNT_FALLING_EDGE }
- enum ECAP_InputCaptureSignals { ECAP_INPUT_INPUTXBAR1, ECAP_INPUT_INPUTXBAR2, ECAP_INPUT_INPUTXBAR3, ECAP_INPUT_INPUTXBAR4, ECAP_INPUT_INPUTXBAR5, ECAP_INPUT_INPUTXBAR6, ECAP_INPUT_INPUTXBAR7, ECAP_INPUT_INPUTXBAR8, ECAP_INPUT_INPUTXBAR9, ECAP_INPUT_INPUTXBAR10, ECAP_INPUT_INPUTXBAR11, ECAP_INPUT_INPUTXBAR12, ECAP_INPUT_INPUTXBAR13, ECAP_INPUT_INPUTXBAR14,

ECAP_INPUT_INPUTXBAR15, ECAP_INPUT_INPUTXBAR16,
ECAP_INPUT_CANA_INT0, ECAP_INPUT_CANB_INT0,
ECAP_INPUT_ECAP_DELAY_CLOCK, ECAP_INPUT_OUTPUTXBAR1,
ECAP_INPUT_OUTPUTXBAR2, ECAP_INPUT_OUTPUTXBAR3,
ECAP_INPUT_OUTPUTXBAR4, ECAP_INPUT_OUTPUTXBAR5,
ECAP_INPUT_OUTPUTXBAR6, ECAP_INPUT_OUTPUTXBAR7,
ECAP_INPUT_OUTPUTXBAR8, ECAP_INPUT_ADC_C_EVENT4,
ECAP_INPUT_ADC_C_EVENT3, ECAP_INPUT_ADC_C_EVENT2,
ECAP_INPUT_ADC_C_EVENT1, ECAP_INPUT_ADC_B_EVENT4,
ECAP_INPUT_ADC_B_EVENT3, ECAP_INPUT_ADC_B_EVENT2,
ECAP_INPUT_ADC_B_EVENT1, ECAP_INPUT_ADC_A_EVENT4,
ECAP_INPUT_ADC_A_EVENT3, ECAP_INPUT_ADC_A_EVENT2,
ECAP_INPUT_ADC_A_EVENT1, ECAP_INPUT_SDFM1_FLT1_COMPARE_LOW,
ECAP_INPUT_SDFM1_FLT2_COMPARE_LOW,
ECAP_INPUT_SDFM1_FLT3_COMPARE_LOW,
ECAP_INPUT_SDFM1_FLT4_COMPARE_LOW,
ECAP_INPUT_SDFM1_FLT1_COMPARE_HIGH,
ECAP_INPUT_SDFM1_FLT2_COMPARE_HIGH,
ECAP_INPUT_SDFM1_FLT3_COMPARE_HIGH,
ECAP_INPUT_SDFM1_FLT4_COMPARE_HIGH,
ECAP_INPUT_SDFM1_FLT1_COMPARE_HIGH_OR_LOW,
ECAP_INPUT_SDFM1_FLT2_COMPARE_HIGH_OR_LOW,
ECAP_INPUT_SDFM1_FLT3_COMPARE_HIGH_OR_LOW,
ECAP_INPUT_SDFM1_FLT4_COMPARE_HIGH_OR_LOW,
ECAP_INPUT_CMPSS1_CTRIP_LOW,
ECAP_INPUT_CMPSS2_CTRIP_LOW, ECAP_INPUT_CMPSS3_CTRIP_LOW,
ECAP_INPUT_CMPSS4_CTRIP_LOW, ECAP_INPUT_CMPSS5_CTRIP_LOW,
ECAP_INPUT_CMPSS6_CTRIP_LOW, ECAP_INPUT_CMPSS7_CTRIP_LOW,
ECAP_INPUT_CMPSS1_CTRIP_HIGH, ECAP_INPUT_CMPSS2_CTRIP_HIGH,
ECAP_INPUT_CMPSS3_CTRIP_HIGH, ECAP_INPUT_CMPSS4_CTRIP_HIGH,
ECAP_INPUT_CMPSS5_CTRIP_HIGH, ECAP_INPUT_CMPSS6_CTRIP_HIGH,
ECAP_INPUT_CMPSS7_CTRIP_HIGH, ECAP_INPUT_CMPSS1_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS2_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS3_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS4_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS5_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS6_CTRIP_HIGH_OR_LOW,
ECAP_INPUT_CMPSS7_CTRIP_HIGH_OR_LOW }

Functions

- static void ECAP_setEventPrescaler (uint32_t base, uint16_t preScalerValue)
- static void ECAP_setEventPolarity (uint32_t base, ECAP_Events event, ECAP_EventPolarity polarity)
- static void ECAP_setCaptureMode (uint32_t base, ECAP_CaptureMode mode, ECAP_Events event)
- static void ECAP_reArm (uint32_t base)
- static void ECAP_enableInterrupt (uint32_t base, uint16_t intFlags)
- static void ECAP_disableInterrupt (uint32_t base, uint16_t intFlags)
- static uint16_t ECAP_getInterruptSource (uint32_t base)
- static bool ECAP_getGlobalInterruptStatus (uint32_t base)
- static void ECAP_clearInterrupt (uint32_t base, uint16_t intFlags)

- static void ECAP_clearGlobalInterrupt (uint32_t base)
- static void ECAP_forceInterrupt (uint32_t base, uint16_t intFlags)
- static void ECAP_enableCaptureMode (uint32_t base)
- static void ECAP_enableAPWMMode (uint32_t base)
- static void ECAP_enableCounterResetOnEvent (uint32_t base, ECAP_Events event)
- static void ECAP_disableCounterResetOnEvent (uint32_t base, ECAP_Events event)
- static void ECAP_enableTimeStampCapture (uint32_t base)
- static void ECAP_disableTimeStampCapture (uint32_t base)
- static void ECAP_setPhaseShiftCount (uint32_t base, uint32_t shiftCount)
- static void ECAP_enableLoadCounter (uint32_t base)
- static void ECAP_disableLoadCounter (uint32_t base)
- static void ECAP_loadCounter (uint32_t base)
- static void ECAP_setSyncOutMode (uint32_t base, ECAP_SyncOutMode mode)
- static void ECAP_stopCounter (uint32_t base)
- static void ECAP_startCounter (uint32_t base)
- static void ECAP_setAPWMPolarity (uint32_t base, ECAP_APWMPolarity polarity)
- static void ECAP_setAPWMPeriod (uint32_t base, uint32_t periodCount)
- static void ECAP_setAPWMCompare (uint32_t base, uint32_t compareCount)
- static void ECAP_setAPWMShadowPeriod (uint32_t base, uint32_t periodCount)
- static void ECAP_setAPWMShadowCompare (uint32_t base, uint32_t compareCount)
- static uint32_t ECAP_getTimeBaseCounter (uint32_t base)
- static uint32_t ECAP_getEventTimeStamp (uint32_t base, ECAP_Events event)
- static void ECAP_selectECAPInput (uint32_t base, ECAP_InputCaptureSignals input)
- static void ECAP_resetCounters (uint32_t base)
- static void ECAP_setDMASource (uint32_t base, ECAP_Events event)
- static ECAP_Events ECAP_getModuloCounterStatus (uint32_t base)
- void ECAP_setEmulationMode (uint32_t base, ECAP_EmulationMode mode)

16.2.1 Detailed Description

The code for this module is contained in `driverlib/ecap.c`, with `driverlib/ecap.h` containing the API declarations for use by applications.

16.2.2 Macro Definition Documentation

16.2.2.1 #define ECAP_ISR_SOURCE_CAPTURE_EVENT_1

Event 1 ISR source

16.2.2.2 #define ECAP_ISR_SOURCE_CAPTURE_EVENT_2

Event 2 ISR source

16.2.2.3 #define ECAP_ISR_SOURCE_CAPTURE_EVENT_3

Event 3 ISR source

16.2.2.4 #define ECAP_ISR_SOURCE_CAPTURE_EVENT_4

Event 4 ISR source

16.2.2.5 #define ECAP_ISR_SOURCE_COUNTER_OVERFLOW

Counter overflow ISR source

16.2.2.6 #define ECAP_ISR_SOURCE_COUNTER_PERIOD

Counter equals period ISR source

16.2.2.7 #define ECAP_ISR_SOURCE_COUNTER_COMPARE

Counter equals compare ISR source

16.2.3 Enumeration Type Documentation

16.2.3.1 enum **ECAP_EmulationMode**

Values that can be passed to ECAP_setEmulationMode() as the *mode* parameter.

Enumerator

ECAP_EMULATION_STOP TSCTR is stopped on emulation suspension.

ECAP_EMULATION_RUN_TO_ZERO TSCTR runs until 0 before stopping on emulation suspension.

ECAP_EMULATION_FREE_RUN TSCTR is not affected by emulation suspension.

16.2.3.2 enum **ECAP_CaptureMode**

Values that can be passed to ECAP_setCaptureMode() as the *mode* parameter.

Enumerator

ECAP_CONTINUOUS_CAPTURE_MODE eCAP operates in continuous capture mode

ECAP_ONE_SHOT_CAPTURE_MODE eCAP operates in one shot capture mode

16.2.3.3 enum **ECAP_Events**

Values that can be passed to ECAP_setEventPolarity(), ECAP_setCaptureMode(), ECAP_enableCounterResetOnEvent(), ECAP_disableCounterResetOnEvent(), ECAP_getEventTimeStamp(), ECAP_setDMASource() as the *event* parameter.

Enumerator

ECAP_EVENT_1 eCAP event 1

ECAP_EVENT_2 eCAP event 2
ECAP_EVENT_3 eCAP event 3
ECAP_EVENT_4 eCAP event 4

16.2.3.4 enum **ECAP_SyncOutMode**

Values that can be passed to ECAP_setSyncOutMode() as the *mode* parameter.

Enumerator
ECAP_SYNC_OUT_SYNCI sync out on the sync in signal and software force
ECAP_SYNC_OUT_COUNTER_PRD sync out on counter equals period
ECAP_SYNC_OUT_DISABLED Disable sync out signal.

16.2.3.5 enum **ECAP_APWMPolarity**

Values that can be passed to ECAP_setAPWMPolarity() as the *polarity* parameter.

Enumerator
ECAP_APWM_ACTIVE_HIGH APWM is active high.
ECAP_APWM_ACTIVE_LOW APWM is active low.

16.2.3.6 enum **ECAP_EventPolarity**

Values that can be passed to ECAP_setEventPolarity() as the *polarity* parameter.

Enumerator
ECAP_EVNT_RISING_EDGE Rising edge polarity.
ECAP_EVNT_FALLING_EDGE Falling edge polarity.

16.2.3.7 enum **ECAP_InputCaptureSignals**

Values that can be passed to ECAP_selectECAPInput() as the *input* parameter.

Enumerator
ECAP_INPUT_INPUTXBAR1 GPIO Input Crossbar output signal-1.
ECAP_INPUT_INPUTXBAR2 GPIO Input Crossbar output signal-2.
ECAP_INPUT_INPUTXBAR3 GPIO Input Crossbar output signal-3.
ECAP_INPUT_INPUTXBAR4 GPIO Input Crossbar output signal-4.
ECAP_INPUT_INPUTXBAR5 GPIO Input Crossbar output signal-5.
ECAP_INPUT_INPUTXBAR6 GPIO Input Crossbar output signal-6.
ECAP_INPUT_INPUTXBAR7 GPIO Input Crossbar output signal-7.
ECAP_INPUT_INPUTXBAR8 GPIO Input Crossbar output signal-8.
ECAP_INPUT_INPUTXBAR9 GPIO Input Crossbar output signal-9.
ECAP_INPUT_INPUTXBAR10 GPIO Input Crossbar output signal-10.
ECAP_INPUT_INPUTXBAR11 GPIO Input Crossbar output signal-11.

ECAP_INPUT_INPUTXBAR12 GPIO Input Crossbar output signal-12.
ECAP_INPUT_INPUTXBAR13 GPIO Input Crossbar output signal-13.
ECAP_INPUT_INPUTXBAR14 GPIO Input Crossbar output signal-14.
ECAP_INPUT_INPUTXBAR15 GPIO Input Crossbar output signal-15.
ECAP_INPUT_INPUTXBAR16 GPIO Input Crossbar output signal-16.
ECAP_INPUT_CANA_INT0 CANA INT0 Input.
ECAP_INPUT_CANB_INT0 CANB INT0 Input.
ECAP_INPUT_ECAP_DELAY_CLOCK Delay clock for measurement.
ECAP_INPUT_OUTPUTXBAR1 Output Xbar Output-1.
ECAP_INPUT_OUTPUTXBAR2 Output Xbar Output-2.
ECAP_INPUT_OUTPUTXBAR3 Output Xbar Output-3.
ECAP_INPUT_OUTPUTXBAR4 Output Xbar Output-4.
ECAP_INPUT_OUTPUTXBAR5 Output Xbar Output-5.
ECAP_INPUT_OUTPUTXBAR6 Output Xbar Output-6.
ECAP_INPUT_OUTPUTXBAR7 Output Xbar Output-7.
ECAP_INPUT_OUTPUTXBAR8 Output Xbar Output-8.
ECAP_INPUT_ADC_C_EVENT4 ADCC Event4.
ECAP_INPUT_ADC_C_EVENT3 ADCC Event3.
ECAP_INPUT_ADC_C_EVENT2 ADCC Event2.
ECAP_INPUT_ADC_C_EVENT1 ADCC Event1.
ECAP_INPUT_ADC_B_EVENT4 ADCB Event4.
ECAP_INPUT_ADC_B_EVENT3 ADCB Event3.
ECAP_INPUT_ADC_B_EVENT2 ADCB Event2.
ECAP_INPUT_ADC_B_EVENT1 ADCB Event1.
ECAP_INPUT_ADC_A_EVENT4 ADCA Event4.
ECAP_INPUT_ADC_A_EVENT3 ADCA Event3.
ECAP_INPUT_ADC_A_EVENT2 ADCA Event2.
ECAP_INPUT_ADC_A_EVENT1 ADCA Event1.
ECAP_INPUT_SDFM1_FLT1_COMPARE_LOW SDFM-1 Filter-1 Compare Low Trip.
ECAP_INPUT_SDFM1_FLT2_COMPARE_LOW SDFM-1 Filter-2 Compare Low Trip.
ECAP_INPUT_SDFM1_FLT3_COMPARE_LOW SDFM-1 Filter-3 Compare Low Trip.
ECAP_INPUT_SDFM1_FLT4_COMPARE_LOW SDFM-1 Filter-4 Compare Low Trip.
ECAP_INPUT_SDFM1_FLT1_COMPARE_HIGH SDFM-1 Filter-1 Compare High Trip.
ECAP_INPUT_SDFM1_FLT2_COMPARE_HIGH SDFM-1 Filter-2 Compare High Trip.
ECAP_INPUT_SDFM1_FLT3_COMPARE_HIGH SDFM-1 Filter-3 Compare High Trip.
ECAP_INPUT_SDFM1_FLT4_COMPARE_HIGH SDFM-1 Filter-4 Compare High Trip.
ECAP_INPUT_SDFM1_FLT1_COMPARE_HIGH_OR_LOW SDFM-1 Filter-1 Compare High Trip or Low Trip.
ECAP_INPUT_SDFM1_FLT2_COMPARE_HIGH_OR_LOW SDFM-1 Filter-2 Compare High Trip or Low Trip.
ECAP_INPUT_SDFM1_FLT3_COMPARE_HIGH_OR_LOW SDFM-1 Filter-3 Compare High Trip or Low Trip.
ECAP_INPUT_SDFM1_FLT4_COMPARE_HIGH_OR_LOW SDFM-1 Filter-4 Compare High Trip or Low Trip.
ECAP_INPUT_CMPSS1_CTRIP_LOW Compare Subsystem-1 Low Trip.
ECAP_INPUT_CMPSS2_CTRIP_LOW Compare Subsystem-2 Low Trip.

ECAP_INPUT_CMPSS3_CTRIP_LOW Compare Subsystem-3 Low Trip.
ECAP_INPUT_CMPSS4_CTRIP_LOW Compare Subsystem-4 Low Trip.
ECAP_INPUT_CMPSS5_CTRIP_LOW Compare Subsystem-5 Low Trip.
ECAP_INPUT_CMPSS6_CTRIP_LOW Compare Subsystem-6 Low Trip.
ECAP_INPUT_CMPSS7_CTRIP_LOW Compare Subsystem-7 Low Trip.
ECAP_INPUT_CMPSS1_CTRIP_HIGH Compare Subsystem-1 High Trip.
ECAP_INPUT_CMPSS2_CTRIP_HIGH Compare Subsystem-2 High Trip.
ECAP_INPUT_CMPSS3_CTRIP_HIGH Compare Subsystem-3 High Trip.
ECAP_INPUT_CMPSS4_CTRIP_HIGH Compare Subsystem-4 High Trip.
ECAP_INPUT_CMPSS5_CTRIP_HIGH Compare Subsystem-5 High Trip.
ECAP_INPUT_CMPSS6_CTRIP_HIGH Compare Subsystem-6 High Trip.
ECAP_INPUT_CMPSS7_CTRIP_HIGH Compare Subsystem-7 High Trip.
ECAP_INPUT_CMPSS1_CTRIP_HIGH_OR_LOW Compare Subsystem-1 High Trip or Low Trip.
ECAP_INPUT_CMPSS2_CTRIP_HIGH_OR_LOW Compare Subsystem-2 High Trip or Low Trip.
ECAP_INPUT_CMPSS3_CTRIP_HIGH_OR_LOW Compare Subsystem-3 High Trip or Low Trip.
ECAP_INPUT_CMPSS4_CTRIP_HIGH_OR_LOW Compare Subsystem-4 High Trip or Low Trip.
ECAP_INPUT_CMPSS5_CTRIP_HIGH_OR_LOW Compare Subsystem-5 High Trip or Low Trip.
ECAP_INPUT_CMPSS6_CTRIP_HIGH_OR_LOW Compare Subsystem-6 High Trip or Low Trip.
ECAP_INPUT_CMPSS7_CTRIP_HIGH_OR_LOW Compare Subsystem-7 High Trip or Low Trip.

16.2.4 Function Documentation

16.2.4.1 static void ECAP_setEventPrescaler (uint32_t *base*, uint16_t *preScalerValue*) `[inline]`, `[static]`

Sets the input prescaler.

Parameters

base	is the base address of the ECAP module.
preScalerValue	is the pre scaler value for ECAP input

This function divides the ECAP input scaler. The pre scale value is doubled inside the module. For example a preScalerValue of 5 will divide the scaler by 10. Use a value of 1 to divide the pre scaler by 1. The value of preScalerValue should be less than **ECAP_MAX_PRESCALER_VALUE**.

Returns

None.

16.2.4.2 static void ECAP_setEventPolarity (uint32_t *base*, **ECAP_Events** *event*, **ECAP_EventPolarity** *polarity*) [inline], [static]

Sets the Capture event polarity.

Parameters

base	is the base address of the ECAP module.
event	is the event number.
polarity	is the polarity of the event.

This function sets the polarity of a given event. The value of event is between **ECAP_EVENT_1** and **ECAP_EVENT_4** inclusive corresponding to the four available events.For each event the polarity value determines the edge on which the capture is activated. For a rising edge use a polarity value of **ECAP_EVNT_RISING_EDGE** and for a falling edge use a polarity of **ECAP_EVNT_FALLING_EDGE**.

Returns

None.

16.2.4.3 static void ECAP_setCaptureMode (uint32_t *base*, **ECAP_CaptureMode** *mode*, **ECAP_Events** *event*) [inline], [static]

Sets the capture mode.

Parameters

base	is the base address of the ECAP module.
mode	is the capture mode.
event	is the event number at which the counter stops or wraps.

This function sets the eCAP module to a continuous or one-shot mode. The value of mode should be either **ECAP_CONTINUOUS_CAPTURE_MODE** or **ECAP_ONE_SHOT_CAPTURE_MODE** corresponding to continuous or one-shot mode respectively.

The value of event determines the event number at which the counter stops (in one-shot mode) or the counter wraps (in continuous mode). The value of event should be between **ECAP_EVENT_1** and **ECAP_EVENT_4** corresponding to the valid event numbers.

Returns

None.

16.2.4.4 static void ECAP_reArm (uint32_t *base*) [inline], [static]

Re-arms the eCAP module.

Parameters

base	is the base address of the ECAP module.

This function re-arms the eCAP module.

Returns
None.

16.2.4.5 static void ECAP_enableInterrupt (uint32_t *base*, uint16_t *intFlags*) [inline], [static]

Enables interrupt source.

Parameters

base	is the base address of the ECAP module.
intFlags	is the interrupt source to be enabled.

This function sets and enables eCAP interrupt source. The following are valid interrupt sources.

- ECAP_ISR_SOURCE_CAPTURE_EVENT_1 - Event 1 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_2 - Event 2 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_3 - Event 3 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_4 - Event 4 generates interrupt
- ECAP_ISR_SOURCE_COUNTER_OVERFLOW - Counter overflow generates interrupt
- ECAP_ISR_SOURCE_COUNTER_PERIOD - Counter equal period generates interrupt
- ECAP_ISR_SOURCE_COUNTER_COMPARE - Counter equal compare generates interrupt

Returns

None.

16.2.4.6 static void ECAP_disableInterrupt (uint32_t *base,* uint16_t *intFlags*) `[inline]`, `[static]`

Disables interrupt source.

Parameters

base	is the base address of the ECAP module.
intFlags	is the interrupt source to be disabled.

This function clears and disables eCAP interrupt source. The following are valid interrupt sources.

- ECAP_ISR_SOURCE_CAPTURE_EVENT_1 - Event 1 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_2 - Event 2 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_3 - Event 3 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_4 - Event 4 generates interrupt
- ECAP_ISR_SOURCE_COUNTER_OVERFLOW - Counter overflow generates interrupt
- ECAP_ISR_SOURCE_COUNTER_PERIOD - Counter equal period generates interrupt
- ECAP_ISR_SOURCE_COUNTER_COMPARE - Counter equal compare generates interrupt

Returns

None.

16.2.4.7 static uint16_t ECAP_getInterruptSource (uint32_t *base*) `[inline]`, `[static]`

Returns the interrupt flag.

Parameters

base	is the base address of the ECAP module.

This function returns the eCAP interrupt flag. The following are valid interrupt sources corresponding to the eCAP interrupt flag.

Returns

Returns the eCAP interrupt that has occurred. The following are valid return values.

- ECAP_ISR_SOURCE_CAPTURE_EVENT_1 - Event 1 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_2 - Event 2 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_3 - Event 3 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_4 - Event 4 generates interrupt
- ECAP_ISR_SOURCE_COUNTER_OVERFLOW - Counter overflow generates interrupt
- ECAP_ISR_SOURCE_COUNTER_PERIOD - Counter equal period generates interrupt
- ECAP_ISR_SOURCE_COUNTER_COMPARE - Counter equal compare generates interrupt

Note

- User can check if a combination of various interrupts have occurred by ORing the above return values.

16.2.4.8 static bool ECAP_getGlobalInterruptStatus (uint32_t *base*) `[inline]`, `[static]`

Returns the Global interrupt flag.

Parameters

base	is the base address of the ECAP module.

This function returns the eCAP Global interrupt flag.

Returns

Returns true if there is a global eCAP interrupt, false otherwise.

16.2.4.9 static void ECAP_clearInterrupt (uint32_t *base*, uint16_t *intFlags*) `[inline]`, `[static]`

Clears interrupt flag.

Parameters

base	is the base address of the ECAP module.
intFlags	is the interrupt source.

This function clears eCAP interrupt flags. The following are valid interrupt sources.

- ECAP_ISR_SOURCE_CAPTURE_EVENT_1 - Event 1 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_2 - Event 2 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_3 - Event 3 generates interrupt

- ECAP_ISR_SOURCE_CAPTURE_EVENT_4 - Event 4 generates interrupt
- ECAP_ISR_SOURCE_COUNTER_OVERFLOW - Counter overflow generates interrupt
- ECAP_ISR_SOURCE_COUNTER_PERIOD - Counter equal period generates interrupt
- ECAP_ISR_SOURCE_COUNTER_COMPARE - Counter equal compare generates interrupt

Returns
None.

16.2.4.10 static void ECAP_clearGlobalInterrupt (uint32_t *base*) [inline], [static]

Clears global interrupt flag

Parameters

base	is the base address of the ECAP module.

This function clears the global interrupt bit.

Returns
None.

16.2.4.11 static void ECAP_forceInterrupt (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

Forces interrupt source.

Parameters

base	is the base address of the ECAP module.
intFlags	is the interrupt source.

This function forces and enables eCAP interrupt source. The following are valid interrupt sources.

- ECAP_ISR_SOURCE_CAPTURE_EVENT_1 - Event 1 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_2 - Event 2 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_3 - Event 3 generates interrupt
- ECAP_ISR_SOURCE_CAPTURE_EVENT_4 - Event 4 generates interrupt
- ECAP_ISR_SOURCE_COUNTER_OVERFLOW - Counter overflow generates interrupt
- ECAP_ISR_SOURCE_COUNTER_PERIOD - Counter equal period generates interrupt
- ECAP_ISR_SOURCE_COUNTER_COMPARE - Counter equal compare generates interrupt

Returns
None.

16.2.4.12 static void ECAP_enableCaptureMode (uint32_t *base*) [inline], [static]

Sets eCAP in Capture mode.

Parameters

base	is the base address of the ECAP module.

This function sets the eCAP module to operate in Capture mode.

Returns
None.

16.2.4.13 static void ECAP_enableAPWMMode (uint32_t *base*) [inline], [static]

Sets eCAP in APWM mode.

Parameters

base	is the base address of the ECAP module.

This function sets the eCAP module to operate in APWM mode.

Returns
None.

16.2.4.14 static void ECAP_enableCounterResetOnEvent (uint32_t *base*, **ECAP_Events** *event*) [inline], [static]

Enables counter reset on an event.

Parameters

base	is the base address of the ECAP module.
event	is the event number the time base gets reset.

This function enables the base timer, TSCTR, to be reset on capture event provided by the variable event. Valid inputs for event are **ECAP_EVENT_1** to **ECAP_EVENT_4**.

Returns
None.

16.2.4.15 static void ECAP_disableCounterResetOnEvent (uint32_t *base*, **ECAP_Events** *event*) [inline], [static]

Disables counter reset on events.

Parameters

base	is the base address of the ECAP module.
event	is the event number the time base gets reset.

This function disables the base timer, TSCTR, from being reset on capture event provided by the variable event. Valid inputs for event are 1 to 4.

Returns
None.

16.2.4.16 static void ECAP_enableTimeStampCapture (uint32_t *base*) [inline], [static]

Enables time stamp capture.

Parameters

base	is the base address of the ECAP module.

This function enables time stamp count to be captured

Returns

None.

16.2.4.17 static void ECAP_disableTimeStampCapture (uint32_t *base*) [inline], [static]

Disables time stamp capture.

Parameters

base	is the base address of the ECAP module.

This function disables time stamp count to be captured

Returns

None.

16.2.4.18 static void ECAP_setPhaseShiftCount (uint32_t *base,* uint32_t *shiftCount*) [inline], [static]

Sets a phase shift value count.

Parameters

base	is the base address of the ECAP module.
shiftCount	is the phase shift value.

This function writes a phase shift value to be loaded into the main time stamp counter.

Returns

None.

16.2.4.19 static void ECAP_enableLoadCounter (uint32_t *base*) [inline], [static]

Enable counter loading with phase shift value.

Parameters

base	is the base address of the ECAP module.

This function enables loading of the counter with the value present in the phase shift counter as defined by the ECAP_setPhaseShiftCount() function.

Returns

None.

16.2.4.20 static void ECAP_disableLoadCounter (uint32_t *base*) [inline], [static]

Disable counter loading with phase shift value.

Parameters

base	is the base address of the ECAP module.

This function disables loading of the counter with the value present in the phase shift counter as defined by the ECAP_setPhaseShiftCount() function.

Returns

None.

16.2.4.21 static void ECAP_loadCounter (uint32_t *base*) [inline], [static]

Load time stamp counter

Parameters

base	is the base address of the ECAP module.

This function forces the value in the phase shift counter register to be loaded into Time stamp counter register. Make sure to enable loading of Time stamp counter by calling ECAP_enableLoadCounter() function before calling this function.

Returns

None.

16.2.4.22 static void ECAP_setSyncOutMode (uint32_t *base*, **ECAP_SyncOutMode** *mode*) [inline], [static]

Configures Sync out signal mode.

Parameters

base	is the base address of the ECAP module.
mode	is the sync out mode.

This function sets the sync out mode. Valid parameters for mode are:

- ECAP_SYNC_OUT_SYNCI - Trigger sync out on sync-in event.
- ECAP_SYNC_OUT_COUNTER_PRD - Trigger sync out when counter equals period.
- ECAP_SYNC_OUT_DISABLED - Disable sync out.

Returns

None.

16.2.4.23 static void ECAP_stopCounter (uint32_t *base*) [inline], [static]

Stops Time stamp counter.

Parameters

base	is the base address of the ECAP module.

This function stops the time stamp counter.

Returns

None.

16.2.4.24 static void ECAP_startCounter (uint32_t *base*) [inline], [static]

Starts Time stamp counter.

Parameters

base	is the base address of the ECAP module.

This function starts the time stamp counter.

Returns

None.

16.2.4.25 static void ECAP_setAPWMPolarity (uint32_t *base*, **ECAP_APWMPolarity** *polarity*) [inline], [static]

Set eCAP APWM polarity.

Parameters

base	is the base address of the ECAP module.
polarity	is the polarity of APWM

This function sets the polarity of the eCAP in APWM mode. Valid inputs for polarity are:

- ECAP_APWM_ACTIVE_HIGH - For active high.
- ECAP_APWM_ACTIVE_LOW - For active low.

Returns

None.

16.2.4.26 static void ECAP_setAPWMPeriod (uint32_t *base*, uint32_t *periodCount*) [inline], [static]

Set eCAP APWM period.

Parameters

base	is the base address of the ECAP module.

periodCount	is the period count for APWM.

This function sets the period count of the APWM waveform. periodCount takes the actual count which is written to the register. The user is responsible for converting the desired frequency or time into the period count.

Returns
None.

16.2.4.27 static void ECAP_setAPWMCompare (uint32_t *base,* uint32_t *compareCount*) [inline], [static]

Set eCAP APWM on or off time count.

Parameters

base	is the base address of the ECAP module.
compareCount	is the on or off count for APWM.

This function sets the on or off time count of the APWM waveform depending on the polarity of the output. If the output , as set by ECAP_setAPWMPolarity(), is active high then compareCount determines the on time. If the output is active low then compareCount determines the off time. compareCount takes the actual count which is written to the register. The user is responsible for converting the desired frequency or time into the appropriate count value.

Returns
None.

16.2.4.28 static void ECAP_setAPWMShadowPeriod (uint32_t *base,* uint32_t *periodCount*) [inline], [static]

Load eCAP APWM shadow period.

Parameters

base	is the base address of the ECAP module.
periodCount	is the shadow period count for APWM.

This function sets the shadow period count of the APWM waveform. periodCount takes the actual count which is written to the register. The user is responsible for converting the desired frequency or time into the period count.

Returns
None.

16.2.4.29 static void ECAP_setAPWMShadowCompare (uint32_t *base,* uint32_t *compareCount*) [inline], [static]

Set eCAP APWM shadow on or off time count.

Parameters

base	is the base address of the ECAP module.
compareCount	is the on or off count for APWM.

This function sets the shadow on or off time count of the APWM waveform depending on the polarity of the output. If the output , as set by ECAP_setAPWMPolarity() , is active high then compareCount determines the on time. If the output is active low then compareCount determines the off time. compareCount takes the actual count which is written to the register. The user is responsible for converting the desired frequency or time into the appropriate count value.

Returns

None.

16.2.4.30 static uint32_t ECAP_getTimeBaseCounter (uint32_t *base*) [static]

Returns the time base counter value.

Parameters

base	is the base address of the ECAP module.

This function returns the time base counter value.

Returns

Returns the time base counter value.

16.2.4.31 static uint32_t ECAP_getEventTimeStamp (uint32_t *base*, **ECAP_Events** *event*) [inline], [static]

Returns event time stamp.

Parameters

base	is the base address of the ECAP module.
event	is the event number.

This function returns the current time stamp count of the given event. Valid values for event are **ECAP_EVENT_1** to **ECAP_EVENT_4**.

Returns

Event time stamp value or 0 if *event* is invalid.

References ECAP_EVENT_1, ECAP_EVENT_2, ECAP_EVENT_3, and ECAP_EVENT_4.

16.2.4.32 static void ECAP_selectECAPInput (uint32_t *base*, **ECAP_InputCaptureSignals** *input*) [inline], [static]

Select eCAP input.

Parameters

base	is the base address of the ECAP module.
input	is the eCAP input signal.

This function selects the eCAP input signal.

Returns

None.

16.2.4.33 static void ECAP_resetCounters (uint32_t *base*) [inline], [static]

Resets eCAP counters and flags.

Parameters

base	is the base address of the ECAP module.

This function resets the main counter (TSCTR register), event filter, modulo counter, capture events and counter overflow flags

Returns

None.

16.2.4.34 static void ECAP_setDMASource (uint32_t *base*, **ECAP_Events** *event*) [inline], [static]

Sets the eCAP DMA source.

Parameters

base	is the base address of the ECAP module.
event	is the eCAP event for the DMA

This function sets the eCAP event source for the DMA trigger.

Returns

None.

16.2.4.35 static **ECAP_Events** ECAP_getModuloCounterStatus (uint32_t *base*) [inline], [static]

Return the Modulo counter status.

Parameters

base	is the base address of the ECAP module.

This function returns the modulo counter status, indicating which register gets loaded on the next capture event.

Returns

Returns an **ECAP_EVENT_n** value indicating that CAPn is the register to be loaded on the next event.

16.2.4.36 void ECAP_setEmulationMode (uint32_t *base*, **ECAP_EmulationMode** *mode*)

Configures emulation mode.

Parameters

base	is the base address of the ECAP module.
mode	is the emulation mode.

This function configures the eCAP counter, TSCTR, to the desired emulation mode when emulation suspension occurs. Valid inputs for mode are:

- ECAP_EMULATION_STOP - Counter is stopped immediately.
- ECAP_EMULATION_RUN_TO_ZERO - Counter runs till it reaches 0.
- ECAP_EMULATION_FREE_RUN - Counter is not affected.

Returns

None.

17 HRCAP Module

Introduction 189
API Functions 189

17.1 HRCAP Introduction

The High Resolution Enhanced Capture (HRCAP) API provides a set of functions for configuring and using the high resolution capability of the HRCAP module. The functions provided allow for the setup and configuration of the high resolution capture capability. The necessary APIs to configure the high resolution capture calibration hardware are also provided.

17.2 API Functions

Macros

- #define HRCAP_GLOBAL_CALIBRATION_INTERRUPT
- #define HRCAP_CALIBRATION_DONE
- #define HRCAP_CALIBRATION_PERIOD_OVERFLOW

Enumerations

- enum HRCAP_CalibrationClockSource { HRCAP_CALIBRATION_CLOCK_SYSCLK, HRCAP_CALIBRATION_CLOCK_HRCLK }

Functions

- static void HRCAP_enableHighResolution (uint32_t base)
- static void HRCAP_disableHighResolution (uint32_t base)
- static void HRCAP_enableHighResolutionClock (uint32_t base)
- static void HRCAP_disbleHighResolutionClock (uint32_t base)
- static void HRCAP_startCalibration (uint32_t base)
- static void HRCAP_setCalibrationMode (uint32_t base)
- static void HRCAP_enableCalibrationInterrupt (uint32_t base, uint16_t intFlags)
- static void HRCAP_disableCalibrationInterrupt (uint32_t base, uint16_t intFlags)
- static uint16_t HRCAP_getCalibrationFlags (uint32_t base)
- static void HRCAP_clearCalibrationFlags (uint32_t base, uint16_t flags)
- static void HRCAP_setCalibrationPeriod (uint32_t base, uint32_t sysclkHz)
- static uint32_t HRCAP_getCalibrationClockPeriod (uint32_t base, HRCAP_CalibrationClockSource clockSource)
- static float32_t HRCAP_getScaleFactor (uint32_t base)
- static float32_t HRCAP_convertEventTimeStampNanoseconds (uint32_t timeStamp, float32_t scaleFactor)

17.2.1 Detailed Description

The code for this module is contained in `driverlib/hrcap.c`, with `driverlib/hrcap.h` containing the API declarations for use by applications.

17.2.2 Macro Definition Documentation

17.2.2.1 #define HRCAP_GLOBAL_CALIBRATION_INTERRUPT

Global calibration interrupt flag

Referenced by HRCAP_clearCalibrationFlags().

17.2.2.2 #define HRCAP_CALIBRATION_DONE

Calibration done flag

Referenced by HRCAP_clearCalibrationFlags(), HRCAP_disableCalibrationInterrupt(), and HRCAP_enableCalibrationInterrupt().

17.2.2.3 #define HRCAP_CALIBRATION_PERIOD_OVERFLOW

Calibration period overflow flag

Referenced by HRCAP_clearCalibrationFlags(), HRCAP_disableCalibrationInterrupt(), and HRCAP_enableCalibrationInterrupt().

17.2.3 Enumeration Type Documentation

17.2.3.1 enum **HRCAP_CalibrationClockSource**

Values that can be passed to HRCAP_getCalibrationClockPeriod() as the *clockSource* parameter.

Enumerator

HRCAP_CALIBRATION_CLOCK_SYSCLK Use SYSCLK for period match.

HRCAP_CALIBRATION_CLOCK_HRCLK Use HRCLK for period match.

17.2.4 Function Documentation

17.2.4.1 static void HRCAP_enableHighResolution (uint32_t *base*) `[inline]`, `[static]`

enables HRCAP.

Parameters

base	is the base address of the HRCAP instance used.

This function enables High Resolution Capture module.

Note

High resolution clock must be enabled before High Resolution Module is enabled.

Returns

None.

17.2.4.2 static void HRCAP_disableHighResolution (uint32_t *base*) [inline], [static]

Disables HRCAP.

Parameters

base	is the base address of the HRCAP instance used.

This function disable High Resolution Capture module.

Returns

None.

17.2.4.3 static void HRCAP_enableHighResolutionClock (uint32_t *base*) [inline], [static]

Enables high resolution clock.

Parameters

base	is the base address of the HRCAP instance used.

This function enables High Resolution clock.

Returns

None.

17.2.4.4 static void HRCAP_disbleHighResolutionClock (uint32_t *base*) [inline], [static]

Disables High resolution clock.

Parameters

base	is the base address of the HRCAP instance used.

This function disables High Resolution clock.

Returns

None.

17.2.4.5 static void HRCAP_startCalibration (uint32_t *base*) [inline], [static]

Starts calibration.

Parameters

base	is the base address of the HRCAP instance used.

This function starts calibration.

Returns

None.

17.2.4.6 static void HRCAP_setCalibrationMode (uint32_t *base*) [inline], [static]

Sets the calibration mode.

Parameters

base	is the base address of the HRCAP instance used.

This function sets the the calibration mode by turning on continuous calibration.

Returns

None.

17.2.4.7 static void HRCAP_enableCalibrationInterrupt (uint32_t *base*, uint16_t *intFlags*) [inline], [static]

Enables calibration interrupt.

Parameters

base	is the base address of the HRCAP module.
intFlags	is the calibration interrupt flags to be enabled.

This function enables HRCAP calibration interrupt flags. Valid values for intFlags are:

- HRCAP_CALIBRATION_DONE - Calibration done interrupt.
- HRCAP_CALIBRATION_PERIOD_OVERFLOW - Calibration period overflow check interrupt.

  Returns

  None.

References HRCAP_CALIBRATION_DONE, and HRCAP_CALIBRATION_PERIOD_OVERFLOW.

17.2.4.8 static void HRCAP_disableCalibrationInterrupt (uint32_t *base*, uint16_t *intFlags*) [inline], [static]

Disables calibration interrupt source.

Parameters

base	is the base address of the HRCAP module.
intFlags	is the calibration interrupt flags to be disabled.

This function disables HRCAP calibration interrupt flags. Valid values for intFlags are:

- HRCAP_CALIBRATION_DONE - Calibration done interrupt.

- HRCAP_CALIBRATION_PERIOD_OVERFLOW - Calibration period check interrupt.

Returns
None.

References HRCAP_CALIBRATION_DONE, and HRCAP_CALIBRATION_PERIOD_OVERFLOW.

17.2.4.9 static uint16_t HRCAP_getCalibrationFlags (uint32_t *base*) [inline], [static]

Returns the calibration interrupt source.

Parameters

base	is the base address of the HRCAP module.

This function returns the HRCAP calibration interrupt source.

Returns
Returns the HRCAP interrupt that has occurred. The following are valid return values.
- HRCAP_GLOBAL_CALIBRATION_INTERRUPT - Global calibration interrupt.
- HRCAP_CALIBRATION_DONE - Calibration done interrupt.
- HRCAP_CALIBRATION_PERIOD_OVERFLOW - Calibration period overflow interrupt.

Note
- User can check if a combination of the interrupts have occurred by ORing the above return values.

17.2.4.10 static void HRCAP_clearCalibrationFlags (uint32_t *base,* uint16_t *flags*) [inline], [static]

Clears calibration flags.

Parameters

base	is the base address of the HRCAP module.
flags	is the calibration flags to be cleared.

This function clears HRCAP calibration flags. The following are valid values for flags.

- HRCAP_GLOBAL_CALIBRATION_INTERRUPT - Global calibration interrupt.
- HRCAP_CALIBRATION_DONE - Calibration done flag.
- HRCAP_CALIBRATION_PERIOD_OVERFLOW - Calibration period overflow flag.

Returns
None.

References HRCAP_CALIBRATION_DONE, HRCAP_CALIBRATION_PERIOD_OVERFLOW, and HRCAP_GLOBAL_CALIBRATION_INTERRUPT.

17.2.4.11 static void HRCAP_setCalibrationPeriod (uint32_t *base*, uint32_t *sysclkHz*) [inline], [static]

Sets the calibration period count

Parameters

base	is the base address of the HRCAP instance used.
sysclkHz	is the rate of the SYSCLK in Hz.

This function sets the calibration period count value to achieve a period of 1.6 microseconds given the SYSCLK frequency in Hz (the *sysclkHz* parameter).

Returns

None.

17.2.4.12 static uint32_t HRCAP_getCalibrationClockPeriod (uint32_t *base*, **HRCAP_CalibrationClockSource** *clockSource*) [inline], [static]

Returns the calibration clock period

Parameters

base	is the base address of the HRCAP instance used.
clockSource	is the calibration clock source (**HRCAP_CALIBRATION_CLOCK_SYSCLK** or **HRCAP_CALIBRATION_CLOCK_HRCLK**).

This function returns the period match value of the calibration clock. The return value has a valid count when a period match occurs.

Returns

This function returns the captured value of the clock counter specified by clockSource.

Referenced by HRCAP_getScaleFactor().

17.2.4.13 static float32_t HRCAP_getScaleFactor (uint32_t *base*) [inline], [static]

Calculates the scale factor

Parameters

base	is the base address of the HRCAP instance used.

This function reads the SYSCLK and HRCLK calibration periods and then uses them to calculate the scale factor.

Returns

This function returns the calculated scale factor.

References HRCAP_CALIBRATION_CLOCK_HRCLK, HRCAP_CALIBRATION_CLOCK_SYSCLK, and HRCAP_getCalibrationClockPeriod().

17.2.4.14 static float32_t HRCAP_convertEventTimeStampNanoseconds (uint32_t *timeStamp*, float32_t *scaleFactor*) [inline], [static]

Returns event time stamp in nanoseconds

Parameters

timeStamp	is a raw time stamp count returned by ECAP_getEventTimeStamp().
scaleFactor	is the calculated scale factor returned by HRCAP_getScaleFactor().

This function converts a raw CAP time stamp (the *timeStamp* parameter) to nanoseconds using the provided scale factor (the *scaleFactor* parameter).

Returns

Returns the converted time stamp in nanoseconds.

18 EPWM Module

Introduction 197
API Functions 197

18.1 EPWM Introduction

The ePWM (enhanced Pulse width Modulator) API provides a set of functions for configuring and using the ePWM module. The provided functions provide the capability to generate and alter PWM wave forms by providing access to the following ePWM sub-modules.

- Time Base
- Counter Compare
- Action Qualifier
- Dead Band Generator
- Trip Zone
- Event Trigger
- Digital Compare

18.2 API Functions

Macros

- #define EPWM_TIME_BASE_STATUS_COUNT_UP
- #define EPWM_TIME_BASE_STATUS_COUNT_DOWN
- #define EPWM_DB_INPUT_EPWMA
- #define EPWM_DB_INPUT_EPWMB
- #define EPWM_DB_INPUT_DB_RED
- #define EPWM_TZ_SIGNAL_CBC1
- #define EPWM_TZ_SIGNAL_CBC2
- #define EPWM_TZ_SIGNAL_CBC3
- #define EPWM_TZ_SIGNAL_CBC4
- #define EPWM_TZ_SIGNAL_CBC5
- #define EPWM_TZ_SIGNAL_CBC6
- #define EPWM_TZ_SIGNAL_DCAEVT2
- #define EPWM_TZ_SIGNAL_DCBEVT2
- #define EPWM_TZ_SIGNAL_OSHT1
- #define EPWM_TZ_SIGNAL_OSHT2
- #define EPWM_TZ_SIGNAL_OSHT3
- #define EPWM_TZ_SIGNAL_OSHT4
- #define EPWM_TZ_SIGNAL_OSHT5
- #define EPWM_TZ_SIGNAL_OSHT6
- #define EPWM_TZ_SIGNAL_DCAEVT1
- #define EPWM_TZ_SIGNAL_DCBEVT1
- #define EPWM_TZ_INTERRUPT_CBC
- #define EPWM_TZ_INTERRUPT_OST
- #define EPWM_TZ_INTERRUPT_DCAEVT1

- #define EPWM_TZ_INTERRUPT_DCAEVT2
- #define EPWM_TZ_INTERRUPT_DCBEVT1
- #define EPWM_TZ_INTERRUPT_DCBEVT2
- #define EPWM_TZ_FLAG_CBC
- #define EPWM_TZ_FLAG_OST
- #define EPWM_TZ_FLAG_DCAEVT1
- #define EPWM_TZ_FLAG_DCAEVT2
- #define EPWM_TZ_FLAG_DCBEVT1
- #define EPWM_TZ_FLAG_DCBEVT2
- #define EPWM_TZ_INTERRUPT
- #define EPWM_TZ_CBC_FLAG_1
- #define EPWM_TZ_CBC_FLAG_2
- #define EPWM_TZ_CBC_FLAG_3
- #define EPWM_TZ_CBC_FLAG_4
- #define EPWM_TZ_CBC_FLAG_5
- #define EPWM_TZ_CBC_FLAG_6
- #define EPWM_TZ_CBC_FLAG_DCAEVT2
- #define EPWM_TZ_CBC_FLAG_DCBEVT2
- #define EPWM_TZ_OST_FLAG_OST1
- #define EPWM_TZ_OST_FLAG_OST2
- #define EPWM_TZ_OST_FLAG_OST3
- #define EPWM_TZ_OST_FLAG_OST4
- #define EPWM_TZ_OST_FLAG_OST5
- #define EPWM_TZ_OST_FLAG_OST6
- #define EPWM_TZ_OST_FLAG_DCAEVT1
- #define EPWM_TZ_OST_FLAG_DCBEVT1
- #define EPWM_TZ_FORCE_EVENT_CBC
- #define EPWM_TZ_FORCE_EVENT_OST
- #define EPWM_TZ_FORCE_EVENT_DCAEVT1
- #define EPWM_TZ_FORCE_EVENT_DCAEVT2
- #define EPWM_TZ_FORCE_EVENT_DCBEVT1
- #define EPWM_TZ_FORCE_EVENT_DCBEVT2
- #define EPWM_INT_TBCTR_ZERO
- #define EPWM_INT_TBCTR_PERIOD
- #define EPWM_INT_TBCTR_ZERO_OR_PERIOD
- #define EPWM_INT_TBCTR_U_CMPA
- #define EPWM_INT_TBCTR_U_CMPC
- #define EPWM_INT_TBCTR_D_CMPA
- #define EPWM_INT_TBCTR_D_CMPC
- #define EPWM_INT_TBCTR_U_CMPB
- #define EPWM_INT_TBCTR_U_CMPD
- #define EPWM_INT_TBCTR_D_CMPB
- #define EPWM_INT_TBCTR_D_CMPD
- #define EPWM_DC_COMBINATIONAL_TRIPIN1
- #define EPWM_DC_COMBINATIONAL_TRIPIN2
- #define EPWM_DC_COMBINATIONAL_TRIPIN3
- #define EPWM_DC_COMBINATIONAL_TRIPIN4
- #define EPWM_DC_COMBINATIONAL_TRIPIN5
- #define EPWM_DC_COMBINATIONAL_TRIPIN6
- #define EPWM_DC_COMBINATIONAL_TRIPIN7
- #define EPWM_DC_COMBINATIONAL_TRIPIN8
- #define EPWM_DC_COMBINATIONAL_TRIPIN9
- #define EPWM_DC_COMBINATIONAL_TRIPIN10
- #define EPWM_DC_COMBINATIONAL_TRIPIN11
- #define EPWM_DC_COMBINATIONAL_TRIPIN12
- #define EPWM_DC_COMBINATIONAL_TRIPIN14
- #define EPWM_DC_COMBINATIONAL_TRIPIN15
- #define EPWM_GL_REGISTER_TBPRD_TBPRDHR

- #define EPWM_GL_REGISTER_CMPA_CMPAHR
- #define EPWM_GL_REGISTER_CMPB_CMPBHR
- #define EPWM_GL_REGISTER_CMPC
- #define EPWM_GL_REGISTER_CMPD
- #define EPWM_GL_REGISTER_DBRED_DBREDHR
- #define EPWM_GL_REGISTER_DBFED_DBFEDHR
- #define EPWM_GL_REGISTER_DBCTL
- #define EPWM_GL_REGISTER_AQCTLA_AQCTLA2
- #define EPWM_GL_REGISTER_AQCTLB_AQCTLB2
- #define EPWM_GL_REGISTER_AQCSFRC

Enumerations

- enum EPWM_EmulationMode { EPWM_EMULATION_STOP_AFTER_NEXT_TB, EPWM_EMULATION_STOP_AFTER_FULL_CYCLE, EPWM_EMULATION_FREE_RUN }
- enum EPWM_SyncCountMode { EPWM_COUNT_MODE_DOWN_AFTER_SYNC, EPWM_COUNT_MODE_UP_AFTER_SYNC }
- enum EPWM_ClockDivider {
  EPWM_CLOCK_DIVIDER_1, EPWM_CLOCK_DIVIDER_2, EPWM_CLOCK_DIVIDER_4,
  EPWM_CLOCK_DIVIDER_8,
  EPWM_CLOCK_DIVIDER_16, EPWM_CLOCK_DIVIDER_32,
  EPWM_CLOCK_DIVIDER_64, EPWM_CLOCK_DIVIDER_128 }
- enum EPWM_HSClockDivider {
  EPWM_HSCLOCK_DIVIDER_1, EPWM_HSCLOCK_DIVIDER_2,
  EPWM_HSCLOCK_DIVIDER_4, EPWM_HSCLOCK_DIVIDER_6,
  EPWM_HSCLOCK_DIVIDER_8, EPWM_HSCLOCK_DIVIDER_10,
  EPWM_HSCLOCK_DIVIDER_12, EPWM_HSCLOCK_DIVIDER_14 }
- enum EPWM_SyncOutPulseMode {
  EPWM_SYNC_OUT_PULSE_ON_SOFTWARE,
  EPWM_SYNC_OUT_PULSE_ON_EPWMxSYNCIN,
  EPWM_SYNC_OUT_PULSE_ON_COUNTER_ZERO,
  EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_B,
  EPWM_SYNC_OUT_PULSE_DISABLED,
  EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_C,
  EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_D }
- enum EPWM_PeriodLoadMode { EPWM_PERIOD_SHADOW_LOAD, EPWM_PERIOD_DIRECT_LOAD }
- enum EPWM_TimeBaseCountMode { EPWM_COUNTER_MODE_UP, EPWM_COUNTER_MODE_DOWN, EPWM_COUNTER_MODE_UP_DOWN, EPWM_COUNTER_MODE_STOP_FREEZE }
- enum EPWM_PeriodShadowLoadMode {
  EPWM_SHADOW_LOAD_MODE_COUNTER_ZERO,
  EPWM_SHADOW_LOAD_MODE_COUNTER_SYNC,
  EPWM_SHADOW_LOAD_MODE_SYNC }
- enum EPWM_CurrentLink {
  EPWM_LINK_WITH_EPWM_1, EPWM_LINK_WITH_EPWM_2,
  EPWM_LINK_WITH_EPWM_3, EPWM_LINK_WITH_EPWM_4,
  EPWM_LINK_WITH_EPWM_5, EPWM_LINK_WITH_EPWM_6,
  EPWM_LINK_WITH_EPWM_7, EPWM_LINK_WITH_EPWM_8 }
- enum EPWM_LinkComponent {
  EPWM_LINK_TBPRD, EPWM_LINK_COMP_A, EPWM_LINK_COMP_B,
  EPWM_LINK_COMP_C,
  EPWM_LINK_COMP_D, EPWM_LINK_GLDCTL2 }

- enum EPWM_CounterCompareModule { EPWM_COUNTER_COMPARE_A, EPWM_COUNTER_COMPARE_B, EPWM_COUNTER_COMPARE_C, EPWM_COUNTER_COMPARE_D }
- enum EPWM_CounterCompareLoadMode { EPWM_COMP_LOAD_ON_CNTR_ZERO, EPWM_COMP_LOAD_ON_CNTR_PERIOD, EPWM_COMP_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_COMP_LOAD_FREEZE, EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO, EPWM_COMP_LOAD_ON_SYNC_CNTR_PERIOD, EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO_PERIOD, EPWM_COMP_LOAD_ON_SYNC_ONLY }
- enum EPWM_ActionQualifierModule { EPWM_ACTION_QUALIFIER_A, EPWM_ACTION_QUALIFIER_B }
- enum EPWM_ActionQualifierLoadMode { EPWM_AQ_LOAD_ON_CNTR_ZERO, EPWM_AQ_LOAD_ON_CNTR_PERIOD, EPWM_AQ_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_AQ_LOAD_FREEZE, EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO, EPWM_AQ_LOAD_ON_SYNC_CNTR_PERIOD, EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO_PERIOD, EPWM_AQ_LOAD_ON_SYNC_ONLY }
- enum EPWM_ActionQualifierTriggerSource { EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_1, EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_2, EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_1, EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_2, EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_1, EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_2, EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_3, EPWM_AQ_TRIGGER_EVENT_TRIG_EPWM_SYNCIN }
- enum EPWM_ActionQualifierOutputEvent { EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO, EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD, EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA, EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA, EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPB, EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB, EPWM_AQ_OUTPUT_ON_T1_COUNT_UP, EPWM_AQ_OUTPUT_ON_T1_COUNT_DOWN, EPWM_AQ_OUTPUT_ON_T2_COUNT_UP, EPWM_AQ_OUTPUT_ON_T2_COUNT_DOWN }
- enum EPWM_ActionQualifierOutput { EPWM_AQ_OUTPUT_NO_CHANGE, EPWM_AQ_OUTPUT_LOW, EPWM_AQ_OUTPUT_HIGH, EPWM_AQ_OUTPUT_TOGGLE }
- enum EPWM_ActionQualifierSWOutput { EPWM_AQ_SW_DISABLED, EPWM_AQ_SW_OUTPUT_LOW, EPWM_AQ_SW_OUTPUT_HIGH }
- enum EPWM_ActionQualifierEventAction { EPWM_AQ_OUTPUT_NO_CHANGE_ZERO, EPWM_AQ_OUTPUT_LOW_ZERO, EPWM_AQ_OUTPUT_HIGH_ZERO, EPWM_AQ_OUTPUT_TOGGLE_ZERO, EPWM_AQ_OUTPUT_NO_CHANGE_PERIOD, EPWM_AQ_OUTPUT_LOW_PERIOD, EPWM_AQ_OUTPUT_HIGH_PERIOD, EPWM_AQ_OUTPUT_TOGGLE_PERIOD, EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPA, EPWM_AQ_OUTPUT_LOW_UP_CMPA, EPWM_AQ_OUTPUT_HIGH_UP_CMPA, EPWM_AQ_OUTPUT_TOGGLE_UP_CMPA, EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPA, EPWM_AQ_OUTPUT_LOW_DOWN_CMPA, EPWM_AQ_OUTPUT_HIGH_DOWN_CMPA, EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPA, EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPB, EPWM_AQ_OUTPUT_LOW_UP_CMPB,

EPWM_AQ_OUTPUT_HIGH_UP_CMPB, EPWM_AQ_OUTPUT_TOGGLE_UP_CMPB, EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPB, EPWM_AQ_OUTPUT_LOW_DOWN_CMPB, EPWM_AQ_OUTPUT_HIGH_DOWN_CMPB, EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPB }

- enum EPWM_AdditionalActionQualifierEventAction { EPWM_AQ_OUTPUT_NO_CHANGE_UP_T1, EPWM_AQ_OUTPUT_LOW_UP_T1, EPWM_AQ_OUTPUT_HIGH_UP_T1, EPWM_AQ_OUTPUT_TOGGLE_UP_T1, EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_T1, EPWM_AQ_OUTPUT_LOW_DOWN_T1, EPWM_AQ_OUTPUT_HIGH_DOWN_T1, EPWM_AQ_OUTPUT_TOGGLE_DOWN_T1, EPWM_AQ_OUTPUT_NO_CHANGE_UP_T2, EPWM_AQ_OUTPUT_LOW_UP_T2, EPWM_AQ_OUTPUT_HIGH_UP_T2, EPWM_AQ_OUTPUT_TOGGLE_UP_T2, EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_T2, EPWM_AQ_OUTPUT_LOW_DOWN_T2, EPWM_AQ_OUTPUT_HIGH_DOWN_T2, EPWM_AQ_OUTPUT_TOGGLE_DOWN_T2 }
- enum EPWM_ActionQualifierOutputModule { EPWM_AQ_OUTPUT_A, EPWM_AQ_OUTPUT_B }
- enum EPWM_ActionQualifierContForce { EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO, EPWM_AQ_SW_SH_LOAD_ON_CNTR_PERIOD, EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_AQ_SW_IMMEDIATE_LOAD }
- enum EPWM_DeadBandOutput { EPWM_DB_OUTPUT_A, EPWM_DB_OUTPUT_B }
- enum EPWM_DeadBandDelayMode { EPWM_DB_RED, EPWM_DB_FED }
- enum EPWM_DeadBandPolarity { EPWM_DB_POLARITY_ACTIVE_HIGH, EPWM_DB_POLARITY_ACTIVE_LOW }
- enum EPWM_DeadBandControlLoadMode { EPWM_DB_LOAD_ON_CNTR_ZERO, EPWM_DB_LOAD_ON_CNTR_PERIOD, EPWM_DB_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_DB_LOAD_FREEZE }
- enum EPWM_RisingEdgeDelayLoadMode { EPWM_RED_LOAD_ON_CNTR_ZERO, EPWM_RED_LOAD_ON_CNTR_PERIOD, EPWM_RED_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_RED_LOAD_FREEZE }
- enum EPWM_FallingEdgeDelayLoadMode { EPWM_FED_LOAD_ON_CNTR_ZERO, EPWM_FED_LOAD_ON_CNTR_PERIOD, EPWM_FED_LOAD_ON_CNTR_ZERO_PERIOD, EPWM_FED_LOAD_FREEZE }
- enum EPWM_DeadBandClockMode { EPWM_DB_COUNTER_CLOCK_FULL_CYCLE, EPWM_DB_COUNTER_CLOCK_HALF_CYCLE }
- enum EPWM_TripZoneDigitalCompareOutput { EPWM_TZ_DC_OUTPUT_A1, EPWM_TZ_DC_OUTPUT_A2, EPWM_TZ_DC_OUTPUT_B1, EPWM_TZ_DC_OUTPUT_B2 }
- enum EPWM_TripZoneDigitalCompareOutputEvent { EPWM_TZ_EVENT_DC_DISABLED, EPWM_TZ_EVENT_DCXH_LOW, EPWM_TZ_EVENT_DCXH_HIGH, EPWM_TZ_EVENT_DCXL_LOW, EPWM_TZ_EVENT_DCXL_HIGH, EPWM_TZ_EVENT_DCXL_HIGH_DCXH_LOW }
- enum EPWM_TripZoneEvent { EPWM_TZ_ACTION_EVENT_TZA, EPWM_TZ_ACTION_EVENT_TZB, EPWM_TZ_ACTION_EVENT_DCAEVT1, EPWM_TZ_ACTION_EVENT_DCAEVT2, EPWM_TZ_ACTION_EVENT_DCBEVT1, EPWM_TZ_ACTION_EVENT_DCBEVT2 }
- enum EPWM_TripZoneAction { EPWM_TZ_ACTION_HIGH_Z, EPWM_TZ_ACTION_HIGH, EPWM_TZ_ACTION_LOW, EPWM_TZ_ACTION_DISABLE }
- enum EPWM_TripZoneAdvancedEvent { EPWM_TZ_ADV_ACTION_EVENT_TZB_D, EPWM_TZ_ADV_ACTION_EVENT_TZB_U, EPWM_TZ_ADV_ACTION_EVENT_TZA_D, EPWM_TZ_ADV_ACTION_EVENT_TZA_U }
- enum EPWM_TripZoneAdvancedAction { EPWM_TZ_ADV_ACTION_HIGH_Z, EPWM_TZ_ADV_ACTION_HIGH, EPWM_TZ_ADV_ACTION_LOW, EPWM_TZ_ADV_ACTION_TOGGLE, EPWM_TZ_ADV_ACTION_DISABLE }

- enum EPWM_TripZoneAdvDigitalCompareEvent {
  EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_U,
  EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_D,
  EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_U,
  EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_D }
- enum EPWM_CycleByCycleTripZoneClearMode {
  EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO,
  EPWM_TZ_CBC_PULSE_CLR_CNTR_PERIOD,
  EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO_PERIOD }
- enum EPWM_ADCStartOfConversionType { EPWM_SOC_A, EPWM_SOC_B }
- enum EPWM_ADCStartOfConversionSource {
  EPWM_SOC_DCxEVT1, EPWM_SOC_TBCTR_ZERO, EPWM_SOC_TBCTR_PERIOD,
  EPWM_SOC_TBCTR_ZERO_OR_PERIOD,
  EPWM_SOC_TBCTR_U_CMPA, EPWM_SOC_TBCTR_U_CMPC,
  EPWM_SOC_TBCTR_D_CMPA, EPWM_SOC_TBCTR_D_CMPC,
  EPWM_SOC_TBCTR_U_CMPB, EPWM_SOC_TBCTR_U_CMPD,
  EPWM_SOC_TBCTR_D_CMPB, EPWM_SOC_TBCTR_D_CMPD }
- enum EPWM_DigitalCompareType { EPWM_DC_TYPE_DCAH, EPWM_DC_TYPE_DCAL,
  EPWM_DC_TYPE_DCBH, EPWM_DC_TYPE_DCBL }
- enum EPWM_DigitalCompareTripInput {
  EPWM_DC_TRIP_TRIPIN1, EPWM_DC_TRIP_TRIPIN2, EPWM_DC_TRIP_TRIPIN3,
  EPWM_DC_TRIP_TRIPIN4,
  EPWM_DC_TRIP_TRIPIN5, EPWM_DC_TRIP_TRIPIN6, EPWM_DC_TRIP_TRIPIN7,
  EPWM_DC_TRIP_TRIPIN8,
  EPWM_DC_TRIP_TRIPIN9, EPWM_DC_TRIP_TRIPIN10, EPWM_DC_TRIP_TRIPIN11,
  EPWM_DC_TRIP_TRIPIN12,
  EPWM_DC_TRIP_TRIPIN14, EPWM_DC_TRIP_TRIPIN15,
  EPWM_DC_TRIP_COMBINATION }
- enum EPWM_DigitalCompareBlankingPulse {
  EPWM_DC_WINDOW_START_TBCTR_PERIOD,
  EPWM_DC_WINDOW_START_TBCTR_ZERO,
  EPWM_DC_WINDOW_START_TBCTR_ZERO_PERIOD }
- enum EPWM_DigitalCompareFilterInput { EPWM_DC_WINDOW_SOURCE_DCAEVT1,
  EPWM_DC_WINDOW_SOURCE_DCAEVT2, EPWM_DC_WINDOW_SOURCE_DCBEVT1,
  EPWM_DC_WINDOW_SOURCE_DCBEVT2 }
- enum EPWM_DigitalCompareModule { EPWM_DC_MODULE_A, EPWM_DC_MODULE_B }
- enum EPWM_DigitalCompareEvent { EPWM_DC_EVENT_1, EPWM_DC_EVENT_2 }
- enum EPWM_DigitalCompareEventSource {
  EPWM_DC_EVENT_SOURCE_ORIG_SIGNAL,
  EPWM_DC_EVENT_SOURCE_FILT_SIGNAL }
- enum EPWM_DigitalCompareSyncMode { EPWM_DC_EVENT_INPUT_SYNCED,
  EPWM_DC_EVENT_INPUT_NOT_SYNCED }
- enum EPWM_GlobalLoadTrigger {
  EPWM_GL_LOAD_PULSE_CNTR_ZERO, EPWM_GL_LOAD_PULSE_CNTR_PERIOD,
  EPWM_GL_LOAD_PULSE_CNTR_ZERO_PERIOD, EPWM_GL_LOAD_PULSE_SYNC,
  EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_ZERO,
  EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_PERIOD,
  EPWM_GL_LOAD_PULSE_SYNC_CNTR_ZERO_PERIOD,
  EPWM_GL_LOAD_PULSE_GLOBAL_FORCE }
- enum EPWM_ValleyTriggerSource {
  EPWM_VALLEY_TRIGGER_EVENT_SOFTWARE,
  EPWM_VALLEY_TRIGGER_EVENT_CNTR_ZERO,
  EPWM_VALLEY_TRIGGER_EVENT_CNTR_PERIOD,

EPWM_VALLEY_TRIGGER_EVENT_CNTR_ZERO_PERIOD,
EPWM_VALLEY_TRIGGER_EVENT_DCAEVT1,
EPWM_VALLEY_TRIGGER_EVENT_DCAEVT2,
EPWM_VALLEY_TRIGGER_EVENT_DCBEVT1,
EPWM_VALLEY_TRIGGER_EVENT_DCBEVT2 }
- enum EPWM_ValleyCounterEdge { EPWM_VALLEY_COUNT_START_EDGE, EPWM_VALLEY_COUNT_STOP_EDGE }
- enum EPWM_ValleyDelayMode { EPWM_VALLEY_DELAY_MODE_SW_DELAY, EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SW_DELAY, EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_1_SW_DELAY, EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_2_SW_DELAY, EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_4_SW_DELAY }
- enum EPWM_DigitalCompareEdgeFilterMode { EPWM_DC_EDGEFILT_MODE_RISING, EPWM_DC_EDGEFILT_MODE_FALLING, EPWM_DC_EDGEFILT_MODE_BOTH }
- enum EPWM_DigitalCompareEdgeFilterEdgeCount { EPWM_DC_EDGEFILT_EDGECNT_0, EPWM_DC_EDGEFILT_EDGECNT_1, EPWM_DC_EDGEFILT_EDGECNT_2, EPWM_DC_EDGEFILT_EDGECNT_3, EPWM_DC_EDGEFILT_EDGECNT_4, EPWM_DC_EDGEFILT_EDGECNT_5, EPWM_DC_EDGEFILT_EDGECNT_6, EPWM_DC_EDGEFILT_EDGECNT_7 }
- enum EPWM_LockRegisterGroup { EPWM_REGISTER_GROUP_GLOBAL_LOAD, EPWM_REGISTER_GROUP_TRIP_ZONE, EPWM_REGISTER_GROUP_TRIP_ZONE_CLEAR, EPWM_REGISTER_GROUP_DIGITAL_COMPARE }

Functions

- static void EPWM_setTimeBaseCounter (uint32_t base, uint16_t count)
- static void EPWM_setCountModeAfterSync (uint32_t base, EPWM_SyncCountMode mode)
- static void EPWM_setClockPrescaler (uint32_t base, EPWM_ClockDivider prescaler, EPWM_HSClockDivider highSpeedPrescaler)
- static void EPWM_forceSyncPulse (uint32_t base)
- static void EPWM_setSyncOutPulseMode (uint32_t base, EPWM_SyncOutPulseMode mode)
- static void EPWM_setPeriodLoadMode (uint32_t base, EPWM_PeriodLoadMode loadMode)
- static void EPWM_enablePhaseShiftLoad (uint32_t base)
- static void EPWM_disablePhaseShiftLoad (uint32_t base)
- static void EPWM_setTimeBaseCounterMode (uint32_t base, EPWM_TimeBaseCountMode counterMode)
- static void EPWM_selectPeriodLoadEvent (uint32_t base, EPWM_PeriodShadowLoadMode shadowLoadMode)
- static void EPWM_enableOneShotSync (uint32_t base)
- static void EPWM_disableOneShotSync (uint32_t base)
- static void EPWM_startOneShotSync (uint32_t base)
- static bool EPWM_getTimeBaseCounterOverflowStatus (uint32_t base)
- static void EPWM_clearTimeBaseCounterOverflowEvent (uint32_t base)
- static bool EPWM_getSyncStatus (uint32_t base)
- static void EPWM_clearSyncEvent (uint32_t base)
- static uint16_t EPWM_getTimeBaseCounterDirection (uint32_t base)
- static void EPWM_setPhaseShift (uint32_t base, uint16_t phaseCount)
- static void EPWM_setTimeBasePeriod (uint32_t base, uint16_t periodCount)
- static uint16_t EPWM_getTimeBasePeriod (uint32_t base)
- static void EPWM_setupEPWMLinks (uint32_t base, EPWM_CurrentLink epwmLink, EPWM_LinkComponent linkComp)

- static void EPWM_setCounterCompareShadowLoadMode (uint32_t base, EPWM_CounterCompareModule compModule, EPWM_CounterCompareLoadMode loadMode)
- static void EPWM_disableCounterCompareShadowLoadMode (uint32_t base, EPWM_CounterCompareModule compModule)
- static void EPWM_setCounterCompareValue (uint32_t base, EPWM_CounterCompareModule compModule, uint16_t compCount)
- static uint16_t EPWM_getCounterCompareValue (uint32_t base, EPWM_CounterCompareModule compModule)
- static bool EPWM_getCounterCompareShadowStatus (uint32_t base, EPWM_CounterCompareModule compModule)
- static void EPWM_setActionQualifierShadowLoadMode (uint32_t base, EPWM_ActionQualifierModule aqModule, EPWM_ActionQualifierLoadMode loadMode)
- static void EPWM_disableActionQualifierShadowLoadMode (uint32_t base, EPWM_ActionQualifierModule aqModule)
- static void EPWM_setActionQualifierT1TriggerSource (uint32_t base, EPWM_ActionQualifierTriggerSource trigger)
- static void EPWM_setActionQualifierT2TriggerSource (uint32_t base, EPWM_ActionQualifierTriggerSource trigger)
- static void EPWM_setActionQualifierAction (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput, EPWM_ActionQualifierOutput output, EPWM_ActionQualifierOutputEvent event)
- static void EPWM_setActionQualifierActionComplete (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput, EPWM_ActionQualifierEventAction action)
- static void EPWM_setAdditionalActionQualifierActionComplete (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput, EPWM_AdditionalActionQualifierEventAction action)
- static void EPWM_setActionQualifierContSWForceShadowMode (uint32_t base, EPWM_ActionQualifierContForce mode)
- static void EPWM_setActionQualifierContSWForceAction (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput, EPWM_ActionQualifierSWOutput output)
- static void EPWM_setActionQualifierSWAction (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput, EPWM_ActionQualifierOutput output)
- static void EPWM_forceActionQualifierSWAction (uint32_t base, EPWM_ActionQualifierOutputModule epwmOutput)
- static void EPWM_setDeadBandOutputSwapMode (uint32_t base, EPWM_DeadBandOutput output, bool enableSwapMode)
- static void EPWM_setDeadBandDelayMode (uint32_t base, EPWM_DeadBandDelayMode delayMode, bool enableDelayMode)
- static void EPWM_setDeadBandDelayPolarity (uint32_t base, EPWM_DeadBandDelayMode delayMode, EPWM_DeadBandPolarity polarity)
- static void EPWM_setRisingEdgeDeadBandDelayInput (uint32_t base, uint16_t input)
- static void EPWM_setFallingEdgeDeadBandDelayInput (uint32_t base, uint16_t input)
- static void EPWM_setDeadBandControlShadowLoadMode (uint32_t base, EPWM_DeadBandControlLoadMode loadMode)
- static void EPWM_disableDeadBandControlShadowLoadMode (uint32_t base)
- static void EPWM_setRisingEdgeDelayCountShadowLoadMode (uint32_t base, EPWM_RisingEdgeDelayLoadMode loadMode)
- static void EPWM_disableRisingEdgeDelayCountShadowLoadMode (uint32_t base)
- static void EPWM_setFallingEdgeDelayCountShadowLoadMode (uint32_t base, EPWM_FallingEdgeDelayLoadMode loadMode)
- static void EPWM_disableFallingEdgeDelayCountShadowLoadMode (uint32_t base)
- static void EPWM_setDeadBandCounterClock (uint32_t base, EPWM_DeadBandClockMode clockMode)

- static void EPWM_setRisingEdgeDelayCount (uint32_t base, uint16_t redCount)
- static void EPWM_setFallingEdgeDelayCount (uint32_t base, uint16_t fedCount)
- static void EPWM_enableChopper (uint32_t base)
- static void EPWM_disableChopper (uint32_t base)
- static void EPWM_setChopperDutyCycle (uint32_t base, uint16_t dutyCycleCount)
- static void EPWM_setChopperFreq (uint32_t base, uint16_t freqDiv)
- static void EPWM_setChopperFirstPulseWidth (uint32_t base, uint16_t firstPulseWidth)
- static void EPWM_enableTripZoneSignals (uint32_t base, uint16_t tzSignal)
- static void EPWM_disableTripZoneSignals (uint32_t base, uint16_t tzSignal)
- static void EPWM_setTripZoneDigitalCompareEventCondition (uint32_t base, EPWM_TripZoneDigitalCompareOutput dcType, EPWM_TripZoneDigitalCompareOutputEvent dcEvent)
- static void EPWM_enableTripZoneAdvAction (uint32_t base)
- static void EPWM_disableTripZoneAdvAction (uint32_t base)
- static void EPWM_setTripZoneAction (uint32_t base, EPWM_TripZoneEvent tzEvent, EPWM_TripZoneAction tzAction)
- static void EPWM_setTripZoneAdvAction (uint32_t base, EPWM_TripZoneAdvancedEvent tzAdvEvent, EPWM_TripZoneAdvancedAction tzAdvAction)
- static void EPWM_setTripZoneAdvDigitalCompareActionA (uint32_t base, EPWM_TripZoneAdvDigitalCompareEvent tzAdvDCEvent, EPWM_TripZoneAdvancedAction tzAdvDCAction)
- static void EPWM_setTripZoneAdvDigitalCompareActionB (uint32_t base, EPWM_TripZoneAdvDigitalCompareEvent tzAdvDCEvent, EPWM_TripZoneAdvancedAction tzAdvDCAction)
- static void EPWM_enableTripZoneInterrupt (uint32_t base, uint16_t tzInterrupt)
- static void EPWM_disableTripZoneInterrupt (uint32_t base, uint16_t tzInterrupt)
- static uint16_t EPWM_getTripZoneFlagStatus (uint32_t base)
- static uint16_t EPWM_getCycleByCycleTripZoneFlagStatus (uint32_t base)
- static uint16_t EPWM_getOneShotTripZoneFlagStatus (uint32_t base)
- static void EPWM_selectCycleByCycleTripZoneClearEvent (uint32_t base, EPWM_CycleByCycleTripZoneClearMode clearEvent)
- static void EPWM_clearTripZoneFlag (uint32_t base, uint16_t tzFlags)
- static void EPWM_clearCycleByCycleTripZoneFlag (uint32_t base, uint16_t tzCBCFlags)
- static void EPWM_clearOneShotTripZoneFlag (uint32_t base, uint16_t tzOSTFlags)
- static void EPWM_forceTripZoneEvent (uint32_t base, uint16_t tzForceEvent)
- static void EPWM_enableInterrupt (uint32_t base)
- static void EPWM_disableInterrupt (uint32_t base)
- static void EPWM_setInterruptSource (uint32_t base, uint16_t interruptSource)
- static void EPWM_setInterruptEventCount (uint32_t base, uint16_t eventCount)
- static bool EPWM_getEventTriggerInterruptStatus (uint32_t base)
- static void EPWM_clearEventTriggerInterruptFlag (uint32_t base)
- static void EPWM_enableInterruptEventCountInit (uint32_t base)
- static void EPWM_disableInterruptEventCountInit (uint32_t base)
- static void EPWM_forceInterruptEventCountInit (uint32_t base)
- static void EPWM_setInterruptEventCountInitValue (uint32_t base, uint16_t eventCount)
- static uint16_t EPWM_getInterruptEventCount (uint32_t base)
- static void EPWM_forceEventTriggerInterrupt (uint32_t base)
- static void EPWM_enableADCTrigger (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_disableADCTrigger (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_setADCTriggerSource (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType, EPWM_ADCStartOfConversionSource socSource)
- static void EPWM_setADCTriggerEventPrescale (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType, uint16_t preScaleCount)

- static bool EPWM_getADCTriggerFlagStatus (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_clearADCTriggerFlag (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_enableADCTriggerEventCountInit (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_disableADCTriggerEventCountInit (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_forceADCTriggerEventCountInit (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_setADCTriggerEventCountInitValue (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType, uint16_t eventCount)
- static uint16_t EPWM_getADCTriggerEventCount (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_forceADCTrigger (uint32_t base, EPWM_ADCStartOfConversionType adcSOCType)
- static void EPWM_selectDigitalCompareTripInput (uint32_t base, EPWM_DigitalCompareTripInput tripSource, EPWM_DigitalCompareType dcType)
- static void EPWM_enableDigitalCompareBlankingWindow (uint32_t base)
- static void EPWM_disableDigitalCompareBlankingWindow (uint32_t base)
- static void EPWM_enableDigitalCompareWindowInverseMode (uint32_t base)
- static void EPWM_disableDigitalCompareWindowInverseMode (uint32_t base)
- static void EPWM_setDigitalCompareBlankingEvent (uint32_t base, EPWM_DigitalCompareBlankingPulse blankingPulse)
- static void EPWM_setDigitalCompareFilterInput (uint32_t base, EPWM_DigitalCompareFilterInput filterInput)
- static void EPWM_enableDigitalCompareEdgeFilter (uint32_t base)
- static void EPWM_disableDigitalCompareEdgeFilter (uint32_t base)
- static void EPWM_setDigitalCompareEdgeFilterMode (uint32_t base, EPWM_DigitalCompareEdgeFilterMode edgeMode)
- static void EPWM_setDigitalCompareEdgeFilterEdgeCount (uint32_t base, uint16_t edgeCount)
- static uint16_t EPWM_getDigitalCompareEdgeFilterEdgeCount (uint32_t base)
- static uint16_t EPWM_getDigitalCompareEdgeFilterEdgeStatus (uint32_t base)
- static void EPWM_setDigitalCompareWindowOffset (uint32_t base, uint16_t windowOffsetCount)
- static void EPWM_setDigitalCompareWindowLength (uint32_t base, uint16_t windowLengthCount)
- static uint16_t EPWM_getDigitalCompareBlankingWindowOffsetCount (uint32_t base)
- static uint16_t EPWM_getDigitalCompareBlankingWindowLengthCount (uint32_t base)
- static void EPWM_setDigitalCompareEventSource (uint32_t base, EPWM_DigitalCompareModule dcModule, EPWM_DigitalCompareEvent dcEvent, EPWM_DigitalCompareEventSource dcEventSource)
- static void EPWM_setDigitalCompareEventSyncMode (uint32_t base, EPWM_DigitalCompareModule dcModule, EPWM_DigitalCompareEvent dcEvent, EPWM_DigitalCompareSyncMode syncMode)
- static void EPWM_enableDigitalCompareADCTrigger (uint32_t base, EPWM_DigitalCompareModule dcModule)
- static void EPWM_disableDigitalCompareADCTrigger (uint32_t base, EPWM_DigitalCompareModule dcModule)
- static void EPWM_enableDigitalCompareSyncEvent (uint32_t base, EPWM_DigitalCompareModule dcModule)
- static void EPWM_disableDigitalCompareSyncEvent (uint32_t base, EPWM_DigitalCompareModule dcModule)
- static void EPWM_enableDigitalCompareCounterCapture (uint32_t base)
- static void EPWM_disableDigitalCompareCounterCapture (uint32_t base)

- static void EPWM_setDigitalCompareCounterShadowMode (uint32_t base, bool enableShadowMode)
- static bool EPWM_getDigitalCompareCaptureStatus (uint32_t base)
- static uint16_t EPWM_getDigitalCompareCaptureCount (uint32_t base)
- static void EPWM_enableDigitalCompareTripCombinationInput (uint32_t base, uint16_t tripInput, EPWM_DigitalCompareType dcType)
- static void EPWM_disableDigitalCompareTripCombinationInput (uint32_t base, uint16_t tripInput, EPWM_DigitalCompareType dcType)
- static void EPWM_enableValleyCapture (uint32_t base)
- static void EPWM_disableValleyCapture (uint32_t base)
- static void EPWM_startValleyCapture (uint32_t base)
- static void EPWM_setValleyTriggerSource (uint32_t base, EPWM_ValleyTriggerSource trigger)
- static void EPWM_setValleyTriggerEdgeCounts (uint32_t base, uint16_t startCount, uint16_t stopCount)
- static void EPWM_enableValleyHWDelay (uint32_t base)
- static void EPWM_disableValleyHWDelay (uint32_t base)
- static void EPWM_setValleySWDelayValue (uint32_t base, uint16_t delayOffsetValue)
- static void EPWM_setValleyDelayDivider (uint32_t base, EPWM_ValleyDelayMode delayMode)
- static bool EPWM_getValleyEdgeStatus (uint32_t base, EPWM_ValleyCounterEdge edge)
- static uint16_t EPWM_getValleyCount (uint32_t base)
- static uint16_t EPWM_getValleyHWDelay (uint32_t base)
- static void EPWM_enableGlobalLoad (uint32_t base)
- static void EPWM_disableGlobalLoad (uint32_t base)
- static void EPWM_setGlobalLoadTrigger (uint32_t base, EPWM_GlobalLoadTrigger loadTrigger)
- static void EPWM_setGlobalLoadEventPrescale (uint32_t base, uint16_t prescalePulseCount)
- static uint16_t EPWM_getGlobalLoadEventCount (uint32_t base)
- static void EPWM_disableGlobalLoadOneShotMode (uint32_t base)
- static void EPWM_enableGlobalLoadOneShotMode (uint32_t base)
- static void EPWM_setGlobalLoadOneShotLatch (uint32_t base)
- static void EPWM_forceGlobalLoadOneShotEvent (uint32_t base)
- static void EPWM_enableGlobalLoadRegisters (uint32_t base, uint16_t loadRegister)
- static void EPWM_disableGlobalLoadRegisters (uint32_t base, uint16_t loadRegister)
- static void EPWM_lockRegisters (uint32_t base, EPWM_LockRegisterGroup registerGroup)
- void EPWM_setEmulationMode (uint32_t base, EPWM_EmulationMode emulationMode)

18.2.1 Detailed Description

The code for this module is contained in `driverlib/epwm.c`, with `driverlib/epwm.h` containing the API declarations for use by applications.

18.2.2 Macro Definition Documentation

18.2.2.1 #define EPWM_TIME_BASE_STATUS_COUNT_UP

Time base counter is counting up

18.2.2.2 #define EPWM_TIME_BASE_STATUS_COUNT_DOWN

Time base counter is counting down

18.2.2.3 #define EPWM_DB_INPUT_EPWMA

Input signal is ePWMA

Referenced by EPWM_setFallingEdgeDeadBandDelayInput(), and EPWM_setRisingEdgeDeadBandDelayInput().

18.2.2.4 #define EPWM_DB_INPUT_EPWMB

Input signal is ePWMA

Referenced by EPWM_setFallingEdgeDeadBandDelayInput(), and EPWM_setRisingEdgeDeadBandDelayInput().

18.2.2.5 #define EPWM_DB_INPUT_DB_RED

Input signal is the output of Rising Edge delay

Referenced by EPWM_setFallingEdgeDeadBandDelayInput().

18.2.2.6 #define EPWM_TZ_SIGNAL_CBC1

TZ1 Cycle By Cycle

18.2.2.7 #define EPWM_TZ_SIGNAL_CBC2

TZ2 Cycle By Cycle

18.2.2.8 #define EPWM_TZ_SIGNAL_CBC3

TZ3 Cycle By Cycle

18.2.2.9 #define EPWM_TZ_SIGNAL_CBC4

TZ4 Cycle By Cycle

18.2.2.10 #define EPWM_TZ_SIGNAL_CBC5

TZ5 Cycle By Cycle

18.2.2.11 #define EPWM_TZ_SIGNAL_CBC6

TZ6 Cycle By Cycle

18.2.2.12 #define EPWM_TZ_SIGNAL_DCAEVT2

DCAEVT2 Cycle By Cycle

18.2.2.13 #define EPWM_TZ_SIGNAL_DCBEVT2

DCBEVT2 Cycle By Cycle

18.2.2.14 #define EPWM_TZ_SIGNAL_OSHT1

One-shot TZ1

18.2.2.15 #define EPWM_TZ_SIGNAL_OSHT2

One-shot TZ2

18.2.2.16 #define EPWM_TZ_SIGNAL_OSHT3

One-shot TZ3

18.2.2.17 #define EPWM_TZ_SIGNAL_OSHT4

One-shot TZ4

18.2.2.18 #define EPWM_TZ_SIGNAL_OSHT5

One-shot TZ5

18.2.2.19 #define EPWM_TZ_SIGNAL_OSHT6

One-shot TZ6

18.2.2.20 #define EPWM_TZ_SIGNAL_DCAEVT1

One-shot DCAEVT1

18.2.2.21 #define EPWM_TZ_SIGNAL_DCBEVT1

One-shot DCBEVT1

18.2.2.22 #define EPWM_TZ_INTERRUPT_CBC

Trip Zones Cycle By Cycle interrupt

18.2.2.23 #define EPWM_TZ_INTERRUPT_OST

Trip Zones One Shot interrupt

18.2.2.24 #define EPWM_TZ_INTERRUPT_DCAEVT1

Digital Compare A Event 1 interrupt

18.2.2.25 #define EPWM_TZ_INTERRUPT_DCAEVT2

Digital Compare A Event 2 interrupt

18.2.2.26 #define EPWM_TZ_INTERRUPT_DCBEVT1

Digital Compare B Event 1 interrupt

18.2.2.27 #define EPWM_TZ_INTERRUPT_DCBEVT2

Digital Compare B Event 2 interrupt

18.2.2.28 #define EPWM_TZ_FLAG_CBC

Trip Zones Cycle By Cycle flag

18.2.2.29 #define EPWM_TZ_FLAG_OST

Trip Zones One Shot flag

18.2.2.30 #define EPWM_TZ_FLAG_DCAEVT1

Digital Compare A Event 1 flag

18.2.2.31 #define EPWM_TZ_FLAG_DCAEVT2

Digital Compare A Event 2 flag

18.2.2.32 #define EPWM_TZ_FLAG_DCBEVT1

Digital Compare B Event 1 flag

18.2.2.33 #define EPWM_TZ_FLAG_DCBEVT2

Digital Compare B Event 2 flag

18.2.2.34 #define EPWM_TZ_INTERRUPT

Trip Zone interrupt

18.2.2.35 #define EPWM_TZ_CBC_FLAG_1

CBC flag 1

18.2.2.36 #define EPWM_TZ_CBC_FLAG_2

CBC flag 2

18.2.2.37 #define EPWM_TZ_CBC_FLAG_3

CBC flag 3

18.2.2.38 #define EPWM_TZ_CBC_FLAG_4

CBC flag 4

18.2.2.39 #define EPWM_TZ_CBC_FLAG_5

CBC flag 5

18.2.2.40 #define EPWM_TZ_CBC_FLAG_6

CBC flag 6

18.2.2.41 #define EPWM_TZ_CBC_FLAG_DCAEVT2

CBC flag Digital compare event A2

18.2.2.42 #define EPWM_TZ_CBC_FLAG_DCBEVT2

CBC flag Digital compare event B2

18.2.2.43 #define EPWM_TZ_OST_FLAG_OST1

OST flag OST1

18.2.2.44 #define EPWM_TZ_OST_FLAG_OST2

OST flag OST2

18.2.2.45 #define EPWM_TZ_OST_FLAG_OST3

OST flag OST3

18.2.2.46 #define EPWM_TZ_OST_FLAG_OST4

OST flag OST4

18.2.2.47 #define EPWM_TZ_OST_FLAG_OST5

OST flag OST5

18.2.2.48 #define EPWM_TZ_OST_FLAG_OST6

OST flag OST6

18.2.2.49 #define EPWM_TZ_OST_FLAG_DCAEVT1

OST flag Digital compare event A1

18.2.2.50 #define EPWM_TZ_OST_FLAG_DCBEVT1

OST flag Digital compare event B1

18.2.2.51 #define EPWM_TZ_FORCE_EVENT_CBC

Force Cycle By Cycle trip event

18.2.2.52 #define EPWM_TZ_FORCE_EVENT_OST

Force a One-Shot Trip Event

18.2.2.53 #define EPWM_TZ_FORCE_EVENT_DCAEVT1

ForceDigital Compare Output A Event 1

18.2.2.54 #define EPWM_TZ_FORCE_EVENT_DCAEVT2

ForceDigital Compare Output A Event 2

18.2.2.55 #define EPWM_TZ_FORCE_EVENT_DCBEVT1

ForceDigital Compare Output B Event 1

18.2.2.56 #define EPWM_TZ_FORCE_EVENT_DCBEVT2

ForceDigital Compare Output B Event 2

18.2.2.57 #define EPWM_INT_TBCTR_ZERO

Time-base counter equal to zero

18.2.2.58 #define EPWM_INT_TBCTR_PERIOD

Time-base counter equal to period

18.2.2.59 #define EPWM_INT_TBCTR_ZERO_OR_PERIOD

Time-base counter equal to zero or period

18.2.2.60 #define EPWM_INT_TBCTR_U_CMPA

time-base counter equal to CMPA when the timer is incrementing

Referenced by EPWM_setInterruptSource().

18.2.2.61 #define EPWM_INT_TBCTR_U_CMPC

time-base counter equal to CMPC when the timer is incrementing

Referenced by EPWM_setInterruptSource().

18.2.2.62 #define EPWM_INT_TBCTR_D_CMPA

time-base counter equal to CMPA when the timer is decrementing

Referenced by EPWM_setInterruptSource().

18.2.2.63 #define EPWM_INT_TBCTR_D_CMPC

time-base counter equal to CMPC when the timer is decrementing

Referenced by EPWM_setInterruptSource().

18.2.2.64 #define EPWM_INT_TBCTR_U_CMPB

time-base counter equal to CMPB when the timer is incrementing

Referenced by EPWM_setInterruptSource().

18.2.2.65 #define EPWM_INT_TBCTR_U_CMPD

time-base counter equal to CMPD when the timer is incrementing

Referenced by EPWM_setInterruptSource().

18.2.2.66 #define EPWM_INT_TBCTR_D_CMPB

time-base counter equal to CMPB when the timer is decrementing

Referenced by EPWM_setInterruptSource().

18.2.2.67 #define EPWM_INT_TBCTR_D_CMPD

time-base counter equal to CMPD when the timer is decrementing

Referenced by EPWM_setInterruptSource().

18.2.2.68 #define EPWM_DC_COMBINATIONAL_TRIPIN1

Combinational Trip 1 input

18.2.2.69 #define EPWM_DC_COMBINATIONAL_TRIPIN2

Combinational Trip 2 input

18.2.2.70 #define EPWM_DC_COMBINATIONAL_TRIPIN3

Combinational Trip 3 input

18.2.2.71 #define EPWM_DC_COMBINATIONAL_TRIPIN4

Combinational Trip 4 input

18.2.2.72 #define EPWM_DC_COMBINATIONAL_TRIPIN5

Combinational Trip 5 input

18.2.2.73 #define EPWM_DC_COMBINATIONAL_TRIPIN6

Combinational Trip 6 input

18.2.2.74 #define EPWM_DC_COMBINATIONAL_TRIPIN7

Combinational Trip 7 input

18.2.2.75 #define EPWM_DC_COMBINATIONAL_TRIPIN8

Combinational Trip 8 input

18.2.2.76 #define EPWM_DC_COMBINATIONAL_TRIPIN9

Combinational Trip 9 input

18.2.2.77 #define EPWM_DC_COMBINATIONAL_TRIPIN10

Combinational Trip 10 input

18.2.2.78 #define EPWM_DC_COMBINATIONAL_TRIPIN11

Combinational Trip 11 input

18.2.2.79 #define EPWM_DC_COMBINATIONAL_TRIPIN12

Combinational Trip 12 input

18.2.2.80 #define EPWM_DC_COMBINATIONAL_TRIPIN14

Combinational Trip 14 input

18.2.2.81 #define EPWM_DC_COMBINATIONAL_TRIPIN15

Combinational Trip 15 input

18.2.2.82 #define EPWM_GL_REGISTER_TBPRD_TBPRDHR

Global load TBPRD:TBPRDHR

18.2.2.83 #define EPWM_GL_REGISTER_CMPA_CMPAHR

Global load CMPA:CMPAHR

18.2.2.84 #define EPWM_GL_REGISTER_CMPB_CMPBHR

Global load CMPB:CMPBHR

18.2.2.85 #define EPWM_GL_REGISTER_CMPC

Global load CMPC

18.2.2.86 #define EPWM_GL_REGISTER_CMPD

Global load CMPD

18.2.2.87 #define EPWM_GL_REGISTER_DBRED_DBREDHR

Global load DBRED:DBREDHR

18.2.2.88 #define EPWM_GL_REGISTER_DBFED_DBFEDHR

Global load DBFED:DBFEDHR

18.2.2.89 #define EPWM_GL_REGISTER_DBCTL

Global load DBCTL

18.2.2.90 #define EPWM_GL_REGISTER_AQCTLA_AQCTLA2

Global load AQCTLA/A2

18.2.2.91 #define EPWM_GL_REGISTER_AQCTLB_AQCTLB2

Global load AQCTLB/B2

18.2.2.92 #define EPWM_GL_REGISTER_AQCSFRC

Global load AQCSFRC

18.2.3 Enumeration Type Documentation

18.2.3.1 enum **EPWM_EmulationMode**

Values that can be passed to EPWM_setEmulationMode() as the *emulationMode* parameter.

Enumerator

EPWM_EMULATION_STOP_AFTER_NEXT_TB Stop after next Time Base counter increment or decrement.

EPWM_EMULATION_STOP_AFTER_FULL_CYCLE Stop when counter completes whole cycle.

EPWM_EMULATION_FREE_RUN Free run.

18.2.3.2 enum **EPWM_SyncCountMode**

Values that can be passed to EPWM_setCountModeAfterSync() as the *mode* parameter.

Enumerator

EPWM_COUNT_MODE_DOWN_AFTER_SYNC Count down after sync event.

EPWM_COUNT_MODE_UP_AFTER_SYNC Count up after sync event.

18.2.3.3 enum **EPWM_ClockDivider**

Values that can be passed to EPWM_setClockPrescaler() as the *prescaler* parameter.

Enumerator

EPWM_CLOCK_DIVIDER_1 Divide clock by 1.

EPWM_CLOCK_DIVIDER_2 Divide clock by 2.

EPWM_CLOCK_DIVIDER_4 Divide clock by 4.
EPWM_CLOCK_DIVIDER_8 Divide clock by 8.
EPWM_CLOCK_DIVIDER_16 Divide clock by 16.
EPWM_CLOCK_DIVIDER_32 Divide clock by 32.
EPWM_CLOCK_DIVIDER_64 Divide clock by 64.
EPWM_CLOCK_DIVIDER_128 Divide clock by 128.

18.2.3.4 enum **EPWM_HSClockDivider**

Values that can be passed to EPWM_setClockPrescaler() as the *highSpeedPrescaler* parameter.

Enumerator

EPWM_HSCLOCK_DIVIDER_1 Divide clock by 1.
EPWM_HSCLOCK_DIVIDER_2 Divide clock by 2.
EPWM_HSCLOCK_DIVIDER_4 Divide clock by 4.
EPWM_HSCLOCK_DIVIDER_6 Divide clock by 6.
EPWM_HSCLOCK_DIVIDER_8 Divide clock by 8.
EPWM_HSCLOCK_DIVIDER_10 Divide clock by 10.
EPWM_HSCLOCK_DIVIDER_12 Divide clock by 12.
EPWM_HSCLOCK_DIVIDER_14 Divide clock by 14.

18.2.3.5 enum **EPWM_SyncOutPulseMode**

Values that can be passed to EPWM_setSyncOutPulseMode() as the *mode* parameter.

Enumerator

EPWM_SYNC_OUT_PULSE_ON_SOFTWARE sync pulse is generated by software
EPWM_SYNC_OUT_PULSE_ON_EPWMxSYNCIN sync pulse is passed from EPWMxSYNCIN
EPWM_SYNC_OUT_PULSE_ON_COUNTER_ZERO sync pulse is generated when time base counter equals zero
EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_B sync pulse is generated when time base counter equals compare B value.
EPWM_SYNC_OUT_PULSE_DISABLED sync pulse is disabled
EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_C sync pulse is generated when time base counter equals compare D value.
EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_D sync pulse is disabled.

18.2.3.6 enum **EPWM_PeriodLoadMode**

Values that can be passed to EPWM_setPeriodLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_PERIOD_SHADOW_LOAD PWM Period register access is through shadow register.

EPWM_PERIOD_DIRECT_LOAD PWM Period register access is directly.

18.2.3.7 enum **EPWM_TimeBaseCountMode**

Values that can be passed to EPWM_setTimeBaseCounterMode() as the *counterMode* parameter.

Enumerator

EPWM_COUNTER_MODE_UP Up - count mode.

EPWM_COUNTER_MODE_DOWN Down - count mode.

EPWM_COUNTER_MODE_UP_DOWN Up - down - count mode.

EPWM_COUNTER_MODE_STOP_FREEZE Stop - Freeze counter.

18.2.3.8 enum **EPWM_PeriodShadowLoadMode**

Values that can be passed to EPWM_selectPeriodLoadEvent() as the *shadowLoadMode* parameter.

Enumerator

EPWM_SHADOW_LOAD_MODE_COUNTER_ZERO shadow to active load occurs when time base counter reaches 0.

EPWM_SHADOW_LOAD_MODE_COUNTER_SYNC shadow to active load occurs when time base counter reaches 0 and a SYNC occurs

EPWM_SHADOW_LOAD_MODE_SYNC shadow to active load occurs only when a SYNC occurs

18.2.3.9 enum **EPWM_CurrentLink**

Values that can be passed to EPWM_setupEPWMLinks() as the *epwmLink* parameter.

Enumerator

EPWM_LINK_WITH_EPWM_1 link current ePWM with ePWM1

EPWM_LINK_WITH_EPWM_2 link current ePWM with ePWM2

EPWM_LINK_WITH_EPWM_3 link current ePWM with ePWM3

EPWM_LINK_WITH_EPWM_4 link current ePWM with ePWM4

EPWM_LINK_WITH_EPWM_5 link current ePWM with ePWM5

EPWM_LINK_WITH_EPWM_6 link current ePWM with ePWM6

EPWM_LINK_WITH_EPWM_7 link current ePWM with ePWM7

EPWM_LINK_WITH_EPWM_8 link current ePWM with ePWM8

18.2.3.10 enum **EPWM_LinkComponent**

Values that can be passed to EPWM_setupEPWMLinks() as the *linkComp* parameter.

Enumerator

EPWM_LINK_TBPRD link TBPRD:TBPRDHR registers

EPWM_LINK_COMP_A link COMPA registers

EPWM_LINK_COMP_B link COMPB registers

EPWM_LINK_COMP_C link COMPC registers

EPWM_LINK_COMP_D link COMPD registers
EPWM_LINK_GLDCTL2 link GLDCTL2 registers

18.2.3.11 enum **EPWM_CounterCompareModule**

Values that can be passed to the EPWM_getCounterCompareShadowStatus(), EPWM_setCounterCompareValue(), EPWM_setCounterCompareShadowLoadMode(), EPWM_disableCounterCompareShadowLoadMode() as the *compModule* parameter.

Enumerator

EPWM_COUNTER_COMPARE_A counter compare A
EPWM_COUNTER_COMPARE_B counter compare B
EPWM_COUNTER_COMPARE_C counter compare C
EPWM_COUNTER_COMPARE_D counter compare D

18.2.3.12 enum **EPWM_CounterCompareLoadMode**

Values that can be passed to EPWM_setCounterCompareShadowLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_COMP_LOAD_ON_CNTR_ZERO load when counter equals zero
EPWM_COMP_LOAD_ON_CNTR_PERIOD load when counter equals period
EPWM_COMP_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period
EPWM_COMP_LOAD_FREEZE Freeze shadow to active load.
EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO load when counter equals zero
EPWM_COMP_LOAD_ON_SYNC_CNTR_PERIOD load when counter equals period
EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO_PERIOD load when counter equals zero or period
EPWM_COMP_LOAD_ON_SYNC_ONLY load on sync only

18.2.3.13 enum **EPWM_ActionQualifierModule**

Values that can be passed to EPWM_setActionQualifierShadowLoadMode() and EPWM_disableActionQualifierShadowLoadMode() as the *aqModule* parameter.

Enumerator

EPWM_ACTION_QUALIFIER_A Action Qualifier A.
EPWM_ACTION_QUALIFIER_B Action Qualifier B.

18.2.3.14 enum **EPWM_ActionQualifierLoadMode**

Values that can be passed to EPWM_setActionQualifierShadowLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_AQ_LOAD_ON_CNTR_ZERO load when counter equals zero
EPWM_AQ_LOAD_ON_CNTR_PERIOD load when counter equals period
EPWM_AQ_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period
EPWM_AQ_LOAD_FREEZE Freeze shadow to active load.
EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO load on sync or when counter equals zero
EPWM_AQ_LOAD_ON_SYNC_CNTR_PERIOD load on sync or when counter equals period
EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO_PERIOD load on sync or when counter equals zero or period
EPWM_AQ_LOAD_ON_SYNC_ONLY load on sync only

18.2.3.15 enum **EPWM_ActionQualifierTriggerSource**

Values that can be passed to EPWM_setActionQualifierT1TriggerSource() and EPWM_setActionQualifierT2TriggerSource() as the *trigger* parameter.

Enumerator

EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_1 Digital compare event A 1.
EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_2 Digital compare event A 2.
EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_1 Digital compare event B 1.
EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_2 Digital compare event B 2.
EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_1 Trip zone 1.
EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_2 Trip zone 2.
EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_3 Trip zone 3.
EPWM_AQ_TRIGGER_EVENT_TRIG_EPWM_SYNCIN ePWM sync

18.2.3.16 enum **EPWM_ActionQualifierOutputEvent**

Values that can be passed to EPWM_setActionQualifierAction() as the *event* parameter.

Enumerator

EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO Time base counter equals zero.
EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD Time base counter equals period.
EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA Time base counter up equals COMPA.
EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA Time base counter down equals COMPA.
EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPB Time base counter up equals COMPB.
EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB Time base counter down equals COMPB.
EPWM_AQ_OUTPUT_ON_T1_COUNT_UP T1 event on count up.
EPWM_AQ_OUTPUT_ON_T1_COUNT_DOWN T1 event on count down.
EPWM_AQ_OUTPUT_ON_T2_COUNT_UP T2 event on count up.
EPWM_AQ_OUTPUT_ON_T2_COUNT_DOWN T2 event on count down.

18.2.3.17 enum **EPWM_ActionQualifierOutput**

Values that can be passed to EPWM_setActionQualifierSWAction(), EPWM_setActionQualifierAction() as the *outPut* parameter.

Enumerator

EPWM_AQ_OUTPUT_NO_CHANGE No change in the output pins.

EPWM_AQ_OUTPUT_LOW Set output pins to low.

EPWM_AQ_OUTPUT_HIGH Set output pins to High.

EPWM_AQ_OUTPUT_TOGGLE Toggle the output pins.

18.2.3.18 enum **EPWM_ActionQualifierSWOutput**

Values that can be passed to EPWM_setActionQualifierContSWForceAction() as the *outPut* parameter.

Enumerator

EPWM_AQ_SW_DISABLED Software forcing disabled.

EPWM_AQ_SW_OUTPUT_LOW Set output pins to low.

EPWM_AQ_SW_OUTPUT_HIGH Set output pins to High.

18.2.3.19 enum **EPWM_ActionQualifierEventAction**

Values that can be passed to EPWM_setActionQualifierActionComplete() as the *action* parameter.

Enumerator

EPWM_AQ_OUTPUT_NO_CHANGE_ZERO Time base counter equals zero and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_ZERO Time base counter equals zero and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_ZERO Time base counter equals zero and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_ZERO Time base counter equals zero and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_PERIOD Time base counter equals period and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_PERIOD Time base counter equals period and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_PERIOD Time base counter equals period and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_PERIOD Time base counter equals period and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPA Time base counter up equals COMPA and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_UP_CMPA Time base counter up equals COMPA and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_UP_CMPA Time base counter up equals COMPA and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_UP_CMPA Time base counter up equals COMPA and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPA Time base counter down equals COMPA and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_DOWN_CMPA Time base counter down equals COMPA and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_DOWN_CMPA Time base counter down equals COMPA and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPA Time base counter down equals COMPA and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPB Time base counter up equals COMPB and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_UP_CMPB Time base counter up equals COMPB and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_UP_CMPB Time base counter up equals COMPB and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_UP_CMPB Time base counter up equals COMPB and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPB Time base counter down equals COMPB and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_DOWN_CMPB Time base counter down equals COMPB and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_DOWN_CMPB Time base counter down equals COMPB and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPB Time base counter down equals COMPB and toggle the output pins.

18.2.3.20 enum **EPWM_AdditionalActionQualifierEventAction**

Values that can be passed to EPWM_setAdditionalActionQualifierActionComplete() as the *action* parameter.

Enumerator

EPWM_AQ_OUTPUT_NO_CHANGE_UP_T1 T1 event on count up and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_UP_T1 T1 event on count up and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_UP_T1 T1 event on count up and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_UP_T1 T1 event on count up and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_T1 T1 event on count down and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_DOWN_T1 T1 event on count down and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_DOWN_T1 T1 event on count down and set output pins to high.

EPWM_AQ_OUTPUT_TOGGLE_DOWN_T1 T1 event on count down and toggle the output pins.

EPWM_AQ_OUTPUT_NO_CHANGE_UP_T2 T2 event on count up and no change in the output pins.

EPWM_AQ_OUTPUT_LOW_UP_T2 T2 event on count up and set output pins to low.

EPWM_AQ_OUTPUT_HIGH_UP_T2 T2 event on count up and set output pins to high.
EPWM_AQ_OUTPUT_TOGGLE_UP_T2 T2 event on count up and toggle the output pins.
EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_T2 T2 event on count down and no change in the output pins.
EPWM_AQ_OUTPUT_LOW_DOWN_T2 T2 event on count down and set output pins to low.
EPWM_AQ_OUTPUT_HIGH_DOWN_T2 T2 event on count down and set output pins to high.
EPWM_AQ_OUTPUT_TOGGLE_DOWN_T2 T2 event on count down and toggle the output pins.

18.2.3.21 enum **EPWM_ActionQualifierOutputModule**

Values that can be passed to EPWM_forceActionQualifierSWAction(), EPWM_setActionQualifierSWAction(), EPWM_setActionQualifierAction() EPWM_setActionQualifierContSWForceAction() as the *epwmOutput* parameter.

Enumerator

EPWM_AQ_OUTPUT_A ePWMxA output
EPWM_AQ_OUTPUT_B ePWMxB output

18.2.3.22 enum **EPWM_ActionQualifierContForce**

Values that can be passed to EPWM_setActionQualifierContSWForceShadowMode() as the *mode* parameter.

Enumerator

EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO shadow mode load when counter equals zero
EPWM_AQ_SW_SH_LOAD_ON_CNTR_PERIOD shadow mode load when counter equals period
EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO_PERIOD shadow mode load when counter equals zero or period
EPWM_AQ_SW_IMMEDIATE_LOAD No shadow load mode. Immediate mode only.

18.2.3.23 enum **EPWM_DeadBandOutput**

Values that can be passed to EPWM_setDeadBandOutputSwapMode() as the *output* parameter.

Enumerator

EPWM_DB_OUTPUT_A DB output is ePWMA.
EPWM_DB_OUTPUT_B DB output is ePWMB.

18.2.3.24 enum **EPWM_DeadBandDelayMode**

Values that can be passed to EPWM_setDeadBandDelayPolarity(), EPWM_setDeadBandDelayMode() as the *delayMode* parameter.

Enumerator

EPWM_DB_RED DB RED (Rising Edge Delay) mode.

EPWM_DB_FED DB FED (Falling Edge Delay) mode.

18.2.3.25 enum **EPWM_DeadBandPolarity**

Values that can be passed to EPWM_setDeadBandDelayPolarity as the *polarity* parameter.

Enumerator

EPWM_DB_POLARITY_ACTIVE_HIGH DB polarity is not inverted.

EPWM_DB_POLARITY_ACTIVE_LOW DB polarity is inverted.

18.2.3.26 enum **EPWM_DeadBandControlLoadMode**

Values that can be passed to EPWM_setDeadBandControlShadowLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_DB_LOAD_ON_CNTR_ZERO load when counter equals zero

EPWM_DB_LOAD_ON_CNTR_PERIOD load when counter equals period

EPWM_DB_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period

EPWM_DB_LOAD_FREEZE Freeze shadow to active load.

18.2.3.27 enum **EPWM_RisingEdgeDelayLoadMode**

Values that can be passed to EPWM_setRisingEdgeDelayCountShadowLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_RED_LOAD_ON_CNTR_ZERO load when counter equals zero

EPWM_RED_LOAD_ON_CNTR_PERIOD load when counter equals period

EPWM_RED_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period

EPWM_RED_LOAD_FREEZE Freeze shadow to active load.

18.2.3.28 enum **EPWM_FallingEdgeDelayLoadMode**

Values that can be passed to EPWM_setFallingEdgeDelayCountShadowLoadMode() as the *loadMode* parameter.

Enumerator

EPWM_FED_LOAD_ON_CNTR_ZERO load when counter equals zero

EPWM_FED_LOAD_ON_CNTR_PERIOD load when counter equals period

EPWM_FED_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period

EPWM_FED_LOAD_FREEZE Freeze shadow to active load.

18.2.3.29 enum **EPWM_DeadBandClockMode**

Values that can be passed to EPWM_setDeadBandCounterClock() as the *clockMode* parameter.

Enumerator

EPWM_DB_COUNTER_CLOCK_FULL_CYCLE Dead band counter runs at TBCLK rate.

EPWM_DB_COUNTER_CLOCK_HALF_CYCLE Dead band counter runs at 2∗TBCLK rate.

18.2.3.30 enum **EPWM_TripZoneDigitalCompareOutput**

Values that can be passed to EPWM_setTripZoneDigitalCompareEventCondition() as the *dcType* parameter.

Enumerator

EPWM_TZ_DC_OUTPUT_A1 Digital Compare output 1 A.

EPWM_TZ_DC_OUTPUT_A2 Digital Compare output 2 A.

EPWM_TZ_DC_OUTPUT_B1 Digital Compare output 1 B.

EPWM_TZ_DC_OUTPUT_B2 Digital Compare output 2 B.

18.2.3.31 enum **EPWM_TripZoneDigitalCompareOutputEvent**

Values that can be passed to EPWM_setTripZoneDigitalCompareEventCondition() as the *dcEvent* parameter.

Enumerator

EPWM_TZ_EVENT_DC_DISABLED Event is disabled.

EPWM_TZ_EVENT_DCXH_LOW Event when DCxH low.

EPWM_TZ_EVENT_DCXH_HIGH Event when DCxH high.

EPWM_TZ_EVENT_DCXL_LOW Event when DCxL low.

EPWM_TZ_EVENT_DCXL_HIGH Event when DCxL high.

EPWM_TZ_EVENT_DCXL_HIGH_DCXH_LOW Event when DCxL high DCxH low.

18.2.3.32 enum **EPWM_TripZoneEvent**

Values that can be passed to EPWM_setTripZoneAction() as the *tzEvent* parameter.

Enumerator

EPWM_TZ_ACTION_EVENT_TZA TZ1 - TZ6, DCAEVT2, DCAEVT1.

EPWM_TZ_ACTION_EVENT_TZB TZ1 - TZ6, DCBEVT2, DCBEVT1.

EPWM_TZ_ACTION_EVENT_DCAEVT1 DCAEVT1 (Digital Compare A event 1)

EPWM_TZ_ACTION_EVENT_DCAEVT2 DCAEVT2 (Digital Compare A event 2)

EPWM_TZ_ACTION_EVENT_DCBEVT1 DCBEVT1 (Digital Compare B event 1)

EPWM_TZ_ACTION_EVENT_DCBEVT2 DCBEVT2 (Digital Compare B event 2)

18.2.3.33 enum **EPWM_TripZoneAction**

Values that can be passed to EPWM_setTripZoneAction() as the *tzAction* parameter.

Enumerator

EPWM_TZ_ACTION_HIGH_Z high impedance output

EPWM_TZ_ACTION_HIGH high voltage state

EPWM_TZ_ACTION_LOW low voltage state

EPWM_TZ_ACTION_DISABLE disable action

18.2.3.34 enum **EPWM_TripZoneAdvancedEvent**

Values that can be passed to EPWM_setTripZoneAdvAction() as the *tzAdvEvent* parameter.

Enumerator

EPWM_TZ_ADV_ACTION_EVENT_TZB_D TZ1 - TZ6, DCBEVT2, DCBEVT1 while counting down.

EPWM_TZ_ADV_ACTION_EVENT_TZB_U TZ1 - TZ6, DCBEVT2, DCBEVT1 while counting up.

EPWM_TZ_ADV_ACTION_EVENT_TZA_D TZ1 - TZ6, DCAEVT2, DCAEVT1 while counting down.

EPWM_TZ_ADV_ACTION_EVENT_TZA_U TZ1 - TZ6, DCAEVT2, DCAEVT1 while counting up.

18.2.3.35 enum **EPWM_TripZoneAdvancedAction**

Values that can be passed to EPWM_setTripZoneAdvDigitalCompareActionA(), EPWM_setTripZoneAdvDigitalCompareActionB(), EPWM_setTripZoneAdvAction() as the *tzAdvDCAction* parameter.

Enumerator

EPWM_TZ_ADV_ACTION_HIGH_Z high impedance output

EPWM_TZ_ADV_ACTION_HIGH high voltage state

EPWM_TZ_ADV_ACTION_LOW low voltage state

EPWM_TZ_ADV_ACTION_TOGGLE toggle the output

EPWM_TZ_ADV_ACTION_DISABLE disable action

18.2.3.36 enum **EPWM_TripZoneAdvDigitalCompareEvent**

Values that can be passed to EPWM_setTripZoneAdvDigitalCompareActionA() and EPWM_setTripZoneAdvDigitalCompareActionB() as the *tzAdvDCEvent* parameter.

Enumerator

EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_U Digital Compare event A/B 1 while counting up.

EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_D Digital Compare event A/B 1 while counting down.

EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_U Digital Compare event A/B 2 while counting up.

EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_D Digital Compare event A/B 2 while counting down.

18.2.3.37 enum **EPWM_CycleByCycleTripZoneClearMode**

Values that can be passed to EPWM_selectCycleByCycleTripZoneClearEvent() as the *clearMode* parameter.

Enumerator

EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO Clear CBC pulse when counter equals zero.

EPWM_TZ_CBC_PULSE_CLR_CNTR_PERIOD Clear CBC pulse when counter equals period.

EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO_PERIOD Clear CBC pulse when counter equals zero or period.

18.2.3.38 enum **EPWM_ADCStartOfConversionType**

Values that can be passed to EPWM_enableADCTrigger(), EPWM_disableADCTrigger(), EPWM_setADCTriggerSource(), EPWM_setADCTriggerEventPrescale(), EPWM_getADCTriggerFlagStatus(), EPWM_clearADCTriggerFlag(), EPWM_enableADCTriggerEventCountInit(), EPWM_disableADCTriggerEventCountInit(), EPWM_forceADCTriggerEventCountInit(), EPWM_setADCTriggerEventCountInitValue(), EPWM_getADCTriggerEventCount(), EPWM_forceADCTrigger() as the *adcSOCType* parameter

Enumerator

EPWM_SOC_A SOC A.

EPWM_SOC_B SOC B.

18.2.3.39 enum **EPWM_ADCStartOfConversionSource**

Values that can be passed to EPWM_setADCTriggerSource() as the *socSource* parameter.

Enumerator

EPWM_SOC_DCxEVT1 Event is based on DCxEVT1.

EPWM_SOC_TBCTR_ZERO Time-base counter equal to zero.

EPWM_SOC_TBCTR_PERIOD Time-base counter equal to period.

EPWM_SOC_TBCTR_ZERO_OR_PERIOD Time-base counter equal to zero or period.

EPWM_SOC_TBCTR_U_CMPA time-base counter equal to CMPA when the timer is incrementing

EPWM_SOC_TBCTR_U_CMPC time-base counter equal to CMPC when the timer is incrementing

EPWM_SOC_TBCTR_D_CMPA time-base counter equal to CMPA when the timer is decrementing

EPWM_SOC_TBCTR_D_CMPC time-base counter equal to CMPC when the timer is decrementing

EPWM_SOC_TBCTR_U_CMPB time-base counter equal to CMPB when the timer is incrementing

EPWM_SOC_TBCTR_U_CMPD time-base counter equal to CMPD when the timer is incrementing

EPWM_SOC_TBCTR_D_CMPB time-base counter equal to CMPB when the timer is decrementing

EPWM_SOC_TBCTR_D_CMPD time-base counter equal to CMPD when the timer is decrementing

18.2.3.40 enum **EPWM_DigitalCompareType**

Values that can be passed to EPWM_selectDigitalCompareTripInput(), EPWM_enableDigitalCompareTripCombinationInput(), EPWM_disableDigitalCompareTripCombinationInput() as the *dcType* parameter.

Enumerator

EPWM_DC_TYPE_DCAH Digital Compare A High.

EPWM_DC_TYPE_DCAL Digital Compare A Low.

EPWM_DC_TYPE_DCBH Digital Compare B High.

EPWM_DC_TYPE_DCBL Digital Compare B Low.

18.2.3.41 enum **EPWM_DigitalCompareTripInput**

Values that can be passed to EPWM_selectDigitalCompareTripInput() as the *tripSource* parameter.

Enumerator

EPWM_DC_TRIP_TRIPIN1 Trip 1.

EPWM_DC_TRIP_TRIPIN2 Trip 2.

EPWM_DC_TRIP_TRIPIN3 Trip 3.

EPWM_DC_TRIP_TRIPIN4 Trip 4.

EPWM_DC_TRIP_TRIPIN5 Trip 5.

EPWM_DC_TRIP_TRIPIN6 Trip 6.

EPWM_DC_TRIP_TRIPIN7 Trip 7.

EPWM_DC_TRIP_TRIPIN8 Trip 8.

EPWM_DC_TRIP_TRIPIN9 Trip 9.

EPWM_DC_TRIP_TRIPIN10 Trip 10.

EPWM_DC_TRIP_TRIPIN11 Trip 11.

EPWM_DC_TRIP_TRIPIN12 Trip 12.

EPWM_DC_TRIP_TRIPIN14 Trip 14.

EPWM_DC_TRIP_TRIPIN15 Trip 15.

EPWM_DC_TRIP_COMBINATION All Trips (Trip1 - Trip 15) are selected.

18.2.3.42 enum **EPWM_DigitalCompareBlankingPulse**

Values that can be passed to EPWM_setDigitalCompareBlankingEvent() as the the *blankingPulse* parameter.

Enumerator

EPWM_DC_WINDOW_START_TBCTR_PERIOD Time base counter equals period.

EPWM_DC_WINDOW_START_TBCTR_ZERO Time base counter equals zero.

EPWM_DC_WINDOW_START_TBCTR_ZERO_PERIOD Time base counter equals zero.

18.2.3.43 enum **EPWM_DigitalCompareFilterInput**

Values that can be passed to EPWM_setDigitalCompareFilterInput() as the *filterInput* parameter.

Enumerator

EPWM_DC_WINDOW_SOURCE_DCAEVT1 DC filter signal source is DCAEVT1.

EPWM_DC_WINDOW_SOURCE_DCAEVT2 DC filter signal source is DCAEVT2.

EPWM_DC_WINDOW_SOURCE_DCBEVT1 DC filter signal source is DCBEVT1.

EPWM_DC_WINDOW_SOURCE_DCBEVT2 DC filter signal source is DCBEVT2.

18.2.3.44 enum **EPWM_DigitalCompareModule**

Values that can be assigned to EPWM_setDigitalCompareEventSource(), EPWM_setDigitalCompareEventSyncMode(), EPWM_enableDigitalCompareSyncEvent() EPWM_enableDigitalCompareADCTrigger(), EPWM_disableDigitalCompareSyncEvent() EPWM_disableDigitalCompareADCTrigger() as the *dcModule* parameter.

Enumerator

EPWM_DC_MODULE_A Digital Compare Module A.

EPWM_DC_MODULE_B Digital Compare Module B.

18.2.3.45 enum **EPWM_DigitalCompareEvent**

Values that can be passed to EPWM_setDigitalCompareEventSource(), EPWM_setDigitalCompareEventSyncMode as the *dcEvent* parameter.

Enumerator

EPWM_DC_EVENT_1 Digital Compare Event number 1.

EPWM_DC_EVENT_2 Digital Compare Event number 2.

18.2.3.46 enum **EPWM_DigitalCompareEventSource**

Values that can be passed to EPWM_setDigitalCompareEventSource() as the *dcEventSource* parameter.

Enumerator

EPWM_DC_EVENT_SOURCE_ORIG_SIGNAL signal source is unfiltered (DCAEVT1/2)

EPWM_DC_EVENT_SOURCE_FILT_SIGNAL signal source is filtered (DCEVTFILT)

18.2.3.47 enum **EPWM_DigitalCompareSyncMode**

Values that can be passed to EPWM_setDigitalCompareEventSyncMode() as the *syncMode* parameter.

Enumerator

EPWM_DC_EVENT_INPUT_SYNCED DC input signal is synced with TBCLK.

EPWM_DC_EVENT_INPUT_NOT_SYNCED DC input signal is not synced with TBCLK.

18.2.3.48 enum **EPWM_GlobalLoadTrigger**

Values that can be passed to EPWM_setGlobalLoadTrigger() as the *loadTrigger* parameter.

Enumerator

EPWM_GL_LOAD_PULSE_CNTR_ZERO load when counter is equal to zero

EPWM_GL_LOAD_PULSE_CNTR_PERIOD load when counter is equal to period

EPWM_GL_LOAD_PULSE_CNTR_ZERO_PERIOD load when counter is equal to zero or period

EPWM_GL_LOAD_PULSE_SYNC load on sync event

EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_ZERO load on sync event or when counter is equal to zero

EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_PERIOD load on sync event or when counter is equal to period

EPWM_GL_LOAD_PULSE_SYNC_CNTR_ZERO_PERIOD load on sync event or when counter is equal to period or zero

EPWM_GL_LOAD_PULSE_GLOBAL_FORCE load on global force

18.2.3.49 enum **EPWM_ValleyTriggerSource**

Values that can be passed to EPWM_setValleyTriggerSource() as the *trigger* parameter.

Enumerator

EPWM_VALLEY_TRIGGER_EVENT_SOFTWARE Valley capture trigged by software.

EPWM_VALLEY_TRIGGER_EVENT_CNTR_ZERO Valley capture trigged by when counter is equal to zero.

EPWM_VALLEY_TRIGGER_EVENT_CNTR_PERIOD Valley capture trigged by when counter is equal period.

EPWM_VALLEY_TRIGGER_EVENT_CNTR_ZERO_PERIOD Valley capture trigged when counter is equal to zero or period.

EPWM_VALLEY_TRIGGER_EVENT_DCAEVT1 Valley capture trigged by DCAEVT1 (Digital Compare A event 1)

EPWM_VALLEY_TRIGGER_EVENT_DCAEVT2 Valley capture trigged by DCAEVT2 (Digital Compare A event 2)

EPWM_VALLEY_TRIGGER_EVENT_DCBEVT1 Valley capture trigged by DCBEVT1 (Digital Compare B event 1)
EPWM_VALLEY_TRIGGER_EVENT_DCBEVT2 Valley capture trigged by DCBEVT2 (Digital Compare B event 2)

18.2.3.50 enum **EPWM_ValleyCounterEdge**

Values that can be passed to EPWM_getValleyCountEdgeStatus() as the *edge* parameter.

Enumerator
EPWM_VALLEY_COUNT_START_EDGE Valley count start edge.
EPWM_VALLEY_COUNT_STOP_EDGE Valley count stop edge.

18.2.3.51 enum **EPWM_ValleyDelayMode**

Values that can be passed to EPWM_setValleyDelayValue() as the *delayMode* parameter.

Enumerator
EPWM_VALLEY_DELAY_MODE_SW_DELAY Delay value equals the offset value defines by software.
EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SW_DELAY Delay value equals the sum of the Hardware counter value and the offset value defines by software
EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_1_SW_DELAY Delay value equals the the Hardware counter shifted by (1 + the offset value defines by software)
EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_2_SW_DELAY Delay value equals the the Hardware counter shifted by (2 + the offset value defines by software)
EPWM_VALLEY_DELAY_MODE_VCNT_DELAY_SHIFT_4_SW_DELAY Delay value equals the the Hardware counter shifted by (4 + the offset value defines by software)

18.2.3.52 enum **EPWM_DigitalCompareEdgeFilterMode**

Values that can be passed to EPWM_setDigitalCompareEdgeFilterMode() as the *edgeMode* parameter.

Enumerator
EPWM_DC_EDGEFILT_MODE_RISING Digital Compare Edge filter low to high edge mode
EPWM_DC_EDGEFILT_MODE_FALLING Digital Compare Edge filter high to low edge mode
EPWM_DC_EDGEFILT_MODE_BOTH Digital Compare Edge filter both edges mode

18.2.3.53 enum **EPWM_DigitalCompareEdgeFilterEdgeCount**

Values that can be passed to EPWM_setDigitalCompareEdgeFilterEdgeCount() as the *edgeCount* parameter.

Enumerator
EPWM_DC_EDGEFILT_EDGECNT_0 Digital Compare Edge filter edge count = 0

EPWM_DC_EDGEFILT_EDGECNT_1 Digital Compare Edge filter edge count = 1
EPWM_DC_EDGEFILT_EDGECNT_2 Digital Compare Edge filter edge count = 2
EPWM_DC_EDGEFILT_EDGECNT_3 Digital Compare Edge filter edge count = 3
EPWM_DC_EDGEFILT_EDGECNT_4 Digital Compare Edge filter edge count = 4
EPWM_DC_EDGEFILT_EDGECNT_5 Digital Compare Edge filter edge count = 5
EPWM_DC_EDGEFILT_EDGECNT_6 Digital Compare Edge filter edge count = 6
EPWM_DC_EDGEFILT_EDGECNT_7 Digital Compare Edge filter edge count = 7

18.2.3.54 enum **EPWM_LockRegisterGroup**

Values that can be passed to EPWM_lockRegisters() as the *registerGroup* parameter.

Enumerator

EPWM_REGISTER_GROUP_GLOBAL_LOAD Global load register group.
EPWM_REGISTER_GROUP_TRIP_ZONE Trip zone register group.
EPWM_REGISTER_GROUP_TRIP_ZONE_CLEAR Trip zone clear group.
EPWM_REGISTER_GROUP_DIGITAL_COMPARE Digital compare group.

18.2.4 Function Documentation

18.2.4.1 static void EPWM_setTimeBaseCounter (uint32_t *base*, uint16_t *count*) [inline], [static]

Set the time base count

Parameters

base	is the base address of the EPWM module.
count	is the time base count value.

This function sets the 16 bit counter value of the time base counter.

Returns

None.

18.2.4.2 static void EPWM_setCountModeAfterSync (uint32_t *base*, **EPWM_SyncCountMode** *mode*) [inline], [static]

Set count mode after phase shift sync

Parameters

base	is the base address of the EPWM module.
mode	is the count mode.

This function sets the time base count to count up or down after a new phase value set by the EPWM_setPhaseShift(). The count direction is determined by the variable mode. Valid inputs for mode are:

- EPWM_COUNT_MODE_UP_AFTER_SYNC - Count up after sync

- EPWM_COUNT_MODE_DOWN_AFTER_SYNC - Count down after sync

Returns
None.

References EPWM_COUNT_MODE_UP_AFTER_SYNC.

18.2.4.3 static void EPWM_setClockPrescaler (uint32_t *base,* **EPWM_ClockDivider** *prescaler,* **EPWM_HSClockDivider** *highSpeedPrescaler*) [inline], [static]

Set the time base clock and the high speed time base clock count pre-scaler

Parameters

base	is the base address of the EPWM module.
prescaler	is the time base count pre scale value.
highSpeed-Prescaler	is the high speed time base count pre scale value.

This function sets the pre scaler(divider)value for the time base clock counter and the high speed time base clock counter. Valid values for pre-scaler and highSpeedPrescaler are EPWM_CLOCK_DIVIDER_X, where X is 1, 2, 4, 8, 16, 32, 64 or 128. The actual numerical values for these macros represent values 0, 1...7. The equation for the output clock is: TBCLK = EPWMCLK/(highSpeedPrescaler $*$ pre-scaler)

Note: EPWMCLK is a scaled version of SYSCLK. At reset EPWMCLK is half SYSCLK.

Returns
None.

18.2.4.4 static void EPWM_forceSyncPulse (uint32_t *base*) [inline], [static]

Force a software sync pulse

Parameters

base	is the base address of the EPWM module.

This function causes a single software initiated sync pulse. Make sure the appropriate mode is selected using EPWM_setupSyncOutputMode() before using this function.

Returns
None.

18.2.4.5 static void EPWM_setSyncOutPulseMode (uint32_t *base,* **EPWM_SyncOutPulseMode** *mode*) [inline], [static]

Set up the sync out pulse event

Parameters

base	is the base address of the EPWM module.
mode	is the sync out mode.

This function set the sync out pulse mode. Valid values for mode are:

- EPWM_SYNC_OUT_PULSE_ON_SOFTWARE - sync pulse is generated by software when EPWM_forceSyncPulse() function is called or by EPWMxSYNCI signal.
- EPWM_SYNC_OUT_PULSE_ON_COUNTER_ZERO - sync pulse is generated when time base counter equals zero.
- EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_B - sync pulse is generated when time base counter equals compare B value.
- EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_C - sync pulse is generated when time base counter equals compare C value.
- EPWM_SYNC_OUT_PULSE_ON_COUNTER_COMPARE_D - sync pulse is generated when time base counter equals compare D value.
- EPWM_SYNC_OUT_PULSE_DISABLED - sync pulse is disabled.

Returns

None.

References EPWM_SYNC_OUT_PULSE_DISABLED.

18.2.4.6 static void EPWM_setPeriodLoadMode (uint32_t *base*, **EPWM_PeriodLoadMode** *loadMode*) [inline], [static]

Set PWM period load mode.

Parameters

base	is the base address of the EPWM module.
loadMode	is the PWM period load mode.

This function sets the load mode for the PWM period. If loadMode is set to EPWM_PERIOD_SHADOW_LOAD, a write or read to the TBPRD (PWM Period count register) accesses the shadow register. If loadMode is set to EPWM_PERIOD_DIRECT_LOAD, a write or read to the TBPRD register accesses the register directly.

Returns

None.

References EPWM_PERIOD_SHADOW_LOAD.

18.2.4.7 static void EPWM_enablePhaseShiftLoad (uint32_t *base*) [inline], [static]

Enable phase shift load

Parameters

base	is the base address of the EPWM module.

This function enables loading of phase shift when the appropriate sync event occurs.

Returns
None.

18.2.4.8 static void EPWM_disablePhaseShiftLoad (uint32_t *base*) [inline], [static]

Disable phase shift load

Parameters

base	is the base address of the EPWM module.

This function disables loading of phase shift. occurs.

Returns
None.

18.2.4.9 static void EPWM_setTimeBaseCounterMode (uint32_t *base*, **EPWM_TimeBaseCountMode** *counterMode*) [inline], [static]

Set time base counter mode

Parameters

base	is the base address of the EPWM module.
counterMode	is the time base counter mode.

This function sets up the time base counter mode. Valid values for counterMode are:

- EPWM_COUNTER_MODE_UP - Up - count mode.
- EPWM_COUNTER_MODE_DOWN - Down - count mode.
- EPWM_COUNTER_MODE_UP_DOWN - Up - down - count mode.
- EPWM_COUNTER_MODE_STOP_FREEZE - Stop - Freeze counter.

Returns
None.

18.2.4.10 static void EPWM_selectPeriodLoadEvent (uint32_t *base*, **EPWM_PeriodShadowLoadMode** *shadowLoadMode*) [inline], [static]

Set shadow to active period load on sync mode

Parameters

base	is the base address of the EPWM module.
shadowLoad-Mode	is the shadow to active load mode.

This function sets up the shadow to active Period register load mode with respect to a sync event. Valid values for shadowLoadMode are:

- EPWM_SHADOW_LOAD_MODE_COUNTER_ZERO - shadow to active load occurs when time base counter reaches 0.
- EPWM_SHADOW_LOAD_MODE_COUNTER_SYNC - shadow to active load occurs when time base counter reaches 0 and a SYNC occurs.
- EPWM_SHADOW_LOAD_MODE_SYNC - shadow to active load occurs only when a SYNC occurs.

Returns
None.

18.2.4.11 static void EPWM_enableOneShotSync (uint32_t *base*) `[inline]`, `[static]`

Enable one shot sync mode

Parameters

base	is the base address of the EPWM module.

This function enables one shot sync mode.

Returns
None.

18.2.4.12 static void EPWM_disableOneShotSync (uint32_t *base*) `[inline]`, `[static]`

Disable one shot sync mode

Parameters

base	is the base address of the EPWM module.

This function disables one shot sync mode.

Returns
None.

18.2.4.13 static void EPWM_startOneShotSync (uint32_t *base*) `[inline]`, `[static]`

Start one shot sync mode

Parameters

base	is the base address of the EPWM module.

This function propagates a one shot sync pulse.

Returns

None.

18.2.4.14 static bool EPWM_getTimeBaseCounterOverflowStatus (uint32_t *base*) [inline], [static]

Return time base counter maximum status.

Parameters

base	is the base address of the EPWM module.

This function returns the status of the time base max counter.

Returns

Returns true if the counter has reached 0xFFFF. Returns false if the counter hasn't reached 0xFFFF.

18.2.4.15 static void EPWM_clearTimeBaseCounterOverflowEvent (uint32_t *base*) [inline], [static]

Clear max time base counter event.

Parameters

base	is the base address of the EPWM module.

This function clears the max time base counter latch event. The latch event occurs when the time base counter reaches its maximum value of 0xFFFF.

Returns

None.

18.2.4.16 static bool EPWM_getSyncStatus (uint32_t *base*) [inline], [static]

Return external sync signal status.

Parameters

base	is the base address of the EPWM module.

This function returns the external sync signal status.

Returns

Returns true if if an external sync signal event Returns false if there is no event.

18.2.4.17 static void EPWM_clearSyncEvent (uint32_t *base*) [inline], [static]

Clear external sync signal event.

Parameters

base	is the base address of the EPWM module.

This function clears the external sync signal latch event.

Returns
None.

18.2.4.18 static uint16_t EPWM_getTimeBaseCounterDirection (uint32_t *base*) [inline], [static]

Return time base counter direction.

Parameters

base	is the base address of the EPWM module.

This function returns the direction of the time base counter.

Returns
returns EPWM_TIME_BASE_STATUS_COUNT_UP if the counter is counting up or EPWM_TIME_BASE_STATUS_COUNT_DOWN if the counter is counting down.

18.2.4.19 static void EPWM_setPhaseShift (uint32_t *base*, uint16_t *phaseCount*) [inline], [static]

Sets the phase shift offset counter value.

Parameters

base	is the base address of the EPWM module.
phaseCount	is the phase shift count value.

This function sets the 16 bit time-base counter phase of the ePWM relative to the time-base that is supplying the synchronization input signal. Call the EPWM_enablePhaseShiftLoad() function to enable loading of the phaseCount phase shift value when a sync event occurs.

Returns
None.

18.2.4.20 static void EPWM_setTimeBasePeriod (uint32_t *base*, uint16_t *periodCount*) [inline], [static]

Sets the PWM period count.

Parameters

base	is the base address of the EPWM module.
periodCount	is period count value.

This function sets the period of the PWM count. The value of periodCount is the value written to the register. User should map the desired period or frequency of the waveform into the correct periodCount. Invoke the function EPWM_selectPeriodLoadEvent() with the appropriate parameter to set the load mode of the Period count. periodCount has a maximum valid value of 0xFFFF

Returns
None.

18.2.4.21 static uint16_t EPWM_getTimeBasePeriod (uint32_t *base*) [inline], [static]

Gets the PWM period count.

Parameters

base	is the base address of the EPWM module.

This function gets the period of the PWM count.

Returns
The period count value.

18.2.4.22 static void EPWM_setupEPWMLinks (uint32_t *base*, **EPWM_CurrentLink** *epwmLink*, **EPWM_LinkComponent** *linkComp*) [inline], [static]

Sets the EPWM links.

Parameters

base	is the base address of the EPWM module.
epwmLink	is the ePWM instance to link with.
linkComp	is the ePWM component to link.

This function links the component defined in linkComp in the current ePWM instance with the linkComp component of the ePWM instance defined by epwmLink. A change (a write) in the value of linkComp component of epwmLink instance, causes a change in the current ePWM linkComp component. For example if the current ePWM is ePWM3 and the values of epwmLink and linkComp are EPWM_LINK_WITH_EPWM_1 and EPWM_LINK_COMP_C respectively, then a write to COMPC register in ePWM1, will result in a simultaneous write to COMPC register in ePWM3. Valid values for epwmLink are:

- EPWM_LINK_WITH_EPWM_1 - link current ePWM with ePWM1
- EPWM_LINK_WITH_EPWM_2 - link current ePWM with ePWM2
- EPWM_LINK_WITH_EPWM_3 - link current ePWM with ePWM3
- EPWM_LINK_WITH_EPWM_4 - link current ePWM with ePWM4
- EPWM_LINK_WITH_EPWM_5 - link current ePWM with ePWM5
- EPWM_LINK_WITH_EPWM_6 - link current ePWM with ePWM6
- EPWM_LINK_WITH_EPWM_7 - link current ePWM with ePWM7

- EPWM_LINK_WITH_EPWM_8 - link current ePWM with ePWM8

Valid values for linkComp are:

- EPWM_LINK_TBPRD - link TBPRD:TBPRDHR registers
- EPWM_LINK_COMP_A - link COMPA registers
- EPWM_LINK_COMP_B - link COMPB registers
- EPWM_LINK_COMP_C - link COMPC registers
- EPWM_LINK_COMP_D - link COMPD registers
- EPWM_LINK_GLDCTL2 - link GLDCTL2 registers

Returns
None.

18.2.4.23 static void EPWM_setCounterCompareShadowLoadMode (uint32_t *base*, **EPWM_CounterCompareModule** *compModule*, **EPWM_CounterCompareLoadMode** *loadMode*) [inline], [static]

Sets up the Counter Compare shadow load mode

Parameters

base	is the base address of the EPWM module.
compModule	is the counter compare module.
loadMode	is the shadow to active load mode.

This function enables and sets up the counter compare shadow load mode. Valid values for the variables are:

- compModule
  - EPWM_COUNTER_COMPARE_A - counter compare A.
  - EPWM_COUNTER_COMPARE_B - counter compare B.
  - EPWM_COUNTER_COMPARE_C - counter compare C.
  - EPWM_COUNTER_COMPARE_D - counter compare D.
- loadMode
  - EPWM_COMP_LOAD_ON_CNTR_ZERO - load when counter equals zero
  - EPWM_COMP_LOAD_ON_CNTR_PERIOD - load when counter equals period
  - EPWM_COMP_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period
  - EPWM_COMP_LOAD_FREEZE - Freeze shadow to active load
  - EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO - load when counter equals zero
  - EPWM_COMP_LOAD_ON_SYNC_CNTR_PERIOD -load when counter equals period
  - EPWM_COMP_LOAD_ON_SYNC_CNTR_ZERO_PERIOD - load when counter equals zero or period
  - EPWM_COMP_LOAD_ON_SYNC_ONLY - load on sync only

Returns
None.

References EPWM_COUNTER_COMPARE_A, and EPWM_COUNTER_COMPARE_C.

18.2.4.24 static void EPWM_disableCounterCompareShadowLoadMode (uint32_t *base*, **EPWM_CounterCompareModule** *compModule*) [inline], [static]

Disable Counter Compare shadow load mode

Parameters

base	is the base address of the EPWM module.
compModule	is the counter compare module.

This function disables counter compare shadow load mode. Valid values for the variables are:

- compModule
  - EPWM_COUNTER_COMPARE_A - counter compare A.
  - EPWM_COUNTER_COMPARE_B - counter compare B.
  - EPWM_COUNTER_COMPARE_C - counter compare C.
  - EPWM_COUNTER_COMPARE_D - counter compare D.

Returns

None.

References EPWM_COUNTER_COMPARE_A, and EPWM_COUNTER_COMPARE_C.

18.2.4.25 static void EPWM_setCounterCompareValue (uint32_t *base*, **EPWM_CounterCompareModule** *compModule*, uint16_t *compCount*) [inline], [static]

Set counter compare values.

Parameters

base	is the base address of the EPWM module.
compModule	is the Counter Compare value module.
compCount	is the counter compare count value.

This function sets the counter compare value for counter compare registers. The maximum value for compCount is 0xFFFF. Valid values for compModule are:

- EPWM_COUNTER_COMPARE_A - counter compare A.
- EPWM_COUNTER_COMPARE_B - counter compare B.
- EPWM_COUNTER_COMPARE_C - counter compare C.
- EPWM_COUNTER_COMPARE_D - counter compare D.

Returns

None.

References EPWM_COUNTER_COMPARE_A, and EPWM_COUNTER_COMPARE_B.

18.2.4.26 static uint16_t EPWM_getCounterCompareValue (uint32_t *base*, **EPWM_CounterCompareModule** *compModule*) [inline], [static]

Get counter compare values.

Parameters

base	is the base address of the EPWM module.
compModule	is the Counter Compare value module.

This function gets the counter compare value for counter compare registers. Valid values for compModule are:

- EPWM_COUNTER_COMPARE_A - counter compare A.
- EPWM_COUNTER_COMPARE_B - counter compare B.
- EPWM_COUNTER_COMPARE_C - counter compare C.
- EPWM_COUNTER_COMPARE_D - counter compare D.

Returns
The counter compare count value.

References EPWM_COUNTER_COMPARE_A, and EPWM_COUNTER_COMPARE_B.

18.2.4.27 static bool EPWM_getCounterCompareShadowStatus (uint32_t *base*, **EPWM_CounterCompareModule** *compModule*) [inline], [static]

Return the counter compare shadow register full status.

Parameters

base	is the base address of the EPWM module.
compModule	is the Counter Compare value module.

This function returns the counter Compare shadow register full status flag. Valid values for compModule are:

- EPWM_COUNTER_COMPARE_A - counter compare A.
- EPWM_COUNTER_COMPARE_B - counter compare B.

Returns
Returns true if the shadow register is full. Returns false if the shadow register is not full.

18.2.4.28 static void EPWM_setActionQualifierShadowLoadMode (uint32_t *base*, **EPWM_ActionQualifierModule** *aqModule*, **EPWM_ActionQualifierLoadMode** *loadMode*) [inline], [static]

Sets the Action Qualifier shadow load mode

Parameters

base	is the base address of the EPWM module.
aqModule	is the Action Qualifier module value.

loadMode	is the shadow to active load mode.

This function enables and sets the Action Qualifier shadow load mode. Valid values for the variables are:

- aqModule
  - EPWM_ACTION_QUALIFIER_A - Action Qualifier A.
  - EPWM_ACTION_QUALIFIER_B - Action Qualifier B.
- loadMode
  - EPWM_AQ_LOAD_ON_CNTR_ZERO - load when counter equals zero
  - EPWM_AQ_LOAD_ON_CNTR_PERIOD - load when counter equals period
  - EPWM_AQ_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period
  - EPWM_AQ_LOAD_FREEZE - Freeze shadow to active load
  - EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO - load on sync or when counter equals zero
  - EPWM_AQ_LOAD_ON_SYNC_CNTR_PERIOD - load on sync or when counter equals period
  - EPWM_AQ_LOAD_ON_SYNC_CNTR_ZERO_PERIOD - load on sync or when counter equals zero or period
  - EPWM_AQ_LOAD_ON_SYNC_ONLY - load on sync only

Returns

None.

18.2.4.29 static void EPWM_disableActionQualifierShadowLoadMode (uint32_t *base,* **EPWM_ActionQualifierModule** *aqModule*) [inline], [static]

Disable Action Qualifier shadow load mode

Parameters

base	is the base address of the EPWM module.
aqModule	is the Action Qualifier module value.

This function disables the Action Qualifier shadow load mode. Valid values for the variables are:

- aqModule
  - EPWM_ACTION_QUALIFIER_A - Action Qualifier A.
  - EPWM_ACTION_QUALIFIER_B - Action Qualifier B.

Returns

None.

18.2.4.30 static void EPWM_setActionQualifierT1TriggerSource (uint32_t *base,* **EPWM_ActionQualifierTriggerSource** *trigger*) [inline], [static]

Set up Action qualifier trigger source for event T1

Parameters

base	is the base address of the EPWM module.
trigger	sources for Action Qualifier triggers.

This function sets up the sources for Action Qualifier event T1. Valid values for trigger are:

- EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_1 - Digital compare event A 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_2 - Digital compare event A 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_1 - Digital compare event B 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_2 - Digital compare event B 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_1 - Trip zone 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_2 - Trip zone 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_3 - Trip zone 3
- EPWM_AQ_TRIGGER_EVENT_TRIG_EPWM_SYNCIN - ePWM sync

Returns

None.

18.2.4.31 static void EPWM_setActionQualifierT2TriggerSource (uint32_t *base*, **EPWM_ActionQualifierTriggerSource** *trigger*) [inline], [static]

Set up Action qualifier trigger source for event T2

Parameters

base	is the base address of the EPWM module.
trigger	sources for Action Qualifier triggers.

This function sets up the sources for Action Qualifier event T2. Valid values for trigger are:

- EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_1 - Digital compare event A 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCA_2 - Digital compare event A 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_1 - Digital compare event B 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_DCB_2 - Digital compare event B 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_1 - Trip zone 1
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_2 - Trip zone 2
- EPWM_AQ_TRIGGER_EVENT_TRIG_TZ_3 - Trip zone 3
- EPWM_AQ_TRIGGER_EVENT_TRIG_EPWM_SYNCIN - ePWM sync

Returns

None.

18.2.4.32 static void EPWM_setActionQualifierAction (uint32_t *base*, **EPWM_ActionQualifierOutputModule** *epwmOutput*, **EPWM_ActionQualifierOutput** *output*, **EPWM_ActionQualifierOutputEvent** *event*) [inline], [static]

Set up Action qualifier outputs

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.
output	is the Action Qualifier output.
event	is the event that causes a change in output.

This function sets up the Action Qualifier output on ePWM A or ePWMB, depending on the value of epwmOutput, to a value specified by outPut based on the input events - specified by event. The following are valid values for the parameters.

- epwmOutput
  - EPWM_AQ_OUTPUT_A - ePWMxA output
  - EPWM_AQ_OUTPUT_B - ePWMxB output
- output
  - EPWM_AQ_OUTPUT_NO_CHANGE - No change in the output pins
  - EPWM_AQ_OUTPUT_LOW - Set output pins to low
  - EPWM_AQ_OUTPUT_HIGH - Set output pins to High
  - EPWM_AQ_OUTPUT_TOGGLE - Toggle the output pins
- event
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_ZERO - Time base counter equals zero
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_PERIOD - Time base counter equals period
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPA - Time base counter up equals COMPA
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPA - Time base counter down equals COMPA
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_UP_CMPB - Time base counter up equals COMPB
  - EPWM_AQ_OUTPUT_ON_TIMEBASE_DOWN_CMPB - Time base counter down equals COMPB
  - EPWM_AQ_OUTPUT_ON_T1_COUNT_UP - T1 event on count up
  - EPWM_AQ_OUTPUT_ON_T1_COUNT_DOWN - T1 event on count down
  - EPWM_AQ_OUTPUT_ON_T2_COUNT_UP - T2 event on count up
  - EPWM_AQ_OUTPUT_ON_T2_COUNT_DOWN - T2 event on count down

Returns
None.

18.2.4.33 static void EPWM_setActionQualifierActionComplete (uint32_t *base*, **EPWM_ActionQualifierOutputModule** *epwmOutput*, **EPWM_ActionQualifierEventAction** *action*) [inline], [static]

Set up Action qualifier event outputs

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.
action	is the desired action when the specified event occurs

This function sets up the Action Qualifier output on ePWMA or ePWMB, depending on the value of epwmOutput, to a value specified by action The following are valid values for the parameters.

- epwmOutput
  - EPWM_AQ_OUTPUT_A - ePWMxA output
  - EPWM_AQ_OUTPUT_B - ePWMxB output
- action
  - EPWM_AQ_OUTPUT_NO_CHANGE_ZERO - Time base counter equals zero and no change in output pins
  - EPWM_AQ_OUTPUT_LOW_ZERO - Time base counter equals zero and set output pins to low
  - EPWM_AQ_OUTPUT_HIGH_ZERO - Time base counter equals zero and set output pins to high
  - EPWM_AQ_OUTPUT_TOGGLE_ZERO - Time base counter equals zero and toggle the output pins
  - EPWM_AQ_OUTPUT_NO_CHANGE_PERIOD - Time base counter equals period and no change in output pins
  - EPWM_AQ_OUTPUT_LOW_PERIOD - Time base counter equals period and set output pins to low
  - EPWM_AQ_OUTPUT_HIGH_PERIOD - Time base counter equals period and set output pins to high
  - EPWM_AQ_OUTPUT_TOGGLE_PERIOD - Time base counter equals period and toggle the output pins
  - EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPA - Time base counter up equals COMPA and no change in the output pins
  - EPWM_AQ_OUTPUT_LOW_UP_CMPA - Time base counter up equals COMPA and set output pins low
  - EPWM_AQ_OUTPUT_HIGH_UP_CMPA - Time base counter up equals COMPA and set output pins high
  - EPWM_AQ_OUTPUT_TOGGLE_UP_CMPA - Time base counter up equals COMPA and toggle output pins
  - EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPA- Time base counter down equals COMPA and no change in the output pins
  - EPWM_AQ_OUTPUT_LOW_DOWN_CMPA - Time base counter down equals COMPA and set output pins low
  - EPWM_AQ_OUTPUT_HIGH_DOWN_CMPA - Time base counter down equals COMPA and set output pins high
  - EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPA - Time base counter down equals COMPA and toggle output pins
  - EPWM_AQ_OUTPUT_NO_CHANGE_UP_CMPB - Time base counter up equals COMPB and no change in the output pins
  - EPWM_AQ_OUTPUT_LOW_UP_CMPB - Time base counter up equals COMPB and set output pins low

- EPWM_AQ_OUTPUT_HIGH_UP_CMPB - Time base counter up equals COMPB and set output pins high
- EPWM_AQ_OUTPUT_TOGGLE_UP_CMPB - Time base counter up equals COMPB and toggle output pins
- EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_CMPB- Time base counter down equals COMPB and no change in the output pins
- EPWM_AQ_OUTPUT_LOW_DOWN_CMPB - Time base counter down equals COMPB and set output pins low
- EPWM_AQ_OUTPUT_HIGH_DOWN_CMPB - Time base counter down equals COMPB and set output pins high
- EPWM_AQ_OUTPUT_TOGGLE_DOWN_CMPB - Time base counter down equals COMPB and toggle output pins

Returns
None.

18.2.4.34 static void EPWM_setAdditionalActionQualifierActionComplete (uint32_t *base*, **EPWM_ActionQualifierOutputModule** *epwmOutput*, **EPWM_AdditionalActionQualifierEventAction** *action*) [inline], [static]

Set up Additional action qualifier event outputs

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.
action	is the desired action when the specified event occurs

This function sets up the Additional Action Qualifier output on ePWMA or ePWMB depending on the value of epwmOutput, to a value specified by action The following are valid values for the parameters.

- epwmOutput
  - EPWM_AQ_OUTPUT_A - ePWMxA output
  - EPWM_AQ_OUTPUT_B - ePWMxB output
- action
  - EPWM_AQ_OUTPUT_NO_CHANGE_UP_TI - T1 event on count up and no change in output pins
  - EPWM_AQ_OUTPUT_LOW_UP_TI - T1 event on count up and set output pins to low
  - EPWM_AQ_OUTPUT_HIGH_UP_TI - T1 event on count up and set output pins to high
  - EPWM_AQ_OUTPUT_TOGGLE_UP_TI - T1 event on count up and toggle the output pins
  - EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_TI- T1 event on count down and no change in output pins
  - EPWM_AQ_OUTPUT_LOW_DOWN_TI - T1 event on count down and set output pins to low
  - EPWM_AQ_OUTPUT_HIGH_DOWN_TI - T1 event on count down and set output pins to high

- EPWM_AQ_OUTPUT_TOGGLE_DOWN_TI - T1 event on count down and toggle the output pins
- EPWM_AQ_OUTPUT_NO_CHANGE_UP_T2 - T2 event on count up and no change in output pins
- EPWM_AQ_OUTPUT_LOW_UP_T2 - T2 event on count up and set output pins to low
- EPWM_AQ_OUTPUT_HIGH_UP_T2 - T2 event on count up and set output pins to high
- EPWM_AQ_OUTPUT_TOGGLE_UP_T2 - T2 event on count up and toggle the output pins
- EPWM_AQ_OUTPUT_NO_CHANGE_DOWN_T2- T2 event on count down and no change in output pins
- EPWM_AQ_OUTPUT_LOW_DOWN_T2 - T2 event on count down and set output pins to low
- EPWM_AQ_OUTPUT_HIGH_DOWN_T2 - T2 event on count down and set output pins to high
- EPWM_AQ_OUTPUT_TOGGLE_DOWN_T2 - T2 event on count down and toggle the output pins

Returns

None.

18.2.4.35 static void EPWM_setActionQualifierContSWForceShadowMode (uint32_t *base*, **EPWM_ActionQualifierContForce** *mode*) [inline], [static]

Sets up Action qualifier continuous software load mode.

Parameters

base	is the base address of the EPWM module.
mode	is the mode for shadow to active load mode.

This function sets up the AQCFRSC register load mode for continuous software force reload mode. The software force actions are determined by the EPWM_setActionQualifierContSWForceAction() function. Valid values for mode are:

- EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO - shadow mode load when counter equals zero
- EPWM_AQ_SW_SH_LOAD_ON_CNTR_PERIOD - shadow mode load when counter equals period
- EPWM_AQ_SW_SH_LOAD_ON_CNTR_ZERO_PERIOD - shadow mode load when counter equals zero or period
- EPWM_AQ_SW_IMMEDIATE_LOAD - immediate mode load only

Returns

None.

18.2.4.36 static void EPWM_setActionQualifierContSWForceAction (uint32_t *base*, **EPWM_ActionQualifierOutputModule** *epwmOutput*, **EPWM_ActionQualifierSWOutput** *output*) [inline], [static]

Triggers a continuous software forced event.

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.
output	is the Action Qualifier output.

This function triggers a continuous software forced Action Qualifier output on ePWM A or B based on the value of epwmOutput. Valid values for the parameters are:

- epwmOutput
  - EPWM_AQ_OUTPUT_A - ePWMxA output
  - EPWM_AQ_OUTPUT_B - ePWMxB output
- output
  - EPWM_AQ_SW_DISABLED - Software forcing disabled.
  - EPWM_AQ_OUTPUT_LOW - Set output pins to low
  - EPWM_AQ_OUTPUT_HIGH - Set output pins to High

Returns

None.

References EPWM_AQ_OUTPUT_A.

18.2.4.37 static void EPWM_setActionQualifierSWAction (uint32_t *base,* **EPWM_ActionQualifierOutputModule** *epwmOutput,* **EPWM_ActionQualifierOutput** *output*) [inline], [static]

Set up one time software forced Action qualifier outputs

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.
output	is the Action Qualifier output.

This function sets up the one time software forced Action Qualifier output on ePWM A or ePWMB, depending on the value of epwmOutput to a value specified by outPut. The following are valid values for the parameters.

- epwmOutput
  - EPWM_AQ_OUTPUT_A - ePWMxA output
  - EPWM_AQ_OUTPUT_B - ePWMxB output
- output
  - EPWM_AQ_OUTPUT_NO_CHANGE - No change in the output pins
  - EPWM_AQ_OUTPUT_LOW - Set output pins to low
  - EPWM_AQ_OUTPUT_HIGH - Set output pins to High
  - EPWM_AQ_OUTPUT_TOGGLE - Toggle the output pins

Returns

None.

References EPWM_AQ_OUTPUT_A.

18.2.4.38 static void EPWM_forceActionQualifierSWAction (uint32_t *base*, **EPWM_ActionQualifierOutputModule** *epwmOutput*) [inline], [static]

Triggers a one time software forced event on Action qualifier

Parameters

base	is the base address of the EPWM module.
epwmOutput	is the ePWM pin type.

This function triggers a one time software forced Action Qualifier event on ePWM A or B based on the value of epwmOutput. Valid values for epwmOutput are:

- EPWM_AQ_OUTPUT_A - ePWMxA output
- EPWM_AQ_OUTPUT_B - ePWMxB output

Returns

None.

References EPWM_AQ_OUTPUT_A.

18.2.4.39 static void EPWM_setDeadBandOutputSwapMode (uint32_t *base*, **EPWM_DeadBandOutput** *output*, bool *enableSwapMode*) [inline], [static]

Sets Dead Band signal output swap mode.

Parameters

base	is the base address of the EPWM module.
output	is the ePWM Dead Band output.
enableSwapMode	is the output swap mode.

This function sets up the output signal swap mode. For example if the output variable is set to EPWM_DB_OUTPUT_A and enableSwapMode is true, then the ePWM A output gets its signal from the ePWM B signal path. Valid values for the input variables are: output

- EPWM_DB_OUTPUT_A - ePWM output A
- EPWM_DB_OUTPUT_B - ePWM output B enableSwapMode
- true - the output is swapped
- false - the output and the signal path are the same.

Returns

None.

18.2.4.40 static void EPWM_setDeadBandDelayMode (uint32_t *base*, **EPWM_DeadBandDelayMode** *delayMode*, bool *enableDelayMode*) [inline], [static]

Sets Dead Band signal output mode.

Parameters

base	is the base address of the EPWM module.
delayMode	is the Dead Band delay type.
enableDelay-Mode	is the dead band delay mode.

This function sets up the dead band delay mode. The delayMode variable determines if the applied delay is Rising Edge or Falling Edge. The enableDelayMode determines if a dead band delay should be applied. Valid values for the variables are: delayMode

- EPWM_DB_RED - Rising Edge delay
- EPWM_DB_FED - Falling Edge delay enableDelayMode
- true - Falling edge or Rising edge delay is applied.
- false - Dead Band delay is bypassed.

Returns

None.

18.2.4.41 static void EPWM_setDeadBandDelayPolarity (uint32_t *base,* **EPWM_DeadBandDelayMode** *delayMode,* **EPWM_DeadBandPolarity** *polarity*) [inline], [static]

Sets Dead Band delay polarity.

Parameters

base	is the base address of the EPWM module.
delayMode	is the Dead Band delay type.
polarity	is the polarity of the delayed signal.

This function sets up the polarity as determined by the variable polarity of the Falling Edge or Rising Edge delay depending on the value of delayMode. Valid values for the variables are: delayMode

- EPWM_DB_RED - Rising Edge delay
- EPWM_DB_FED - Falling Edge delay polarity
- EPWM_DB_POLARITY_ACTIVE_HIGH - polarity is not inverted.
- EPWM_DB_POLARITY_ACTIVE_LOW - polarity is inverted.

Returns

None.

18.2.4.42 static void EPWM_setRisingEdgeDeadBandDelayInput (uint32_t *base,* uint16_t *input*) [inline], [static]

Sets Rising Edge Dead Band delay input.

Parameters

base	is the base address of the EPWM module.
input	is the input signal to the dead band.

This function sets up the rising Edge delay input signal. Valid values for input are:

- EPWM_DB_INPUT_EPWMA - Input signal is ePWMA(Valid for both Falling Edge and Rising Edge)
- EPWM_DB_INPUT_EPWMB - Input signal is ePWMA(Valid for both Falling Edge and Rising Edge)

Returns

None.

References EPWM_DB_INPUT_EPWMA, and EPWM_DB_INPUT_EPWMB.

18.2.4.43 static void EPWM_setFallingEdgeDeadBandDelayInput (uint32_t *base*, uint16_t *input*) [inline], [static]

Sets Dead Band delay input.

Parameters

base	is the base address of the EPWM module.
input	is the input signal to the dead band.

This function sets up the rising Edge delay input signal. Valid values for input are:

- EPWM_DB_INPUT_EPWMA - Input signal is ePWMA(Valid for both Falling Edge and Rising Edge)
- EPWM_DB_INPUT_EPWMB - Input signal is ePWMA(Valid for both Falling Edge and Rising Edge)
- EPWM_DB_INPUT_DB_RED - Input signal is the output of Rising Edge delay. (Valid only for Falling Edge delay)

Returns

None.

References EPWM_DB_INPUT_DB_RED, EPWM_DB_INPUT_EPWMA, and EPWM_DB_INPUT_EPWMB.

18.2.4.44 static void EPWM_setDeadBandControlShadowLoadMode (uint32_t *base*, **EPWM_DeadBandControlLoadMode** *loadMode*) [inline], [static]

Set the Dead Band control shadow load mode.

Parameters

base	is the base address of the EPWM module.
loadMode	is the shadow to active load mode.

This function enables and sets the Dead Band control register shadow load mode. Valid values for the parameters are: loadMode

- EPWM_DB_LOAD_ON_CNTR_ZERO - load when counter equals zero.
- EPWM_DB_LOAD_ON_CNTR_PERIOD - load when counter equals period.
- EPWM_DB_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period.
- EPWM_DB_LOAD_FREEZE - Freeze shadow to active load.

Returns
None.

18.2.4.45 static void EPWM_disableDeadBandControlShadowLoadMode (uint32_t *base*) [inline], [static]

Disable Dead Band control shadow load mode.

Parameters

base	is the base address of the EPWM module.

This function disables the Dead Band control register shadow load mode.

Returns
None.

18.2.4.46 static void EPWM_setRisingEdgeDelayCountShadowLoadMode (uint32_t *base*, **EPWM_RisingEdgeDelayLoadMode** *loadMode*) [inline], [static]

Set the RED (Rising Edge Delay) shadow load mode.

Parameters

base	is the base address of the EPWM module.
loadMode	is the shadow to active load event.

This function sets the Rising Edge Delay register shadow load mode. Valid values for the parameters are: loadMode

- EPWM_RED_LOAD_ON_CNTR_ZERO - load when counter equals zero.
- EPWM_RED_LOAD_ON_CNTR_PERIOD - load when counter equals period.
- EPWM_RED_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period.
- EPWM_RED_LOAD_FREEZE - Freeze shadow to active load.

Returns
None.

18.2.4.47 static void EPWM_disableRisingEdgeDelayCountShadowLoadMode (uint32_t *base*) [inline], [static]

Disable the RED (Rising Edge Delay) shadow load mode.

Parameters

base	is the base address of the EPWM module.

This function disables the Rising Edge Delay register shadow load mode.

Returns

None.

18.2.4.48 static void EPWM_setFallingEdgeDelayCountShadowLoadMode (uint32_t *base*, **EPWM_FallingEdgeDelayLoadMode** *loadMode*) [inline], [static]

Set the FED (Falling Edge Delay) shadow load mode.

Parameters

base	is the base address of the EPWM module.
loadMode	is the shadow to active load event.

This function enables and sets the Falling Edge Delay register shadow load mode. Valid values for the parameters are: loadMode

- EPWM_FED_LOAD_ON_CNTR_ZERO - load when counter equals zero.
- EPWM_FED_LOAD_ON_CNTR_PERIOD - load when counter equals period.
- EPWM_FED_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period.
- EPWM_FED_LOAD_FREEZE - Freeze shadow to active load.

Returns

None.

18.2.4.49 static void EPWM_disableFallingEdgeDelayCountShadowLoadMode (uint32_t *base*) [inline], [static]

Disables the FED (Falling Edge Delay) shadow load mode.

Parameters

base	is the base address of the EPWM module.

This function disables the Falling Edge Delay register shadow load mode. Valid values for the parameters are:

Returns

None.

18.2.4.50 static void EPWM_setDeadBandCounterClock (uint32_t *base*, **EPWM_DeadBandClockMode** *clockMode*) [inline], [static]

Sets Dead Band Counter clock rate.

Parameters

base	is the base address of the EPWM module.
clockMode	is the Dead Band counter clock mode.

This function sets up the Dead Band counter clock rate with respect to TBCLK (ePWM time base counter). Valid values for clockMode are:

- EPWM_DB_COUNTER_CLOCK_FULL_CYCLE -Dead band counter runs at TBCLK (ePWM Time Base Counter) rate.
- EPWM_DB_COUNTER_CLOCK_HALF_CYCLE -Dead band counter runs at 2∗TBCLK (twice ePWM Time Base Counter)rate.

Returns
None.

18.2.4.51 static void EPWM_setRisingEdgeDelayCount (uint32_t *base,* uint16_t *redCount*) [inline], [static]

Set ePWM RED count

Parameters

base	is the base address of the EPWM module.
redCount	is the RED(Rising Edge Delay) count.

This function sets the RED (Rising Edge Delay) count value. The value of redCount should be less than 0x4000U.

Returns
None.

18.2.4.52 static void EPWM_setFallingEdgeDelayCount (uint32_t *base,* uint16_t *fedCount*) [inline], [static]

Set ePWM FED count

Parameters

base	is the base address of the EPWM module.
fedCount	is the FED(Falling Edge Delay) count.

This function sets the FED (Falling Edge Delay) count value. The value of fedCount should be less than 0x4000U.

Returns
None.

18.2.4.53 static void EPWM_enableChopper (uint32_t *base*) [inline], [static]

Enable chopper mode

Parameters

base	is the base address of the EPWM module.

This function enables ePWM chopper module.

Returns
None.

18.2.4.54 static void EPWM_disableChopper (uint32_t *base*) [inline], [static]

Disable chopper mode

Parameters

base	is the base address of the EPWM module.

This function disables ePWM chopper module.

Returns
None.

18.2.4.55 static void EPWM_setChopperDutyCycle (uint32_t *base,* uint16_t *dutyCycleCount*) [inline], [static]

Set chopper duty cycle.

Parameters

base	is the base address of the EPWM module.
dutyCycleCount	is the chopping clock duty cycle count.

This function sets the chopping clock duty cycle. The value of dutyCycleCount should be less than 7. The dutyCycleCount value is converted to the actual chopper duty cycle value base on the following equation: chopper duty cycle = (dutyCycleCount + 1) / 8

Returns
None.

18.2.4.56 static void EPWM_setChopperFreq (uint32_t *base,* uint16_t *freqDiv*) [inline], [static]

Set chopper clock frequency scaler.

Parameters

base	is the base address of the EPWM module.
freqDiv	is the chopping clock frequency divider.

This function sets the scaler for the chopping clock frequency. The value of freqDiv should be less than 8. The chopping clock frequency is altered based on the following equation. chopper clock frequency = SYSCLKOUT / (1 + freqDiv)

Returns
None.

18.2.4.57 static void EPWM_setChopperFirstPulseWidth (uint32_t *base,* uint16_t *firstPulseWidth*) [inline], [static]

Set chopper clock frequency scaler.

Parameters

base	is the base address of the EPWM module.
firstPulseWidth	is the width of the first pulse.

This function sets the first pulse width of chopper output waveform. The value of firstPulseWidth should be less than 0x10. The value of the first pulse width in seconds is given using the following equation: first pulse width = 1 / (((firstPulseWidth + 1) $*$ SYSCLKOUT)/8)

Returns
None.

18.2.4.58 static void EPWM_enableTripZoneSignals (uint32_t *base,* uint16_t *tzSignal*) [inline], [static]

Enables Trip Zone signal.

Parameters

base	is the base address of the EPWM module.
tzSignal	is the Trip Zone signal.

This function enables the Trip Zone signals specified by tzSignal as a source for the Trip Zone module. Valid values for tzSignal are:

- EPWM_TZ_SIGNAL_CBC1 - TZ1 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC2 - TZ2 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC3 - TZ3 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC4 - TZ4 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC5 - TZ5 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC6 - TZ6 Cycle By Cycle
- EPWM_TZ_SIGNAL_DCAEVT2 - DCAEVT2 Cycle By Cycle
- EPWM_TZ_SIGNAL_DCBEVT2 - DCBEVT2 Cycle By Cycle
- EPWM_TZ_SIGNAL_OSHT1 - One-shot TZ1
- EPWM_TZ_SIGNAL_OSHT2 - One-shot TZ2
- EPWM_TZ_SIGNAL_OSHT3 - One-shot TZ3
- EPWM_TZ_SIGNAL_OSHT4 - One-shot TZ4
- EPWM_TZ_SIGNAL_OSHT5 - One-shot TZ5
- EPWM_TZ_SIGNAL_OSHT6 - One-shot TZ6
- EPWM_TZ_SIGNAL_DCAEVT1 - One-shot DCAEVT1

- EPWM_TZ_SIGNAL_DCBEVT1 - One-shot DCBEVT1

note: A logical OR of the valid values can be passed as the tzSignal parameter.

Returns
None.

18.2.4.59 static void EPWM_disableTripZoneSignals (uint32_t *base*, uint16_t *tzSignal*) [inline], [static]

Disables Trip Zone signal.

Parameters

base	is the base address of the EPWM module.
tzSignal	is the Trip Zone signal.

This function disables the Trip Zone signal specified by tzSignal as a source for the Trip Zone module. Valid values for tzSignal are:

- EPWM_TZ_SIGNAL_CBC1 - TZ1 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC2 - TZ2 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC3 - TZ3 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC4 - TZ4 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC5 - TZ5 Cycle By Cycle
- EPWM_TZ_SIGNAL_CBC6 - TZ6 Cycle By Cycle
- EPWM_TZ_SIGNAL_DCAEVT2 - DCAEVT2 Cycle By Cycle
- EPWM_TZ_SIGNAL_DCBEVT2 - DCBEVT2 Cycle By Cycle
- EPWM_TZ_SIGNAL_OSHT1 - One-shot TZ1
- EPWM_TZ_SIGNAL_OSHT2 - One-shot TZ2
- EPWM_TZ_SIGNAL_OSHT3 - One-shot TZ3
- EPWM_TZ_SIGNAL_OSHT4 - One-shot TZ4
- EPWM_TZ_SIGNAL_OSHT5 - One-shot TZ5
- EPWM_TZ_SIGNAL_OSHT6 - One-shot TZ6
- EPWM_TZ_SIGNAL_DCAEVT1 - One-shot DCAEVT1
- EPWM_TZ_SIGNAL_DCBEVT1 - One-shot DCBEVT1

note: A logical OR of the valid values can be passed as the tzSignal parameter.

Returns
None.

18.2.4.60 static void EPWM_setTripZoneDigitalCompareEventCondition (uint32_t *base*, **EPWM_TripZoneDigitalCompareOutput** *dcType*, **EPWM_TripZoneDigitalCompareOutputEvent** *dcEvent*) [inline], [static]

Set Digital compare conditions that cause Trip Zone event.

Parameters

base	is the base address of the EPWM module.
dcType	is the Digital compare output type.
dcEvent	is the Digital Compare output event.

This function sets up the Digital Compare output Trip Zone event sources. The dcType variable specifies the event source to be whether Digital Compare output A or Digital Compare output B. The dcEvent parameter specifies the event that causes Trip Zone. Valid values for the parameters are: dcType

- EPWM_TZ_DC_OUTPUT_A1 - Digital Compare output 1 A
- EPWM_TZ_DC_OUTPUT_A2 - Digital Compare output 2 A
- EPWM_TZ_DC_OUTPUT_B1 - Digital Compare output 1 B
- EPWM_TZ_DC_OUTPUT_B2 - Digital Compare output 2 B dcEvent
- EPWM_TZ_EVENT_DC_DISABLED - Event Trigger is disabled
- EPWM_TZ_EVENT_DCXH_LOW - Trigger event when DCxH low
- EPWM_TZ_EVENT_DCXH_HIGH - Trigger event when DCxH high
- EPWM_TZ_EVENT_DCXL_LOW - Trigger event when DCxL low
- EPWM_TZ_EVENT_DCXL_HIGH - Trigger event when DCxL high
- EPWM_TZ_EVENT_DCXL_HIGH_DCXH_LOW - Trigger event when DCxL high DCxH low

Note

x in DCxH/DCxL represents DCAH/DCAL or DCBH/DCBL

Returns

None.

18.2.4.61 static void EPWM_enableTripZoneAdvAction (uint32_t *base*) `[inline]`, `[static]`

Enable advanced Trip Zone event Action.

Parameters

base	is the base address of the EPWM module.

This function enables the advanced actions of the Trip Zone events. The advanced features combine the trip zone events with the direction of the counter.

Returns

None.

18.2.4.62 static void EPWM_disableTripZoneAdvAction (uint32_t *base*) `[inline]`, `[static]`

Disable advanced Trip Zone event Action.

Parameters

base	is the base address of the EPWM module.

This function disables the advanced actions of the Trip Zone events.

Returns

None.

18.2.4.63 static void EPWM_setTripZoneAction (uint32_t *base*, **EPWM_TripZoneEvent** *tzEvent*, **EPWM_TripZoneAction** *tzAction*) [inline], [static]

Set Trip Zone Action.

Parameters

base	is the base address of the EPWM module.
tzEvent	is the Trip Zone event type.
tzAction	is the Trip zone Action.

This function sets the Trip Zone Action to be taken when a Trip Zone event occurs. Valid values for the parameters are: tzEvent

- EPWM_TZ_ACTION_EVENT_DCBEVT2 - DCBEVT2 (Digital Compare B event 2)
- EPWM_TZ_ACTION_EVENT_DCBEVT1 - DCBEVT1 (Digital Compare B event 1)
- EPWM_TZ_ACTION_EVENT_DCAEVT2 - DCAEVT2 (Digital Compare A event 2)
- EPWM_TZ_ACTION_EVENT_DCAEVT1 - DCAEVT1 (Digital Compare A event 1)
- EPWM_TZ_ACTION_EVENT_TZB - TZ1 - TZ6, DCBEVT2, DCBEVT1
- EPWM_TZ_ACTION_EVENT_TZA - TZ1 - TZ6, DCAEVT2, DCAEVT1 tzAction
- EPWM_TZ_ACTION_HIGH_Z - high impedance output
- EPWM_TZ_ACTION_HIGH - high output
- EPWM_TZ_ACTION_LOW - low low
- EPWM_TZ_ACTION_DISABLE - disable action

Note

Disable the advanced Trip Zone event using EPWM_disableTripZoneAdvAction() before calling this function.
This function operates on both ePWMA and ePWMB depending on the tzEvent parameter.

Returns

None.

18.2.4.64 static void EPWM_setTripZoneAdvAction (uint32_t *base*, **EPWM_TripZoneAdvancedEvent** *tzAdvEvent*, **EPWM_TripZoneAdvancedAction** *tzAdvAction*) [inline], [static]

Set Advanced Trip Zone Action.

Parameters

base	is the base address of the EPWM module.
tzAdvEvent	is the Trip Zone event type.
tzAdvAction	is the Trip zone Action.

This function sets the Advanced Trip Zone Action to be taken when an advanced Trip Zone event occurs.

Valid values for the parameters are: tzAdvEvent

- EPWM_TZ_ADV_ACTION_EVENT_TZB_D - TZ1 - TZ6, DCBEVT2, DCBEVT1 while counting down
- EPWM_TZ_ADV_ACTION_EVENT_TZB_U - TZ1 - TZ6, DCBEVT2, DCBEVT1 while counting up
- EPWM_TZ_ADV_ACTION_EVENT_TZA_D - TZ1 - TZ6, DCAEVT2, DCAEVT1 while counting down
- EPWM_TZ_ADV_ACTION_EVENT_TZA_U - TZ1 - TZ6, DCAEVT2, DCAEVT1 while counting up tzAdvAction
- EPWM_TZ_ADV_ACTION_HIGH_Z - high impedance output
- EPWM_TZ_ADV_ACTION_HIGH - high voltage state
- EPWM_TZ_ADV_ACTION_LOW - low voltage state
- EPWM_TZ_ADV_ACTION_TOGGLE - Toggle output
- EPWM_TZ_ADV_ACTION_DISABLE - disable action

Note

This function enables the advanced Trip Zone event.
This function operates on both ePWMA and ePWMB depending on the tzAdvEvent parameter.
Advanced Trip Zone events take into consideration the direction of the counter in addition to Trip Zone events.

Returns

None.

18.2.4.65 static void EPWM_setTripZoneAdvDigitalCompareActionA (uint32_t *base*, **EPWM_TripZoneAdvDigitalCompareEvent** *tzAdvDCEvent*, **EPWM_TripZoneAdvancedAction** *tzAdvDCAction*) [inline], [static]

Set Advanced Digital Compare Trip Zone Action on ePWMA.

Parameters

base	is the base address of the EPWM module.
tzAdvDCEvent	is the Digital Compare Trip Zone event type.

tzAdvDCAction	is the Digital Compare Trip zone Action.

This function sets the Digital Compare (DC) Advanced Trip Zone Action to be taken on ePWMA when an advanced Digital Compare Trip Zone A event occurs. Valid values for the parameters are: tzAdvDCEvent

- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_D - Digital Compare event A2 while counting down
- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_U - Digital Compare event A2 while counting up
- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_D - Digital Compare event A1 while counting down
- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_U - Digital Compare event A1 while counting up tzAdvDCAction
- EPWM_TZ_ADV_ACTION_HIGH_Z - high impedance output
- EPWM_TZ_ADV_ACTION_HIGH - high voltage state
- EPWM_TZ_ADV_ACTION_LOW - low voltage state
- EPWM_TZ_ADV_ACTION_TOGGLE - Toggle output
- EPWM_TZ_ADV_ACTION_DISABLE - disable action

Note

This function enables the advanced Trip Zone event.
Advanced Trip Zone events take into consideration the direction of the counter in addition to Digital Compare Trip Zone events.

Returns

None.

18.2.4.66 static void EPWM_setTripZoneAdvDigitalCompareActionB (uint32_t *base*, **EPWM_TripZoneAdvDigitalCompareEvent** *tzAdvDCEvent*, **EPWM_TripZoneAdvancedAction** *tzAdvDCAction*) [inline], [static]

Set Advanced Digital Compare Trip Zone Action on ePWMB.

Parameters

base	is the base address of the EPWM module.
tzAdvDCEvent	is the Digital Compare Trip Zone event type.
tzAdvDCAction	is the Digital Compare Trip zone Action.

This function sets the Digital Compare (DC) Advanced Trip Zone Action to be taken on ePWMB when an advanced Digital Compare Trip Zone B event occurs. Valid values for the parameters are: tzAdvDCEvent

- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_D - Digital Compare event B2 while counting down
- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT2_U - Digital Compare event B2 while counting up
- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_D - Digital Compare event B1 while counting down

- EPWM_TZ_ADV_ACTION_EVENT_DCxEVT1_U - Digital Compare event B1 while counting up tzAdvDCAction
- EPWM_TZ_ADV_ACTION_HIGH_Z - high impedance output
- EPWM_TZ_ADV_ACTION_HIGH - high voltage state
- EPWM_TZ_ADV_ACTION_LOW - low voltage state
- EPWM_TZ_ADV_ACTION_TOGGLE - Toggle output
- EPWM_TZ_ADV_ACTION_DISABLE - disable action

Note

This function enables the advanced Trip Zone event.
Advanced Trip Zone events take into consideration the direction of the counter in addition to Digital Compare Trip Zone events.

Returns

None.

18.2.4.67 static void EPWM_enableTripZoneInterrupt (uint32_t *base,* uint16_t *tzInterrupt*) [inline], [static]

Enable Trip Zone interrupts.

Parameters

base	is the base address of the EPWM module.
tzInterrupt	is the Trip Zone interrupt.

This function enables the Trip Zone interrupts. Valid values for tzInterrupt are:

- EPWM_TZ_INTERRUPT_CBC - Trip Zones Cycle By Cycle interrupt
- EPWM_TZ_INTERRUPT_OST - Trip Zones One Shot interrupt
- EPWM_TZ_INTERRUPT_DCAEVT1 - Digital Compare A Event 1 interrupt
- EPWM_TZ_INTERRUPT_DCAEVT2 - Digital Compare A Event 2 interrupt
- EPWM_TZ_INTERRUPT_DCBEVT1 - Digital Compare B Event 1 interrupt
- EPWM_TZ_INTERRUPT_DCBEVT2 - Digital Compare B Event 2 interrupt

note: A logical OR of the valid values can be passed as the tzInterrupt parameter.

Returns

None.

18.2.4.68 static void EPWM_disableTripZoneInterrupt (uint32_t *base,* uint16_t *tzInterrupt*) [inline], [static]

Disable Trip Zone interrupts.

Parameters

base	is the base address of the EPWM module.
tzInterrupt	is the Trip Zone interrupt.

This function disables the Trip Zone interrupts. Valid values for tzInterrupt are:

- EPWM_TZ_INTERRUPT_CBC - Trip Zones Cycle By Cycle interrupt
- EPWM_TZ_INTERRUPT_OST - Trip Zones One Shot interrupt
- EPWM_TZ_INTERRUPT_DCAEVT1 - Digital Compare A Event 1 interrupt
- EPWM_TZ_INTERRUPT_DCAEVT2 - Digital Compare A Event 2 interrupt
- EPWM_TZ_INTERRUPT_DCBEVT1 - Digital Compare B Event 1 interrupt
- EPWM_TZ_INTERRUPT_DCBEVT2 - Digital Compare B Event 2 interrupt

note: A logical OR of the valid values can be passed as the tzInterrupt parameter.

Returns
None.

18.2.4.69 static uint16_t EPWM_getTripZoneFlagStatus (uint32_t *base*) [inline], [static]

Gets the Trip Zone status flag

Parameters

base	is the base address of the EPWM module.

This function returns the Trip Zone status flag.

Returns
The function returns the following or the bitwise OR value of the following values.

- EPWM_TZ_INTERRUPT - Trip Zone interrupt was generated due to the following TZ events.
- EPWM_TZ_FLAG_CBC - Trip Zones Cycle By Cycle event status flag
- EPWM_TZ_FLAG_OST - Trip Zones One Shot event status flag
- EPWM_TZ_FLAG_DCAEVT1 - Digital Compare A Event 1 status flag
- EPWM_TZ_FLAG_DCAEVT2 - Digital Compare A Event 2 status flag
- EPWM_TZ_FLAG_DCBEVT1 - Digital Compare B Event 1 status flag
- EPWM_TZ_FLAG_DCBEVT2 - Digital Compare B Event 2 status flag

18.2.4.70 static uint16_t EPWM_getCycleByCycleTripZoneFlagStatus (uint32_t *base*) [inline], [static]

Gets the Trip Zone Cycle by Cycle flag status

Parameters

base	is the base address of the EPWM module.

This function returns the specific Cycle by Cycle Trip Zone flag status.

Returns

The function returns the following values.

- EPWM_TZ_CBC_FLAG_1 - CBC 1 status flag
- EPWM_TZ_CBC_FLAG_2 - CBC 2 status flag
- EPWM_TZ_CBC_FLAG_3 - CBC 3 status flag
- EPWM_TZ_CBC_FLAG_4 - CBC 4 status flag
- EPWM_TZ_CBC_FLAG_5 - CBC 5 status flag
- EPWM_TZ_CBC_FLAG_6 - CBC 6 status flag
- EPWM_TZ_CBC_FLAG_DCAEVT2 - CBC status flag for Digital compare event A2
- EPWM_TZ_CBC_FLAG_DCBEVT2 - CBC status flag for Digital compare event B2

18.2.4.71 static uint16_t EPWM_getOneShotTripZoneFlagStatus (uint32_t *base*) [inline], [static]

Gets the Trip Zone One Shot flag status

Parameters

base	is the base address of the EPWM module.

This function returns the specific One Shot Trip Zone flag status.

Returns

The function returns the bitwise OR of the following flags.

- EPWM_TZ_OST_FLAG_OST1 - OST status flag for OST1
- EPWM_TZ_OST_FLAG_OST2 - OST status flag for OST2
- EPWM_TZ_OST_FLAG_OST3 - OST status flag for OST3
- EPWM_TZ_OST_FLAG_OST4 - OST status flag for OST4
- EPWM_TZ_OST_FLAG_OST5 - OST status flag for OST5
- EPWM_TZ_OST_FLAG_OST6 - OST status flag for OST6
- EPWM_TZ_OST_FLAG_DCAEVT1 - OST status flag for Digital compare event A1
- EPWM_TZ_OST_FLAG_DCBEVT1 - OST status flag for Digital compare event B1

18.2.4.72 static void EPWM_selectCycleByCycleTripZoneClearEvent (uint32_t *base*, **EPWM_CycleByCycleTripZoneClearMode** *clearEvent*) [inline], [static]

Set the Trip Zone CBC pulse clear event.

Parameters

base	is the base address of the EPWM module.
clearEvent	is the CBC trip zone clear event.

This function set the event which automatically clears the CBC (Cycle by Cycle) latch. Valid values for clearEvent are:

- EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO - Clear CBC pulse when counter equals zero
- EPWM_TZ_CBC_PULSE_CLR_CNTR_PERIOD - Clear CBC pulse when counter equals period
- EPWM_TZ_CBC_PULSE_CLR_CNTR_ZERO_PERIOD - Clear CBC pulse when counter equals zero or period

Returns
None.

18.2.4.73 static void EPWM_clearTripZoneFlag (uint32_t *base,* uint16_t *tzFlags*) [inline], [static]

Clear Trip Zone flag

Parameters

base	is the base address of the EPWM module.
tzFlags	is the Trip Zone flags.

This function clears the Trip Zone flags Valid values for tzFlags are:

- EPWM_TZ_INTERRUPT - Global Trip Zone interrupt flag
- EPWM_TZ_FLAG_CBC - Trip Zones Cycle By Cycle flag
- EPWM_TZ_FLAG_OST - Trip Zones One Shot flag
- EPWM_TZ_FLAG_DCAEVT1 - Digital Compare A Event 1 flag
- EPWM_TZ_FLAG_DCAEVT2 - Digital Compare A Event 2 flag
- EPWM_TZ_FLAG_DCBEVT1 - Digital Compare B Event 1 flag
- EPWM_TZ_FLAG_DCBEVT2 - Digital Compare B Event 2 flag

note: A bitwise OR of the valid values can be passed as the tzFlags parameter.

Returns
None.

18.2.4.74 static void EPWM_clearCycleByCycleTripZoneFlag (uint32_t *base,* uint16_t *tzCBCFlags*) [inline], [static]

Clear the Trip Zone Cycle by Cycle flag.

Parameters

base	is the base address of the EPWM module.
tzCBCFlags	is the CBC flag to be cleared.

This function clears the specific Cycle by Cycle Trip Zone flag. The following are valid values for tzCBCFlags.

- EPWM_TZ_CBC_FLAG_1 - CBC 1 flag
- EPWM_TZ_CBC_FLAG_2 - CBC 2 flag
- EPWM_TZ_CBC_FLAG_3 - CBC 3 flag
- EPWM_TZ_CBC_FLAG_4 - CBC 4 flag
- EPWM_TZ_CBC_FLAG_5 - CBC 5 flag
- EPWM_TZ_CBC_FLAG_6 - CBC 6 flag
- EPWM_TZ_CBC_FLAG_DCAEVT2 - CBC flag Digital compare event A2
- EPWM_TZ_CBC_FLAG_DCBEVT2 - CBC flag Digital compare event B2

Returns
None.

18.2.4.75 static void EPWM_clearOneShotTripZoneFlag (uint32_t *base*, uint16_t *tzOSTFlags*) [inline], [static]

Clear the Trip Zone One Shot flag.

Parameters

base	is the base address of the EPWM module.
tzOSTFlags	is the OST flags to be cleared.

This function clears the specific One Shot (OST) Trip Zone flag. The following are valid values for tzOSTFlags.

- EPWM_TZ_OST_FLAG_OST1 - OST flag for OST1
- EPWM_TZ_OST_FLAG_OST2 - OST flag for OST2
- EPWM_TZ_OST_FLAG_OST3 - OST flag for OST3
- EPWM_TZ_OST_FLAG_OST4 - OST flag for OST4
- EPWM_TZ_OST_FLAG_OST5 - OST flag for OST5
- EPWM_TZ_OST_FLAG_OST6 - OST flag for OST6
- EPWM_TZ_OST_FLAG_DCAEVT1 - OST flag for Digital compare event A1
- EPWM_TZ_OST_FLAG_DCBEVT1 - OST flag for Digital compare event B1

Returns
None.

18.2.4.76 static void EPWM_forceTripZoneEvent (uint32_t *base*, uint16_t *tzForceEvent*) [inline], [static]

Force Trip Zone events.

Parameters

base	is the base address of the EPWM module.
tzForceEvent	is the forced Trip Zone event.

This function forces a Trip Zone event. Valid values for tzForceEvent are:

- EPWM_TZ_FORCE_EVENT_CBC - Force Trip Zones Cycle By Cycle event
- EPWM_TZ_FORCE_EVENT_OST - Force Trip Zones One Shot Event
- EPWM_TZ_FORCE_EVENT_DCAEVT1 - Force Digital Compare A Event 1
- EPWM_TZ_FORCE_EVENT_DCAEVT2 - Force Digital Compare A Event 2
- EPWM_TZ_FORCE_EVENT_DCBEVT1 - Force Digital Compare B Event 1
- EPWM_TZ_FORCE_EVENT_DCBEVT2 - Force Digital Compare B Event 2

Returns
None.

18.2.4.77 static void EPWM_enableInterrupt (uint32_t *base*) `[inline]`, `[static]`

Enable ePWM interrupt.

Parameters

base	is the base address of the EPWM module.

This function enables the ePWM interrupt.

Returns
None.

18.2.4.78 static void EPWM_disableInterrupt (uint32_t *base*) `[inline]`, `[static]`

disable ePWM interrupt.

Parameters

base	is the base address of the EPWM module.

This function disables the ePWM interrupt.

Returns
None.

18.2.4.79 static void EPWM_setInterruptSource (uint32_t *base,* uint16_t *interruptSource*) `[inline]`, `[static]`

Sets the ePWM interrupt source.

Parameters

base	is the base address of the EPWM module.
interruptSource	is the ePWM interrupt source.

This function sets the ePWM interrupt source. Valid values for interruptSource are:

- EPWM_INT_TBCTR_ZERO - Time-base counter equal to zero
- EPWM_INT_TBCTR_PERIOD - Time-base counter equal to period
- EPWM_INT_TBCTR_ZERO_OR_PERIOD - Time-base counter equal to zero or period
- EPWM_INT_TBCTR_U_CMPx - Where x is A, B, C or D Time-base counter equal to CMPA, CMPB, CMPC or CMPD (depending the value of x) when the timer is incrementing
- EPWM_INT_TBCTR_D_CMPx - Where x is A, B, C or D Time-base counter equal to CMPA, CMPB, CMPC or CMPD (depending the value of x) when the timer is decrementing

Returns
None.

References EPWM_INT_TBCTR_D_CMPA, EPWM_INT_TBCTR_D_CMPB, EPWM_INT_TBCTR_D_CMPC, EPWM_INT_TBCTR_D_CMPD, EPWM_INT_TBCTR_U_CMPA, EPWM_INT_TBCTR_U_CMPB, EPWM_INT_TBCTR_U_CMPC, and EPWM_INT_TBCTR_U_CMPD.

18.2.4.80 static void EPWM_setInterruptEventCount (uint32_t *base,* uint16_t *eventCount*) [inline], [static]

Sets the ePWM interrupt event counts.

Parameters

base	is the base address of the EPWM module.
eventCount	is the event count for interrupt scale

This function sets the interrupt event count that determines the number of events that have to occur before an interrupt is issued. Maximum value for eventCount is 15.

Returns
None.

18.2.4.81 static bool EPWM_getEventTriggerInterruptStatus (uint32_t *base*) [inline], [static]

Return the interrupt status.

Parameters

base	is the base address of the EPWM module.

This function returns the ePWM interrupt status. **Note** This function doesn't return the Trip Zone status.

Returns
Returns true if ePWM interrupt was generated. Returns false if no interrupt was generated

18.2.4.82 static void EPWM_clearEventTriggerInterruptFlag (uint32_t *base*) [inline], [static]

Clear interrupt flag.

Parameters

base	is the base address of the EPWM module.

This function clears the ePWM interrupt flag.

Returns
None

18.2.4.83 static void EPWM_enableInterruptEventCountInit (uint32_t *base*) [inline], [static]

Enable Pre-interrupt count load.

Parameters

base	is the base address of the EPWM module.

This function enables the ePWM interrupt counter to be pre-interrupt loaded with a count value.

Note
This is valid only for advanced/expanded interrupt mode

Returns
None.

18.2.4.84 static void EPWM_disableInterruptEventCountInit (uint32_t *base*) [inline], [static]

Disable interrupt count load.

Parameters

base	is the base address of the EPWM module.

This function disables the ePWM interrupt counter from being loaded with pre-interrupt count value.

Returns
None.

18.2.4.85 static void EPWM_forceInterruptEventCountInit (uint32_t *base*) [inline], [static]

Force a software pre interrupt event counter load.

Parameters

base	is the base address of the EPWM module.

This function forces the ePWM interrupt counter to be loaded with the contents set by EPWM_setPreInterruptEventCount().

Note
make sure the EPWM_enablePreInterruptEventCountLoad() function is is called before invoking this function.

Returns
None.

18.2.4.86 static void EPWM_setInterruptEventCountInitValue (uint32_t *base,* uint16_t *eventCount*) [inline], [static]

Set interrupt count.

Parameters

base	is the base address of the EPWM module.
eventCount	is the ePWM interrupt count value.

This function sets the ePWM interrupt count. eventCount is the value of the pre-interrupt value that is to be loaded. The maximum value of eventCount is 15.

Returns
None.

18.2.4.87 static uint16_t EPWM_getInterruptEventCount (uint32_t *base*) [inline], [static]

Get the interrupt count.

Parameters

base	is the base address of the EPWM module.

This function returns the ePWM interrupt event count.

Returns
The interrupt event counts that have occurred.

18.2.4.88 static void EPWM_forceEventTriggerInterrupt (uint32_t *base*) [inline], [static]

Force ePWM interrupt.

Parameters

base	is the base address of the EPWM module.

This function forces an ePWM interrupt.

Returns
None

18.2.4.89 static void EPWM_enableADCTrigger (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Enable ADC SOC event.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function enables the ePWM module to trigger an ADC SOC event. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns

None.

References EPWM_SOC_A.

18.2.4.90 static void EPWM_disableADCTrigger (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) `[inline]`, `[static]`

Disable ADC SOC event.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function disables the ePWM module from triggering an ADC SOC event. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns

None.

References EPWM_SOC_A.

18.2.4.91 static void EPWM_setADCTriggerSource (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*, **EPWM_ADCStartOfConversionSource** *socSource*) `[inline]`, `[static]`

Sets the ePWM SOC source.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.
socSource	is the SOC source.

This function sets the ePWM ADC SOC source. Valid values for socSource are: adcSOCType

- EPWM_SOC_A - SOC A

- EPWM_SOC_B - SOC B socSource
  - EPWM_SOC_DCxEVT1 - Event is based on DCxEVT1
  - EPWM_SOC_TBCTR_ZERO - Time-base counter equal to zero
  - EPWM_SOC_TBCTR_PERIOD - Time-base counter equal to period
  - EPWM_SOC_TBCTR_ZERO_OR_PERIOD - Time-base counter equal to zero or period
  - EPWM_SOC_TBCTR_U_CMPx - Where x is A, B, C or D Time-base counter equal to CMPA, CMPB, CMPC or CMPD(depending the value of x) when the timer is incrementing
  - EPWM_SOC_TBCTR_D_CMPx - Where x is A, B, C or D Time-base counter equal to CMPA, CMPB, CMPC or CMPD(depending the value of x) when the timer is decrementing

Returns

None.

References EPWM_SOC_A, EPWM_SOC_TBCTR_D_CMPA, EPWM_SOC_TBCTR_D_CMPB, EPWM_SOC_TBCTR_D_CMPC, EPWM_SOC_TBCTR_D_CMPD, EPWM_SOC_TBCTR_U_CMPA, EPWM_SOC_TBCTR_U_CMPB, EPWM_SOC_TBCTR_U_CMPC, and EPWM_SOC_TBCTR_U_CMPD.

18.2.4.92 static void EPWM_setADCTriggerEventPrescale (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*, uint16_t *preScaleCount*) [inline], [static]

Sets the ePWM SOC event counts.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.
preScaleCount	is the event count number.

This function sets the SOC event count that determines the number of events that have to occur before an SOC is issued. Valid values for the parameters are: adcSOCType

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B preScaleCount
  - [1 - 15] - Generate SOC pulse every preScaleCount upto 15 events. **Note**. A preScaleCount value of 0 disables the presale.

Returns

None.

References EPWM_SOC_A.

18.2.4.93 static bool EPWM_getADCTriggerFlagStatus (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Return the SOC event status.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function returns the ePWM SOC status. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns

Returns true if the selected adcSOCType SOC was generated. Returns false if the selected adcSOCType SOC was not generated.

18.2.4.94 static void EPWM_clearADCTriggerFlag (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Clear SOC flag.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function clears the ePWM SOC flag. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns

None

18.2.4.95 static void EPWM_enableADCTriggerEventCountInit (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Enable Pre-SOC event count load.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function enables the ePWM SOC event counter which is set by the EPWM_setADCTriggerEventCountInitValue() function to be loaded before an SOC event. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Note

This is valid only for advanced/expanded SOC mode

Returns
None.

18.2.4.96 static void EPWM_disableADCTriggerEventCountInit (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Disable Pre-SOC event count load.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function disables the ePWM SOC event counter from being loaded before an SOC event (only an SOC event causes an increment of the counter value). Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Note
This is valid only for advanced/expanded SOC mode

Returns
None.

18.2.4.97 static void EPWM_forceADCTriggerEventCountInit (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Force a software pre SOC event counter load.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type

This function forces the ePWM SOC counter to be loaded with the contents set by EPWM_setPreADCStartOfConversionEventCount().

Note
make sure the EPWM_enableADCTriggerEventCountInit() function is called before invoking this function.

Returns
None.

18.2.4.98 static void EPWM_setADCTriggerEventCountInitValue (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*, uint16_t *eventCount*) [inline], [static]

Set ADC Trigger count values.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.
eventCount	is the ePWM interrupt count value.

This function sets the ePWM ADC Trigger count values. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B The eventCount has a maximum value of 15.

Returns

None.

References EPWM_SOC_A.

18.2.4.99 static uint16_t EPWM_getADCTriggerEventCount (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Get the SOC event count.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function returns the ePWM SOC event count. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns

The SOC event counts that have occurred.

References EPWM_SOC_A.

18.2.4.100 static void EPWM_forceADCTrigger (uint32_t *base*, **EPWM_ADCStartOfConversionType** *adcSOCType*) [inline], [static]

Force SOC event.

Parameters

base	is the base address of the EPWM module.
adcSOCType	is the ADC SOC type.

This function forces an ePWM SOC event. Valid values for adcSOCType are:

- EPWM_SOC_A - SOC A
- EPWM_SOC_B - SOC B

Returns
None

18.2.4.101 static void EPWM_selectDigitalCompareTripInput (uint32_t *base*, **EPWM_DigitalCompareTripInput** *tripSource*, **EPWM_DigitalCompareType** *dcType*) [inline], [static]

Set the DC trip input.

Parameters

base	is the base address of the EPWM module.
tripSource	is the tripSource.
dcType	is the Digital Compare type.

This function sets the trip input to the Digital Compare (DC). For a given dcType the function sets the tripSource to be the input to the DC. Valid values for the parameter are: dcType

- EPWM_DC_TYPE_DCAH - Digital Compare A High
- EPWM_DC_TYPE_DCAL - Digital Compare A Low
- EPWM_DC_TYPE_DCBH - Digital Compare B High
- EPWM_DC_TYPE_DCBL - Digital Compare B Low tripSource

EPWM_DC_TRIP_TRIPINx - Trip x, where x ranges from 1 to 15 excluding 13.

- EPWM_DC_TRIP_COMBINATION - selects all the Trip signals whose input is enabled by the EPWM_enableDCTripCombInput() function.

Returns
None

18.2.4.102 static void EPWM_enableDigitalCompareBlankingWindow (uint32_t *base*) [inline], [static]

Enable DC filter blanking window.

Parameters

base	is the base address of the EPWM module.

This function enables the DC filter blanking window.

Returns
None

18.2.4.103 static void EPWM_disableDigitalCompareBlankingWindow (uint32_t *base*) [inline], [static]

Disable DC filter blanking window.

Parameters

base	is the base address of the EPWM module.

This function disables the DC filter blanking window.

Returns

None

18.2.4.104 static void EPWM_enableDigitalCompareWindowInverseMode (uint32_t *base*) [inline], [static]

Enable Digital Compare Window inverse mode.

Parameters

base	is the base address of the EPWM module.

This function enables the Digital Compare Window inverse mode. This will invert the blanking window.

Returns

None

18.2.4.105 static void EPWM_disableDigitalCompareWindowInverseMode (uint32_t *base*) [inline], [static]

Disable Digital Compare Window inverse mode.

Parameters

base	is the base address of the EPWM module.

This function disables the Digital Compare Window inverse mode.

Returns

None

18.2.4.106 static void EPWM_setDigitalCompareBlankingEvent (uint32_t *base*, **EPWM_DigitalCompareBlankingPulse** *blankingPulse*) [inline], [static]

Set the Digital Compare filter blanking pulse.

Parameters

base	is the base address of the EPWM module.
blankingPulse	is Pulse that starts blanking window.

This function sets the input pulse that starts the Digital Compare blanking window. Valid values for blankingPulse are:

- EPWM_DC_WINDOW_START_TBCTR_PERIOD - Time base counter equals period
- EPWM_DC_WINDOW_START_TBCTR_ZERO - Time base counter equals zero

- EPWM_DC_WINDOW_START_TBCTR_ZERO_PERIOD - Time base counter equals zero or period.

Returns
None

18.2.4.107 static void EPWM_setDigitalCompareFilterInput (uint32_t *base*, **EPWM_DigitalCompareFilterInput** *filterInput*) [inline], [static]

Set up the Digital Compare filter input.

Parameters

base	is the base address of the EPWM module.
filterInput	is Digital Compare signal source.

This function sets the signal input source that will be filtered by the Digital Compare module. Valid values for filterInput are:

- EPWM_DC_WINDOW_SOURCE_DCAEVT1 - DC filter signal source is DCAEVT1
- EPWM_DC_WINDOW_SOURCE_DCAEVT2 - DC filter signal source is DCAEVT2
- EPWM_DC_WINDOW_SOURCE_DCBEVT1 - DC filter signal source is DCBEVT1
- EPWM_DC_WINDOW_SOURCE_DCBEVT2 - DC filter signal source is DCBEVT2

Returns
None

18.2.4.108 static void EPWM_enableDigitalCompareEdgeFilter (uint32_t *base*) [inline], [static]

Enable Digital Compare Edge Filter.

Parameters

base	is the base address of the EPWM module.

This function enables the Digital Compare Edge filter to generate event after configured number of edges.

Returns
None

18.2.4.109 static void EPWM_disableDigitalCompareEdgeFilter (uint32_t *base*) [inline], [static]

Disable Digital Compare Edge Filter.

Parameters

base	is the base address of the EPWM module.

This function disables the Digital Compare Edge filter.

Returns

None

18.2.4.110 static void EPWM_setDigitalCompareEdgeFilterMode (uint32_t *base*, **EPWM_DigitalCompareEdgeFilterMode** *edgeMode*) [inline], [static]

Set the Digital Compare Edge Filter Mode.

Parameters

base	is the base address of the EPWM module.
edgeMode	is Digital Compare Edge filter mode.

This function sets the Digital Compare Event filter mode. Valid values for edgeMode are:

- EPWM_DC_EDGEFILT_MODE_RISING - DC edge filter mode is rising edge
- EPWM_DC_EDGEFILT_MODE_FALLING - DC edge filter mode is falling edge
- EPWM_DC_EDGEFILT_MODE_BOTH - DC edge filter mode is both edges

Returns

None

18.2.4.111 static void EPWM_setDigitalCompareEdgeFilterEdgeCount (uint32_t *base*, uint16_t *edgeCount*) [inline], [static]

Set the Digital Compare Edge Filter Edge Count.

Parameters

base	is the base address of the EPWM module.
edgeMode	is Digital Compare Edge filter mode.

This function sets the Digital Compare Event filter Edge Count to genrate events. Valid values for edgeCount can be:

- EPWM_DC_EDGEFILT_EDGECNT_0 - No edge is required to generate event
- EPWM_DC_EDGEFILT_EDGECNT_1 - 1 edge is required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_2 - 2 edges are required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_3 - 3 edges are required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_4 - 4 edges are required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_5 - 5 edges are required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_6 - 6 edges are required for event generation
- EPWM_DC_EDGEFILT_EDGECNT_7 - 7 edges are required for event generation

Returns
None

18.2.4.112 static uint16_t EPWM_getDigitalCompareEdgeFilterEdgeCount (uint32_t *base*) [inline], [static]

Returns the Digital Compare Edge Filter Edge Count.

Parameters

base	is the base address of the EPWM module.

This function returns the configured Digital Compare Edge filter edge count required to generate events. It can return values from 0-7.

Returns
Returns the configured DigitalCompare Edge filter edge count.

18.2.4.113 static uint16_t EPWM_getDigitalCompareEdgeFilterEdgeStatus (uint32_t *base*) [inline], [static]

Returns the Digital Compare Edge filter captured edge count status.

Parameters

base	is the base address of the EPWM module.

This function returns the count of edges captured by Digital Compare Edge filter. It can return values from 0-7.

Returns
Returns the count of captured edges

18.2.4.114 static void EPWM_setDigitalCompareWindowOffset (uint32_t *base,* uint16_t *windowOffsetCount*) [inline], [static]

Set up the Digital Compare filter window offset

Parameters

base	is the base address of the EPWM module.
windowOffset-Count	is blanking window offset length.

This function sets the offset between window start pulse and blanking window in TBCLK count. The function take a 16bit count value for the offset value.

Returns
None

18.2.4.115 static void EPWM_setDigitalCompareWindowLength (uint32_t *base,* uint16_t *windowLengthCount*) [inline], [static]

Set up the Digital Compare filter window length

Parameters

base	is the base address of the EPWM module.
windowLength-Count	is blanking window length.

This function sets up the Digital Compare filter blanking window length in TBCLK count.The function takes a 16bit count value for the window length.

Returns
None

18.2.4.116 static uint16_t EPWM_getDigitalCompareBlankingWindowOffsetCount (uint32_t *base*) [inline], [static]

Return DC filter blanking window offset count.

Parameters

base	is the base address of the EPWM module.

This function returns DC filter blanking window offset count.

Returns
None

18.2.4.117 static uint16_t EPWM_getDigitalCompareBlankingWindowLengthCount (uint32_t *base*) [inline], [static]

Return DC filter blanking window length count.

Parameters

base	is the base address of the EPWM module.

This function returns DC filter blanking window length count.

Returns
None

18.2.4.118 static void EPWM_setDigitalCompareEventSource (uint32_t *base*, **EPWM_DigitalCompareModule** *dcModule*, **EPWM_DigitalCompareEvent** *dcEvent*, **EPWM_DigitalCompareEventSource** *dcEventSource*) [inline], [static]

Set up the Digital Compare Event source.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.
dcEvent	is the Digital Compare Event number.
dcEventSource	is the - Digital Compare Event source.

This function sets up the Digital Compare module Event sources. The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B dcEvent
- EPWM_DC_EVENT_1 - Digital Compare Event number 1
- EPWM_DC_EVENT_2 - Digital Compare Event number 2 dcEventSource
- EPWM_DC_EVENT_SOURCE_FILT_SIGNAL - signal source is filtered

  Note

  The signal source for this option is DCxEVTy, where the value of x is dependent on dcModule and the value of y is dependent on dcEvent. Possible signal sources are DCAEVT1, DCBEVT1, DCAEVT2 or DCBEVT2 depending on the value of both dcModule and dcEvent.
- EPWM_DC_EVENT_SOURCE_ORIG_SIGNAL - signal source is unfiltered The signal source for this option is DCEVTFILT.

  Returns

  None

References EPWM_DC_EVENT_1.

18.2.4.119 static void EPWM_setDigitalCompareEventSyncMode (uint32_t *base,* **EPWM_DigitalCompareModule** *dcModule,* **EPWM_DigitalCompareEvent** *dcEvent,* **EPWM_DigitalCompareSyncMode** *syncMode*) [inline], [static]

Set up the Digital Compare input sync mode.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.
dcEvent	is the Digital Compare Event number.
syncMode	is the Digital Compare Event sync mode.

This function sets up the Digital Compare module Event sources. The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B dcEvent
- EPWM_DC_EVENT_1 - Digital Compare Event number 1
- EPWM_DC_EVENT_2 - Digital Compare Event number 2 syncMode
- EPWM_DC_EVENT_INPUT_SYNCED - DC input signal is synced with TBCLK
- EPWM_DC_EVENT_INPUT_NOT SYNCED - DC input signal is not synced with TBCLK

Returns
None

References EPWM_DC_EVENT_1.

18.2.4.120 static void EPWM_enableDigitalCompareADCTrigger (uint32_t *base*, **EPWM_DigitalCompareModule** *dcModule*) [inline], [static]

Enable Digital Compare to generate Start of Conversion.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.

This function enables the Digital Compare Event 1 to generate Start of Conversion. The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B

Returns
None

18.2.4.121 static void EPWM_disableDigitalCompareADCTrigger (uint32_t *base*, **EPWM_DigitalCompareModule** *dcModule*) [inline], [static]

Disable Digital Compare from generating Start of Conversion.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.

This function disables the Digital Compare Event 1 from generating Start of Conversion. The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B

Returns
None

18.2.4.122 static void EPWM_enableDigitalCompareSyncEvent (uint32_t *base*, **EPWM_DigitalCompareModule** *dcModule*) [inline], [static]

Enable Digital Compare to generate sync out pulse.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.

This function enables the Digital Compare Event 1 to generate sync out pulse The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B

Returns
None

18.2.4.123 static void EPWM_disableDigitalCompareSyncEvent (uint32_t *base*, **EPWM_DigitalCompareModule** *dcModule*) [inline], [static]

Disable Digital Compare from generating Start of Conversion.

Parameters

base	is the base address of the EPWM module.
dcModule	is the Digital Compare module.

This function disables the Digital Compare Event 1 from generating synch out pulse. The following are valid values for the parameters. dcModule

- EPWM_DC_MODULE_A - Digital Compare Module A
- EPWM_DC_MODULE_B - Digital Compare Module B

Returns
None

18.2.4.124 static void EPWM_enableDigitalCompareCounterCapture (uint32_t *base*) [inline], [static]

Enables the Time Base Counter Capture controller.

Parameters

base	is the base address of the EPWM module.

This function enables the time Base Counter Capture.

Returns
None.

18.2.4.125 static void EPWM_disableDigitalCompareCounterCapture (uint32_t *base*) [inline], [static]

Disables the Time Base Counter Capture controller.

Parameters

base	is the base address of the EPWM module.

This function disable the time Base Counter Capture.

Returns

None.

18.2.4.126 static void EPWM_setDigitalCompareCounterShadowMode (uint32_t *base,* bool *enableShadowMode*) [inline], [static]

Set the Time Base Counter Capture mode.

Parameters

base	is the base address of the EPWM module.
enableShadow-Mode	is the shadow read mode flag.

This function sets the mode the Time Base Counter value is read from. If enableShadowMode is true, CPU reads of the DCCAP register will return the shadow register contents.If enableShadowMode is false, CPU reads of the DCCAP register will return the active register contents.

Returns

None.

18.2.4.127 static bool EPWM_getDigitalCompareCaptureStatus (uint32_t *base*) [inline], [static]

Return the DC Capture event status.

Parameters

base	is the base address of the EPWM module.

This function returns the DC capture event status.

Returns

Returns true if a DC capture event has occurs. Returns false if no DC Capture event has occurred.
None.

18.2.4.128 static uint16_t EPWM_getDigitalCompareCaptureCount (uint32_t *base*) [inline], [static]

Return the DC Time Base Counter capture value.

Parameters

base	is the base address of the EPWM module.

This function returns the DC Time Base Counter capture value. The value read is determined by the mode as set in the EPWM_setTimeBaseCounterReadMode() function.

Returns
Returns the DC Time Base Counter Capture count value.

18.2.4.129 static void EPWM_enableDigitalCompareTripCombinationInput (uint32_t *base,* uint16_t *tripInput,* **EPWM_DigitalCompareType** *dcType*) [inline], [static]

Enable DC TRIP combinational input.

Parameters

base	is the base address of the EPWM module.
tripInput	is the Trip number.
dcType	is the Digital Compare module.

This function enables the specified Trip input. Valid values for the parameters are: tripInput

- EPWM_DC_COMBINATIONAL_TRIPINx, where x is 1, 2, ...12, 14, 15 dcType
- EPWM_DC_TYPE_DCAH - Digital Compare A High
- EPWM_DC_TYPE_DCAL - Digital Compare A Low
- EPWM_DC_TYPE_DCBH - Digital Compare B High
- EPWM_DC_TYPE_DCBL - Digital Compare B Low

Returns
None.

18.2.4.130 static void EPWM_disableDigitalCompareTripCombinationInput (uint32_t *base,* uint16_t *tripInput,* **EPWM_DigitalCompareType** *dcType*) [inline], [static]

Disable DC TRIP combinational input.

Parameters

base	is the base address of the EPWM module.
tripInput	is the Trip number.
dcType	is the Digital Compare module.

This function disables the specified Trip input. Valid values for the parameters are: tripInput

- EPWM_DC_COMBINATIONAL_TRIPINx, where x is 1, 2, ...12, 14, 15 dcType
- EPWM_DC_TYPE_DCAH - Digital Compare A High
- EPWM_DC_TYPE_DCAL - Digital Compare A Low
- EPWM_DC_TYPE_DCBH - Digital Compare B High

- EPWM_DC_TYPE_DCBL - Digital Compare B Low

Returns
None.

18.2.4.131 static void EPWM_enableValleyCapture (uint32_t *base*) [inline], [static]

Enable valley capture mode.

Parameters

base	is the base address of the EPWM module.

This function enables Valley Capture mode.

Returns
None.

18.2.4.132 static void EPWM_disableValleyCapture (uint32_t *base*) [inline], [static]

Disable valley capture mode.

Parameters

base	is the base address of the EPWM module.

This function disables Valley Capture mode.

Returns
None.

18.2.4.133 static void EPWM_startValleyCapture (uint32_t *base*) [inline], [static]

Start valley capture mode.

Parameters

base	is the base address of the EPWM module.

This function starts Valley Capture sequence.

Make sure you invoke EPWM_setValleyTriggerSource with the trigger variable set to EPWM_VALLEY_TRIGGER_EVENT_SOFTWARE before calling this function.

Returns
None.

18.2.4.134 static void EPWM_setValleyTriggerSource (uint32_t *base*, **EPWM_ValleyTriggerSource** *trigger*) [inline], [static]

Set valley capture trigger.

Parameters

base	is the base address of the EPWM module.
trigger	is the Valley counter trigger.

This function sets the trigger value that initiates Valley Capture sequence

Set the number of Trigger source events for starting and stopping the valley capture using EPWM_setValleyTriggerEdgeCounts().

Returns
None.

18.2.4.135 static void EPWM_setValleyTriggerEdgeCounts (uint32_t *base*, uint16_t *startCount*, uint16_t *stopCount*) [inline], [static]

Set valley capture trigger source count.

Parameters

base	is the base address of the EPWM module.
startCount	
stopCount	This function sets the number of trigger events required to start and stop the valley capture count. Maximum values for both startCount and stopCount is 15 corresponding to the 15th edge of the trigger event.

Note: A startCount value of 0 prevents starting the valley counter. A stopCount value of 0 prevents the valley counter from stopping.

Returns
None.

18.2.4.136 static void EPWM_enableValleyHWDelay (uint32_t *base*) [inline], [static]

Enable valley switching delay.

Parameters

base	is the base address of the EPWM module.

This function enables Valley switching delay.

Returns
None.

18.2.4.137 static void EPWM_disableValleyHWDelay (uint32_t *base*) [inline], [static]

Disable valley switching delay.

Parameters

base	is the base address of the EPWM module.

This function disables Valley switching delay.

Returns

None.

18.2.4.138 static void EPWM_setValleySWDelayValue (uint32_t *base,* uint16_t *delayOffsetValue*) [inline], [static]

Set Valley delay values.

Parameters

base	is the base address of the EPWM module.
delayOffset-Value	is the software defined delay offset value.

This function sets the Valley delay value.

Returns

None.

18.2.4.139 static void EPWM_setValleyDelayDivider (uint32_t *base,* **EPWM_ValleyDelayMode** *delayMode*) [inline], [static]

Set Valley delay mode.

Parameters

base	is the base address of the EPWM module.
delayMode	is the Valley delay mode.

This function sets the Valley delay mode values.

Returns

None.

18.2.4.140 static bool EPWM_getValleyEdgeStatus (uint32_t *base,* **EPWM_ValleyCounterEdge** *edge*) [inline], [static]

Get the valley edge status bit.

Parameters

base	is the base address of the EPWM module.

edge	is the start or stop edge.

This function returns the status of the start or stop valley status depending on the value of edge. If a start or stop edge has occurred, the function returns true, if not it returns false.

Returns
Returns true if the specified edge has occurred, Returns false if the specified edge has not occurred.

References EPWM_VALLEY_COUNT_START_EDGE.

18.2.4.141 static uint16_t EPWM_getValleyCount (uint32_t *base*) [inline], [static]

Get the Valley Counter value.

Parameters

base	is the base address of the EPWM module.

This function returns the valley time base count value which is captured upon occurrence of the stop edge condition selected by EPWM_setValleyTriggerSource() and by the stopCount variable of the EPWM_setValleyTriggerEdgeCounts() function.

Returns
Returns the valley base time count.

18.2.4.142 static uint16_t EPWM_getValleyHWDelay (uint32_t *base*) [inline], [static]

Get the Valley delay value.

Parameters

base	is the base address of the EPWM module.

This function returns the hardware valley delay count.

Returns
Returns the valley delay count.

18.2.4.143 static void EPWM_enableGlobalLoad (uint32_t *base*) [inline], [static]

Enable Global shadow load mode.

Parameters

base	is the base address of the EPWM module.

This function enables Global shadow to active load mode of registers. The trigger source for loading shadow to active is determined by EPWM_setGlobalLoadTrigger() function.

Returns
None.

18.2.4.144 static void EPWM_disableGlobalLoad (uint32_t *base*) [inline], [static]

Disable Global shadow load mode.

Parameters

base	is the base address of the EPWM module.

This function disables Global shadow to active load mode of registers. Loading shadow to active is determined individually.

Returns
None.

18.2.4.145 static void EPWM_setGlobalLoadTrigger (uint32_t *base*, EPWM_GlobalLoadTrigger *loadTrigger*) [inline], [static]

Set the Global shadow load pulse.

Parameters

base	is the base address of the EPWM module.
loadTrigger	is the pulse that causes global shadow load.

This function sets the pulse that causes Global shadow to active load. Valid values for the loadTrigger parameter are:

- EPWM_GL_LOAD_PULSE_CNTR_ZERO - load when counter is equal to zero
- EPWM_GL_LOAD_PULSE_CNTR_PERIOD - load when counter is equal to period
- EPWM_GL_LOAD_PULSE_CNTR_ZERO_PERIOD - load when counter is equal to zero or period
- EPWM_GL_LOAD_PULSE_SYNC - load on sync event
- EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_ZERO - load on sync event or when counter is equal to zero
- EPWM_GL_LOAD_PULSE_SYNC_OR_CNTR_PERIOD - load on sync event or when counter is equal to period
- EPWM_GL_LOAD_PULSE_SYNC_CNTR_ZERO_PERIOD - load on sync event or when counter is equal to period or zero
- EPWM_GL_LOAD_PULSE_GLOBAL_FORCE - load on global force

Returns
None.

18.2.4.146 static void EPWM_setGlobalLoadEventPrescale (uint32_t *base*, uint16_t *prescalePulseCount*) [inline], [static]

Set the number of Global load pulse event counts

Parameters

base	is the base address of the EPWM module.
prescalePulseCount	is the pulse event counts.

This function sets the number of Global Load pulse events that have to occurred before a global load pulse is issued. Valid values for prescaleCount range from 0 to 7. 0 being no event (disables counter), and 7 representing 7 events.

Returns

None.

18.2.4.147 static uint16_t EPWM_getGlobalLoadEventCount (uint32_t *base*) [inline], [static]

Return the number of Global load pulse event counts

Parameters

base	is the base address of the EPWM module.

This function returns the number of Global Load pulse events that have occurred. These pulse events are set by the EPWM_setGlobalLoadTrigger() function.

Returns

None.

18.2.4.148 static void EPWM_disableGlobalLoadOneShotMode (uint32_t *base*) [inline], [static]

Enable continuous global shadow to active load.

Parameters

base	is the base address of the EPWM module.

This function enables global continuous shadow to active load. Register load happens every time the event set by the EPWM_setGlobalLoadTrigger() occurs.

Returns

None.

18.2.4.149 static void EPWM_enableGlobalLoadOneShotMode (uint32_t *base*) [inline], [static]

Enable One shot global shadow to active load.

Parameters

base	is the base address of the EPWM module.

This function enables a one time global shadow to active load. Register load happens every time the event set by the EPWM_setGlobalLoadTrigger() occurs.

Returns
None.

18.2.4.150 static void EPWM_setGlobalLoadOneShotLatch (uint32_t *base*) [inline], [static]

Set One shot global shadow to active load pulse.

Parameters

base	is the base address of the EPWM module.

This function sets a one time global shadow to active load pulse. The pulse propagates to generate a load signal if any of the events set by EPWM_setGlobalLoadTrigger() occur.

Returns
None.

18.2.4.151 static void EPWM_forceGlobalLoadOneShotEvent (uint32_t *base*) [inline], [static]

Force a software One shot global shadow to active load pulse.

Parameters

base	is the base address of the EPWM module.

This function forces a software a one time global shadow to active load pulse.

Returns
None.

18.2.4.152 static void EPWM_enableGlobalLoadRegisters (uint32_t *base,* uint16_t *loadRegister*) [inline], [static]

Enable a register to be loaded Globally.

Parameters

base	is the base address of the EPWM module.
loadRegister	is the register.

This function enables the register specified by loadRegister to be globally loaded. Valid values for loadRegister are:

- EPWM_GL_REGISTER_TBPRD_TBPRDHR - Register TBPRD:TBPRDHR

- EPWM_GL_REGISTER_CMPA_CMPAHR - Register CMPA:CMPAHR
- EPWM_GL_REGISTER_CMPB_CMPBHR - Register CMPB:CMPBHR
- EPWM_GL_REGISTER_CMPC - Register CMPC
- EPWM_GL_REGISTER_CMPD - Register CMPD
- EPWM_GL_REGISTER_DBRED_DBREDHR - Register DBRED:DBREDHR
- EPWM_GL_REGISTER_DBFED_DBFEDHR - Register DBFED:DBFEDHR
- EPWM_GL_REGISTER_DBCTL - Register DBCTL
- EPWM_GL_REGISTER_AQCTLA_AQCTLA2 - Register AQCTLA/A2
- EPWM_GL_REGISTER_AQCTLB_AQCTLB2 - Register AQCTLB/B2
- EPWM_GL_REGISTER_AQCSFRC - Register AQCSFRC

Returns

None.

18.2.4.153 static void EPWM_disableGlobalLoadRegisters (uint32_t *base,* uint16_t *loadRegister*) [inline], [static]

Disable a register to be loaded Globally.

Parameters

base	is the base address of the EPWM module.
loadRegister	is the register.

This function disables the register specified by loadRegister from being loaded globally. The shadow to active load happens as specified by the register control Valid values for loadRegister are:

- EPWM_GL_REGISTER_TBPRD_TBPRDHR - Register TBPRD:TBPRDHR
- EPWM_GL_REGISTER_CMPA_CMPAHR - Register CMPA:CMPAHR
- EPWM_GL_REGISTER_CMPB_CMPBHR - Register CMPB:CMPBHR
- EPWM_GL_REGISTER_CMPC - Register CMPC
- EPWM_GL_REGISTER_CMPD - Register CMPD
- EPWM_GL_REGISTER_DBRED_DBREDHR - Register DBRED:DBREDHR
- EPWM_GL_REGISTER_DBFED_DBFEDHR - Register DBFED:DBFEDHR
- EPWM_GL_REGISTER_DBCTL - Register DBCTL
- EPWM_GL_REGISTER_AQCTLA_AQCTLA2 - Register AQCTLA/A2
- EPWM_GL_REGISTER_AQCTLB_AQCTLB2 - Register AQCTLB/B2
- EPWM_GL_REGISTER_AQCSFRC - Register AQCSFRC

Returns

None.

18.2.4.154 static void EPWM_lockRegisters (uint32_t *base,* **EPWM_LockRegisterGroup** *registerGroup*) [inline], [static]

Lock EALLOW protected register groups

Parameters

base	is the base address of the EPWM module.
registerGroup	is the EALLOW register groups.

This functions locks the EALLOW protected register groups specified by the registerGroup variable.

Returns
None.

18.2.4.155 void EPWM_setEmulationMode (uint32_t *base*, **EPWM_EmulationMode** *emulationMode*)

Set emulation mode

Parameters

base	is the base address of the EPWM module.
emulationMode	is the emulation mode.

This function sets the emulation behaviours of the time base counter. Valid values for emulationMode are:

- EPWM_EMULATION_STOP_AFTER_NEXT_TB - Stop after next Time Base counter increment or decrement.
- EPWM_EMULATION_STOP_AFTER_FULL_CYCLE - Stop when counter completes whole cycle.
- EPWM_EMULATION_FREE_RUN - Free run.

Returns
None.

19 HRPWM Module

Introduction 311
API Functions 311

19.1 HRPWM Introduction

The HRPWM (High Resolution Pulse width Modulator) API provides a set of functions for configuring and using the HRPWM module. The functions provided give access to the HRPWM module which extends the time resolution capability of the ePWM module thus achieving a finer resolution than would be attainable just using the main CPU clock. */

19.2 API Functions

Enumerations

- enum HRPWM_Channel { HRPWM_CHANNEL_A, HRPWM_CHANNEL_B }
- enum HRPWM_MEPEdgeMode { HRPWM_MEP_CTRL_DISABLE, HRPWM_MEP_CTRL_RISING_EDGE, HRPWM_MEP_CTRL_FALLING_EDGE, HRPWM_MEP_CTRL_RISING_AND_FALLING_EDGE }
- enum HRPWM_MEPCtrlMode { HRPWM_MEP_DUTY_PERIOD_CTRL, HRPWM_MEP_PHASE_CTRL }
- enum HRPWM_LoadMode { HRPWM_LOAD_ON_CNTR_ZERO, HRPWM_LOAD_ON_CNTR_PERIOD, HRPWM_LOAD_ON_CNTR_ZERO_PERIOD }
- enum HRPWM_ChannelBOutput { HRPWM_OUTPUT_ON_B_NORMAL, HRPWM_OUTPUT_ON_B_INV_A }
- enum HRPWM_SyncPulseSource { HRPWM_PWMSYNC_SOURCE_PERIOD, HRPWM_PWMSYNC_SOURCE_ZERO, HRPWM_PWMSYNC_SOURCE_COMPC_UP, HRPWM_PWMSYNC_SOURCE_COMPC_DOWN, HRPWM_PWMSYNC_SOURCE_COMPD_UP, HRPWM_PWMSYNC_SOURCE_COMPD_DOWN }
- enum HRPWM_CounterCompareModule { HRPWM_COUNTER_COMPARE_A, HRPWM_COUNTER_COMPARE_B }
- enum HRPWM_MEPDeadBandEdgeMode { HRPWM_DB_MEP_CTRL_DISABLE, HRPWM_DB_MEP_CTRL_RED, HRPWM_DB_MEP_CTRL_FED, HRPWM_DB_MEP_CTRL_RED_FED }
- enum HRPWM_LockRegisterGroup { HRPWM_REGISTER_GROUP_HRPWM, HRPWM_REGISTER_GROUP_GLOBAL_LOAD, HRPWM_REGISTER_GROUP_TRIP_ZONE, HRPWM_REGISTER_GROUP_TRIP_ZONE_CLEAR, HRPWM_REGISTER_GROUP_DIGITAL_COMPARE }

Functions

- static void HRPWM_setPhaseShift (uint32_t base, uint32_t phaseCount)

- static void HRPWM_setTimeBasePeriod (uint32_t base, uint32_t periodCount)
- static uint32_t HRPWM_getTimeBasePeriod (uint32_t base)
- static void HRPWM_setMEPEdgeSelect (uint32_t base, HRPWM_Channel channel, HRPWM_MEPEdgeMode mepEdgeMode)
- static void HRPWM_setMEPControlMode (uint32_t base, HRPWM_Channel channel, HRPWM_MEPCtrlMode mepCtrlMode)
- static void HRPWM_setCounterCompareShadowLoadEvent (uint32_t base, HRPWM_Channel channel, HRPWM_LoadMode loadEvent)
- static void HRPWM_setOutputSwapMode (uint32_t base, bool enableOutputSwap)
- static void HRPWM_setChannelBOutputPath (uint32_t base, HRPWM_ChannelBOutput outputOnB)
- static void HRPWM_enableAutoConversion (uint32_t base)
- static void HRPWM_disableAutoConversion (uint32_t base)
- static void HRPWM_enablePeriodControl (uint32_t base)
- static void HRPWM_disablePeriodControl (uint32_t base)
- static void HRPWM_enablePhaseShiftLoad (uint32_t base)
- static void HRPWM_disablePhaseShiftLoad (uint32_t base)
- static void HRPWM_setSyncPulseSource (uint32_t base, HRPWM_SyncPulseSource syncPulseSource)
- static void HRPWM_setCounterCompareValue (uint32_t base, HRPWM_CounterCompareModule compModule, uint32_t compCount)
- static uint32_t HRPWM_getCounterCompareValue (uint32_t base, HRPWM_CounterCompareModule compModule)
- static void HRPWM_setRisingEdgeDelay (uint32_t base, uint32_t redCount)
- static void HRPWM_setFallingEdgeDelay (uint32_t base, uint32_t fedCount)
- static void HRPWM_setMEPStep (uint32_t base, uint16_t mepCount)
- static void HRPWM_setDeadbandMEPEdgeSelect (uint32_t base, HRPWM_MEPDeadBandEdgeMode mepDBEdge)
- static void HRPWM_setRisingEdgeDelayLoadMode (uint32_t base, HRPWM_LoadMode loadEvent)
- static void HRPWM_setFallingEdgeDelayLoadMode (uint32_t base, HRPWM_LoadMode loadEvent)
- static void HRPWM_lockRegisters (uint32_t base, HRPWM_LockRegisterGroup registerGroup)

19.2.1 Detailed Description

The code for this module is contained in `driverlib/hrpwm.c`, with `driverlib/hrpwm.h` containing the API declarations for use by applications.

19.2.2 Enumeration Type Documentation

19.2.2.1 enum **HRPWM_Channel**

Values that can be passed to HRPWM_setMEPEdgeSelect(), HRPWM_setMEPControlMode(), HRPWM_setCounterCompareShadowLoadEvent() as the *channel* parameter.

Enumerator

HRPWM_CHANNEL_A HRPWM A.

HRPWM_CHANNEL_B HRPWM B.

19.2.2.2 enum **HRPWM_MEPEdgeMode**

Values that can be passed to HRPWM_setMEPEdgeSelect() as the *mepEdgeMode* parameter.

Enumerator

HRPWM_MEP_CTRL_DISABLE HRPWM is disabled.

HRPWM_MEP_CTRL_RISING_EDGE MEP controls rising edge.

HRPWM_MEP_CTRL_FALLING_EDGE MEP controls falling edge.

HRPWM_MEP_CTRL_RISING_AND_FALLING_EDGE MEP controls both rising and falling edge.

19.2.2.3 enum **HRPWM_MEPCtrlMode**

Values that can be passed to HRPWM_setHRMEPCtrlMode() as the *parameter*.

Enumerator

HRPWM_MEP_DUTY_PERIOD_CTRL CMPAHR/CMPBHR or TBPRDHR controls MEP edge.

HRPWM_MEP_PHASE_CTRL TBPHSHR controls MEP edge.

19.2.2.4 enum **HRPWM_LoadMode**

Values that can be passed to HRPWM_setCounterCompareShadowLoadEvent(), HRPWM_setRisingEdgeDelayLoadMode() and HRPWM_setFallingEdgeDelayLoadMode as the *loadEvent* parameter.

Enumerator

HRPWM_LOAD_ON_CNTR_ZERO load when counter equals zero

HRPWM_LOAD_ON_CNTR_PERIOD load when counter equals period

HRPWM_LOAD_ON_CNTR_ZERO_PERIOD load when counter equals zero or period

19.2.2.5 enum **HRPWM_ChannelBOutput**

Values that can be passed to HRPWM_setChannelBOutputPath() as the *outputOnB* parameter.

Enumerator

HRPWM_OUTPUT_ON_B_NORMAL ePWMxB output is normal.

HRPWM_OUTPUT_ON_B_INV_A version of ePWMxA signal ePWMxB output is inverted

19.2.2.6 enum **HRPWM_SyncPulseSource**

Values that can be passed to HRPWM_setSyncPulseSource() as the *syncPulseSource* parameter.

Enumerator

HRPWM_PWMSYNC_SOURCE_PERIOD Counter equals Period.

HRPWM_PWMSYNC_SOURCE_ZERO Counter equals zero.
HRPWM_PWMSYNC_SOURCE_COMPC_UP Counter equals COMPC when counting up.
HRPWM_PWMSYNC_SOURCE_COMPC_DOWN Counter equals COMPC when counting down.
HRPWM_PWMSYNC_SOURCE_COMPD_UP Counter equals COMPD when counting up.
HRPWM_PWMSYNC_SOURCE_COMPD_DOWN Counter equals COMPD when counting down.

19.2.2.7 enum **HRPWM_CounterCompareModule**

Values that can be passed to HRPWM_setCounterCompareValue() as the *compModule* parameter.

Enumerator

HRPWM_COUNTER_COMPARE_A counter compare A
HRPWM_COUNTER_COMPARE_B counter compare B

19.2.2.8 enum **HRPWM_MEPDeadBandEdgeMode**

Values that can be passed to HRPWM_setDeadbandMEPEdgeSelect() as the *mepDBEdge*.

Enumerator

HRPWM_DB_MEP_CTRL_DISABLE HRPWM is disabled.
HRPWM_DB_MEP_CTRL_RED MEP controls Rising Edge Delay.
HRPWM_DB_MEP_CTRL_FED MEP controls Falling Edge Delay.
HRPWM_DB_MEP_CTRL_RED_FED MEP controls both Falling and Rising edge delay.

19.2.2.9 enum **HRPWM_LockRegisterGroup**

Values that can be passed to HRPWM_lockRegisters() as the *registerGroup* parameter.

Enumerator

HRPWM_REGISTER_GROUP_HRPWM HRPWM register group.
HRPWM_REGISTER_GROUP_GLOBAL_LOAD Global load register group.
HRPWM_REGISTER_GROUP_TRIP_ZONE Trip zone register group.
HRPWM_REGISTER_GROUP_TRIP_ZONE_CLEAR Trip zone clear group.
HRPWM_REGISTER_GROUP_DIGITAL_COMPARE Digital compare group.

19.2.3 Function Documentation

19.2.3.1 static void HRPWM_setPhaseShift (uint32_t *base*, uint32_t *phaseCount*) `[inline]`, `[static]`

Sets the high resolution phase shift value.

Parameters

base	is the base address of the EPWM module.
phaseCount	is the high resolution phase shift count value.

This function sets the high resolution phase shift value. Call the HRPWM_enableHRPhaseShiftLoad() function to enable loading of the phaseCount

Note: phaseCount is a 24 bit value

Returns
None.

19.2.3.2 static void HRPWM_setTimeBasePeriod (uint32_t *base,* uint32_t *periodCount*) [inline], [static]

Sets the period of the high resolution time base counter.

Parameters

base	is the base address of the EPWM module.
periodCount	is high resolution period count value.

This function sets the period of the high resolution time base counter. The value of periodCount is the value written to the register. User should map the desired period or frequency of the waveform into the correct periodCount.

Note: periodCount is a 24 bit value

Returns
None.

19.2.3.3 static uint32_t HRPWM_getTimeBasePeriod (uint32_t *base*) [inline], [static]

Gets the HRPWM period count.

Parameters

base	is the base address of the EPWM module.

This function gets the period of the HRPWM count.

Returns
The period count value.

19.2.3.4 static void HRPWM_setMEPEdgeSelect (uint32_t *base,* **HRPWM_Channel** *channel,* **HRPWM_MEPEdgeMode** *mepEdgeMode*) [inline], [static]

Sets the high resolution edge controlled by MEP (Micro Edge Positioner).

Parameters

base	is the base address of the EPWM module.
channel	is high resolution period module.
mepEdgeMode	edge of the PWM that is controlled by MEP (Micro Edge Positioner).

This function sets the edge of the PWM that is controlled by MEP (Micro Edge Positioner). Valid values for the parameters are: channel

- HRPWM_CHANNEL_A - HRPWM A
- HRPWM_CHANNEL_B - HRPWM B mepEdgeMode
- HRPWM_MEP_CTRL_DISABLE - HRPWM is disabled
- HRPWM_MEP_CTRL_RISING_EDGE - MEP (Micro Edge Positioner) controls rising edge.
- HRPWM_MEP_CTRL_FALLING_EDGE - MEP (Micro Edge Positioner) controls falling edge.
- HRPWM_MEP_CTRL_RISING_AND_FALLING_EDGE - MEP (Micro Edge Positioner) controls both edges.

Returns

None.

19.2.3.5 static void HRPWM_setMEPControlMode (uint32_t *base*, **HRPWM_Channel** *channel*, **HRPWM_MEPCtrlMode** *mepCtrlMode*) [inline], [static]

Sets the MEP (Micro Edge Positioner) control mode.

Parameters

base	is the base address of the EPWM module.
channel	is high resolution period module.
mepCtrlMode	is the MEP (Micro Edge Positioner) control mode.

This function sets the mode (register type) the MEP (Micro Edge Positioner) will control. Valid values for the parameters are: channel

- HRPWM_CHANNEL_A - HRPWM A
- HRPWM_CHANNEL_B - HRPWM B mepCtrlMode
- HRPWM_MEP_DUTY_PERIOD_CTRL - MEP (Micro Edge Positioner) is controled by value of CMPAHR/ CMPBHR(depedning on the value of channel) or TBPRDHR.
- HRPWM_MEP_PHASE_CTRL - MEP (Micro Edge Positioner) is controlled by TBPHSHR.

Returns

None.

19.2.3.6 static void HRPWM_setCounterCompareShadowLoadEvent (uint32_t *base*, **HRPWM_Channel** *channel*, **HRPWM_LoadMode** *loadEvent*) [inline], [static]

Sets the high resolution comparator load mode.

Parameters

base	is the base address of the EPWM module.
channel	is high resolution period module.
loadEvent	is the MEP (Micro Edge Positioner) control mode.

This function sets the shadow load mode of the high resolution comparator. The function sets the COMPA or COMPB register depending on the channel variable. Valid values for the parameters are: channel

- HRPWM_CHANNEL_A - HRPWM A
- HRPWM_CHANNEL_B - HRPWM B loadEvent
- HRPWM_LOAD_ON_CNTR_ZERO - load when counter equals zero
- HRPWM_LOAD_ON_CNTR_PERIOD - load when counter equals period
- HRPWM_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period

Returns
None.

19.2.3.7 static void HRPWM_setOutputSwapMode (uint32_t *base*, bool *enableOutputSwap*) [inline], [static]

Sets the high resolution output swap mode.

Parameters

base	is the base address of the EPWM module.
enableOutputSwap	is the output swap flag.

This function sets the HRPWM output swap mode. If enableOutputSwap is true, ePWMxA signal appears on ePWMxB output and ePWMxB signal appears on ePWMxA output. If it is false ePWMxA and ePWMxB outputs are unchanged

Returns
None.

19.2.3.8 static void HRPWM_setChannelBOutputPath (uint32_t *base*, **HRPWM_ChannelBOutput** *outputOnB*) [inline], [static]

Sets the high resolution output on ePWMxB

Parameters

base	is the base address of the EPWM module.
outputOnB	is the output signal on ePWMxB.

This function sets the HRPWM output signal on ePWMxB. If outputOnB is HRPWM_OUTPUT_ON_B_INV_A, ePWMxB output is an inverted version of ePWMxA. If outputOnB is HRPWM_OUTPUT_ON_B_NORMAL, ePWMxB output is ePWMxB.

Returns
None.

19.2.3.9 static void HRPWM_enableAutoConversion (uint32_t *base*) [inline], [static]

Enables MEP (Micro Edge Positioner) automatic scale mode.

Parameters

base	is the base address of the EPWM module.

This function enables the MEP (Micro Edge Positioner) to automatically scale HRMSTEP.

Returns
None.

19.2.3.10 static void HRPWM_disableAutoConversion (uint32_t *base*) [inline], [static]

Disables MEP automatic scale mode.

Parameters

base	is the base address of the EPWM module.

This function disables the MEP (Micro Edge Positioner) from automatically scaling HRMSTEP.

Returns
None.

19.2.3.11 static void HRPWM_enablePeriodControl (uint32_t *base*) [inline], [static]

Enable high resolution period feature.

Parameters

base	is the base address of the EPWM module.

This function enables the high resolution period feature.

Returns
None.

19.2.3.12 static void HRPWM_disablePeriodControl (uint32_t *base*) [inline], [static]

Disable high resolution period feature.

Parameters

base	is the base address of the EPWM module.

This function disables the high resolution period feature.

Returns

None.

19.2.3.13 static void HRPWM_enablePhaseShiftLoad (uint32_t *base*) [inline], [static]

Enable high resolution phase load

Parameters

base	is the base address of the EPWM module.

This function enables loading of high resolution phase shift value which is set by the function HRPWM_setPhaseShift().

Returns

None.

19.2.3.14 static void HRPWM_disablePhaseShiftLoad (uint32_t *base*) [inline], [static]

Disable high resolution phase load

Parameters

base	is the base address of the EPWM module.

This function disables loading of high resolution phase shift value.

Returns

19.2.3.15 static void HRPWM_setSyncPulseSource (uint32_t *base*, **HRPWM_SyncPulseSource** *syncPulseSource*) [inline], [static]

Set high resolution PWMSYNC source.

Parameters

base	is the base address of the EPWM module.
syncPulseSource	is the PWMSYNC source.

This function sets the high resolution PWMSYNC pulse source. Valid values for syncPulseSource are:

- HRPWM_PWMSYNC_SOURCE_PERIOD - Counter equals Period.

- HRPWM_PWMSYNC_SOURCE_ZERO - Counter equals zero.
- HRPWM_PWMSYNC_SOURCE_COMPC_UP - Counter equals COMPC when counting up.
- HRPWM_PWMSYNC_SOURCE_COMPC_DOWN - Counter equals COMPC when counting down.
- HRPWM_PWMSYNC_SOURCE_COMPD_UP - Counter equals COMPD when counting up.
- HRPWM_PWMSYNC_SOURCE_COMPD_DOWN - Counter equals COMPD when counting down.

Returns

None.

References HRPWM_PWMSYNC_SOURCE_COMPC_UP.

19.2.3.16 static void HRPWM_setCounterCompareValue (uint32_t *base*, **HRPWM_CounterCompareModule** *compModule*, uint32_t *compCount*) `[inline]`, `[static]`

Set high resolution counter compare values.

Parameters

base	is the base address of the EPWM module.
compModule	is the Compare value module.
compCount	is the counter compare count value.

This function sets the high resolution counter compare value for counter compare registers. Valid values for compModule are:

- HRPWM_COUNTER_COMPARE_A - counter compare A.
- HRPWM_COUNTER_COMPARE_B - counter compare B.

Note: compCount is a 24 bit value

Returns

None.

References HRPWM_COUNTER_COMPARE_A.

19.2.3.17 static uint32_t HRPWM_getCounterCompareValue (uint32_t *base*, **HRPWM_CounterCompareModule** *compModule*) `[inline]`, `[static]`

Gets high resolution counter compare values.

Parameters

base	is the base address of the EPWM module.
compModule	is the Compare value module.

This function gets the high resolution counter compare value for counter compare registers specified. Valid values for compModule are:

- HRPWM_COUNTER_COMPARE_A - counter compare A.

- HRPWM_COUNTER_COMPARE_B - counter compare B.

Returns
None.

References HRPWM_COUNTER_COMPARE_A.

19.2.3.18 static void HRPWM_setRisingEdgeDelay (uint32_t *base*, uint32_t *redCount*) [inline], [static]

Set High Resolution RED count

Parameters

base	is the base address of the EPWM module.
redCount	is the high resolution RED count.

This function sets the high resolution RED (Rising Edge Delay) count value. The value of redCount should be less than 0x200000.

Note: redCount is a 21 bit value

Returns
None.

19.2.3.19 static void HRPWM_setFallingEdgeDelay (uint32_t *base*, uint32_t *fedCount*) [inline], [static]

Set High Resolution FED count

Parameters

base	is the base address of the EPWM module.
fedCount	is the high resolution FED count.

This function sets the high resolution FED (Falling Edge Delay) count value. The value of fedCount should be less than 0x200000.

Note: fedCount is a 21 bit value

Returns
None.

19.2.3.20 static void HRPWM_setMEPStep (uint32_t *base*, uint16_t *mepCount*) [inline], [static]

Set high resolution MEP (Micro Edge Positioner) step.

Parameters

base	is the base address of the EPWM module.
mepCount	is the high resolution MEP (Micro Edge Positioner) step count.

This function sets the high resolution MEP (Micro Edge Positioner) step count. The maximum value for the MEP count step is 255.

Returns
None.

19.2.3.21 static void HRPWM_setDeadbandMEPEdgeSelect (uint32_t *base,* **HRPWM_MEPDeadBandEdgeMode** *mepDBEdge*) [inline], [static]

Set high resolution Dead Band MEP (Micro Edge Positioner) control.

Parameters

base	is the base address of the EPWM module.
mepDBEdge	is the high resolution MEP (Micro Edge Positioner) control edge.

This function sets the high resolution Dead Band edge that the MEP (Micro Edge Positioner) controls Valid values for mepDBEdge are:

- HRPWM_DB_MEP_CTRL_DISABLE - HRPWM is disabled
- HRPWM_DB_MEP_CTRL_RED - MEP (Micro Edge Positioner) controls Rising Edge Delay
- HRPWM_DB_MEP_CTRL_FED - MEP (Micro Edge Positioner) controls Falling Edge Delay
- HRPWM_DB_MEP_CTRL_RED_FED - MEP (Micro Edge Positioner) controls both Falling and Rising edge delays

Returns
None.

19.2.3.22 static void HRPWM_setRisingEdgeDelayLoadMode (uint32_t *base,* **HRPWM_LoadMode** *loadEvent*) [inline], [static]

Set the high resolution Dead Band RED load mode.

Parameters

base	is the base address of the EPWM module.
loadEvent	is the shadow to active load event.

This function sets the high resolution Rising Edge Delay(RED)Dead Band count load mode. Valid values for loadEvent are:

- HRPWM_LOAD_ON_CNTR_ZERO - load when counter equals zero.
- HRPWM_LOAD_ON_CNTR_PERIOD - load when counter equals period
- HRPWM_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period.

Returns
None.

19.2.3.23 static void HRPWM_setFallingEdgeDelayLoadMode (uint32_t *base*, **HRPWM_LoadMode** *loadEvent*) `[inline]`, `[static]`

Set the high resolution Dead Band FED load mode.

Parameters

base	is the base address of the EPWM module.
loadEvent	is the shadow to active load event.

This function sets the high resolution Falling Edge Delay(FED) Dead Band count load mode. Valid values for loadEvent are:

- HRPWM_LOAD_ON_CNTR_ZERO - load when counter equals zero.
- HRPWM_LOAD_ON_CNTR_PERIOD - load when counter equals period
- HRPWM_LOAD_ON_CNTR_ZERO_PERIOD - load when counter equals zero or period.

Returns
None.

19.2.3.24 static void HRPWM_lockRegisters (uint32_t *base*, **HRPWM_LockRegisterGroup** *registerGroup*) [inline], [static]

Lock EALLOW protected register groups

Parameters

base	is the base address of the EPWM module.
registerGroup	is the EALLOW register groups.

This functions locks the EALLOW protected register groups specified by the registerGroup variable.

Returns
None.

20 EQEP Module

Introduction 327
API Functions 327

20.1 EQEP Introduction

The enhanced quadrature encoder pulse (eQEP) API provides a set of functions to configure an interface to an encoder. The functions provide the ability to configure the device's eQEP module to properly decode incoming pulse signals, to configure module outputs, and to get direction, position, and speed information. There are also APIs to setup the possible interrupt events that the module can generate.

20.2 API Functions

Enumerations

- enum EQEP_PositionResetMode { EQEP_POSITION_RESET_IDX, EQEP_POSITION_RESET_MAX_POS, EQEP_POSITION_RESET_1ST_IDX, EQEP_POSITION_RESET_UNIT_TIME_OUT }
- enum EQEP_CAPCLKPrescale { EQEP_CAPTURE_CLK_DIV_1, EQEP_CAPTURE_CLK_DIV_2, EQEP_CAPTURE_CLK_DIV_4, EQEP_CAPTURE_CLK_DIV_8, EQEP_CAPTURE_CLK_DIV_16, EQEP_CAPTURE_CLK_DIV_32, EQEP_CAPTURE_CLK_DIV_64, EQEP_CAPTURE_CLK_DIV_128 }
- enum EQEP_UPEVNTPrescale { EQEP_UNIT_POS_EVNT_DIV_1, EQEP_UNIT_POS_EVNT_DIV_2, EQEP_UNIT_POS_EVNT_DIV_4, EQEP_UNIT_POS_EVNT_DIV_8, EQEP_UNIT_POS_EVNT_DIV_16, EQEP_UNIT_POS_EVNT_DIV_32, EQEP_UNIT_POS_EVNT_DIV_64, EQEP_UNIT_POS_EVNT_DIV_128, EQEP_UNIT_POS_EVNT_DIV_256, EQEP_UNIT_POS_EVNT_DIV_512, EQEP_UNIT_POS_EVNT_DIV_1024, EQEP_UNIT_POS_EVNT_DIV_2048 }
- enum EQEP_StrobeSource { EQEP_STROBE_FROM_GPIO, EQEP_STROBE_OR_ADCSOCA, EQEP_STROBE_OR_ADCSOCB }
- enum EQEP_QMAMode { EQEP_QMA_MODE_BYPASS, EQEP_QMA_MODE_1, EQEP_QMA_MODE_2 }
- enum EQEP_EmulationMode { EQEP_EMULATIONMODE_STOPIMMEDIATELY, EQEP_EMULATIONMODE_STOPATROLLOVER, EQEP_EMULATIONMODE_RUNFREE }

Functions

- static void EQEP_enableModule (uint32_t base)
- static void EQEP_disableModule (uint32_t base)
- static void EQEP_setDecoderConfig (uint32_t base, uint16_t config)
- static void EQEP_setPositionCounterConfig (uint32_t base, EQEP_PositionResetMode mode, uint32_t maxPosition)

- static uint32_t EQEP_getPosition (uint32_t base)
- static void EQEP_setPosition (uint32_t base, uint32_t position)
- static int16_t EQEP_getDirection (uint32_t base)
- static void EQEP_enableInterrupt (uint32_t base, uint16_t intFlags)
- static void EQEP_disableInterrupt (uint32_t base, uint16_t intFlags)
- static uint16_t EQEP_getInterruptStatus (uint32_t base)
- static void EQEP_clearInterruptStatus (uint32_t base, uint16_t intFlags)
- static void EQEP_forceInterrupt (uint32_t base, uint16_t intFlags)
- static bool EQEP_getError (uint32_t base)
- static uint16_t EQEP_getStatus (uint32_t base)
- static void EQEP_clearStatus (uint32_t base, uint16_t statusFlags)
- static void EQEP_setCaptureConfig (uint32_t base, EQEP_CAPCLKPrescale capPrescale, EQEP_UPEVNTPrescale evntPrescale)
- static void EQEP_enableCapture (uint32_t base)
- static void EQEP_disableCapture (uint32_t base)
- static uint16_t EQEP_getCapturePeriod (uint32_t base)
- static uint16_t EQEP_getCaptureTimer (uint32_t base)
- static void EQEP_enableCompare (uint32_t base)
- static void EQEP_disableCompare (uint32_t base)
- static void EQEP_setComparePulseWidth (uint32_t base, uint16_t cycles)
- static void EQEP_enableUnitTimer (uint32_t base, uint32_t period)
- static void EQEP_disableUnitTimer (uint32_t base)
- static void EQEP_enableWatchdog (uint32_t base, uint16_t period)
- static void EQEP_disableWatchdog (uint32_t base)
- static void EQEP_setWatchdogTimerValue (uint32_t base, uint16_t value)
- static uint16_t EQEP_getWatchdogTimerValue (uint32_t base)
- static void EQEP_setPositionInitMode (uint32_t base, uint16_t initMode)
- static void EQEP_setSWPositionInit (uint32_t base, bool initialize)
- static void EQEP_setInitialPosition (uint32_t base, uint32_t position)
- static void EQEP_setLatchMode (uint32_t base, uint32_t latchMode)
- static uint32_t EQEP_getIndexPositionLatch (uint32_t base)
- static uint32_t EQEP_getStrobePositionLatch (uint32_t base)
- static uint32_t EQEP_getPositionLatch (uint32_t base)
- static uint16_t EQEP_getCaptureTimerLatch (uint32_t base)
- static uint16_t EQEP_getCapturePeriodLatch (uint32_t base)
- static void EQEP_setQMAModuleMode (uint32_t base, EQEP_QMAMode qmaMode)
- static void EQEP_setStrobeSource (uint32_t base, EQEP_StrobeSource strobeSrc)
- static void EQEP_setEmulationMode (uint32_t base, EQEP_EmulationMode emuMode)
- void EQEP_setCompareConfig (uint32_t base, uint16_t config, uint32_t compareValue, uint16_t cycles)
- void EQEP_setInputPolarity (uint32_t base, bool invertQEPA, bool invertQEPB, bool invertIndex, bool invertStrobe)

20.2.1 Detailed Description

The code for this module is contained in `driverlib/eqep.c`, with `driverlib/eqep.h` containing the API declarations for use by applications.

20.2.2 Enumeration Type Documentation

20.2.2.1 enum **EQEP_PositionResetMode**

Values that can be passed to EQEP_setPositionCounterConfig() as the *mode* parameter.

Enumerator

EQEP_POSITION_RESET_IDX Reset position on index pulse.
EQEP_POSITION_RESET_MAX_POS Reset position on maximum position.
EQEP_POSITION_RESET_1ST_IDX Reset position on the first index pulse.
EQEP_POSITION_RESET_UNIT_TIME_OUT Reset position on a unit time event.

20.2.2.2 enum **EQEP_CAPCLKPrescale**

Values that can be passed to EQEP_setCaptureConfig() as the *capPrescale* parameter. CAPCLK is the capture timer clock frequency.

Enumerator

EQEP_CAPTURE_CLK_DIV_1 CAPCLK = SYSCLKOUT/1.
EQEP_CAPTURE_CLK_DIV_2 CAPCLK = SYSCLKOUT/2.
EQEP_CAPTURE_CLK_DIV_4 CAPCLK = SYSCLKOUT/4.
EQEP_CAPTURE_CLK_DIV_8 CAPCLK = SYSCLKOUT/8.
EQEP_CAPTURE_CLK_DIV_16 CAPCLK = SYSCLKOUT/16.
EQEP_CAPTURE_CLK_DIV_32 CAPCLK = SYSCLKOUT/32.
EQEP_CAPTURE_CLK_DIV_64 CAPCLK = SYSCLKOUT/64.
EQEP_CAPTURE_CLK_DIV_128 CAPCLK = SYSCLKOUT/128.

20.2.2.3 enum **EQEP_UPEVNTPrescale**

Values that can be passed to EQEP_setCaptureConfig() as the *evntPrescale* parameter. UPEVNT is the unit position event frequency.

Enumerator

EQEP_UNIT_POS_EVNT_DIV_1 UPEVNT = QCLK/1.
EQEP_UNIT_POS_EVNT_DIV_2 UPEVNT = QCLK/2.
EQEP_UNIT_POS_EVNT_DIV_4 UPEVNT = QCLK/4.
EQEP_UNIT_POS_EVNT_DIV_8 UPEVNT = QCLK/8.
EQEP_UNIT_POS_EVNT_DIV_16 UPEVNT = QCLK/16.
EQEP_UNIT_POS_EVNT_DIV_32 UPEVNT = QCLK/32.
EQEP_UNIT_POS_EVNT_DIV_64 UPEVNT = QCLK/64.
EQEP_UNIT_POS_EVNT_DIV_128 UPEVNT = QCLK/128.
EQEP_UNIT_POS_EVNT_DIV_256 UPEVNT = QCLK/256.
EQEP_UNIT_POS_EVNT_DIV_512 UPEVNT = QCLK/512.
EQEP_UNIT_POS_EVNT_DIV_1024 UPEVNT = QCLK/1024.
EQEP_UNIT_POS_EVNT_DIV_2048 UPEVNT = QCLK/2048.

20.2.2.4 enum **EQEP_StrobeSource**

Values that can be passed to EQEP_setStrobeSource() as the *strobeSrc* parameter.

Enumerator

EQEP_STROBE_FROM_GPIO Strobe signal comes from GPIO.

EQEP_STROBE_OR_ADCSOCA Strobe signal is OR'd with ADCSOCA.
EQEP_STROBE_OR_ADCSOCB Strobe signal is OR'd with ADCSOCB.

20.2.2.5 enum **EQEP_QMAMode**

Values that can be passed to EQEP_setQMAModuleMode() as the *qmaMode* parameter.

Enumerator

EQEP_QMA_MODE_BYPASS QMA module is bypassed.
EQEP_QMA_MODE_1 QMA mode-1 operation is selected.
EQEP_QMA_MODE_2 QMA mode-2 operation is selected.

20.2.2.6 enum **EQEP_EmulationMode**

Values that can be passed to EQEP_setEmulationMode() as the *emuMode* parameter.

Enumerator

EQEP_EMULATIONMODE_STOPIMMEDIATELY Counters stop immediately.
EQEP_EMULATIONMODE_STOPATROLLOVER Counters stop at period rollover.
EQEP_EMULATIONMODE_RUNFREE Counter unaffected by suspend.

20.2.3 Function Documentation

20.2.3.1 static void EQEP_enableModule (uint32_t *base*) `[inline]`, `[static]`

Enables the eQEP module.

Parameters

base	is the base address of the eQEP module.

This function enables operation of the enhanced quadrature encoder pulse (eQEP) module. The module must be configured before it is enabled.

See Also

EQEP_setConfig()

Returns

None.

20.2.3.2 static void EQEP_disableModule (uint32_t *base*) `[inline]`, `[static]`

Disables the eQEP module.

Parameters

base	is the base address of the enhanced quadrature encoder pulse (eQEP) module

This function disables operation of the eQEP module.

Returns
None.

20.2.3.3 static void EQEP_setDecoderConfig (uint32_t *base,* uint16_t *config*) [inline], [static]

Configures eQEP module's quadrature decoder unit.

Parameters

base	is the base address of the eQEP module.
config	is the configuration for the eQEP module decoder unit.

This function configures the operation of the eQEP module's quadrature decoder unit. The *config* parameter provides the configuration of the decoder and is the logical OR of several values:

- **EQEP_CONFIG_2X_RESOLUTION** or **EQEP_CONFIG_1X_RESOLUTION** specify if both rising and falling edges should be counted or just rising edges.
- **EQEP_CONFIG_QUADRATURE**, **EQEP_CONFIG_CLOCK_DIR**, **EQEP_CONFIG_UP_COUNT**, or **EQEP_CONFIG_DOWN_COUNT** specify if quadrature signals are being provided on QEPA and QEPB, if a direction signal and a clock are being provided, or if the direction should be hard-wired for a single direction with QEPA used for input.
- **EQEP_CONFIG_NO_SWAP** or **EQEP_CONFIG_SWAP** to specify if the signals provided on QEPA and QEPB should be swapped before being processed.

Returns
None.

20.2.3.4 static void EQEP_setPositionCounterConfig (uint32_t *base,* **EQEP_PositionResetMode** *mode,* uint32_t *maxPosition*) [inline], [static]

Configures eQEP module position counter unit.

Parameters

base	is the base address of the eQEP module.
mode	is the configuration for the eQEP module position counter.
maxPosition	specifies the maximum position value.

This function configures the operation of the eQEP module position counter. The *mode* parameter determines the event on which the position counter gets reset. It should be passed one of the following values: **EQEP_POSITION_RESET_IDX**, **EQEP_POSITION_RESET_MAX_POS**, **EQEP_POSITION_RESET_1ST_IDX**, or **EQEP_POSITION_RESET_UNIT_TIME_OUT**.

maxPosition is the maximum value of the position counter and is the value used to reset the position capture when moving in the reverse (negative) direction.

Returns
None.

20.2.3.5 static uint32_t EQEP_getPosition (uint32_t *base*) [inline], [static]

Gets the current encoder position.

Parameters

base	is the base address of the eQEP module.

This function returns the current position of the encoder. Depending upon the configuration of the encoder, and the incident of an index pulse, this value may or may not contain the expected data (that is, if in reset on index mode, if an index pulse has not been encountered, the position counter is not yet aligned with the index pulse).

Returns
The current position of the encoder.

20.2.3.6 static void EQEP_setPosition (uint32_t *base,* uint32_t *position*) [inline], [static]

Sets the current encoder position.

Parameters

base	is the base address of the eQEP module.
position	is the new position for the encoder.

This function sets the current position of the encoder; the encoder position is then measured relative to this value.

Returns
None.

20.2.3.7 static int16_t EQEP_getDirection (uint32_t *base*) [inline], [static]

Gets the current direction of rotation.

Parameters

base	is the base address of the eQEP module.

This function returns the current direction of rotation. In this case, current means the most recently detected direction of the encoder; it may not be presently moving but this is the direction it last moved before it stopped.

Returns
Returns 1 if moving in the forward direction or -1 if moving in the reverse direction.

20.2.3.8 static void EQEP_enableInterrupt (uint32_t *base*, uint16_t *intFlags*) [inline], [static]

Enables individual eQEP module interrupt sources.

Parameters

base	is the base address of the eQEP module.
intFlags	is a bit mask of the interrupt sources to be enabled.

This function enables eQEP module interrupt sources. The *intFlags* parameter can be any of the following values OR'd together:

- **EQEP_INT_POS_CNT_ERROR** - Position counter error
- **EQEP_INT_PHASE_ERROR** - Quadrature phase error
- **EQEP_INT_DIR_CHANGE** - Quadrature direction change
- **EQEP_INT_WATCHDOG** - Watchdog time-out
- **EQEP_INT_UNDERFLOW** - Position counter underflow
- **EQEP_INT_OVERFLOW** - Position counter overflow
- **EQEP_INT_POS_COMP_READY** - Position-compare ready
- **EQEP_INT_POS_COMP_MATCH** - Position-compare match
- **EQEP_INT_STROBE_EVNT_LATCH** - Strobe event latch
- **EQEP_INT_INDEX_EVNT_LATCH** - Index event latch
- **EQEP_INT_UNIT_TIME_OUT** - Unit time-out
- **EQEP_INT_QMA_ERROR** - QMA error

Returns
None.

20.2.3.9 static void EQEP_disableInterrupt (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

Disables individual eQEP module interrupt sources.

Parameters

base	is the base address of the eQEP module.
intFlags	is a bit mask of the interrupt sources to be disabled.

This function disables eQEP module interrupt sources. The *intFlags* parameter can be any of the following values OR'd together:

- **EQEP_INT_POS_CNT_ERROR** - Position counter error
- **EQEP_INT_PHASE_ERROR** - Quadrature phase error
- **EQEP_INT_DIR_CHANGE** - Quadrature direction change
- **EQEP_INT_WATCHDOG** - Watchdog time-out
- **EQEP_INT_UNDERFLOW** - Position counter underflow
- **EQEP_INT_OVERFLOW** - Position counter overflow
- **EQEP_INT_POS_COMP_READY** - Position-compare ready
- **EQEP_INT_POS_COMP_MATCH** - Position-compare match
- **EQEP_INT_STROBE_EVNT_LATCH** - Strobe event latch
- **EQEP_INT_INDEX_EVNT_LATCH** - Index event latch
- **EQEP_INT_UNIT_TIME_OUT** - Unit time-out
- **EQEP_INT_QMA_ERROR** - QMA error

Returns
None.

20.2.3.10 static uint16_t EQEP_getInterruptStatus (uint32_t *base*) [inline], [static]

Gets the current interrupt status.

Parameters

base	is the base address of the eQEP module.

This function returns the interrupt status for the eQEP module module.

Returns
Returns the current interrupt status, enumerated as a bit field of the following values:
- **EQEP_INT_GLOBAL** - Global interrupt flag
- **EQEP_INT_POS_CNT_ERROR** - Position counter error
- **EQEP_INT_PHASE_ERROR** - Quadrature phase error
- **EQEP_INT_DIR_CHANGE** - Quadrature direction change
- **EQEP_INT_WATCHDOG** - Watchdog time-out
- **EQEP_INT_UNDERFLOW** - Position counter underflow
- **EQEP_INT_OVERFLOW** - Position counter overflow
- **EQEP_INT_POS_COMP_READY** - Position-compare ready
- **EQEP_INT_POS_COMP_MATCH** - Position-compare match
- **EQEP_INT_STROBE_EVNT_LATCH** - Strobe event latch
- **EQEP_INT_INDEX_EVNT_LATCH** - Index event latch
- **EQEP_INT_UNIT_TIME_OUT** - Unit time-out
- **EQEP_INT_QMA_ERROR** - QMA error

20.2.3.11 static void EQEP_clearInterruptStatus (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

Clears eQEP module interrupt sources.

Parameters

base	is the base address of the eQEP module.
intFlags	is a bit mask of the interrupt sources to be cleared.

This function clears eQEP module interrupt flags. The *intFlags* parameter can be any of the following values OR'd together:

- **EQEP_INT_GLOBAL** - Global interrupt flag
- **EQEP_INT_POS_CNT_ERROR** - Position counter error
- **EQEP_INT_PHASE_ERROR** - Quadrature phase error
- **EQEP_INT_DIR_CHANGE** - Quadrature direction change
- **EQEP_INT_WATCHDOG** - Watchdog time-out
- **EQEP_INT_UNDERFLOW** - Position counter underflow

- **EQEP_INT_OVERFLOW** - Position counter overflow
- **EQEP_INT_POS_COMP_READY** - Position-compare ready
- **EQEP_INT_POS_COMP_MATCH** - Position-compare match
- **EQEP_INT_STROBE_EVNT_LATCH** - Strobe event latch
- **EQEP_INT_INDEX_EVNT_LATCH** - Index event latch
- **EQEP_INT_UNIT_TIME_OUT** - Unit time-out
- **EQEP_INT_QMA_ERROR** - QMA error

Note that the **EQEP_INT_GLOBAL** value is the global interrupt flag. In order to get any further eQEP interrupts, this flag must be cleared.

Returns
None.

20.2.3.12 static void EQEP_forceInterrupt (uint32_t *base,* uint16_t *intFlags*) [inline], [static]

Forces individual eQEP module interrupts.

Parameters

base	is the base address of the eQEP module.
intFlags	is a bit mask of the interrupt sources to be forced.

This function forces eQEP module interrupt flags. The *intFlags* parameter can be any of the following values OR'd together:

- **EQEP_INT_POS_CNT_ERROR**
- **EQEP_INT_PHASE_ERROR**
- **EQEP_INT_DIR_CHANGE**
- **EQEP_INT_WATCHDOG**
- **EQEP_INT_UNDERFLOW**
- **EQEP_INT_OVERFLOW**
- **EQEP_INT_POS_COMP_READY**
- **EQEP_INT_POS_COMP_MATCH**
- **EQEP_INT_STROBE_EVNT_LATCH**
- **EQEP_INT_INDEX_EVNT_LATCH**
- **EQEP_INT_UNIT_TIME_OUT**
- **EQEP_INT_QMA_ERROR**

Returns
None.

20.2.3.13 static bool EQEP_getError (uint32_t *base*) [inline], [static]

Gets the encoder error indicator.

Parameters

base	is the base address of the eQEP module.

This function returns the error indicator for the eQEP module. It is an error for both of the signals of the quadrature input to change at the same time.

Returns

Returns **true** if an error has occurred and **false** otherwise.

20.2.3.14 static uint16_t EQEP_getStatus (uint32_t *base*) [inline], [static]

Returns content of the eQEP module status register

Parameters

base	is the base address of the eQEP module.

This function returns the contents of the status register. The value it returns is an OR of the following values:

- **EQEP_STS_UNIT_POS_EVNT** - Unit position event detected
- **EQEP_STS_DIR_ON_1ST_IDX** - If set, clockwise rotation (forward movement) occurred on the first index event
- **EQEP_STS_DIR_FLAG** - If set, movement is clockwise rotation
- **EQEP_STS_DIR_LATCH** - If set, clockwise rotation occurred on last index event marker
- **EQEP_STS_CAP_OVRFLW_ERROR** - Overflow occurred in eQEP capture timer
- **EQEP_STS_CAP_DIR_ERROR** - Direction change occurred between position capture events
- **EQEP_STS_1ST_IDX_FLAG** - Set by the occurrence of the first index pulse
- **EQEP_STS_POS_CNT_ERROR** - Position counter error occurred

Returns

Returns the value of the QEP status register.

20.2.3.15 static void EQEP_clearStatus (uint32_t *base,* uint16_t *statusFlags*) [inline], [static]

Clears selected fields of the eQEP module status register

Parameters

base	is the base address of the eQEP module.
statusFlags	is the bit mask of the status flags to be cleared.

This function clears the status register fields indicated by *statusFlags*. The *statusFlags* parameter is the logical OR of any of the following:

- **EQEP_STS_UNIT_POS_EVNT** - Unit position event detected
- **EQEP_STS_CAP_OVRFLW_ERROR** - Overflow occurred in eQEP capture timer

- **EQEP_STS_CAP_DIR_ERROR** - Direction change occurred between position capture events
- **EQEP_STS_1ST_IDX_FLAG** - Set by the occurrence of the first index pulse

Note

Only the above status fields can be cleared. All others are read-only, non-sticky fields.

Returns

None.

20.2.3.16 static void EQEP_setCaptureConfig (uint32_t *base,* **EQEP_CAPCLKPrescale** *capPrescale,* **EQEP_UPEVNTPrescale** *evntPrescale*) [inline], [static]

Configures eQEP module edge-capture unit.

Parameters

base	is the base address of the eQEP module.
capPrescale	is the prescaler setting of the eQEP capture timer clk.
evntPrescale	is the prescaler setting of the unit position event frequency.

This function configures the operation of the eQEP module edge-capture unit. The *capPrescale* parameter provides the configuration of the eQEP capture timer clock rate. It determines by which power of 2 between 1 and 128 inclusive SYSCLKOUT is divided. The macros for this parameter are in the format of EQEP_CAPTURE_CLK_DIV_X, where X is the divide value. For example, **EQEP_CAPTURE_CLK_DIV_32** will give a capture timer clock frequency that is SYSCLKOUT/32.

The *evntPrescale* parameter determines how frequently a unit position event occurs. The macro that can be passed this parameter is in the format EQEP_UNIT_POS_EVNT_DIV_X, where X is the number of quadrature clock periods between unit position events. For example, **EQEP_UNIT_POS_EVNT_DIV_16** will result in a unit position event frequency of QCLK/16.

Returns

None.

20.2.3.17 static void EQEP_enableCapture (uint32_t *base*) [inline], [static]

Enables the eQEP module edge-capture unit.

Parameters

base	is the base address of the eQEP module.

This function enables operation of the eQEP module's edge-capture unit.

Returns

None.

20.2.3.18 static void EQEP_disableCapture (uint32_t *base*) [inline], [static]

Disables the eQEP module edge-capture unit.

Parameters

base	is the base address of the eQEP module.

This function disables operation of the eQEP module's edge-capture unit.

Returns

None.

20.2.3.19 static uint16_t EQEP_getCapturePeriod (uint32_t *base*) [inline], [static]

Gets the encoder capture period.

Parameters

base	is the base address of the eQEP module.

This function returns the period count value between the last successive eQEP position events.

Returns

The period count value between the last successive position events.

20.2.3.20 static uint16_t EQEP_getCaptureTimer (uint32_t *base*) [inline], [static]

Gets the encoder capture timer value.

Parameters

base	is the base address of the eQEP module.

This function returns the time base for the edge capture unit.

Returns

The capture timer value.

20.2.3.21 static void EQEP_enableCompare (uint32_t *base*) [inline], [static]

Enables the eQEP module position-compare unit.

Parameters

base	is the base address of the eQEP module.

This function enables operation of the eQEP module's position-compare unit.

Returns

None.

20.2.3.22 static void EQEP_disableCompare (uint32_t *base*) [inline], [static]

Disables the eQEP module position-compare unit.

Parameters

base	is the base address of the eQEP module.

This function disables operation of the eQEP module's position-compare unit.

Returns
None.

20.2.3.23 static void EQEP_setComparePulseWidth (uint32_t *base,* uint16_t *cycles*) [inline], [static]

Configures the position-compare unit's sync output pulse width.

Parameters

base	is the base address of the eQEP module.
cycles	is the width of the pulse that can be generated on a position-compare event. It is in units of 4 SYSCLKOUT cycles.

This function configures the width of the sync output pulse. The width of the pulse will be *cycles* $*$ 4 $*$ the width of a SYSCLKOUT cycle. The maximum width is 4096 $*$ 4 $*$ SYSCLKOUT cycles.

Returns
None.

20.2.3.24 static void EQEP_enableUnitTimer (uint32_t *base,* uint32_t *period*) [inline], [static]

Enables the eQEP module unit timer.

Parameters

base	is the base address of the eQEP module.
period	is period value at which a unit time-out interrupt is set.

This function enables operation of the eQEP module's peripheral unit timer. The unit timer is clocked by SYSCLKOUT and will set the unit time-out interrupt when it matches the value specified by *period*.

Returns
None.

20.2.3.25 static void EQEP_disableUnitTimer (uint32_t *base*) [inline], [static]

Disables the eQEP module unit timer.

Parameters

base	is the base address of the eQEP module.

This function disables operation of the eQEP module's peripheral unit timer.

Returns
None.

20.2.3.26 static void EQEP_enableWatchdog (uint32_t *base*, uint16_t *period*) [inline], [static]

Enables the eQEP module watchdog timer.

Parameters

base	is the base address of the eQEP module.
period	is watchdog period value at which a time-out will occur if no quadrature-clock event is detected.

This function enables operation of the eQEP module's peripheral watchdog timer.

Note
When selecting *period*, note that the watchdog timer is clocked from SYSCLKOUT/64.

Returns
None.

20.2.3.27 static void EQEP_disableWatchdog (uint32_t *base*) [inline], [static]

Disables the eQEP module watchdog timer.

Parameters

base	is the base address of the eQEP module.

This function disables operation of the eQEP module's peripheral watchdog timer.

Returns
None.

20.2.3.28 static void EQEP_setWatchdogTimerValue (uint32_t *base*, uint16_t *value*) [inline], [static]

Sets the eQEP module watchdog timer value.

Parameters

base	is the base address of the eQEP module.

value	is the value to be written to the watchdog timer.

This function sets the eQEP module's watchdog timer value.

Returns
None.

20.2.3.29 static uint16_t EQEP_getWatchdogTimerValue (uint32_t *base*) [inline], [static]

Gets the eQEP module watchdog timer value.

Parameters

base	is the base address of the eQEP module.

Returns
Returns the current watchdog timer value.

20.2.3.30 static void EQEP_setPositionInitMode (uint32_t *base,* uint16_t *initMode*) [inline], [static]

Configures the mode in which the position counter is initialized.

Parameters

base	is the base address of the eQEP module.
initMode	is the configuration for initializing the position count. See below for a description of this parameter.

This function configures the events on which the position count can be initialized. The *initMode* parameter provides the mode as either **EQEP_INIT_DO_NOTHING** (no action configured) or one of the following strobe events, index events, or a logical OR of both a strobe event and an index event.

- **EQEP_INIT_RISING_STROBE** or **EQEP_INIT_EDGE_DIR_STROBE** specify which strobe event will initialize the position counter.
- **EQEP_INIT_RISING_INDEX** or **EQEP_INIT_FALLING_INDEX** specify which index event will initialize the position counter.

Use EQEP_setSWPositionInit() to cause a software initialization and EQEP_setInitialPosition() to set the value that gets loaded into the position counter upon initialization.

Returns
None.

20.2.3.31 static void EQEP_setSWPositionInit (uint32_t *base,* bool *initialize*) [inline], [static]

Sets the software initialization of the encoder position counter.

Parameters

base	is the base address of the eQEP module.
initialize	is a flag to specify if software initialization of the position counter is enabled.

This function does a software initialization of the position counter when the *initialize* parameter is **true**. When **false**, the QEPCTL[SWI] bit is cleared and no action is taken.

The init value to be loaded into the position counter can be set with EQEP_setInitialPosition(). Additional initialization causes can be configured with EQEP_setPositionInitMode().

Returns

None.

20.2.3.32 static void EQEP_setInitialPosition (uint32_t *base,* uint32_t *position*) [inline], [static]

Sets the init value for the encoder position counter.

Parameters

base	is the base address of the eQEP module.
position	is the value to be written to the position counter upon. initialization.

This function sets the init value for position of the encoder. See EQEP_setPositionInitMode() to set the initialization cause or EQEP_setSWPositionInit() to cause a software initialization.

Returns

None.

20.2.3.33 static void EQEP_setLatchMode (uint32_t *base,* uint32_t *latchMode*) [inline], [static]

Configures the quadrature modes in which the position count can be latched.

Parameters

base	is the base address of the eQEP module.
latchMode	is the configuration for latching of the position count and several other registers. See below for a description of this parameter.

This function configures the events on which the position count and several other registers can be latched. The *latchMode* parameter provides the mode as the logical OR of several values.

- **EQEP_LATCH_CNT_READ_BY_CPU** or **EQEP_LATCH_UNIT_TIME_OUT** specify the event that latches the position counter. This latch register can be read using EQEP_getPositionLatch(). The capture timer and capture period are also latched based on this setting, and can be read using EQEP_getCaptureTimerLatch() and EQEP_getCapturePeriodLatch().
- **EQEP_LATCH_RISING_STROBE** or **EQEP_LATCH_EDGE_DIR_STROBE** specify which strobe event will latch the position counter into the strobe position latch register. This register can be read with EQEP_getStrobePositionLatch().

- **EQEP_LATCH_RISING_INDEX**, **EQEP_LATCH_FALLING_INDEX**, or **EQEP_LATCH_SW_INDEX_MARKER** specify which index event will latch the position counter into the index position latch register. This register can be read with EQEP_getIndexPositionLatch().

Returns
None.

20.2.3.34 static uint32_t EQEP_getIndexPositionLatch (uint32_t *base*) [inline], [static]

Gets the encoder position that was latched on an index event.

Parameters

base	is the base address of the eQEP module.

This function returns the value in the index position latch register. The position counter is latched into this register on either a rising index edge, a falling index edge, or a software index marker. This is configured using EQEP_setLatchMode().

Returns
The position count latched on an index event.

20.2.3.35 static uint32_t EQEP_getStrobePositionLatch (uint32_t *base*) [inline], [static]

Gets the encoder position that was latched on a strobe event.

Parameters

base	is the base address of the eQEP module.

This function returns the value in the strobe position latch register. The position counter can be configured to be latched into this register on rising strobe edges only or on rising strobe edges while moving clockwise and falling strobe edges while moving counter-clockwise. This is configured using EQEP_setLatchMode().

Returns
The position count latched on a strobe event.

20.2.3.36 static uint32_t EQEP_getPositionLatch (uint32_t *base*) [inline], [static]

Gets the encoder position that was latched on a unit time-out event.

Parameters

base	is the base address of the eQEP module.

This function returns the value in the position latch register. The position counter is latched into this register either on a unit time-out event.

Returns

The position count latch register value.

20.2.3.37 static uint16_t EQEP_getCaptureTimerLatch (uint32_t *base*) [inline], [static]

Gets the encoder capture timer latch.

Parameters

base	is the base address of the eQEP module.

This function returns the value in the capture timer latch register. The capture timer value is latched into this register either on a unit time-out event or upon the CPU reading the eQEP position counter. This is configured using EQEP_setLatchMode().

Returns

The edge-capture timer latch value.

20.2.3.38 static uint16_t EQEP_getCapturePeriodLatch (uint32_t *base*) [inline], [static]

Gets the encoder capture period latch.

Parameters

base	is the base address of the eQEP module.

This function returns the value in the capture period latch register. The capture period value is latched into this register either on a unit time-out event or upon the CPU reading the eQEP position counter. This is configured using EQEP_setLatchMode().

Returns

The edge-capture period latch value.

20.2.3.39 static void EQEP_setQMAModuleMode (uint32_t *base*, **EQEP_QMAMode** *qmaMode*) [inline], [static]

Set the quadrature mode adapter (QMA) module mode

Parameters

base	is the base address of the eQEP module.

qmaMode	is the mode in which the QMA module will operate.

This function sets the quadrature mode adapter module mode. The possible modes are passed to the function through the *qmaMode* parameter which can take the values EQEP_QMA_MODE_BYPASS, EQEP_QMA_MODE_1, or EQEP_QMA_MODE_2.

To use the QMA module, you must first put the eQEP module into direction-count mode (**EQEP_CONFIG_CLOCK_DIR**) using EQEP_setConfig().

Returns
None.

20.2.3.40 static void EQEP_setStrobeSource (uint32_t *base*, **EQEP_StrobeSource** *strobeSrc*) [inline], [static]

Set the strobe input source of the eQEP module.

Parameters

base	is the base address of the eQEP module.
strobeSrc	is the source of the strobe signal.

This function sets the source of the eQEP module's strobe signal. The possible values of the *strobeSrc* parameter are

- **EQEP_STROBE_FROM_GPIO** - The strobe is used as-is after passing through the polarity select logic.
- **EQEP_STROBE_OR_ADCSOCA** - The strobe is OR'd with the ADCSOCA signal after passing through the polarity select logic.
- **EQEP_STROBE_OR_ADCSOCB** - The strobe is OR'd with the ADCSOCB signal after passing through the polarity select logic.

Returns
None.

20.2.3.41 static void EQEP_setEmulationMode (uint32_t *base*, **EQEP_EmulationMode** *emuMode*) [inline], [static]

Set the emulation mode of the eQEP module.

Parameters

base	is the base address of the eQEP module.
emuMode	is the mode operation upon an emulation suspend.

This function sets the eQEP module's emulation mode. This mode determines how the timers are affected by an emulation suspend. Valid values for the *emuMode* parameter are the following:

- **EQEP_EMULATIONMODE_STOPIMMEDIATELY** - The position counter, watchdog counter, unit timer, and capture timer all stop immediately.
- **EQEP_EMULATIONMODE_STOPATROLLOVER** - The position counter, watchdog counter, unit timer all count until period rollover. The capture timer counts until the next unit period event.

- **EQEP_EMULATIONMODE_RUNFREE** - The position counter, watchdog counter, unit timer, and capture timer are all unaffected by an emulation suspend.

Returns

None.

20.2.3.42 void EQEP_setCompareConfig (uint32_t *base,* uint16_t *config,* uint32_t *compareValue,* uint16_t *cycles*)

Configures eQEP module position-compare unit.

Parameters

base	is the base address of the eQEP module.
config	is the configuration for the eQEP module position-compare unit. See below for a description of this parameter.
compareValue	is the value to which the position count value is compared for a position-compare event.
cycles	is the width of the pulse that can be generated on a position-compare event. It is in units of 4 SYSCLKOUT cycles.

This function configures the operation of the eQEP module position-compare unit. The *config* parameter provides the configuration of the position-compare unit and is the logical OR of several values:

- **EQEP_COMPARE_NO_SYNC_OUT**, **EQEP_COMPARE_IDX_SYNC_OUT**, or **EQEP_COMPARE_STROBE_SYNC_OUT** specify if there is a sync output pulse and which pin should be used.
- **EQEP_COMPARE_NO_SHADOW**, **EQEP_COMPARE_LOAD_ON_ZERO**, or **EQEP_COMPARE_LOAD_ON_MATCH** specify if a shadow is enabled and when should the load should occur–QPOSCNT = 0 or QPOSCNT = QPOSCOMP.

The *cycles* is used to select the width of the sync output pulse. The width of the resulting pulse will be *cycles* $*$ 4 $*$ the width of a SYSCLKOUT cycle. The maximum width is 4096 $*$ 4 $*$ SYSCLKOUT cycles.

Note

You can set the sync pulse width independently using the EQEP_setComparePulseWidth() function.

Returns

None.

20.2.3.43 void EQEP_setInputPolarity (uint32_t *base,* bool *invertQEPA,* bool *invertQEPB,* bool *invertIndex,* bool *invertStrobe*)

Sets the polarity of the eQEP module's input signals.

Parameters

base	is the base address of the eQEP module.
invertQEPA	is the flag to negate the QEPA input.
invertQEPB	is the flag to negate the QEPA input.
invertIndex	is the flag to negate the index input.
invertStrobe	is the flag to negate the strobe input.

This function configures the polarity of the inputs to the eQEP module. To negate the polarity of any of the input signals, pass **true** into its corresponding parameter in this function. Pass **false** to leave it as-is.

Returns

None.

21 Flash Module

Introduction 349
API Functions 349

21.1 Flash Introduction

The Flash driver provides functions to configure the fallback power modes and the active grace periods of flash banks and pump, and the pump wake-up time. This driver also provides functions to configure the flash wait-states, prefetch, cache and ECC features. It also provides functions to access the Flash ECC test mode registers and the Flash ECC error status registers.

21.2 API Functions

Macros

- #define FLASH_FAIL_0_CLR
- #define FLASH_FAIL_1_CLR
- #define FLASH_UNC_ERR_CLR
- #define FLASH_NO_ERROR
- #define FLASH_SINGLE_ERROR
- #define FLASH_UNC_ERROR

Enumerations

- enum Flash_BankNumber { FLASH_BANK0, FLASH_BANK1 }
- enum Flash_BankPowerMode { FLASH_BANK_PWR_SLEEP, FLASH_BANK_PWR_STANDBY, FLASH_BANK_PWR_ACTIVE }
- enum Flash_PumpPowerMode { FLASH_PUMP_PWR_SLEEP, FLASH_PUMP_PWR_ACTIVE }
- enum Flash_ErrorStatus { FLASH_NO_ERR, FLASH_FAIL_0, FLASH_FAIL_1, FLASH_UNC_ERR }
- enum Flash_ErrorType { FLASH_DATA_ERR, FLASH_ECC_ERR }
- enum Flash_SingleBitErrorIndicator { FLASH_DATA_BITS, FLASH_CHECK_BITS }

Functions

- static void Flash_setWaitstates (uint32_t ctrlBase, uint16_t waitstates)
- static void Flash_setBankPowerMode (uint32_t ctrlBase, Flash_BankNumber bank, Flash_BankPowerMode powerMode)
- static void Flash_setPumpPowerMode (uint32_t ctrlBase, Flash_PumpPowerMode powerMode)
- static void Flash_enablePrefetch (uint32_t ctrlBase)
- static void Flash_disablePrefetch (uint32_t ctrlBase)
- static void Flash_enableCache (uint32_t ctrlBase)
- static void Flash_disableCache (uint32_t ctrlBase)

- static void Flash_enableECC (uint32_t eccBase)
- static void Flash_disableECC (uint32_t eccBase)
- static void Flash_setBankActiveGracePeriod (uint32_t ctrlBase, uint32_t period)
- static void Flash_setPumpActiveGracePeriod (uint32_t ctrlBase, uint16_t period)
- static void Flash_setPumpWakeupTime (uint32_t ctrlBase, uint16_t sysclkCycles)
- static bool Flash_isBankReady (uint32_t ctrlBase, Flash_BankNumber bank)
- static bool Flash_isPumpReady (uint32_t ctrlBase)
- static uint32_t Flash_getSingleBitErrorAddressLow (uint32_t eccBase)
- static uint32_t Flash_getSingleBitErrorAddressHigh (uint32_t eccBase)
- static uint32_t Flash_getUncorrectableErrorAddressLow (uint32_t eccBase)
- static uint32_t Flash_getUncorrectableErrorAddressHigh (uint32_t eccBase)
- static Flash_ErrorStatus Flash_getLowErrorStatus (uint32_t eccBase)
- static Flash_ErrorStatus Flash_getHighErrorStatus (uint32_t eccBase)
- static uint32_t Flash_getLowErrorPosition (uint32_t eccBase)
- static uint32_t Flash_getHighErrorPosition (uint32_t eccBase)
- static void Flash_clearLowErrorPosition (uint32_t eccBase)
- static void Flash_clearHighErrorPosition (uint32_t eccBase)
- static Flash_ErrorType Flash_getLowErrorType (uint32_t eccBase)
- static Flash_ErrorType Flash_getHighErrorType (uint32_t eccBase)
- static void Flash_clearLowErrorStatus (uint32_t eccBase, uint16_t errorStatus)
- static void Flash_clearHighErrorStatus (uint32_t eccBase, uint16_t errorStatus)
- static uint32_t Flash_getErrorCount (uint32_t eccBase)
- static void Flash_setErrorThreshold (uint32_t eccBase, uint16_t threshold)
- static uint32_t Flash_getInterruptFlag (uint32_t eccBase)
- static void Flash_clearSingleErrorInterruptFlag (uint32_t eccBase)
- static void Flash_clearUncorrectableInterruptFlag (uint32_t eccBase)
- static void Flash_setDataLowECCTest (uint32_t eccBase, uint32_t data)
- static void Flash_setDataHighECCTest (uint32_t eccBase, uint32_t data)
- static void Flash_setECCTestAddress (uint32_t eccBase, uint32_t address)
- static void Flash_setECCTestECCBits (uint32_t eccBase, uint16_t ecc)
- static void Flash_enableECCTestMode (uint32_t eccBase)
- static void Flash_disableECCTestMode (uint32_t eccBase)
- static void Flash_selectLowECCBlock (uint32_t eccBase)
- static void Flash_selectHighECCBlock (uint32_t eccBase)
- static void Flash_performECCCalculation (uint32_t eccBase)
- static uint32_t Flash_getTestDataOutHigh (uint32_t eccBase)
- static uint32_t Flash_getTestDataOutLow (uint32_t eccBase)
- static uint32_t Flash_getECCTestStatus (uint32_t eccBase)
- static uint32_t Flash_getECCTestErrorPosition (uint32_t eccBase)
- static
  Flash_SingleBitErrorIndicator Flash_getECCTestSingleBitErrorType (uint32_t eccBase)
- void Flash_initModule (uint32_t ctrlBase, uint32_t eccBase, uint16_t waitstates)
- void Flash_powerDown (uint32_t ctrlBase)

21.2.1 Detailed Description

The code for this module is contained in `driverlib/flash.c`, with `driverlib/flash.h` containing the API declarations for use by applications.

21.2.2 Enumeration Type Documentation

21.2.2.1 enum **Flash_BankNumber**

Values that can be passed to Flash_setBankPowerMode() as the bank parameter.

Enumerator

FLASH_BANK0 Bank 0.

FLASH_BANK1 Bank 1.

21.2.2.2 enum **Flash_BankPowerMode**

Values that can be passed to Flash_setBankPowerMode() as the powerMode parameter.

Enumerator

FLASH_BANK_PWR_SLEEP Sleep fallback mode.

FLASH_BANK_PWR_STANDBY Standby fallback mode.

FLASH_BANK_PWR_ACTIVE Active fallback mode.

21.2.2.3 enum **Flash_PumpPowerMode**

Values that can be passed to Flash_setPumpPowerMode() as the powerMode parameter.

Enumerator

FLASH_PUMP_PWR_SLEEP Sleep fallback mode.

FLASH_PUMP_PWR_ACTIVE Active fallback mode.

21.2.2.4 enum **Flash_ErrorStatus**

Type that correspond to values returned from Flash_getLowErrorStatus() and Flash_getHighErrorStatus() determining the error status code.

Enumerator

FLASH_NO_ERR No error.

FLASH_FAIL_0 Fail on 0.

FLASH_FAIL_1 Fail on 1.

FLASH_UNC_ERR Uncorrectable error.

21.2.2.5 enum **Flash_ErrorType**

Values that can be returned from Flash_getLowErrorType() and Flash_getHighErrorType() determining the error type.

Enumerator

FLASH_DATA_ERR Data error.

FLASH_ECC_ERR ECC error.

21.2.2.6 enum **Flash_SingleBitErrorIndicator**

Values that can be returned from Flash_getECCTestSingleBitErrorType().

Enumerator

FLASH_DATA_BITS Data bits.

FLASH_CHECK_BITS ECC bits.

21.2.3 Function Documentation

21.2.3.1 static void Flash_setWaitstates (uint32_t *ctrlBase*, uint16_t *waitstates*) [inline], [static]

Sets the random read wait state amount.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
waitstates	is the wait-state value.

This function sets the number of wait states for a flash read access. The *waitstates* parameter is a number between 0 and 15. It is **important** to look at your device's datasheet for information about what the required minimum flash wait-state is for your selected SYSCLK frequency.

By default the wait state amount is configured to the maximum 15.

Returns

None.

Referenced by Flash_initModule().

21.2.3.2 static void Flash_setBankPowerMode (uint32_t *ctrlBase*, **Flash_BankNumber** *bank*, **Flash_BankPowerMode** *powerMode*) [inline], [static]

Sets the fallback power mode of a flash bank.

Parameters

ctrlBase	is the base address of the flash wrapper registers.
bank	is the flash bank that is being configured.
powerMode	is the power mode to be entered.

This function sets the fallback power mode of the flash bank specified by them *bank* parameter. The power mode is specified by the *powerMode* parameter with one of the following values:

- **FLASH_BANK_PWR_SLEEP** - Sense amplifiers and sense reference disabled.
- **FLASH_BANK_PWR_STANDBY** - Sense amplifiers disabled but sense reference enabled.
- **FLASH_BANK_PWR_ACTIVE** - Sense amplifiers and sense reference enabled.

Returns

None.

Referenced by Flash_initModule(), and Flash_powerDown().

21.2.3.3 static void Flash_setPumpPowerMode (uint32_t *ctrlBase*, **Flash_PumpPowerMode** *powerMode*) `[inline]`, `[static]`

Sets the fallback power mode of the charge pump.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
powerMode	is the power mode to be entered.

This function sets the fallback power mode flash charge pump.

- **FLASH_PUMP_PWR_SLEEP** - All circuits disabled.
- **FLASH_PUMP_PWR_ACTIVE** - All pump circuits active.

Returns

None.

Referenced by Flash_initModule(), and Flash_powerDown().

21.2.3.4 static void Flash_enablePrefetch (uint32_t *ctrlBase*) `[inline]`, `[static]`

Enables prefetch mechanism.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

Returns

None.

Referenced by Flash_initModule().

21.2.3.5 static void Flash_disablePrefetch (uint32_t *ctrlBase*) `[inline]`, `[static]`

Disables prefetch mechanism.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

Returns

None.

Referenced by Flash_initModule().

21.2.3.6 static void Flash_enableCache (uint32_t *ctrlBase*) `[inline]`, `[static]`

Enables data cache.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

Returns
None.

Referenced by Flash_initModule().

21.2.3.7 static void Flash_disableCache (uint32_t *ctrlBase*) [inline], [static]

Disables data cache.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

Returns
None.

Referenced by Flash_initModule().

21.2.3.8 static void Flash_enableECC (uint32_t *eccBase*) [inline], [static]

Enables flash error correction code (ECC) protection.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

Referenced by Flash_initModule().

21.2.3.9 static void Flash_disableECC (uint32_t *eccBase*) [inline], [static]

Disables flash error correction code (ECC) protection.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.10 static void Flash_setBankActiveGracePeriod (uint32_t *ctrlBase,* uint32_t *period*) [inline], [static]

Sets the bank active grace period.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
period	is the starting count value for the BAGP down counter.

This function sets the bank active grace period specified by the *period* parameter. The *period* is a value between 0 and 255. This value must be greater than 1 when the fallback mode is not Active.

Returns

None.

21.2.3.11 static void Flash_setPumpActiveGracePeriod (uint32_t *ctrlBase,* uint16_t *period*) [inline], [static]

Sets the pump active grace period.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
period	is the starting count value for the PAGP down counter.

This function sets the pump active grace period specified by the *period* parameter. The *period* is a value between 0 and 65535. The counter is reloaded after any flash access. After the counter expires, the charge pump falls back to the power mode determined by FPAC1, bit PMPPWR.

Returns

None.

21.2.3.12 static void Flash_setPumpWakeupTime (uint32_t *ctrlBase,* uint16_t *sysclkCycles*) [inline], [static]

Sets the pump wake up time.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
sysclkCycles	is the number of SYSCLK cycles it takes for the pump to wakeup.

This function sets the wakeup time with *sysclkCycles* parameter. The *sysclkCycles* is a value between 0 and 8190. When the charge pump exits sleep power mode, it will take sysclkCycles to wakeup.

Returns

None.

21.2.3.13 static bool Flash_isBankReady (uint32_t *ctrlBase,* **Flash_BankNumber** *bank*) [inline], [static]

Reads the bank active power state.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
bank	is the flash bank that is being used.

Returns

Returns **true** if the Bank is in Active power state and **false** otherwise.

21.2.3.14 static bool Flash_isPumpReady (uint32_t *ctrlBase*) [inline], [static]

Reads the pump active power state.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

Returns

Returns **true** if the Pump is in Active power state and **false** otherwise.

21.2.3.15 static uint32_t Flash_getSingleBitErrorAddressLow (uint32_t *eccBase*) [inline], [static]

Gets the single error address low.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the 32-bit address of the single bit error that occurred in the lower 64-bits of a 128-bit memory-aligned data. The returned address is to that 64-bit aligned data.

Returns

Returns the 32 bits of a 64-bit aligned address where a single bit error occurred.

21.2.3.16 static uint32_t Flash_getSingleBitErrorAddressHigh (uint32_t *eccBase*) [inline], [static]

Gets the single error address high.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the 32-bit address of the single bit error that occurred in the upper 64-bits of a 128-bit memory-aligned data. The returned address is to that 64-bit aligned data.

Returns

Returns the 32 bits of a 64-bit aligned address where a single bit error occurred.

21.2.3.17 static uint32_t Flash_getUncorrectableErrorAddressLow (uint32_t *eccBase*) [inline], [static]

Gets the uncorrectable error address low.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the 32-bit address of the uncorrectable error that occurred in the lower 64-bits of a 128-bit memory-aligned data. The returned address is to that 64-bit aligned data.

Returns

Returns the 32 bits of a 64-bit aligned address where an uncorrectable error occurred.

21.2.3.18 static uint32_t Flash_getUncorrectableErrorAddressHigh (uint32_t *eccBase*) [inline], [static]

Gets the uncorrectable error address high.

Parameters

eccBase	is the base address of the flash wrapper ECC base.

This function returns the 32-bit address of the uncorrectable error that occurred in the upper 64-bits of a 128-bit memory-aligned data. The returned address is to that 64-bit aligned data.

Returns

Returns the 32 bits of a 64-bit aligned address where an uncorrectable error occurred.

21.2.3.19 static **Flash_ErrorStatus** Flash_getLowErrorStatus (uint32_t *eccBase*) [inline], [static]

Gets the error status of the Lower 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error status of the lower 64-bits of a 128-bit aligned address.

Returns

Returns value of the low error status bits which can be used with Flash_ErrorStatus type.

21.2.3.20 static **Flash_ErrorStatus** Flash_getHighErrorStatus (uint32_t *eccBase*) [inline], [static]

Gets the error status of the Upper 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error status of the upper 64-bits of a 128-bit aligned address.

Returns

Returns value of the high error status bits which can be used with Flash_ErrorStatus type.

21.2.3.21 static uint32_t Flash_getLowErrorPosition (uint32_t *eccBase*) [inline], [static]

Gets the error position of the lower 64-bits for a single bit error.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error position of the lower 64-bits. If the error type is FLASH_ECC_ERR, the position ranges from 0-7 else it ranges from 0-63 for FLASH_DATA_ERR.

Returns

Returns the position of the lower error bit.

21.2.3.22 static uint32_t Flash_getHighErrorPosition (uint32_t *eccBase*) [inline], [static]

Gets the error position of the upper 64-bits for a single bit error.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error position of the upper 64-bits. If the error type is FLASH_ECC_ERR, the position ranges from 0-7 else it ranges from 0-63 for FLASH_DATA_ERR.

Returns

Returns the position of the upper error bit.

21.2.3.23 static void Flash_clearLowErrorPosition (uint32_t *eccBase*) [inline], [static]

Clears the error position bit of the lower 64-bits for a single bit error.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function clears the error position bit of the lower 64-bits.

Returns

None

21.2.3.24 static void Flash_clearHighErrorPosition (uint32_t *eccBase*) [inline], [static]

Clears the error position of the upper 64-bits for a single bit error.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function clears the error position bit of the upper 64-bits.

Returns

None.

21.2.3.25 static **Flash_ErrorType** Flash_getLowErrorType (uint32_t *eccBase*) [inline], [static]

Gets the error type of the lower 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error type of the lower 64-bits. The error type can be FLASH_ECC_ERR or FLASH_DATA_ERR.

Returns

Returns the type of the lower 64-bit error.

21.2.3.26 static **Flash_ErrorType** Flash_getHighErrorType (uint32_t *eccBase*) [inline], [static]

Gets the error type of the upper 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the error type of the upper 64-bits. The error type can be FLASH_ECC_ERR or FLASH_DATA_ERR.

Returns

Returns the type of the upper 64-bit error.

21.2.3.27 static void Flash_clearLowErrorStatus (uint32_t *eccBase,* uint16_t *errorStatus*) [inline], [static]

Clears the errors status of the lower 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
errorStatus	is the error status to clear. errorStatus is a uint16_t. errorStatus is a bitwise OR of the following value: ■ **FLASH_FAIL_0_CLR** ■ **FLASH_FAIL_1_CLR** ■ **FLASH_UNC_ERR_CLR**

Returns

None.

21.2.3.28 static void Flash_clearHighErrorStatus (uint32_t *eccBase,* uint16_t *errorStatus*) [inline], [static]

Clears the errors status of the upper 64-bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
errorStatus	is the error status to clear. errorStatus is a uint16_t. errorStatus is a bitwise OR of the following value: ■ **FLASH_FAIL_0_CLR** ■ **FLASH_FAIL_1_CLR** ■ **FLASH_UNC_ERR_CLR**

Returns

None.

21.2.3.29 static uint32_t Flash_getErrorCount (uint32_t *eccBase*) [inline], [static]

Gets the single bit error count.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

Returns the single bit error count.

21.2.3.30 static void Flash_setErrorThreshold (uint32_t *eccBase,* uint16_t *threshold*) [inline], [static]

Sets the single bit error threshold.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
threshold	is the single bit error threshold. Valid ranges are from 0-65535.

Returns

None.

21.2.3.31 static uint32_t Flash_getInterruptFlag (uint32_t *eccBase*) [inline], [static]

Gets the error interrupt.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the type of error interrupt that occurred. The values can be used with

- **FLASH_NO_ERROR**
- **FLASH_SINGLE_ERROR**
- **FLASH_UNC_ERROR**

Returns
Returns the interrupt flag.

21.2.3.32 static void Flash_clearSingleErrorInterruptFlag (uint32_t *eccBase*) [inline], [static]

Clears the single error interrupt flag.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.33 static void Flash_clearUncorrectableInterruptFlag (uint32_t *eccBase*) [inline], [static]

Clears the uncorrectable error interrupt flag.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.34 static void Flash_setDataLowECCTest (uint32_t *eccBase,* uint32_t *data*) [inline], [static]

Sets the Data Low Test register for ECC testing.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
data	is a 32-bit value that is the low double word of selected 64-bit data

Returns
None.

21.2.3.35 static void Flash_setDataHighECCTest (uint32_t *eccBase*, uint32_t *data*) `[inline]`, `[static]`

Sets the Data High Test register for ECC testing.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
data	is a 32-bit value that is the high double word of selected 64-bit data

Returns

None.

21.2.3.36 static void Flash_setECCTestAddress (uint32_t *eccBase*, uint32_t *address*) [inline], [static]

Sets the test address register for ECC testing.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
address	is a 32-bit value containing an address. Bits 21-3 will be used as the flash word (128-bit) address.

This function left shifts the address 1 bit to convert it to a byte address.

Returns

None.

21.2.3.37 static void Flash_setECCTestECCBits (uint32_t *eccBase*, uint16_t *ecc*) [inline], [static]

Sets the ECC test bits for ECC testing.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.
ecc	is a 32-bit value. The least significant 8 bits are used as the ECC Control Bits in the ECC Test.

Returns

None.

21.2.3.38 static void Flash_enableECCTestMode (uint32_t *eccBase*) [inline], [static]

Enables ECC Test mode.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

None.

21.2.3.39 static void Flash_disableECCTestMode (uint32_t *eccBase*) [inline], [static]

Disables ECC Test mode.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.40 static void Flash_selectLowECCBlock (uint32_t *eccBase*) [inline], [static]

Selects the ECC block on bits [63:0] of bank data.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.41 static void Flash_selectHighECCBlock (uint32_t *eccBase*) [inline], [static]

Selects the ECC block on bits [127:64] of bank data.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.42 static void Flash_performECCCalculation (uint32_t *eccBase*) [inline], [static]

Performs the ECC calculation on the test block.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns
None.

21.2.3.43 static uint32_t Flash_getTestDataOutHigh (uint32_t *eccBase*) [inline], [static]

Gets the ECC Test data out high 63:32 bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

Returns the ECC TEst data out High.

21.2.3.44 static uint32_t Flash_getTestDataOutLow (uint32_t *eccBase*) [inline], [static]

Gets the ECC Test data out low 31:0 bits.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

Returns the ECC Test data out Low.

21.2.3.45 static uint32_t Flash_getECCTestStatus (uint32_t *eccBase*) [inline], [static]

Gets the ECC Test status.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

This function returns the ECC test status. The values can be used with

- **FLASH_NO_ERROR**
- **FLASH_SINGLE_ERROR**
- **FLASH_UNC_ERROR**

Returns

Returns the ECC test status.

21.2.3.46 static uint32_t Flash_getECCTestErrorPosition (uint32_t *eccBase*) [inline], [static]

Gets the ECC Test single bit error position.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

Returns the ECC Test single bit error position. If the error type is check bits than the position can range from 0 to 7. If the error type is data bits than the position can range from 0 to 63.

21.2.3.47 static **Flash_SingleBitErrorIndicator** Flash_getECCTestSingleBitErrorType (uint32_t *eccBase*) [inline], [static]

Gets the single bit error type.

Parameters

eccBase	is the base address of the flash wrapper ECC registers.

Returns

Returns the single bit error type as a Flash_SingleBitErrorIndicator. FLASH_DATA_BITS and FLASH_CHECK_BITS indicate where the single bit error occurred.

21.2.3.48 void Flash_initModule (uint32_t *ctrlBase,* uint32_t *eccBase,* uint16_t *waitstates*)

Initializes the flash control registers.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.
eccBase	is the base address of the flash wrapper ECC registers.
waitstates	is the wait-state value.

This function initializes the flash control registers. At reset bank and pump are in sleep. A flash access will power up the bank and pump automatically. After a flash access, bank and pump go to low power mode (configurable in FBFALLBACK/FPAC1 registers) if there is no further access to flash. This function will power up Flash bank and pump and set the fallback mode of flash and pump as active.

This function also sets the number of wait-states for a flash access (see Flash_setWaitstates() for more details), and enables cache, the prefetch mechanism, and ECC.

Returns

None.

References FLASH_BANK0, FLASH_BANK1, FLASH_BANK_PWR_ACTIVE, Flash_disableCache(), Flash_disablePrefetch(), Flash_enableCache(), Flash_enableECC(), Flash_enablePrefetch(), FLASH_PUMP_PWR_ACTIVE, Flash_setBankPowerMode(), Flash_setPumpPowerMode(), and Flash_setWaitstates().

21.2.3.49 void Flash_powerDown (uint32_t *ctrlBase*)

Powers down the flash.

Parameters

ctrlBase	is the base address of the flash wrapper control registers.

This function powers down the flash bank(s) and the flash pump.

Returns

None.

References FLASH_BANK0, FLASH_BANK1, FLASH_BANK_PWR_SLEEP, FLASH_PUMP_PWR_SLEEP, Flash_setBankPowerMode(), and Flash_setPumpPowerMode().

22 FSI Module

Introduction 371
API Functions 371

22.1 FSI Introduction

The Fast Serial Interface (FSI) API provides a set of functions for configuring and using the FSI module.

22.2 API Functions

Macros

- #define FSI_TX_EVT_FRAME_DONE
- #define FSI_TX_EVTMASK
- #define FSI_TX_MAX_NUM_EXT_TRIGGERS
- #define FSI_TX_INT2_CTRL_S
- #define FSI_RX_EVT_PING_WD_TIMEOUT
- #define FSI_RX_EVTMASK
- #define FSI_RX_MAX_DELAY_LINE_VAL
- #define FSI_MAX_LEN_NWORDS_DATA
- #define FSI_MAX_VALUE_USERDATA
- #define FSI_MAX_VALUE_BUF_PTR_OFF
- #define FSI_CTRL_REG_KEY

Enumerations

- enum FSI_DataWidth
- enum FSI_TxSubmoduleInReset
- enum FSI_TxStartMode
- enum FSI_FrameType
- enum FSI_FrameTag
- enum FSI_PingTimeoutMode
- enum FSI_ECCComputeWidth
- enum FSI_InterruptNum
- enum FSI_RxSubmoduleInReset
- enum FSI_RxDelayTapType
- enum FSI_ExtFrameTriggerSrc

Functions

- static void FSI_sendTxFlush (uint32_t base)
- static void FSI_stopTxFlush (uint32_t base)
- static void FSI_selectTxPLLClock (uint32_t base)
- static void FSI_enableTxClock (uint32_t base, uint16_t preScaleValue)
- static void FSI_disableTxClock (uint32_t base)

- static void FSI_setTxDataWidth (uint32_t base, FSI_DataWidth dataWidth)
- static void FSI_enableTxSPIMode (uint32_t base)
- static void FSI_disableTxSPIMode (uint32_t base)
- static void FSI_setTxStartMode (uint32_t base, FSI_TxStartMode txStartMode)
- static void FSI_setTxPingTimeoutMode (uint32_t base, FSI_PingTimeoutMode pingTimeoutMode)
- static void FSI_setTxExtFrameTrigger (uint32_t base, uint16_t extInputNum)
- static void FSI_enableTxCRCForceError (uint32_t base)
- static void FSI_disableTxCRCForceError (uint32_t base)
- static void FSI_setTxECCComputeWidth (uint32_t base, FSI_ECCComputeWidth eccComputeWidth)
- static void FSI_setTxFrameType (uint32_t base, FSI_FrameType frameType)
- static void FSI_setTxSoftwareFrameSize (uint32_t base, uint16_t nWords)
- static void FSI_startTxTransmit (uint32_t base)
- static void FSI_setTxFrameTag (uint32_t base, FSI_FrameTag frameTag)
- static void FSI_setTxUserDefinedData (uint32_t base, uint16_t userDefData)
- static void FSI_setTxBufferPtr (uint32_t base, uint16_t bufPtrOff)
- static uint16_t FSI_getTxBufferPtr (uint32_t base)
- static uint16_t FSI_getTxWordCount (uint32_t base)
- static void FSI_enableTxPingTimer (uint32_t base, uint32_t refValue, FSI_FrameTag pingFrameTag)
- static void FSI_setTxPingTag (uint32_t base, FSI_FrameTag frameTag)
- static void FSI_disableTxPingTimer (uint32_t base)
- static void FSI_enableTxExtPingTrigger (uint32_t base, uint16_t extTrigSel)
- static void FSI_disableTxExtPingTrigger (uint32_t base)
- static uint32_t FSI_getTxCurrentPingTimeoutCounter (uint32_t base)
- static void FSI_enableTxDMAEvent (uint32_t base)
- static void FSI_disableTxDMAEvent (uint32_t base)
- static void FSI_lockTxCtrl (uint32_t base)
- static uint16_t FSI_getTxEventStatus (uint32_t base)
- static void FSI_forceTxEvents (uint32_t base, uint16_t evtFlags)
- static void FSI_clearTxEvents (uint32_t base, uint16_t evtFlags)
- static void FSI_enableTxUserCRC (uint32_t base, uint16_t userCRCValue)
- static void FSI_disableTxUserCRC (uint32_t base)
- static void FSI_setTxECCdata (uint32_t base, uint32_t data)
- static uint16_t FSI_getTxECCValue (uint32_t base)
- static void FSI_enableTxInterrupt (uint32_t base, FSI_InterruptNum intNum, uint16_t intFlags)
- static void FSI_disableTxInterrupt (uint32_t base, FSI_InterruptNum intNum, uint16_t intFlags)
- static uint32_t FSI_getTxBufferAddress (uint32_t base)
- void FSI_resetTxModule (uint32_t base, FSI_TxSubmoduleInReset submodule)
- void FSI_clearTxModuleReset (uint32_t base, FSI_TxSubmoduleInReset submodule)
- void FSI_writeTxBuffer (uint32_t base, const uint16_t array[ ], uint16_t length, uint16_t bufOffset)
- static void FSI_enableRxInternalLoopback (uint32_t base)
- static void FSI_disableRxInternalLoopback (uint32_t base)
- static void FSI_enableRxSPIPairing (uint32_t base)
- static void FSI_disableRxSPIPairing (uint32_t base)
- static void FSI_setRxDataWidth (uint32_t base, FSI_DataWidth dataWidth)
- static void FSI_enableRxSPIMode (uint32_t base)
- static void FSI_disableRxSPIMode (uint32_t base)
- static void FSI_setRxSoftwareFrameSize (uint32_t base, uint16_t nWords)
- static void FSI_setRxECCComputeWidth (uint32_t base, FSI_ECCComputeWidth eccComputeWidth)
- static void FSI_setRxPingTimeoutMode (uint32_t base, FSI_PingTimeoutMode pingTimeoutMode)

- static FSI_FrameType FSI_getRxFrameType (uint32_t base)
- static void FSI_enableRxDMAEvent (uint32_t base)
- static void FSI_disableRxDMAEvent (uint32_t base)
- static uint16_t FSI_getRxFrameTag (uint32_t base)
- static uint16_t FSI_getRxUserDefinedData (uint32_t base)
- static uint16_t FSI_getRxEventStatus (uint32_t base)
- static void FSI_forceRxEvents (uint32_t base, uint16_t evtFlags)
- static void FSI_clearRxEvents (uint32_t base, uint16_t evtFlags)
- static uint16_t FSI_getRxReceivedCRC (uint32_t base)
- static uint16_t FSI_getRxComputedCRC (uint32_t base)
- static void FSI_setRxBufferPtr (uint32_t base, uint16_t bufPtrOff)
- static uint16_t FSI_getRxBufferPtr (uint32_t base)
- static uint16_t FSI_getRxWordCount (uint32_t base)
- static void FSI_enableRxFrameWatchdog (uint32_t base, uint32_t wdRef)
- static void FSI_disableRxFrameWatchdog (uint32_t base)
- static uint32_t FSI_getRxFrameWatchdogCounter (uint32_t base)
- static void FSI_enableRxPingWatchdog (uint32_t base, uint32_t wdRef)
- static void FSI_disableRxPingWatchdog (uint32_t base)
- static uint32_t FSI_getRxPingWatchdogCounter (uint32_t base)
- static uint16_t FSI_getRxPingTag (uint32_t base)
- static void FSI_lockRxCtrl (uint32_t base)
- static void FSI_setRxECCData (uint32_t base, uint32_t rxECCdata)
- static void FSI_setRxReceivedECCValue (uint32_t base, uint16_t rxECCvalue)
- static uint32_t FSI_getRxECCCorrectedData (uint32_t base)
- static uint16_t FSI_getRxECCLog (uint32_t base)
- static void FSI_enableRxInterrupt (uint32_t base, FSI_InterruptNum intNum, uint16_t intFlags)
- static void FSI_disableRxInterrupt (uint32_t base, FSI_InterruptNum intNum, uint16_t intFlags)
- static uint32_t FSI_getRxBufferAddress (uint32_t base)
- void FSI_resetRxModule (uint32_t base, FSI_RxSubmoduleInReset submodule)
- void FSI_clearRxModuleReset (uint32_t base, FSI_RxSubmoduleInReset submodule)
- void FSI_readRxBuffer (uint32_t base, uint16_t array[ ], uint16_t length, uint16_t bufOffset)
- void FSI_configRxDelayLine (uint32_t base, FSI_RxDelayTapType delayTapType, uint16_t tapValue)
- void FSI_performTxInitialization (uint32_t base, uint16_t prescalar)
- void FSI_performRxInitialization (uint32_t base)
- void FSI_executeTxFlushSequence (uint32_t base, uint16_t prescalar)

22.2.1 Detailed Description

The code for this module is contained in `driverlib/fsi.c`, with `driverlib/fsi.h` containing the API declarations for use by applications.

22.2.2 Macro Definition Documentation

22.2.2.1 #define FSI_TX_EVT_FRAME_DONE

FSI Tx events defines.

Values that can be passed to APIs to enable/disable interrupts and also to set/get/clear event status on FSI Tx operation.

There are 4 supported interrupts related to Tx events- All are available as event status as well excecpt 4th one. 1) frame transmission done 2) transmit buffer is underrun 3) transmit buffer is overrun 4) ping counter timeout

Ping frame transmission upon hardware trigger(ping watchdog or external trigger) is shown as event status.

22.2.2.2 #define FSI_RX_EVT_PING_WD_TIMEOUT

FSI Rx event defines.

Values that can be passed to APIs to enable/disable interrupts and also to set/get/clear event status on FSI Rx operation.

There are 12 supported interrupts related to Rx events- All are available as event status as well. 1) ping watchdog times out 2) frame watchdog times out 3) mismatch between hardware computed CRC and received CRC.This status should be ignored if user chooses SW CRC computation 4) invalid Frame type detected 5) invalid EndofFrame bit-pattern 6) buffer Overrun in Rx buffer 7) received frame without errors 8) software reads empty Rx buffer 9) received error frame 10) received ping frame 11) software didn't clear FRAME_DONE flag after receiving new frame 12) received data frame

22.2.3 Enumeration Type Documentation

22.2.3.1 enum **FSI_DataWidth**

Data lines used for transmit/receive operation.

Supported number of data lines is only 2 - 1 lane or 2 lanes

22.2.3.2 enum **FSI_TxSubmoduleInReset**

List of TX submodules that can be reset, can be used with reset APIs.

Three kind of resets can be made- 1) reset entire Tx Module 2) reset only TX clock 3) reset ping timeout counter

22.2.3.3 enum **FSI_TxStartMode**

Start Mode for Tx frame transmission.

Three start modes(i.e. how transmission will start) are supported-

1. SW write of START bit in **TX_PKT_CTRL** register
2. Rising edge on external trigger
3. Either SW write of START bit or Frame completion

22.2.3.4 enum **FSI_FrameType**

Various FSI frame types.

Three frame types exist-

- **Ping:** Used for checking line integrity, can be sent by software or automatically by hardware.
- **Error:** Used typically during error conditions or when one side wants to signal the other side for attention.
- **Data:** Two subtypes exist based on data-length- a) **Fixed** (1/2/4/6 words) b) **Nwords** Software programs number of data words

Note

4 bit code for frame types- 0x1, 0x2 and 0x8 to 0xE are reserved

22.2.3.5 enum **FSI_FrameTag**

Possible values of a FSI frame.

4 bit field inside FSI frame is available to set tag value(0-15)

22.2.3.6 enum **FSI_PingTimeoutMode**

Ping timeout mode.

Ping timeout can reset and restart only on hardware initiated PING frames (PING Watchdog timeout) OR on any software initiated frame being sent out also based on which mode is selected

22.2.3.7 enum **FSI_InterruptNum**

Interrupt lines supported in FSI.

Any event on FSI Tx or Rx can be enabled to trigger interrupt on 2 interrupt lines to CPU/CLA- **INT1** and **INT2**

22.2.3.8 enum **FSI_RxSubmoduleInReset**

List of RX modules that can be reset, can be used with reset APIs.

Three submodules can be reset- 1) RX master core 2) frame watchdog counter 3) ping watchdog counter

22.2.3.9 enum **FSI_RxDelayTapType**

Available Rx lines for delay tap selection.

Delay tapping can be done on 3 lines- 1)RXCLK 2)RXD0 and 3)RXD1

22.2.3.10 enum **FSI_ExtFrameTriggerSrc**

Indexes of available EPWM SOC triggers.

There are 16 ePWM SOC events as external triggers for FSI frame transfers. Indexes 0:7 and 24:31 are reserved out of total 32 muxed external triggers.

22.2.4 Function Documentation

22.2.4.1 static void FSI_sendTxFlush (uint32_t *base*) [inline], [static]

Validates if FSI-Tx base address is correct.

Parameters

in	*base*	is the base address of the FSI-Tx module

Returns

returns **true** if the base address is valid and **false** otherwise Sends FLUSH pattern

FLUSH pattern (toggle data lines followed by toggle on clocks) should be sent only when FSI Tx is not under **SOFT_RESET** and the clock to the transmit core has been turned ON.

Parameters

in	*base*	is the FSI Tx module base address

Returns

None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_executeTxFlushSequence().

22.2.4.2 static void FSI_stopTxFlush (uint32_t *base*) [inline], [static]

Stops FLUSH pattern transmission.

Transmission of FLUSH pattern should be stopped before starting sending frames. Generally during initilization a pair of send/stop APIs for FLUSH pattern is called to clear data/clock lines.

Parameters

in	*base*	is the FSI Tx module base address

Returns

None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_executeTxFlushSequence().

22.2.4.3 static void FSI_selectTxPLLClock (uint32_t *base*) [inline], [static]

Selects PLL clock as source for clock dividers.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

Referenced by FSI_performTxInitialization().

22.2.4.4 static void FSI_enableTxClock (uint32_t *base*, uint16_t *preScaleValue*) [inline], [static]

sets clock division prescalar and enables the transmit clock

Parameters

in	*base*	is the FSI Tx module base address
in	*preScaleValue*	used to generate transmit clock, it defines the division value of /2, /3, /4, etc. of **PLLcLK**

Returns
None.

Referenced by FSI_performTxInitialization().

22.2.4.5 static void FSI_disableTxClock (uint32_t *base*) [inline], [static]

Disables transmit clock.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.6 static void FSI_setTxDataWidth (uint32_t *base*, **FSI_DataWidth** *dataWidth*) [inline], [static]

Sets Data width for transmission.

Parameters

in	*base*	is the FSI Tx module base address
in	*dataWidth*	selection between 1 or 2 lane transmission

Returns
None.

22.2.4.7 static void FSI_enableTxSPIMode (uint32_t *base*) [inline], [static]

Enables SPI compatible mode.

FSI supports a **compatibility** mode in order to communicate with **legacy** peripherals like **SPI**. Only the 16-bit mode of SPI will be supported. All the frame structures, CRC checks and will be identical to the normal FSI frames.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.8 static void FSI_disableTxSPIMode (uint32_t *base*) [inline], [static]

Disables SPI compatible mode.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.9 static void FSI_setTxStartMode (uint32_t *base*, **FSI_TxStartMode** *txStartMode*) [inline], [static]

Sets start mode for any frame transmission.

Parameters

in	*base*	is the FSI Tx module base address
in	*txStartMode*	is one of supported 3 start modes in transmission

Returns
None.

22.2.4.10 static void FSI_setTxPingTimeoutMode (uint32_t *base*, **FSI_PingTimeoutMode** *pingTimeoutMode*) [inline], [static]

Setting for when Ping timeout can reset and restart.

Parameters

in	*base*	is the FSI Tx module base address
in	*pingTimeout-Mode*	can be HW or both HW/SW initiated

Returns
None.

22.2.4.11 static void FSI_setTxExtFrameTrigger (uint32_t *base*, uint16_t *extInputNum*) [inline], [static]

Sets a particular external input to trigger transmission.

Parameters

in	*base*	is the FSI Tx module base address
in	*extInputNum*	can be one of ports from 0 to 31

Returns
None.

References FSI_TX_MAX_NUM_EXT_TRIGGERS.

22.2.4.12 static void FSI_enableTxCRCForceError (uint32_t *base*) [inline], [static]

Enables CRC value of a data frame to be forced to zero.

CRC value of the data frame will be forced to 0 whenever there is a transmission and buffer over-run or under-run condition happens. The idea is to force a corruption of the CRC since the data is not guaranteed to be reliable

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.13 static void FSI_disableTxCRCForceError (uint32_t *base*) [inline], [static]

Disables forcing of CRC value of a data frame to zero.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.14 static void FSI_setTxECCComputeWidth (uint32_t *base*, **FSI_ECCComputeWidth** *eccComputeWidth*) [inline], [static]

Select between 16-bit and 32-bit ECC computation.

Parameters

in	*base*	is the FSI Tx module base address
in	*eccComputeWidth*	is ECC Computation width

Returns
None.

22.2.4.15 static void FSI_setTxFrameType (uint32_t *base*, **FSI_FrameType** *frameType*) [inline], [static]

Sets frame type for transmission.

Parameters

in	*base*	is the FSI Tx module base address
in	*frameType*	value of frame type

Returns
None.

22.2.4.16 static void FSI_setTxSoftwareFrameSize (uint32_t *base*, uint16_t *nWords*) [inline], [static]

Sets the frame size if frame type is user/software defined frame.

Parameters

in	*base*	is the FSI Tx module base address
in	*nWords*	is number of data words in a software defined frame

Returns
None.

References FSI_MAX_LEN_NWORDS_DATA.

22.2.4.17 static void FSI_startTxTransmit (uint32_t *base*) [inline], [static]

Starts transmitting frames.

Parameters

in	*base*	is the FSI Tx module base address

Returns

None.

22.2.4.18 static void FSI_setTxFrameTag (uint32_t *base*, **FSI_FrameTag** *frameTag*) [inline], [static]

Sets frame tag for transmission.

Parameters

in	*base*	is the FSI Tx module base address
in	*frameTag*	value of frame tag, 4 bit value (0 to 15)

Returns

None.

22.2.4.19 static void FSI_setTxUserDefinedData (uint32_t *base*, uint16_t *userDefData*) [inline], [static]

Sets user defined data for transmission It is an extra data field(8 bit) apart from regular data.

Parameters

in	*base*	is the FSI Tx module base address
in	*userDefData*	8 bit user defined data value

Returns

None.

References FSI_MAX_VALUE_USERDATA.

22.2.4.20 static void FSI_setTxBufferPtr (uint32_t *base*, uint16_t *bufPtrOff*) [inline], [static]

Sets the value for transmit buffer pointer at desired location.

Parameters

in	*base*	is the FSI Tx module base address
in	*bufPtrOff*	4 bit offset pointer in Tx buffer where transmitter will pick the data

Returns

None.

References FSI_MAX_VALUE_BUF_PTR_OFF.

22.2.4.21 static uint16_t FSI_getTxBufferPtr (uint32_t *base*) [inline], [static]

Returns current buffer pointer location.

Parameters

in	*base*	is the FSI Tx module base address

Returns
current buffer pointer location

Note
there could be lag due to synchronization hence value is accurate only when no current transmission is happening

22.2.4.22 static uint16_t FSI_getTxWordCount (uint32_t *base*) [inline], [static]

Returns valid number of data words present in buffer which have not been transmitted yet.

Parameters

in	*base*	is the FSI Tx module base address

Returns
number of data words present in buffer which have not been transmitted yet

Note
there could be lag due to synchronization hence value is accurate only when no current transmission is happening

22.2.4.23 static void FSI_enableTxPingTimer (uint32_t *base,* uint32_t *refValue,* **FSI_FrameTag** *pingFrameTag*) [inline], [static]

Enables ping timer logic and once set time elapses it sends signal to transmitter to send ping frame.

Parameters

in	*base*	is the FSI Tx module base address
in	*refValue*	32 bit reference value for ping time-out counter
in	*pingFrameTag*	4 bit tag value for ping time-out counter

Returns
None.

22.2.4.24 static void FSI_setTxPingTag (uint32_t *base,* **FSI_FrameTag** *frameTag*) [inline], [static]

Sets the ping tag value, used by either timeout counter initiated PING frame transfer or by external ping trigger input.

Parameters

in	*base*	is the FSI Tx module base address
in	*frameTag*	4 bit tag value for ping time-out counter

Returns
None.

22.2.4.25 static void FSI_disableTxPingTimer (uint32_t *base*) [inline], [static]

Disables ping timer logic.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.26 static void FSI_enableTxExtPingTrigger (uint32_t *base,* uint16_t *extTrigSel*) [inline], [static]

Enables external trigger to transmit a ping frame.

Parameters

in	*base*	is the FSI Tx module base address
in	*extTrigSel*	5 bit value which selects among 32 external inputs

Returns
None.

References FSI_TX_MAX_NUM_EXT_TRIGGERS.

22.2.4.27 static void FSI_disableTxExtPingTrigger (uint32_t *base*) [inline], [static]

Disables external trigger logic.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.28 static uint32_t FSI_getTxCurrentPingTimeoutCounter (uint32_t *base*) [inline], [static]

Gives Current value of Ping Timeout Logic Counter.

Parameters

in	*base*	is the FSI Tx module base address

Returns
Current value of counter is returned

22.2.4.29 static void FSI_enableTxDMAEvent (uint32_t *base*) [inline], [static]

Enables to generate DMA event on completion of a frame transfer.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.30 static void FSI_disableTxDMAEvent (uint32_t *base*) [inline], [static]

Disable to generate DMA event on completion of a frame transfer.

Parameters

in	*base*	is the FSI Tx module base address

Returns
None.

22.2.4.31 static void FSI_lockTxCtrl (uint32_t *base*) [inline], [static]

Locks the control of all transmit control registers, once locked further writes will not take effect until system reset occurs.

Parameters

in	*base*	is the FSI Tx module base address

Note
System reset only can unlock registers once locked.

Returns
None.

References FSI_CTRL_REG_KEY.

22.2.4.32 static uint16_t FSI_getTxEventStatus (uint32_t *base*) [inline], [static]

Returns current status of all the error flags.

Parameters

in	*base*	is the FSI Tx module base address

Returns

the status of error flags, each bit of integer is associated with one error flag.

Example Usage - function will set the bits corresponding to respective error flag in return value evtStatus = FSI_getTxEventStatus(FSI_base) if bit value of evtStatus is 12(01100) means FSI_TX_EVT_OVERRUN and FSI_TX_EVT_PING_HW_TRIG flags are set

References FSI_TX_EVTMASK.

22.2.4.33 static void FSI_forceTxEvents (uint32_t *base*, uint16_t *evtFlags*) [inline], [static]

Enables user to set TX error flags.

Parameters

in	*base*	is the FSI Tx module base address
in	*evtFlags*	contains list of event and error flags that are supposed to be set.

Writing a 1 to this bit position will cause the corresponding bit in **TX_EVT_ERR_STATUS** register to get set. The purpose of this register is to allow software to simulate the effect of the event and test the associated software/ISR.

Example Usage evtFlags = FSI_TX_EVT_FRAME_DONE & FSI_TX_EVT_OVERRUN FSI_forceTxEvents(FSI_base, evtFlags) Above call sets error flag to frameDone and overRun events

Returns

None.

References FSI_TX_EVTMASK.

22.2.4.34 static void FSI_clearTxEvents (uint32_t *base*, uint16_t *evtFlags*) [inline], [static]

Enables user to clear TX error flags.

Parameters

in	*base*	is the FSI Tx module base address
in	*evtFlags*	contains list of event and error flags that are supposed to be cleared.

Writing a 1 to this bit position will cause the corresponding bit in the TX_EVT_ERR_STATUS register to get cleared to 0

Returns

None.

References FSI_TX_EVTMASK.

Referenced by FSI_performTxInitialization().

22.2.4.35 static void FSI_enableTxUserCRC (uint32_t *base*, uint16_t *userCRCValue*) `[inline]`, `[static]`

Sets the CRC value to be picked transmission if transmission is configured to use user defined SW CRC.

Parameters

in	*base*	is the FSI Tx module base address
in	*userCRCValue*	is user defined CRC

Returns

None.

22.2.4.36 static void FSI_disableTxUserCRC (uint32_t *base*) [inline], [static]

Sets the CRC value to be picked transmission if transmission is configured to use user defined SW CRC.

Parameters

in	*base*	is the FSI Tx module base address

Returns

None.

22.2.4.37 static void FSI_setTxECCdata (uint32_t *base,* uint32_t *data*) [inline], [static]

Sets data for ECC logic computaion.

Parameters

in	*base*	is the FSI Tx module base address
in	*data*	data value for which ECC needs to be computed

Returns

None.

22.2.4.38 static uint16_t FSI_getTxECCValue (uint32_t *base*) [inline], [static]

Returns ECC value evaluated for 16/32 bit data.

Parameters

in	*base*	is the FSI Tx module base address

Returns

ECC value for input data

22.2.4.39 static void FSI_enableTxInterrupt (uint32_t *base,* **FSI_InterruptNum** *intNum,* uint16_t *intFlags*) [inline], [static]

Enables user to generate interrupt on occurrence of FSI_TxEventList events.

Parameters

in	*base*	is the FSI Tx module base address
in	*intNum*	is the type of interrupt to be generated interrupt1 or interrupt2
in	*intFlags*	contains list of events on which interrupt should be generated.

Example Usage intFlags = FSI_TX_EVT_FRAME_DONE && FSI_TX_EVT_BUF_OVERRUN && FSI_TX_EVT_PING_TIMEOUT FSI_enableTxInterrupt(FSI_base, FSI_INT1, intFlags) above configuration will generate signal on interrupt line 1 upon frameDone, BufOverRun and PingTimeOut event

Returns

None.

References FSI_TX_EVTMASK, and FSI_TX_INT2_CTRL_S.

22.2.4.40 static void FSI_disableTxInterrupt (uint32_t *base*, **FSI_InterruptNum** *intNum*, uint16_t *intFlags*) [inline], [static]

Enables user to disable generation interrupt on occurrence of FSI_TxEventList events.

Parameters

in	*base*	is the FSI Tx module base address
in	*intNum*	is the type of interrupt to be generated interrupt1 or interrupt2
in	*intFlags*	contains list of events on which interrupt generation has to be disabled.

Returns

None.

References FSI_TX_EVTMASK, and FSI_TX_INT2_CTRL_S.

22.2.4.41 static uint32_t FSI_getTxBufferAddress (uint32_t *base*) [inline], [static]

Returns address of Tx data buffer.

Data buffer is consisting of 16 words from offset- 0x40 to 0x4e

Parameters

in	*base*	is the FSI Tx module base address

Returns

Tx data buffer address

22.2.4.42 void FSI_resetTxModule (uint32_t *base*, **FSI_TxSubmoduleInReset** *submodule*)

Resets clock or ping timeout counter or entire TX module.

Parameters

in	*base*	is the FSI Tx module base address
in	*moduleName*	the name of submodule which is supposed to be reset

Returns

None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_performTxInitialization().

22.2.4.43 void FSI_clearTxModuleReset (uint32_t *base*, **FSI_TxSubmoduleInReset** *submodule*)

Clears reset on clock or ping timeout counter or entire TX module.

Parameters

in	*base*	is the FSI Tx module base address
in	*moduleName*	the name of submodule, to be brought out of reset

Returns

None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_performTxInitialization().

22.2.4.44 void FSI_writeTxBuffer (uint32_t *base*, const uint16_t *array[ ]*, uint16_t *length*, uint16_t *bufOffset*)

Writes data in FSI Tx buffer.

Parameters

in	*base*	is the FSI Tx module base address
in	*array*	is the address of the array of words to be transmitted.
in	*length*	is the number of words in the array to be transmitted.
in	*bufOffset*	is the offset in Tx buffer where data will be written

Note

Data Overwrite protection is implemented in this function by ensuring not more than 16 words are written and also wrap around case is taken care when more words need to be written if last write happens at maximum offset in Tx buffer

Returns

None.

References FSI_MAX_VALUE_BUF_PTR_OFF.

22.2.4.45 static void FSI_enableRxInternalLoopback (uint32_t *base*) [inline], [static]

Checks the FSI-Rx base address.

Parameters

base	is the base address of the FSI-Rx module

Returns

returns **true** if the base address is valid and **false** otherwise Enables internal loopback where mux will select internal pins coming from TX module instead of what comes from pins

Parameters

in	*base*	is the FSI Rx module base address

Returns

None.

References FSI_CTRL_REG_KEY.

22.2.4.46 static void FSI_disableRxInternalLoopback (uint32_t *base*) [inline], [static]

Disables internal loopback where mux will not use internal pins coming from TX module.

Parameters

in	*base*	is the FSI Rx module base address

Returns

None.

References FSI_CTRL_REG_KEY.

22.2.4.47 static void FSI_enableRxSPIPairing (uint32_t *base*) [inline], [static]

Receive clock is selected from the internal port coming from TX module.

Parameters

in	*base*	is the FSI Rx module base address

Returns

None.

References FSI_CTRL_REG_KEY.

22.2.4.48 static void FSI_disableRxSPIPairing (uint32_t *base*) [inline], [static]

Selects regular receive clock coming from the pins.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

References FSI_CTRL_REG_KEY.

22.2.4.49 static void FSI_setRxDataWidth (uint32_t *base*, **FSI_DataWidth** *dataWidth*) [inline], [static]

Selects number of data lines used for receiving.

Parameters

in	*base*	is the FSI Rx module base address
in	*dataWidth*	selection between 1 or 2 lane receive operation

Returns
None.

22.2.4.50 static void FSI_enableRxSPIMode (uint32_t *base*) [inline], [static]

Enables SPI compatible mode in FSI Rx.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

22.2.4.51 static void FSI_disableRxSPIMode (uint32_t *base*) [inline], [static]

Disables SPI compatible mode in FSI Rx.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

22.2.4.52 static void FSI_setRxSoftwareFrameSize (uint32_t *base*, uint16_t *nWords*) [inline], [static]

Sets the frame size if frame type is user/software defined frame.

Parameters

in	*base*	is the FSI Rx module base address
in	*nWords*	is number of data words in a software defined frame

Returns
None.

References FSI_MAX_LEN_NWORDS_DATA.

22.2.4.53 static void FSI_setRxECCComputeWidth (uint32_t *base*, **FSI_ECCComputeWidth** *eccComputeWidth*) [inline], [static]

Select between 16-bit and 32-bit ECC computation.

Parameters

in	*base*	is the FSI Rx module base address
in	*eccComputeWidth*	is ECC Computation width

Returns
None.

22.2.4.54 static void FSI_setRxPingTimeoutMode (uint32_t *base*, **FSI_PingTimeoutMode** *pingTimeoutMode*) [inline], [static]

Setting for when Ping timeout can reset and restart.

Parameters

in	*base*	is the FSI Rx module base address
in	*pingTimeoutMode*	can be HW or both HW/SW initiated

Returns
None.

22.2.4.55 static **FSI_FrameType** FSI_getRxFrameType (uint32_t *base*) [inline], [static]

Gets frame type received in the last successful frame.

Parameters

in	*base*	is the FSI Rx module base address

Returns
value of Frame type received on last successful frame

22.2.4.56 static void FSI_enableRxDMAEvent (uint32_t *base*) [inline], [static]

Enables to generate DMA event on completion of a successful frame reception.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

22.2.4.57 static void FSI_disableRxDMAEvent (uint32_t *base*) [inline], [static]

Disables the DMA event generation on completion of a successful frame reception.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

22.2.4.58 static uint16_t FSI_getRxFrameTag (uint32_t *base*) [inline], [static]

Returns Frame tag received for the last successful frame.

Parameters

in	*base*	is the FSI Rx module base address

Returns
frame tag value.

22.2.4.59 static uint16_t FSI_getRxUserDefinedData (uint32_t *base*) [inline], [static]

Returns User-Data(8-bit) field for received data frame.

Parameters

in	*base*	is the FSI Rx module base address

Returns

user data field value.

22.2.4.60 static uint16_t FSI_getRxEventStatus (uint32_t *base*) [inline], [static]

Returns current status of all the evetn/error flags.

Parameters

in	*base*	is the FSI Rx module base address

Returns

the status of error flags, each bit of integer is associated with one error flag.

Example Usage - function will set the bits corresponding to respective error flag in return value evtFlags = FSI_getRxEventStatus(FSI_base) if value of evtFlags is 1036(0100000001100) means FSI_RX_EVT_FRAME_OVERRUN, FSI_RX_EVT_TYPE_ERR and FSI_RX_EVT_CRC_ERR flags are set

References FSI_RX_EVTMASK.

22.2.4.61 static void FSI_forceRxEvents (uint32_t *base,* uint16_t *evtFlags*) [inline], [static]

Enables user to set RX event/error flags.

Parameters

in	*base*	is the FSI Rx module base address
in	*evtFlags*	contains list of error flags to be set

Returns

None.

Example Usage evtFlags = FSI_RX_EVT_EOF_ERR && FSI_RX_EVT_TYPE_ERR FSI_forceRxEvents(FSI_base, evtFlags) Above call sets error flag to FSI_RX_ERR_EOF_ERR and FSI_RX_ERR_TYPE_ERR events

References FSI_RX_EVTMASK.

22.2.4.62 static void FSI_clearRxEvents (uint32_t *base,* uint16_t *evtFlags*) [inline], [static]

Enables user to clear RX event/error flags.

Parameters

in	*base*	is the FSI Rx module base address
in	*evtFlags*	contains list of error flags to be cleared

Returns

None.

References FSI_RX_EVTMASK.

Referenced by FSI_performRxInitialization().

22.2.4.63 static uint16_t FSI_getRxReceivedCRC (uint32_t *base*) [inline], [static]

Returns CRC value received in data frame/frame.

Parameters

in	*base*	is the FSI Rx module base address

Returns

CRC value received in data frame

22.2.4.64 static uint16_t FSI_getRxComputedCRC (uint32_t *base*) [inline], [static]

Computes and returns CRC value for data received.

Parameters

in	*base*	is the FSI Rx module base address

Returns

CRC value computed on received data

22.2.4.65 static void FSI_setRxBufferPtr (uint32_t *base*, uint16_t *bufPtrOff*) [inline], [static]

Sets the value for receive buffer pointer at desired location.

Parameters

in	*base*	is the FSI Rx module base address
in	*bufPtrOff*	4 bit offset pointer in Rx buffer from where received data will be read

Returns

None.

References FSI_MAX_VALUE_BUF_PTR_OFF.

22.2.4.66 static uint16_t FSI_getRxBufferPtr (uint32_t *base*) [inline], [static]

Returns current buffer pointer location.

Parameters

`in`	*base*	is the FSI Rx module base address

Returns

current buffer pointer location

Note

there could be lag due to synchronization, hence value is accurate only when no current reception is happening

22.2.4.67 static uint16_t FSI_getRxWordCount (uint32_t *base*) `[inline]`, `[static]`

Returns valid number of data words present in buffer which have not been read out yet.

Parameters

`in`	*base*	is the FSI Rx module base address

Returns

number of data words present in buffer which have not been read out yet

Note

there could be lag due to synchronization, hence value is accurate only when no current reception is happening

22.2.4.68 static void FSI_enableRxFrameWatchdog (uint32_t *base,* uint32_t *wdRef*) `[inline]`, `[static]`

Enables the frame watchdog counter logic to count every time it start to receive a frame.

Parameters

`in`	*base*	is the FSI Rx module base address
`in`	*wdRef*	reference value for ping watchdog time-out counter

Returns

None.

22.2.4.69 static void FSI_disableRxFrameWatchdog (uint32_t *base*) `[inline]`, `[static]`

Disables the frame watchdog counter logic.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

22.2.4.70 static uint32_t FSI_getRxFrameWatchdogCounter (uint32_t *base*) [inline], [static]

Returns current value of frame watchdog counter.

Parameters

in	*base*	is the FSI Rx module base address

Returns
current value of frame watchdog counter

22.2.4.71 static void FSI_enableRxPingWatchdog (uint32_t *base*, uint32_t *wdRef*) [inline], [static]

Enables the ping watchdog counter logic and once the set time elapses it will indicate ping watchdog time-out has occurred.

Parameters

in	*base*	is the FSI Rx module base address
in	*wdRef*	reference value for ping watchdog time-out counter

Returns
None.

22.2.4.72 static void FSI_disableRxPingWatchdog (uint32_t *base*) [inline], [static]

Enables the ping watchdog counter logic and once the set time elapses it will indicate ping watchdog time-out has occurred.

Parameters

in	*base*	is the FSI Rx module base address
in	*wdRef*	reference value for ping watchdog time-out counter

Returns
None.

22.2.4.73 static uint32_t FSI_getRxPingWatchdogCounter (uint32_t *base*) [inline], [static]

Returns current value of ping watchdog counter.

Parameters

in	*base*	is the FSI Rx module base address

Returns
current value(32 bit) of ping watchdog counter

22.2.4.74 static uint16_t FSI_getRxPingTag (uint32_t *base*) [inline], [static]

Returns the value of tag received for last ping frame.

Parameters

in	*base*	is the FSI Rx module base address

Returns
the tagValue received for last ping frame

22.2.4.75 static void FSI_lockRxCtrl (uint32_t *base*) [inline], [static]

Locks the control of all receive control registers, once locked further writes will not take effect until system reset occurs.

Parameters

in	*base*	is the FSI Rx module base address

Returns
None.

References FSI_CTRL_REG_KEY.

22.2.4.76 static void FSI_setRxECCData (uint32_t *base,* uint32_t *rxECCdata*) [inline], [static]

Sets Rx ECC data on which ECC (SEC-DED) computaion logic runs.

Parameters

in	*base*	is the FSI Rx module base address
in	*rxECCdata*	Data for ECC logic

Returns
None.

22.2.4.77 static void FSI_setRxReceivedECCValue (uint32_t *base,* uint16_t *rxECCvalue*) [inline], [static]

Sets received ECC value on which ECC (SEC-DED) computaion logic runs.

Parameters

`in`	*base*	is the FSI Rx module base address
`in`	*rxECCvalue*	Received ECC value in a data frame

Returns
None.

References FSI_MAX_VALUE_USERDATA.

22.2.4.78 static uint32_t FSI_getRxECCCorrectedData (uint32_t *base*) `[inline]`, `[static]`

Returns ECC Corrected data.

Parameters

`in`	*base*	is the FSI Rx module base address

Returns
32 bit ECC corrected data

22.2.4.79 static uint16_t FSI_getRxECCLog (uint32_t *base*) `[inline]`, `[static]`

Returns ECC Log details.

Parameters

`in`	*base*	is the FSI Rx module base address

Returns
ECC Log value(8 bit)

22.2.4.80 static void FSI_enableRxInterrupt (uint32_t *base,* **FSI_InterruptNum** *intNum,* uint16_t *intFlags*) `[inline]`, `[static]`

Let user generate interrupt on occurrence of Rx events.

Parameters

`in`	*base*	is the FSI Rx module base address
`in`	*intNum*	the type of interrupt to be generated interrupt1 or interrupt2
`in`	*intFlags*	contains list of events on which interrupt should be generated. Each bit will represent one event, bits for the events on which user want to generate interrupt will be set others remain clear

Returns
None.

Example Usage evtFlags = FSI_RX_EVT_PING_WD_TIMEOUT & FSI_RX_INT_TYPE_ERR FSI_enableRxInterrupt(FSI_base, FSI_INT1, evtFlags) Above call will generate interrupt1 on events FSI_RX_INT_PING_WD_TIMEOUT and FSI_RX_INT_TYPE_ERR

References FSI_RX_EVTMASK.

22.2.4.81 static void FSI_disableRxInterrupt (uint32_t *base,* **FSI_InterruptNum** *intNum,* uint16_t *intFlags*) [inline], [static]

Let user disable interrupt generation on Rx events.

Parameters

in	*base*	is the FSI Rx module base address
in	*intNum*	the type of interrupt to be generated interrupt1 or interrupt2
in	*intFlags*	contains list of events on which interrupt generation has to be disabled.

Returns
None.

References FSI_RX_EVTMASK.

22.2.4.82 static uint32_t FSI_getRxBufferAddress (uint32_t *base*) [inline], [static]

Returns address of Rx data buffer.

Data buffer is consisting of 16 words from offset- 0x40 to 0x4e

Parameters

in	*base*	is the FSI Rx module base address

Returns
Rx data buffer address

22.2.4.83 void FSI_resetRxModule (uint32_t *base,* **FSI_RxSubmoduleInReset** *submodule*)

Resets frame watchdog, ping watchdog or entire RX module.

Parameters

in	*base*	is the FSI Rx module base address
in	*moduleName*	the name of module which is supposed to be reset

Returns
None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_performRxInitialization().

22.2.4.84 void FSI_clearRxModuleReset (uint32_t *base*, **FSI_RxSubmoduleInReset** *submodule*)

Clears resets on frame watchdog, ping watchdog or entire RX module.

Parameters

in	*base*	is the FSI Rx module base address
in	*moduleName*	module which is to be brought out of reset

Returns
None.

References FSI_CTRL_REG_KEY.

Referenced by FSI_performRxInitialization().

22.2.4.85 void FSI_readRxBuffer (uint32_t *base*, uint16_t *array[ ]*, uint16_t *length*, uint16_t *bufOffset*)

Reads data from FSI Rx buffer.

Parameters

in	*base*	is the FSI Rx module base address
out	*array*	is the address of the array of words to be transmitted.
in	*length*	is the number of words in the array to be transmitted.
in	*bufOffset*	is the offset in Tx buffer where data will be read

Note
This function ensures that not more than 16 words are written and also wrap around case is taken care when more words need to be read if last read happens at maximum offset in Tx buffer

Returns
None.

References FSI_MAX_VALUE_BUF_PTR_OFF.

22.2.4.86 void FSI_configRxDelayLine (uint32_t *base*, **FSI_RxDelayTapType** *delayTapType*, uint16_t *tapValue*)

Adds delay for selected tap line.

Parameters

in	*base*	is the FSI Rx module base address
in	*delayTapType*	the line for which delay needs to be added it can be either RXCLK, RXD0 or RXD1
in	*tapValue*	5 bit value of the amount of delay to be added

Returns

None.

References FSI_RX_MAX_DELAY_LINE_VAL.

22.2.4.87 void FSI_performTxInitialization (uint32_t *base*, uint16_t *prescalar*)

Initializes FSI Tx module.

Software based initialization of the FSI transmitter IP. This is typically needed only once during initialization or if the module needs to be reset due to an underrun condition that occurred during operation.

Parameters

in	*base*	is the FSI Tx module base address
in	*prescalar*	is the user configurable clock divider for PLL input clock

Returns

None.

References FSI_clearTxEvents(), FSI_clearTxModuleReset(), FSI_enableTxClock(), FSI_resetTxModule(), FSI_selectTxPLLClock(), and FSI_TX_EVTMASK.

22.2.4.88 void FSI_performRxInitialization (uint32_t *base*)

Initializes FSI Rx module.

Software based initialization of the FSI receiver module.This is typically needed only once during initialization. However, if there are framing errors in the received frames, then the receive module needs to be reset so that subsequent frames/packets can be handled fresh.

Parameters

in	*base*	is the FSI Rx module base address

Returns

None.

References FSI_clearRxEvents(), FSI_clearRxModuleReset(), FSI_resetRxModule(), and FSI_RX_EVTMASK.

22.2.4.89 void FSI_executeTxFlushSequence (uint32_t *base*, uint16_t *prescalar*)

Sends Flush pattern sequence.

Flush pattern sequence sent by a FSI transmit module will bring the FSI receive module out of reset so that it will then be ready to receive subsequent frames.

Parameters

in	*base*	is the FSI Tx module base address
in	*prescalar*	is the user configurable clock divider for PLL input clock

Returns

None.

References FSI_sendTxFlush(), and FSI_stopTxFlush().

23 GPIO Module

Introduction 412
API Functions 412

23.1 GPIO Introduction

The GPIO module provides an API to configure, read from, and write to the GPIO pins. Functions fall into the two categories, control and data. Control functions configure properties like direction, pin muxing, and qualification. Data functions allow you to read the value on a pin or write a value to it.

Most functions will configure a single pin at a time. The pin to be configured will be specified using its GPIO number. Refer to the device's datasheet to learn what numbers are valid for that part number. Also note that even if a GPIO number is valid for a part number, it may not be valid for all possible features. For instance, GPIO_setAnalogMode() is only usable for a fraction of the GPIO numbers.

For information and functions to configure a pin for low-power mode wake-up, see the SysCtl module.

23.2 API Functions

Enumerations

- enum GPIO_Direction { GPIO_DIR_MODE_IN, GPIO_DIR_MODE_OUT }
- enum GPIO_IntType { GPIO_INT_TYPE_FALLING_EDGE, GPIO_INT_TYPE_RISING_EDGE, GPIO_INT_TYPE_BOTH_EDGES }
- enum GPIO_QualificationMode { GPIO_QUAL_SYNC, GPIO_QUAL_3SAMPLE, GPIO_QUAL_6SAMPLE, GPIO_QUAL_ASYNC }
- enum GPIO_AnalogMode { GPIO_ANALOG_DISABLED, GPIO_ANALOG_ENABLED }
- enum GPIO_CoreSelect { GPIO_CORE_CPU1, GPIO_CORE_CPU1_CLA1 }
- enum GPIO_Port { GPIO_PORT_A, GPIO_PORT_B, GPIO_PORT_H }
- enum GPIO_ExternalIntNum { GPIO_INT_XINT1, GPIO_INT_XINT2, GPIO_INT_XINT3, GPIO_INT_XINT4, GPIO_INT_XINT5 }

Functions

- static void GPIO_setInterruptType (GPIO_ExternalIntNum extIntNum, GPIO_IntType intType)
- static GPIO_IntType GPIO_getInterruptType (GPIO_ExternalIntNum extIntNum)
- static void GPIO_enableInterrupt (GPIO_ExternalIntNum extIntNum)
- static void GPIO_disableInterrupt (GPIO_ExternalIntNum extIntNum)
- static uint32_t GPIO_readPin (uint32_t pin)
- static void GPIO_writePin (uint32_t pin, uint32_t outVal)
- static void GPIO_togglePin (uint32_t pin)
- static uint32_t GPIO_readPortData (GPIO_Port port)
- static void GPIO_writePortData (GPIO_Port port, uint32_t outVal)

- static void GPIO_setPortPins (GPIO_Port port, uint32_t pinMask)
- static void GPIO_clearPortPins (GPIO_Port port, uint32_t pinMask)
- static void GPIO_togglePortPins (GPIO_Port port, uint32_t pinMask)
- static void GPIO_lockPortConfig (GPIO_Port port, uint32_t pinMask)
- static void GPIO_unlockPortConfig (GPIO_Port port, uint32_t pinMask)
- static void GPIO_commitPortConfig (GPIO_Port port, uint32_t pinMask)
- void GPIO_setDirectionMode (uint32_t pin, GPIO_Direction pinIO)
- GPIO_Direction GPIO_getDirectionMode (uint32_t pin)
- void GPIO_setInterruptPin (uint32_t pin, GPIO_ExternalIntNum extIntNum)
- void GPIO_setPadConfig (uint32_t pin, uint32_t pinType)
- uint32_t GPIO_getPadConfig (uint32_t pin)
- void GPIO_setQualificationMode (uint32_t pin, GPIO_QualificationMode qualification)
- GPIO_QualificationMode GPIO_getQualificationMode (uint32_t pin)
- void GPIO_setQualificationPeriod (uint32_t pin, uint32_t divider)
- void GPIO_setMasterCore (uint32_t pin, GPIO_CoreSelect core)
- void GPIO_setAnalogMode (uint32_t pin, GPIO_AnalogMode mode)
- void GPIO_setPinConfig (uint32_t pinConfig)

23.2.1 Detailed Description

The first step to configuring GPIO is to figure out the peripheral muxing. The function to configure the mux registers is GPIO_setPinConfig(). The values to be passed to this function to specify the functionality the pin should have are found in pin_map.h.

Next, use GPIO_setPadConfig() to configure any properties like internal pullups, open-drain, or an inverted input signal. GPIO_setQualificationMode() and GPIO_setQualificationPeriod() can be used to configure any needed input qualification.

Then, for pins configured as GPIOs, use GPIO_setDirectionMode() to select a direction. Take care to write the desired initial value for that pin using GPIO_writePin() before configuring a pin as an output to avoid any glitches.

Several functions are provided for the configuration of external interrupts. These functions use the device's XINT module. The Input X-BAR is also leveraged to configure the pin on which an event will cause an interrupt. These functions are GPIO_setInterruptType(), GPIO_getInterruptType(), GPIO_enableInterrupt(), GPIO_disableInterrupt(), and GPIO_setInterruptPin().

Most functions operate on one pin at a time. However, there are a few functions that can operate on an entire port at once for the sake of efficiency. These are the data functions GPIO_readPortData(), GPIO_writePortData(), GPIO_setPortPins(), GPIO_clearPortPins(), and GPIO_togglePortPins(). Other data functions that affect a single pin at a time are GPIO_readPin(), GPIO_writePin(), and GPIO_togglePin().

The code for this module is contained in `driverlib/gpio.c`, with `driverlib/gpio.h` containing the API declarations for use by applications.

23.2.2 Enumeration Type Documentation

23.2.2.1 enum **GPIO_Direction**

Values that can be passed to GPIO_setDirectionMode() as the *pinIO* parameter and returned from GPIO_getDirectionMode().

Enumerator

GPIO_DIR_MODE_IN Pin is a GPIO input.

GPIO_DIR_MODE_OUT Pin is a GPIO output.

23.2.2.2 enum **GPIO_IntType**

Values that can be passed to GPIO_setInterruptType() as the *intType* parameter and returned from GPIO_getInterruptType().

Enumerator

GPIO_INT_TYPE_FALLING_EDGE Interrupt on falling edge.

GPIO_INT_TYPE_RISING_EDGE Interrupt on rising edge.

GPIO_INT_TYPE_BOTH_EDGES Interrupt on both edges.

23.2.2.3 enum **GPIO_QualificationMode**

Values that can be passed to GPIO_setQualificationMode() as the *qualification* parameter and returned by GPIO_getQualificationMode().

Enumerator

GPIO_QUAL_SYNC Synchronization to SYSCLKOUT.

GPIO_QUAL_3SAMPLE Qualified with 3 samples.

GPIO_QUAL_6SAMPLE Qualified with 6 samples.

GPIO_QUAL_ASYNC No synchronization.

23.2.2.4 enum **GPIO_AnalogMode**

Values that can be passed to GPIO_setAnalogMode() as the *mode* parameter.

Enumerator

GPIO_ANALOG_DISABLED Pin is in digital mode.

GPIO_ANALOG_ENABLED Pin is in analog mode.

23.2.2.5 enum **GPIO_CoreSelect**

Values that can be passed to GPIO_setMasterCore() as the *core* parameter.

Enumerator

GPIO_CORE_CPU1 CPU1 selected as master core.

GPIO_CORE_CPU1_CLA1 CPU1's CLA1 selected as master core.

23.2.2.6 enum **GPIO_Port**

Values that can be passed to GPIO_readPortData(), GPIO_setPortPins(), GPIO_clearPortPins(), and GPIO_togglePortPins() as the *port* parameter.

Enumerator

GPIO_PORT_A GPIO port A.
GPIO_PORT_B GPIO port B.
GPIO_PORT_H GPIO port H.

23.2.2.7 enum **GPIO_ExternalIntNum**

Values that can be passed to GPIO_setInterruptPin(), GPIO_setInterruptType(), GPIO_getInterruptType(), GPIO_enableInterrupt(), GPIO_disableInterrupt(), as the *extIntNum* parameter.

Enumerator

GPIO_INT_XINT1 External Interrupt 1.
GPIO_INT_XINT2 External Interrupt 2.
GPIO_INT_XINT3 External Interrupt 3.
GPIO_INT_XINT4 External Interrupt 4.
GPIO_INT_XINT5 External Interrupt 5.

23.2.3 Function Documentation

23.2.3.1 static void GPIO_setInterruptType (**GPIO_ExternalIntNum** *extIntNum,* **GPIO_IntType** *intType*) [inline], [static]

Sets the interrupt type for the specified pin.

Parameters

extIntNum	specifies the external interrupt.
intType	specifies the type of interrupt trigger mechanism.

This function sets up the various interrupt trigger mechanisms for the specified pin on the selected GPIO port.

The following defines can be used to specify the external interrupt for the *extIntNum* parameter:

- **GPIO_INT_XINT1**
- **GPIO_INT_XINT2**
- **GPIO_INT_XINT3**
- **GPIO_INT_XINT4**
- **GPIO_INT_XINT5**

One of the following flags can be used to define the *intType* parameter:

- **GPIO_INT_TYPE_FALLING_EDGE** sets detection to edge and trigger to falling
- **GPIO_INT_TYPE_RISING_EDGE** sets detection to edge and trigger to rising

- **GPIO_INT_TYPE_BOTH_EDGES** sets detection to both edges

Returns
None.

23.2.3.2 static **GPIO_IntType** GPIO_getInterruptType (**GPIO_ExternalIntNum** *extIntNum*) [inline], [static]

Gets the interrupt type for a pin.

Parameters

extIntNum	specifies the external interrupt.

This function gets the interrupt type for a interrupt. The interrupt can be configured as a falling-edge, rising-edge, or both-edges detected interrupt.

The following defines can be used to specify the external interrupt for the *extIntNum* parameter:

- **GPIO_INT_XINT1**
- **GPIO_INT_XINT2**
- **GPIO_INT_XINT3**
- **GPIO_INT_XINT4**
- **GPIO_INT_XINT5**

Returns
Returns one of the flags described for GPIO_setInterruptType().

23.2.3.3 static void GPIO_enableInterrupt (**GPIO_ExternalIntNum** *extIntNum*) [inline], [static]

Enables the specified external interrupt.

Parameters

extIntNum	specifies the external interrupt.

This function enables the indicated external interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The following defines can be used to specify the external interrupt for the *extIntNum* parameter:

- **GPIO_INT_XINT1**
- **GPIO_INT_XINT2**
- **GPIO_INT_XINT3**
- **GPIO_INT_XINT4**
- **GPIO_INT_XINT5**

Returns
None.

23.2.3.4 static void GPIO_disableInterrupt (**GPIO_ExternalIntNum** *extIntNum*) `[inline]`, `[static]`

Disables the specified external interrupt.

Parameters

extIntNum	specifies the external interrupt.

This function disables the indicated external interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The following defines can be used to specify the external interrupt for the *extIntNum* parameter:

- **GPIO_INT_XINT1**
- **GPIO_INT_XINT2**
- **GPIO_INT_XINT3**
- **GPIO_INT_XINT4**
- **GPIO_INT_XINT5**

Returns

None.

23.2.3.5 static uint32_t GPIO_readPin (uint32_t *pin*) `[inline]`, `[static]`

Reads the value present on the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.

The value at the specified pin are read, as specified by *pin*. The value is returned for both input and output pins.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns

Returns the value in the data register for the specified pin.

23.2.3.6 static void GPIO_writePin (uint32_t *pin,* uint32_t *outVal*) `[inline]`, `[static]`

Writes a value to the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
outVal	is the value to write to the pin.

Writes the corresponding bit values to the output pin specified by *pin*. Writing to a pin configured as an input pin has no effect.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns

None.

23.2.3.7 static void GPIO_togglePin (uint32_t *pin*) `[inline]`, `[static]`

Toggles the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.

Writes the corresponding bit values to the output pin specified by *pin*. Writing to a pin configured as an input pin has no effect.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns

None.

23.2.3.8 static uint32_t GPIO_readPortData (**GPIO_Port** *port*) `[inline]`, `[static]`

Reads the data on the specified port.

Parameters

port	is the GPIO port being accessed in the form of **GPIO_PORT_X** where X is the port letter.

Returns

Returns the value in the data register for the specified port. Each bit of the the return value represents a pin on the port, where bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

23.2.3.9 static void GPIO_writePortData (**GPIO_Port** *port,* uint32_t *outVal*) `[inline]`, `[static]`

Writes a value to the specified port.

Parameters

port	is the GPIO port being accessed.
outVal	is the value to write to the port.

This function writes the value *outVal* to the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *outVal* is a bit-packed value, where each bit represents a bit on a GPIO port. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns

None.

23.2.3.10 static void GPIO_setPortPins (**GPIO_Port** *port,* uint32_t *pinMask*) `[inline]`, `[static]`

Sets all of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function sets all of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is set identifies the pin to be set. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns
None.

23.2.3.11 static void GPIO_clearPortPins (**GPIO_Port** *port,* uint32_t *pinMask*) `[inline]`, `[static]`

Clears all of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function clears all of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is **set** identifies the pin to be cleared. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns
None.

23.2.3.12 static void GPIO_togglePortPins (**GPIO_Port** *port,* uint32_t *pinMask*) `[inline]`, `[static]`

Toggles all of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function toggles all of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is set identifies the pin to be toggled. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns
None.

23.2.3.13 static void GPIO_lockPortConfig (**GPIO_Port** *port,* uint32_t *pinMask*) [inline], [static]

Locks the configuration of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function locks the configuration registers of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is set identifies the pin to be locked. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, 0xFFFFFFFF represents all pins on that port, and so on.

Note that this function is for locking the configuration of a pin such as the pin muxing, direction, open drain mode, and other settings. It does not affect the ability to change the value of the pin.

Returns
None.

23.2.3.14 static void GPIO_unlockPortConfig (**GPIO_Port** *port,* uint32_t *pinMask*) `[inline]`, `[static]`

Unlocks the configuration of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function locks the configuration registers of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is set identifies the pin to be unlocked. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, 0xFFFFFFFF represents all pins on that port, and so on.

Returns
None.

23.2.3.15 static void GPIO_commitPortConfig (**GPIO_Port** *port,* uint32_t *pinMask*) `[inline]`, `[static]`

Commits the lock configuration of the specified pins on the specified port.

Parameters

port	is the GPIO port being accessed.
pinMask	is a mask of which of the 32 pins on the port are affected.

This function commits the lock configuration registers of the pins specified by the *pinMask* parameter on the port specified by the *port* parameter which takes a value in the form of **GPIO_PORT_X** where X is the port letter. For example, use **GPIO_PORT_A** to affect port A (GPIOs 0-31).

The *pinMask* is a bit-packed value, where each bit that is set identifies the pin to be locked. Bit 0 represents GPIO port pin 0, bit 1 represents GPIO port pin 1, 0xFFFFFFFF represents all pins on that port, and so on.

Note that once this function is called, GPIO_lockPortConfig() and GPIO_unlockPortConfig() will no longer have any effect on the specified pins.

Returns
None.

23.2.3.16 void GPIO_setDirectionMode (uint32_t *pin*, **GPIO_Direction** *pinIO*)

Sets the direction and mode of the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
pinIO	is the pin direction mode.

This function configures the specified pin on the selected GPIO port as either input or output.

The parameter *pinIO* is an enumerated data type that can be one of the following values:

- **GPIO_DIR_MODE_IN**
- **GPIO_DIR_MODE_OUT**

where **GPIO_DIR_MODE_IN** specifies that the pin is programmed as an input and **GPIO_DIR_MODE_OUT** specifies that the pin is programmed as an output.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns
None.

References GPIO_DIR_MODE_OUT.

23.2.3.17 **GPIO_Direction** GPIO_getDirectionMode (uint32_t *pin*)

Gets the direction mode of a pin.

Parameters

pin	is the identifying GPIO number of the pin.

This function gets the direction mode for a specified pin. The pin can be configured as either an input or output The type of direction is returned as an enumerated data type.

Returns
Returns one of the enumerated data types described for GPIO_setDirectionMode().

23.2.3.18 void GPIO_setInterruptPin (uint32_t *pin*, **GPIO_ExternalIntNum** *extIntNum*)

Sets the pin for the specified external interrupt.

Parameters

pin	is the identifying GPIO number of the pin.
extIntNum	specifies the external interrupt.

This function sets which pin triggers the selected external interrupt.

The following defines can be used to specify the external interrupt for the *extIntNum* parameter:

- **GPIO_INT_XINT1**
- **GPIO_INT_XINT2**
- **GPIO_INT_XINT3**
- **GPIO_INT_XINT4**
- **GPIO_INT_XINT5**

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

See Also
XBAR_setInputPin()

Returns
None.

References GPIO_INT_XINT1, GPIO_INT_XINT2, GPIO_INT_XINT3, GPIO_INT_XINT4, GPIO_INT_XINT5, XBAR_INPUT1, XBAR_INPUT13, XBAR_INPUT14, XBAR_INPUT4, XBAR_INPUT5, XBAR_INPUT6, and XBAR_setInputPin().

23.2.3.19 void GPIO_setPadConfig (uint32_t *pin*, uint32_t *pinType*)

Sets the pad configuration for the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
pinType	specifies the pin type.

This function sets the pin type for the specified pin. The parameter *pinType* can be the following values:

- **GPIO_PIN_TYPE_STD** specifies a push-pull output or a floating input
- **GPIO_PIN_TYPE_PULLUP** specifies the pull-up is enabled for an input
- **GPIO_PIN_TYPE_OD** specifies an open-drain output pin
- **GPIO_PIN_TYPE_INVERT** specifies inverted polarity on an input

GPIO_PIN_TYPE_INVERT may be OR-ed with **GPIO_PIN_TYPE_STD** or **GPIO_PIN_TYPE_PULLUP**.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns
None.

23.2.3.20 uint32_t GPIO_getPadConfig (uint32_t *pin*)

Gets the pad configuration for a pin.

Parameters

pin	is the identifying GPIO number of the pin.

This function returns the pin type for the specified pin. The value returned corresponds to the values used in GPIO_setPadConfig().

Returns

Returns a bit field of the values **GPIO_PIN_TYPE_STD**, **GPIO_PIN_TYPE_PULLUP**, **GPIO_PIN_TYPE_OD**, and **GPIO_PIN_TYPE_INVERT**.

23.2.3.21 void GPIO_setQualificationMode (uint32_t *pin,* **GPIO_QualificationMode** *qualification*)

Sets the qualification mode for the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
qualification	specifies the qualification mode of the pin.

This function sets the qualification mode for the specified pin. The parameter *qualification* can be one of the following values:

- **GPIO_QUAL_SYNC**
- **GPIO_QUAL_3SAMPLE**
- **GPIO_QUAL_6SAMPLE**
- **GPIO_QUAL_ASYNC**

To set the qualification sampling period, use GPIO_setQualificationPeriod().

Returns

None.

23.2.3.22 **GPIO_QualificationMode** GPIO_getQualificationMode (uint32_t *pin*)

Gets the qualification type for the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.

Returns

Returns the qualification mode in the form of one of the values **GPIO_QUAL_SYNC**, **GPIO_QUAL_3SAMPLE**, **GPIO_QUAL_6SAMPLE**, or **GPIO_QUAL_ASYNC**.

23.2.3.23 void GPIO_setQualificationPeriod (uint32_t *pin,* uint32_t *divider*)

Sets the qualification period for a set of pins

Parameters

pin	is the identifying GPIO number of the pin.
divider	specifies the output drive strength.

This function sets the qualification period for a set of **8 pins**, specified by the *pin* parameter. For instance, passing in 3 as the value of *pin* will set the qualification period for GPIO0 through GPIO7, and a value of 98 will set the qualification period for GPIO96 through GPIO103. This is because the register field that configures the divider is shared.

To think of this in terms of an equation, configuring *pin* as **n** will configure GPIO (n & ∼(7)) through GPIO ((n & ∼(7)) + 7).

divider is the value by which the frequency of SYSCLKOUT is divided. It can be 1 or an even value between 2 and 510 inclusive.

Returns
None.

23.2.3.24 void GPIO_setMasterCore (uint32_t *pin*, **GPIO_CoreSelect** *core*)

Selects the master core of a specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
core	is the core that is master of the specified pin.

This function configures which core owns the specified pin's data registers (DATA, SET, CLEAR, and TOGGLE). The *core* parameter is an enumerated data type that specifies the core, such as **GPIO_CORE_CPU1_CLA1** to make CPU1's CLA1 master of the pin.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns
None.

23.2.3.25 void GPIO_setAnalogMode (uint32_t *pin*, **GPIO_AnalogMode** *mode*)

Sets the analog mode of the specified pin.

Parameters

pin	is the identifying GPIO number of the pin.
mode	is the selected analog mode.

This function configures the specified pin for either analog or digital mode. Not all GPIO pins have the ability to be switched to analog mode, so refer to the technical reference manual for details. This setting should be thought of as another level of muxing.

The parameter *mode* is an enumerated data type that can be one of the following values:

- **GPIO_ANALOG_DISABLED** - Pin is in digital mode
- **GPIO_ANALOG_ENABLED** - Pin is in analog mode

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns
None.

References GPIO_ANALOG_ENABLED.

23.2.3.26 void GPIO_setPinConfig (uint32_t *pinConfig*)

Configures the alternate function of a GPIO pin.

Parameters

pinConfig	is the pin configuration value, specified as only one of the **GPIO_::_**???? values.

This function configures the pin mux that selects the peripheral function associated with a particular GPIO pin. Only one peripheral function at a time can be associated with a GPIO pin, and each peripheral function should only be associated with a single GPIO pin at a time (despite the fact that many of them can be associated with more than one GPIO pin).

The available mappings are supplied in `pin_map.h`.

Returns
None.

24 I2C Module

Introduction . . . 431
API Functions . . . 431

24.1 I2C Introduction

The inter-integrated circuit (I2C) API provides a set of functions to configure the device's I2C module. The driver supports operation in both master and slave mode and provides functions to initialize the module, to send and receive data, to obtain status information, and to manage interrupts.

24.2 API Functions

Enumerations

- enum I2C_InterruptSource {
  I2C_INTSRC_NONE, I2C_INTSRC_ARB_LOST, I2C_INTSRC_NO_ACK,
  I2C_INTSRC_REG_ACCESS_RDY,
  I2C_INTSRC_RX_DATA_RDY, I2C_INTSRC_TX_DATA_RDY,
  I2C_INTSRC_STOP_CONDITION, I2C_INTSRC_ADDR_SLAVE }
- enum I2C_TxFIFOLevel {
  I2C_FIFO_TXEMPTY, I2C_FIFO_TX0, I2C_FIFO_TX1, I2C_FIFO_TX2,
  I2C_FIFO_TX3, I2C_FIFO_TX4, I2C_FIFO_TX5, I2C_FIFO_TX6,
  I2C_FIFO_TX7, I2C_FIFO_TX8, I2C_FIFO_TX9, I2C_FIFO_TX10,
  I2C_FIFO_TX11, I2C_FIFO_TX12, I2C_FIFO_TX13, I2C_FIFO_TX14,
  I2C_FIFO_TX15, I2C_FIFO_TX16, I2C_FIFO_TXFULL }
- enum I2C_RxFIFOLevel {
  I2C_FIFO_RXEMPTY, I2C_FIFO_RX0, I2C_FIFO_RX1, I2C_FIFO_RX2,
  I2C_FIFO_RX3, I2C_FIFO_RX4, I2C_FIFO_RX5, I2C_FIFO_RX6,
  I2C_FIFO_RX7, I2C_FIFO_RX8, I2C_FIFO_RX9, I2C_FIFO_RX10,
  I2C_FIFO_RX11, I2C_FIFO_RX12, I2C_FIFO_RX13, I2C_FIFO_RX14,
  I2C_FIFO_RX15, I2C_FIFO_RX16, I2C_FIFO_RXFULL }
- enum I2C_BitCount {
  I2C_BITCOUNT_1, I2C_BITCOUNT_2, I2C_BITCOUNT_3, I2C_BITCOUNT_4,
  I2C_BITCOUNT_5, I2C_BITCOUNT_6, I2C_BITCOUNT_7, I2C_BITCOUNT_8 }
- enum I2C_AddressMode { I2C_ADDR_MODE_7BITS, I2C_ADDR_MODE_10BITS }
- enum I2C_EmulationMode { I2C_EMULATION_STOP_SCL_LOW,
  I2C_EMULATION_FREE_RUN }
- enum I2C_DutyCycle { I2C_DUTYCYCLE_33, I2C_DUTYCYCLE_50 }

Functions

- static void I2C_enableModule (uint32_t base)
- static void I2C_disableModule (uint32_t base)
- static void I2C_enableFIFO (uint32_t base)

- static void I2C_disableFIFO (uint32_t base)
- static void I2C_setFIFOInterruptLevel (uint32_t base, I2C_TxFIFOLevel txLevel, I2C_RxFIFOLevel rxLevel)
- static void I2C_getFIFOInterruptLevel (uint32_t base, I2C_TxFIFOLevel ∗txLevel, I2C_RxFIFOLevel ∗rxLevel)
- static I2C_TxFIFOLevel I2C_getTxFIFOStatus (uint32_t base)
- static I2C_RxFIFOLevel I2C_getRxFIFOStatus (uint32_t base)
- static void I2C_setSlaveAddress (uint32_t base, uint16_t slaveAddr)
- static void I2C_setOwnSlaveAddress (uint32_t base, uint16_t slaveAddr)
- static bool I2C_isBusBusy (uint32_t base)
- static uint16_t I2C_getStatus (uint32_t base)
- static void I2C_clearStatus (uint32_t base, uint16_t stsFlags)
- static void I2C_setConfig (uint32_t base, uint16_t config)
- static void I2C_setBitCount (uint32_t base, I2C_BitCount size)
- static void I2C_sendStartCondition (uint32_t base)
- static void I2C_sendStopCondition (uint32_t base)
- static void I2C_sendNACK (uint32_t base)
- static uint16_t I2C_getData (uint32_t base)
- static void I2C_putData (uint32_t base, uint16_t data)
- static bool I2C_getStopConditionStatus (uint32_t base)
- static void I2C_setDataCount (uint32_t base, uint16_t count)
- static void I2C_setAddressMode (uint32_t base, I2C_AddressMode mode)
- static void I2C_setEmulationMode (uint32_t base, I2C_EmulationMode mode)
- static void I2C_enableLoopback (uint32_t base)
- static void I2C_disableLoopback (uint32_t base)
- static I2C_InterruptSource I2C_getInterruptSource (uint32_t base)
- void I2C_initMaster (uint32_t base, uint32_t sysclkHz, uint32_t bitRate, I2C_DutyCycle dutyCycle)
- void I2C_enableInterrupt (uint32_t base, uint32_t intFlags)
- void I2C_disableInterrupt (uint32_t base, uint32_t intFlags)
- uint32_t I2C_getInterruptStatus (uint32_t base)
- void I2C_clearInterruptStatus (uint32_t base, uint32_t intFlags)

24.2.1 Detailed Description

Before initializing the I2C module, the user first must put the module into the reset state by calling I2C_disableModule(). When using the API in master mode, the user must then call I2C_initMaster() which will configure the rate and duty cycle of the master clock. For slave mode, I2C_setOwnSlaveAddress() will need to be called to set the module's address.

For both modes, this is also the time to do any FIFO or interrupt configuration. FIFOs are configured using I2C_enableFIFO() and I2C_disableFIFO() and I2C_setFIFOInterruptLevel() if interrupts are desired. The functions I2C_enableInterrupt(), I2C_disableInterrupt(), I2C_clearInterruptStatus(), and I2C_getInterruptStatus() are for management of interrupts. Note that the I2C module uses separate interrupt lines for its basic and FIFO interrupts although the functions to configure them are the same.

When configuration is complete, I2C_enableModule() should be called to enable the operation of the module.

To do a transfer, for both master and slave modes, I2C_setConfig() should be called to configure the behavior of the module. A master will need to set I2C_setSlaveAddress() to set the address of the slave to which it will communicate. I2C_putData() will place data in the transmit buffer. A start condition can be sent by a master using I2C_sendStartCondition().

When receiving data, the status of data received can be checked using I2C_getStatus() or if in FIFO mode, I2C_getRxFIFOStatus(). I2C_getData() will read the data from the receive buffer and return it.

The code for this module is contained in `driverlib/i2c.c`, with `driverlib/i2c.h` containing the API declarations for use by applications.

24.2.2 Enumeration Type Documentation

24.2.2.1 enum **I2C_InterruptSource**

I2C interrupts to be returned by I2C_getInterruptSource().

Enumerator

I2C_INTSRC_NONE No interrupt pending.
I2C_INTSRC_ARB_LOST Arbitration-lost interrupt.
I2C_INTSRC_NO_ACK NACK interrupt.
I2C_INTSRC_REG_ACCESS_RDY Register-access-ready interrupt.
I2C_INTSRC_RX_DATA_RDY Receive-data-ready interrupt.
I2C_INTSRC_TX_DATA_RDY Transmit-data-ready interrupt.
I2C_INTSRC_STOP_CONDITION Stop condition detected.
I2C_INTSRC_ADDR_SLAVE Addressed as slave interrupt.

24.2.2.2 enum **I2C_TxFIFOLevel**

Values that can be passed to I2C_setFIFOInterruptLevel() as the *txLevel* parameter, returned by I2C_getFIFOInterruptLevel() in the *txLevel* parameter, and returned by I2C_getTxFIFOStatus().

Enumerator

I2C_FIFO_TXEMPTY Transmit FIFO empty.
I2C_FIFO_TX0 Transmit FIFO empty.
I2C_FIFO_TX1 Transmit FIFO 1/16 full.
I2C_FIFO_TX2 Transmit FIFO 2/16 full.
I2C_FIFO_TX3 Transmit FIFO 3/16 full.
I2C_FIFO_TX4 Transmit FIFO 4/16 full.
I2C_FIFO_TX5 Transmit FIFO 5/16 full.
I2C_FIFO_TX6 Transmit FIFO 6/16 full.
I2C_FIFO_TX7 Transmit FIFO 7/16 full.
I2C_FIFO_TX8 Transmit FIFO 8/16 full.
I2C_FIFO_TX9 Transmit FIFO 9/16 full.
I2C_FIFO_TX10 Transmit FIFO 10/16 full.
I2C_FIFO_TX11 Transmit FIFO 11/16 full.
I2C_FIFO_TX12 Transmit FIFO 12/16 full.
I2C_FIFO_TX13 Transmit FIFO 13/16 full.
I2C_FIFO_TX14 Transmit FIFO 14/16 full.
I2C_FIFO_TX15 Transmit FIFO 15/16 full.

I2C_FIFO_TX16 Transmit FIFO full.

I2C_FIFO_TXFULL Transmit FIFO full.

24.2.2.3 enum **I2C_RxFIFOLevel**

Values that can be passed to I2C_setFIFOInterruptLevel() as the *rxLevel* parameter, returned by I2C_getFIFOInterruptLevel() in the *rxLevel* parameter, and returned by I2C_getRxFIFOStatus().

Enumerator

I2C_FIFO_RXEMPTY Receive FIFO empty.

I2C_FIFO_RX0 Receive FIFO empty.

I2C_FIFO_RX1 Receive FIFO 1/16 full.

I2C_FIFO_RX2 Receive FIFO 2/16 full.

I2C_FIFO_RX3 Receive FIFO 3/16 full.

I2C_FIFO_RX4 Receive FIFO 4/16 full.

I2C_FIFO_RX5 Receive FIFO 5/16 full.

I2C_FIFO_RX6 Receive FIFO 6/16 full.

I2C_FIFO_RX7 Receive FIFO 7/16 full.

I2C_FIFO_RX8 Receive FIFO 8/16 full.

I2C_FIFO_RX9 Receive FIFO 9/16 full.

I2C_FIFO_RX10 Receive FIFO 10/16 full.

I2C_FIFO_RX11 Receive FIFO 11/16 full.

I2C_FIFO_RX12 Receive FIFO 12/16 full.

I2C_FIFO_RX13 Receive FIFO 13/16 full.

I2C_FIFO_RX14 Receive FIFO 14/16 full.

I2C_FIFO_RX15 Receive FIFO 15/16 full.

I2C_FIFO_RX16 Receive FIFO full.

I2C_FIFO_RXFULL Receive FIFO full.

24.2.2.4 enum **I2C_BitCount**

Values that can be passed to I2C_setBitCount() as the *size* parameter.

Enumerator

I2C_BITCOUNT_1 1 bit per data byte

I2C_BITCOUNT_2 2 bits per data byte

I2C_BITCOUNT_3 3 bits per data byte

I2C_BITCOUNT_4 4 bits per data byte

I2C_BITCOUNT_5 5 bits per data byte

I2C_BITCOUNT_6 6 bits per data byte

I2C_BITCOUNT_7 7 bits per data byte

I2C_BITCOUNT_8 8 bits per data byte

24.2.2.5 enum **I2C_AddressMode**

Values that can be passed to I2C_setAddressMode() as the *mode* parameter.

Enumerator

I2C_ADDR_MODE_7BITS 7-bit address

I2C_ADDR_MODE_10BITS 10-bit address

24.2.2.6 enum **I2C_EmulationMode**

Values that can be passed to I2C_setEmulationMode() as the *mode* parameter.

Enumerator

I2C_EMULATION_STOP_SCL_LOW If SCL is low, keep it low. If high, stop when it goes low again.

I2C_EMULATION_FREE_RUN Continue I2C operation regardless.

24.2.2.7 enum **I2C_DutyCycle**

Values that can be passed to I2C_initMaster() as the *dutyCycle* parameter.

Enumerator

I2C_DUTYCYCLE_33 Clock duty cycle is 33%.

I2C_DUTYCYCLE_50 Clock duty cycle is 55%.

24.2.3 Function Documentation

24.2.3.1 static void I2C_enableModule (uint32_t *base*) `[inline]`, `[static]`

Enables the I2C module.

Parameters

base	is the base address of the I2C instance used.

This function enables operation of the I2C module.

Returns

None.

24.2.3.2 static void I2C_disableModule (uint32_t *base*) `[inline]`, `[static]`

Disables the I2C module.

Parameters

base	is the base address of the I2C instance used.

This function disables operation of the I2C module.

Returns

None.

24.2.3.3 static void I2C_enableFIFO (uint32_t *base*) [inline], [static]

Enables the transmit and receive FIFOs.

Parameters

base	is the base address of the I2C instance used.

This functions enables the transmit and receive FIFOs in the I2C.

Returns

None.

24.2.3.4 static void I2C_disableFIFO (uint32_t *base*) [inline], [static]

Disables the transmit and receive FIFOs.

Parameters

base	is the base address of the I2C instance used.

This functions disables the transmit and receive FIFOs in the I2C.

Returns

None.

24.2.3.5 static void I2C_setFIFOInterruptLevel (uint32_t *base,* **I2C_TxFIFOLevel** *txLevel,* **I2C_RxFIFOLevel** *rxLevel*) [inline], [static]

Sets the FIFO level at which interrupts are generated.

Parameters

base	is the base address of the I2C instance used.
txLevel	is the transmit FIFO interrupt level, specified as **I2C_FIFO_TX0**, **I2C_FIFO_TX1**, **I2C_FIFO_TX2**, . . . or **I2C_FIFO_TX16**.
rxLevel	is the receive FIFO interrupt level, specified as **I2C_FIFO_RX0**, **I2C_FIFO_RX1**, **I2C_FIFO_RX2**, . . . or **I2C_FIFO_RX16**.

This function sets the FIFO level at which transmit and receive interrupts are generated. The transmit FIFO interrupt flag will be set when the FIFO reaches a value less than or equal to *txLevel*. The receive FIFO flag will be set when the FIFO reaches a value greater than or equal to *rxLevel*.

Returns

None.

24.2.3.6 static void I2C_getFIFOInterruptLevel (uint32_t *base*, **I2C_TxFIFOLevel** ∗ *txLevel*, **I2C_RxFIFOLevel** ∗ *rxLevel*) [inline], [static]

Gets the FIFO level at which interrupts are generated.

Parameters

base	is the base address of the I2C instance used.
txLevel	is a pointer to storage for the transmit FIFO level, returned as one of **I2C_FIFO_TX0**, **I2C_FIFO_TX1**, **I2C_FIFO_TX2**, . . . or **I2C_FIFO_TX16**.
rxLevel	is a pointer to storage for the receive FIFO level, returned as one of **I2C_FIFO_RX0**, **I2C_FIFO_RX1**, **I2C_FIFO_RX2**, . . . or **I2C_FIFO_RX16**.

This function gets the FIFO level at which transmit and receive interrupts are generated. The transmit FIFO interrupt flag will be set when the FIFO reaches a value less than or equal to *txLevel*. The receive FIFO flag will be set when the FIFO reaches a value greater than or equal to *rxLevel*.

Returns

None.

24.2.3.7 static **I2C_TxFIFOLevel** I2C_getTxFIFOStatus (uint32_t *base*) [inline], [static]

Get the transmit FIFO status

Parameters

base	is the base address of the I2C instance used.

This function gets the current number of words in the transmit FIFO.

Returns

Returns the current number of words in the transmit FIFO specified as one of the following: **I2C_FIFO_TX0**, **I2C_FIFO_TX1**, **I2C_FIFO_TX2**, **I2C_FIFO_TX3**, ..., or **I2C_FIFO_TX16**

24.2.3.8 static **I2C_RxFIFOLevel** I2C_getRxFIFOStatus (uint32_t *base*) [inline], [static]

Get the receive FIFO status

Parameters

base	is the base address of the I2C instance used.

This function gets the current number of words in the receive FIFO.

Returns

Returns the current number of words in the receive FIFO specified as one of the following: **I2C_FIFO_RX0**, **I2C_FIFO_RX1**, **I2C_FIFO_RX2**, **I2C_FIFO_RX3**, ..., or **I2C_FIFO_RX16**

24.2.3.9 static void I2C_setSlaveAddress (uint32_t *base*, uint16_t *slaveAddr*) [inline], [static]

Sets the address that the I2C Master places on the bus.

Parameters

base	is the base address of the I2C instance used.
slaveAddr	7-bit or 10-bit slave address

This function configures the address that the I2C Master places on the bus when initiating a transaction.

Returns

None.

24.2.3.10 static void I2C_setOwnSlaveAddress (uint32_t *base,* uint16_t *slaveAddr*) [inline], [static]

Sets the slave address for this I2C module.

Parameters

base	is the base address of the I2C Slave module.
slaveAddr	is the 7-bit or 10-bit slave address

This function writes the specified slave address.

The parameter *slaveAddr* is the value that is compared against the slave address sent by an I2C master.

Returns

None.

24.2.3.11 static bool I2C_isBusBusy (uint32_t *base*) [inline], [static]

Indicates whether or not the I2C bus is busy.

Parameters

base	is the base address of the I2C instance used.

This function returns an indication of whether or not the I2C bus is busy. This function can be used in a multi-master environment to determine if the bus is free for another data transfer.

Returns

Returns **true** if the I2C bus is busy; otherwise, returns **false**.

24.2.3.12 static uint16_t I2C_getStatus (uint32_t *base*) [inline], [static]

Gets the current I2C module status.

Parameters

base	is the base address of the I2C instance used.

This function returns the status for the I2C module.

Returns

The current module status, enumerated as a bit field of

- **I2C_STS_ARB_LOST** - Arbitration-lost
- **I2C_STS_NO_ACK** - No-acknowledgment (NACK)
- **I2C_STS_REG_ACCESS_RDY** - Register-access-ready (ARDY)
- **I2C_STS_RX_DATA_RDY** - Receive-data-ready
- **I2C_STS_TX_DATA_RDY** - Transmit-data-ready
- **I2C_STS_STOP_CONDITION** - Stop condition detected
- **I2C_STS_BYTE_SENT** - Byte transmit complete
- **I2C_STS_ADDR_ZERO** - Address of all zeros detected
- **I2C_STS_ADDR_SLAVE** - Addressed as slave
- **I2C_STS_TX_EMPTY** - Transmit shift register empty
- **I2C_STS_RX_FULL** - Receive shift register full
- **I2C_STS_BUS_BUSY** - Bus busy, wait for STOP or reset
- **I2C_STS_NACK_SENT** - NACK was sent
- **I2C_STS_SLAVE_DIR-** Addressed as slave transmitter

24.2.3.13 static void I2C_clearStatus (uint32_t *base*, uint16_t *stsFlags*) `[inline]`, `[static]`

Clears I2C status flags.

Parameters

base	is the base address of the I2C instance used.
stsFlags	is a bit mask of the status flags to be cleared.

This function clears the specified I2C status flags. The *stsFlags* parameter is the logical OR of the following values:

- **I2C_STS_ARB_LOST**
- **I2C_STS_NO_ACK**,
- **I2C_STS_REG_ACCESS_RDY**
- **I2C_STS_RX_DATA_RDY**
- **I2C_STS_STOP_CONDITION**
- **I2C_STS_BYTE_SENT**
- **I2C_STS_NACK_SENT**
- **I2C_STS_SLAVE_DIR**

Note

Note that some of the status flags returned by I2C_getStatus() cannot be cleared by this function. Some may only be cleared by hardware or a reset of the I2C module.

Returns

None.

24.2.3.14 static void I2C_setConfig (uint32_t *base,* uint16_t *config*) [inline], [static]

Controls the state of the I2C module.

Parameters

base	is the base address of the I2C instance used.
config	is the command to be issued to the I2C module.

This function is used to control the state of the master and slave send and receive operations. The *config* is a logical OR of the following options.

One of the following four options:

- **I2C_MASTER_SEND_MODE** - Master-transmitter mode
- **I2C_MASTER_RECEIVE_MODE** - Master-receiver mode
- **I2C_SLAVE_SEND_MODE** - Slave-transmitter mode
- **I2C_SLAVE_RECEIVE_MODE** - Slave-receiver mode

Any of the following:

- **I2C_REPEAT_MODE** - Sends data until stop bit is set, ignores data count
- **I2C_START_BYTE_MODE** - Use start byte mode
- **I2C_FREE_DATA_FORMAT** - Use free data format, transfers have no address

Returns
None.

24.2.3.15 static void I2C_setBitCount (uint32_t *base,* **I2C_BitCount** *size*) [inline], [static]

Sets the data byte bit count the I2C module.

Parameters

base	is the base address of the I2C instance used.
size	is the number of bits per data byte.

The *size* parameter is a value I2C_BITCOUNT_x where x is the number of bits per data byte. The default and maximum size is 8 bits.

Returns
None.

24.2.3.16 static void I2C_sendStartCondition (uint32_t *base*) [inline], [static]

Issues an I2C START condition.

Parameters

base	is the base address of the I2C instance used.

This function causes the I2C module to generate a start condition. This function is only valid when the I2C module specified by the **base** parameter is a master.

Returns
None.

24.2.3.17 static void I2C_sendStopCondition (uint32_t *base*) [inline], [static]

Issues an I2C STOP condition.

Parameters

base	is the base address of the I2C instance used.

This function causes the I2C module to generate a stop condition. This function is only valid when the I2C module specified by the **base** parameter is a master.

To check on the status of the STOP condition, I2C_getStopConditionStatus() can be used.

Returns

None.

24.2.3.18 static void I2C_sendNACK (uint32_t *base*) [inline], [static]

Issues a no-acknowledge (NACK) bit.

Parameters

base	is the base address of the I2C instance used.

This function causes the I2C module to generate a NACK bit. This is only applicable when the I2C module is acting as a receiver.

Returns

None.

24.2.3.19 static uint16_t I2C_getData (uint32_t *base*) [inline], [static]

Receives a byte that has been sent to the I2C.

Parameters

base	is the base address of the I2C instance used.

This function reads a byte of data from the I2C Data Receive Register.

Returns

Returns the byte received from by the I2C cast as an uint16_t.

24.2.3.20 static void I2C_putData (uint32_t *base,* uint16_t *data*) [inline], [static]

Transmits a byte from the I2C.

Parameters

base	is the base address of the I2C instance used.
data	is the data to be transmitted from the I2C Master.

This function places the supplied data into I2C Data Transmit Register.

Returns

None.

24.2.3.21 static bool I2C_getStopConditionStatus (uint32_t *base*) [inline], [static]

Get stop condition status.

Parameters

base	is the base address of the I2C instance used.

This function reads and returns the stop condition bit status.

Returns

Returns **true** if the STP bit has been set by the device to generate a stop condition when the internal data counter of the I2C module has reached 0. Returns **false** when the STP bit is zero. This bit is automatically cleared after the stop condition has been generated.

24.2.3.22 static void I2C_setDataCount (uint32_t *base,* uint16_t *count*) [inline], [static]

Set number of bytes to be to transfer or receive when repeat mode is off.

Parameters

base	is the base address of the I2C instance used.
count	is the value to be put in the I2C data count register.

This function sets the number of bytes to transfer or receive when repeat mode is off.

Returns

None.

24.2.3.23 static void I2C_setAddressMode (uint32_t *base,* **I2C_AddressMode** *mode*) [inline], [static]

Sets the addressing mode to either 7-bit or 10-bit.

Parameters

base	is the base address of the I2C instance used.
mode	is the address mode, 7-bit or 10-bit.

This function configures the I2C module for either a 7-bit address (default) or a 10-bit address. The *mode* parameter configures the address length to 10 bits when its value is **I2C_ADDR_MODE_10BITS** and 7 bits when **I2C_ADDR_MODE_7BITS**.

Returns

None.

24.2.3.24 static void I2C_setEmulationMode (uint32_t *base,* **I2C_EmulationMode** *mode*) [inline], [static]

Sets I2C emulation mode.

Parameters

base	is the base address of the I2C instance used.
mode	is the emulation mode.

This function sets the behavior of the I2C operation when an emulation suspend occurs. The *mode* parameter can be one of the following:

- **I2C_EMULATION_STOP_SCL_LOW** - If SCL is low when the breakpoint occurs, the I2C module stops immediately. If SCL is high, the I2C module waits until SCL becomes low and then stops.
- **I2C_EMULATION_FREE_RUN** - I2C operation continues regardless of a the suspend.

Returns
None.

24.2.3.25 static void I2C_enableLoopback (uint32_t *base*) [inline], [static]

Enables I2C loopback mode.

Parameters

base	is the base address of the I2C instance used.

This function enables loopback mode. This mode is only valid during master mode and is helpful during device testing as it causes data transmitted out of the data transmit register to be received in data receive register.

Returns
None.

24.2.3.26 static void I2C_disableLoopback (uint32_t *base*) [inline], [static]

Disables I2C loopback mode.

Parameters

base	is the base address of the I2C instance used.

This function disables loopback mode. Loopback mode is disabled by default after reset.

Returns
None.

24.2.3.27 static **I2C_InterruptSource** I2C_getInterruptSource (uint32_t *base*) [inline], [static]

Returns the current I2C interrupt source.

Parameters

base	is the base address of the I2C instance used.

This function returns the event that generated an I2C basic (non-FIFO) interrupt. The possible sources are the following:

- **I2C_INTSRC_NONE**
- **I2C_INTSRC_ARB_LOST**
- **I2C_INTSRC_NO_ACK**
- **I2C_INTSRC_REG_ACCESS_RDY**
- **I2C_INTSRC_RX_DATA_RDY**
- **I2C_INTSRC_TX_DATA_RDY**
- **I2C_INTSRC_STOP_CONDITION**
- **I2C_INTSRC_ADDR_SLAVE**

Calling this function will result in hardware automatically clearing the current interrupt code and if ready, loading the next pending enabled interrupt. It will also clear the corresponding interrupt flag if the source is **I2C_INTSRC_ARB_LOST**, **I2C_INTSRC_NO_ACK**, or **I2C_INTSRC_STOP_CONDITION**.

Note

Note that this function differs from I2C_getInterruptStatus() in that it returns a single interrupt source. I2C_getInterruptSource() will return the status of all interrupt flags possible, including the flags that aren't necessarily enabled to generate interrupts.

Returns

None.

24.2.3.28 void I2C_initMaster (uint32_t *base,* uint32_t *sysclkHz,* uint32_t *bitRate,* **I2C_DutyCycle** *dutyCycle*)

Initializes the I2C Master.

Parameters

base	is the base address of the I2C instance used.
sysclkHz	is the rate of the clock supplied to the I2C module (SYSCLK) in Hz.
bitRate	is the rate of the master clock signal, SCL.
dutyCycle	is duty cycle of the SCL signal.

This function initializes operation of the I2C Master by configuring the bus speed for the master. Note that the I2C module **must** be put into reset before calling this function. You can do this with the function I2C_disableModule().

A programmable prescaler in the I2C module divides down the input clock (rate specified by *sysclkHz*) to produce the module clock (calculated to be around 10 MHz in this function). That clock is then divided down further to configure the SCL signal to run at the rate specified by *bitRate*. The *dutyCycle* parameter determines the percentage of time high and time low on the clock signal. The valid values are **I2C_DUTYCYCLE_33** for 33% and **I2C_DUTYCYCLE_50** for 50%.

The peripheral clock is the system clock. This value is returned by SysCtl_getClock(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtl_getClock()).

Returns
None.

References I2C_DUTYCYCLE_50.

24.2.3.29 void I2C_enableInterrupt (uint32_t *base,* uint32_t *intFlags*)

Enables I2C interrupt sources.

Parameters

base	is the base address of the I2C instance used.
intFlags	is the bit mask of the interrupt sources to be enabled.

This function enables the indicated I2C Master interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The *intFlags* parameter is the logical OR of any of the following:

- **I2C_INT_ARB_LOST** - Arbitration-lost interrupt
- **I2C_INT_NO_ACK** - No-acknowledgment (NACK) interrupt
- **I2C_INT_REG_ACCESS_RDY** - Register-access-ready interrupt
- **I2C_INT_RX_DATA_RDY** - Receive-data-ready interrupt
- **I2C_INT_TX_DATA_RDY** - Transmit-data-ready interrupt
- **I2C_INT_STOP_CONDITION** - Stop condition detected
- **I2C_INT_ADDR_SLAVE** - Addressed as slave interrupt
- **I2C_INT_RXFF** - RX FIFO level interrupt
- **I2C_INT_TXFF** - TX FIFO level interrupt

Note
I2C_INT_RXFF and **I2C_INT_TXFF** are associated with the I2C FIFO interrupt vector. All others are associated with the I2C basic interrupt.

Returns
None.

24.2.3.30 void I2C_disableInterrupt (uint32_t *base,* uint32_t *intFlags*)

Disables I2C interrupt sources.

Parameters

base	is the base address of the I2C instance used.
intFlags	is the bit mask of the interrupt sources to be disabled.

This function disables the indicated I2C Slave interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The *intFlags* parameter has the same definition as the *intFlags* parameter to I2C_enableInterrupt().

Returns

None.

24.2.3.31 uint32_t I2C_getInterruptStatus (uint32_t *base*)

Gets the current I2C interrupt status.

Parameters

base	is the base address of the I2C instance used.

This function returns the interrupt status for the I2C module.

Returns

The current interrupt status, enumerated as a bit field of

- **I2C_INT_ARB_LOST**
- **I2C_INT_NO_ACK**
- **I2C_INT_REG_ACCESS_RDY**
- **I2C_INT_RX_DATA_RDY**
- **I2C_INT_TX_DATA_RDY**
- **I2C_INT_STOP_CONDITION**
- **I2C_INT_ADDR_SLAVE**
- **I2C_INT_RXFF**
- **I2C_INT_TXFF**

Note

This function will only return the status flags associated with interrupts. However, a flag may be set even if its corresponding interrupt is disabled.

24.2.3.32 void I2C_clearInterruptStatus (uint32_t *base,* uint32_t *intFlags*)

Clears I2C interrupt sources.

Parameters

base	is the base address of the I2C instance used.
intFlags	is a bit mask of the interrupt sources to be cleared.

The specified I2C interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *intFlags* parameter has the same definition as the *intFlags* parameter to I2C_enableInterrupt().

Note

I2C_INT_RXFF and **I2C_INT_TXFF** are associated with the I2C FIFO interrupt vector. All others are associated with the I2C basic interrupt.
Also note that some of the status flags returned by I2C_getInterruptStatus() cannot be cleared by this function. Some may only be cleared by hardware or a reset of the I2C module.

Returns

None.

25 Interrupt Module

Introduction 451
API Functions 451

25.1 Interrupt Introduction

The Interrupt API provides a set of functions for dealing with the Peripheral Interrupt Expansion (PIE) Controller as well as CPU-level interrupt configuration. Functions are provided to initialize interrupt-related registers, enable and disable interrupts, and register interrupt handlers.

Interrupt API functions rely on an interrupt number defined to specify which interrupt is being configured. These interrupt numbers are found in inc/hw_ints.h and are in the format **INT_X**. For example, **INT_EPWM2_TZ** would be used to specify the trip zone interrupt for EPWM2 wherever a function has an interruptNumber parameter.

25.2 API Functions

Functions

- static bool Interrupt_enableMaster (void)
- static bool Interrupt_disableMaster (void)
- static void Interrupt_register (uint32_t interruptNumber, void(∗handler)(void))
- static void Interrupt_unregister (uint32_t interruptNumber)
- static void Interrupt_enableInCPU (uint16_t cpuInterrupt)
- static void Interrupt_disableInCPU (uint16_t cpuInterrupt)
- static void Interrupt_clearACKGroup (uint16_t group)
- void Interrupt_initModule (void)
- void Interrupt_initVectorTable (void)
- void Interrupt_enable (uint32_t interruptNumber)
- void Interrupt_disable (uint32_t interruptNumber)

25.2.1 Detailed Description

The Interrupt_ API provides two functions to initialize the module, Interrupt_initModule() and Interrupt_initVectorTable(). The former puts the PIE registers and the interrupt-related registers in the CPU into a known state. It clears all flags, disables interrupts at all levels, and enables vector fetching from the PIE. The latter initializes the PIE Vector Table to a set of default handlers–Interrupt_nmiHandler() for non-maskable interrupts, Interrupt_illegalOperationHandler() for an ITRAP interrupt, and Interrupt_defaultHandler() for all others. These defaults are intended to help with debugging. They should be modified or replaced more appropriate ISRs by the user.

Each interrupt source can be individually enabled and disabled via Interrupt_enable() and Interrupt_disable(). These affect the interrupt both on the PIE and on the CPU's IER register. The processor interrupt can be enabled and disabled via Interrupt_enableMaster() and Interrupt_disableMaster(); this does not affect the individual interrupt enable states. Masking of the processor interrupt can be utilized as a simple critical section (only NMI will interrupt the processor while the processor interrupt is disabled), though this will have adverse effects on the interrupt response time.

When an interrupt occurs, in order for further interrupts on its PIE group to be received, Interrupt_clearACKGroup() must be called. This is typically done at the end of the ISR.

The code for this module is contained in `driverlib/interrupt.c`, with `driverlib/interrupt.h` containing the API declarations for use by applications.

25.2.2 Function Documentation

25.2.2.1 static bool Interrupt_enableMaster (void) `[inline]`, `[static]`

Allows the CPU to process interrupts.

This function clears the global interrupt mask bit (INTM) in the CPU, allowing the processor to respond to interrupts.

Returns

Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

Referenced by Interrupt_disable(), and Interrupt_enable().

25.2.2.2 static bool Interrupt_disableMaster (void) `[inline]`, `[static]`

Stops the CPU from processing interrupts.

This function sets the global interrupt mask bit (INTM) in the CPU, preventing the processor from receiving maskable interrupts.

Returns

Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

Referenced by Interrupt_disable(), Interrupt_enable(), and Interrupt_initModule().

25.2.2.3 static void Interrupt_register (uint32_t *interruptNumber,* void(∗)(void) *handler*) `[inline]`, `[static]`

Registers a function to be called when an interrupt occurs.

Parameters

interruptNumber	specifies the interrupt in question.
handler	is a pointer to the function to be called.

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. When the interrupt occurs, if it is enabled (via Interrupt_enable()), the handler function will be called in interrupt context. Since the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

The available *interruptNumber* values are supplied in `inc/hw_ints.h`.

Note
This function assumes that the PIE has been enabled. See Interrupt_initModule().

Returns
None.

25.2.2.4 static void Interrupt_unregister (uint32_t *interruptNumber*) `[inline]`, `[static]`

Unregisters the function to be called when an interrupt occurs.

Parameters

interruptNumber	specifies the interrupt in question.

This function is used to indicate that a default handler Interrupt_defaultHandler() should be called when the given interrupt is asserted to the processor. Call Interrupt_disable() to disable the interrupt before calling this function.

The available *interruptNumber* values are supplied in `inc/hw_ints.h`.

See Also
Interrupt_register() for important information about registering interrupt handlers.

Returns
None.

25.2.2.5 static void Interrupt_enableInCPU (uint16_t *cpuInterrupt*) `[inline]`, `[static]`

Enables CPU interrupt channels

Parameters

cpuInterrupt	specifies the CPU interrupts to be enabled.

This function enables the specified interrupts in the CPU. The *cpuInterrupt* parameter is a logical OR of the values **INTERRUPT_CPU_INTx** where x is the interrupt number between 1 and 14, **INTERRUPT_CPU_DLOGINT**, and **INTERRUPT_CPU_RTOSINT**.

Note
Note that interrupts 1-12 correspond to the PIE groups with those same numbers.

Returns
None.

25.2.2.6 static void Interrupt_disableInCPU (uint16_t *cpuInterrupt*) `[inline]`, `[static]`

Disables CPU interrupt channels

Parameters

cpuInterrupt	specifies the CPU interrupts to be disabled.

This function disables the specified interrupts in the CPU. The *cpuInterrupt* parameter is a logical OR of the values **INTERRUPT_CPU_INTx** where x is the interrupt number between 1 and 14, **INTERRUPT_CPU_DLOGINT**, and **INTERRUPT_CPU_RTOSINT**.

Note

Note that interrupts 1-12 correspond to the PIE groups with those same numbers.

Returns

None.

25.2.2.7 static void Interrupt_clearACKGroup (uint16_t *group*) [inline], [static]

Acknowledges PIE Interrupt Group

Parameters

group	specifies the interrupt group to be acknowledged.

The specified interrupt group is acknowledged and clears any interrupt flag within that respective group.

The *group* parameter must be a logical OR of the following: **INTERRUPT_ACK_GROUP1**, **INTERRUPT_ACK_GROUP2**, **INTERRUPT_ACK_GROUP3** **INTERRUPT_ACK_GROUP4**, **INTERRUPT_ACK_GROUP5**, **INTERRUPT_ACK_GROUP6** **INTERRUPT_ACK_GROUP7**, **INTERRUPT_ACK_GROUP8**, **INTERRUPT_ACK_GROUP9** **INTERRUPT_ACK_GROUP10**, **INTERRUPT_ACK_GROUP11**, **INTERRUPT_ACK_GROUP12**.

Returns

None.

25.2.2.8 void Interrupt_initModule (void)

Initializes the PIE control registers by setting them to a known state.

This function initializes the PIE control registers. After globally disabling interrupts and enabling the PIE, it clears all of the PIE interrupt enable bits and interrupt flags.

Returns

None.

References Interrupt_disableMaster().

25.2.2.9 void Interrupt_initVectorTable (void)

Initializes the PIE vector table by setting all vectors to a default handler function.

Returns

None.

25.2.2.10 void Interrupt_enable (uint32_t *interruptNumber*)

Enables an interrupt.

Parameters

interruptNumber	specifies the interrupt to be enabled.

The specified interrupt is enabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

The available *interruptNumber* values are supplied in `inc/hw_ints.h`.

Returns

None.

References Interrupt_disableMaster(), and Interrupt_enableMaster().

25.2.2.11 void Interrupt_disable (uint32_t *interruptNumber*)

Disables an interrupt.

Parameters

interruptNumber	specifies the interrupt to be disabled.

The specified interrupt is disabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

The available *interruptNumber* values are supplied in `inc/hw_ints.h`.

Returns

None.

References Interrupt_disableMaster(), and Interrupt_enableMaster().

26 LIN Module

Introduction 457
API Functions 457

26.1 LIN Introduction

The local interconnect network (LIN) API provides a set of functions for configuring and using the LIN serial network protocol. Functions provided allow configuration of the LIN module operating modes and options, ability to send and receive data, and setup interrupt event handling.

26.2 API Functions

Macros

- #define LIN_IO_DFT_KEY
- #define LIN_WAKEUP_KEY
- #define LIN_ID0
- #define LIN_ID1
- #define LIN_ID2
- #define LIN_ID3
- #define LIN_ID4
- #define LIN_ID5

Enumerations

- enum LIN_LoopbackType { LIN_LOOPBACK_DIGITAL, LIN_LOOPBACK_ANALOG }
- enum LIN_AnalogLoopback { LIN_ANALOG_LOOP_NONE, LIN_ANALOG_LOOP_TX, LIN_ANALOG_LOOP_RX }
- enum LIN_CommMode { LIN_COMM_LIN_USELENGTHVAL, LIN_COMM_LIN_ID4ID5LENCTL }
- enum LIN_SCICommMode { LIN_COMM_SCI_IDLELINE, LIN_COMM_SCI_ADDRBIT }
- enum LIN_LINMode { LIN_MODE_LIN_SLAVE, LIN_MODE_LIN_MASTER }
- enum LIN_InterruptLine { LIN_INTERRUPT_LINE0, LIN_INTERRUPT_LINE1 }
- enum LIN_MessageFilter { LIN_MSG_FILTER_IDBYTE, LIN_MSG_FILTER_IDSLAVE }
- enum LIN_ChecksumType { LIN_CHECKSUM_CLASSIC, LIN_CHECKSUM_ENHANCED }
- enum LIN_DebugMode { LIN_DEBUG_FROZEN, LIN_DEBUG_COMPLETE }
- enum LIN_PinSampleMask { LIN_PINMASK_NONE, LIN_PINMASK_CENTER, LIN_PINMASK_CENTER_SCLK, LIN_PINMASK_CENTER_2SCLK }
- enum LIN_SCIParityType { LIN_SCI_PAR_ODD, LIN_SCI_PAR_EVEN }
- enum LIN_SCIStopBits { LIN_SCI_STOP_ONE, LIN_SCI_STOP_TWO }

Functions

- static void LIN_setLINMode (uint32_t base, LIN_LINMode mode)
- static void LIN_setMaximumBaudRate (uint32_t base, uint32_t clock)

- static void LIN_setMessageFiltering (uint32_t base, LIN_MessageFilter type)
- static void LIN_enableParity (uint32_t base)
- static void LIN_disableParity (uint32_t base)
- static uint16_t LIN_generateParityID (uint16_t identifier)
- static void LIN_setIDByte (uint32_t base, uint16_t identifier)
- static void LIN_setIDSlaveTask (uint32_t base, uint16_t identifier)
- static void LIN_sendWakeupSignal (uint32_t base)
- static void LIN_enterSleep (uint32_t base)
- static void LIN_sendChecksum (uint32_t base)
- static void LIN_triggerChecksumCompare (uint32_t base)
- static bool LIN_isTxReady (uint32_t base)
- static void LIN_setFrameLength (uint32_t base, uint16_t length)
- static void LIN_setCommMode (uint32_t base, LIN_CommMode mode)
- static void LIN_setTxMask (uint32_t base, uint16_t mask)
- static void LIN_setRxMask (uint32_t base, uint16_t mask)
- static uint16_t LIN_getTxMask (uint32_t base)
- static uint16_t LIN_getRxMask (uint32_t base)
- static bool LIN_isRxReady (uint32_t base)
- static uint16_t LIN_getRxIdentifier (uint32_t base)
- static bool LIN_isTxMatch (uint32_t base)
- static bool LIN_isRxMatch (uint32_t base)
- static void LIN_enableInterrupt (uint32_t base, uint32_t intFlags)
- static void LIN_disableInterrupt (uint32_t base, uint32_t intFlags)
- static void LIN_clearInterruptStatus (uint32_t base, uint32_t intFlags)
- static void LIN_setInterruptLevel0 (uint32_t base, uint32_t intFlags)
- static void LIN_setInterruptLevel1 (uint32_t base, uint32_t intFlags)
- static void LIN_enableModuleErrors (uint32_t base, uint32_t errors)
- static void LIN_disableModuleErrors (uint32_t base, uint32_t errors)
- static void LIN_enableAutomaticBaudrate (uint32_t base)
- static void LIN_disableAutomaticBaudrate (uint32_t base)
- static void LIN_stopExtendedFrame (uint32_t base)
- static void LIN_setChecksumType (uint32_t base, LIN_ChecksumType type)
- static void LIN_setSyncFields (uint32_t base, uint16_t syncBreak, uint16_t delimiter)
- static void LIN_enableSCIMode (uint32_t base)
- static void LIN_disableSCIMode (uint32_t base)
- static void LIN_setSCICommMode (uint32_t base, LIN_SCICommMode mode)
- static void LIN_enableSCIParity (uint32_t base, LIN_SCIParityType parity)
- static void LIN_disableSCIParity (uint32_t base)
- static void LIN_setSCIStopBits (uint32_t base, LIN_SCIStopBits number)
- static void LIN_enableSCISleepMode (uint32_t base)
- static void LIN_disableSCISleepMode (uint32_t base)
- static void LIN_enterSCILowPower (uint32_t base)
- static void LIN_exitSCILowPower (uint32_t base)
- static void LIN_setSCICharLength (uint32_t base, uint16_t numBits)
- static void LIN_setSCIFrameLength (uint32_t base, uint16_t length)
- static bool LIN_isSCIDataAvailable (uint32_t base)
- static bool LIN_isSCISpaceAvailable (uint32_t base)
- static uint16_t LIN_readSCICharNonBlocking (uint32_t base, bool emulation)
- static uint16_t LIN_readSCICharBlocking (uint32_t base, bool emulation)
- static void LIN_writeSCICharNonBlocking (uint32_t base, uint16_t data)
- static void LIN_writeSCICharBlocking (uint32_t base, uint16_t data)
- static void LIN_enableSCIModuleErrors (uint32_t base, uint32_t errors)
- static void LIN_disableSCIModuleErrors (uint32_t base, uint32_t errors)
- static void LIN_enableSCIInterrupt (uint32_t base, uint32_t intFlags)
- static void LIN_disableSCIInterrupt (uint32_t base, uint32_t intFlags)
- static void LIN_clearSCIInterruptStatus (uint32_t base, uint32_t intFlags)
- static void LIN_setSCIInterruptLevel0 (uint32_t base, uint32_t intFlags)
- static void LIN_setSCIInterruptLevel1 (uint32_t base, uint32_t intFlags)

- static bool LIN_isSCIReceiverIdle (uint32_t base)
- static bool LIN_getSCITxFrameType (uint32_t base)
- static bool LIN_getSCIRxFrameType (uint32_t base)
- static bool LIN_isSCIBreakDetected (uint32_t base)
- static void LIN_enableModule (uint32_t base)
- static void LIN_disableModule (uint32_t base)
- static void LIN_setBaudRatePrescaler (uint32_t base, uint32_t prescaler, uint32_t divider)
- static void LIN_enableDataTransmitter (uint32_t base)
- static void LIN_disableDataTransmitter (uint32_t base)
- static void LIN_enableDataReceiver (uint32_t base)
- static void LIN_disableDataReceiver (uint32_t base)
- static void LIN_performSoftwareReset (uint32_t base)
- static void LIN_enterSoftwareReset (uint32_t base)
- static void LIN_exitSoftwareReset (uint32_t base)
- static bool LIN_isBusBusy (uint32_t base)
- static bool LIN_isTxBufferEmpty (uint32_t base)
- static void LIN_enableExtLoopback (uint32_t base, LIN_LoopbackType loopbackType, LIN_AnalogLoopback path)
- static void LIN_disableExtLoopback (uint32_t base)
- static void LIN_enableIntLoopback (uint32_t base)
- static void LIN_disableIntLoopback (uint32_t base)
- static uint32_t LIN_getInterruptStatus (uint32_t base)
- static uint32_t LIN_getInterruptLevel (uint32_t base)
- static uint16_t LIN_getInterruptLine0Offset (uint32_t base)
- static uint16_t LIN_getInterruptLine1Offset (uint32_t base)
- static void LIN_enableMultibufferMode (uint32_t base)
- static void LIN_disableMultibufferMode (uint32_t base)
- static void LIN_setTransmitDelay (uint32_t base, uint16_t delay)
- static void LIN_setPinSampleMask (uint32_t base, LIN_PinSampleMask mask)
- static void LIN_setDebugSuspendMode (uint32_t base, LIN_DebugMode mode)
- static void LIN_enableGlobalInterrupt (uint32_t base, LIN_InterruptLine line)
- static void LIN_disableGlobalInterrupt (uint32_t base, LIN_InterruptLine line)
- static void LIN_clearGlobalInterruptStatus (uint32_t base, LIN_InterruptLine line)
- static bool LIN_getGlobalInterruptStatus (uint32_t base, LIN_InterruptLine line)
- void LIN_initModule (uint32_t base)
- void LIN_sendData (uint32_t base, uint16_t ∗data)
- void LIN_getData (uint32_t base, uint16_t ∗const data)

26.2.1 Detailed Description

The following describes important details and recommendations when using the LIN API.

Once system control enables the LIN module, **LIN_initModule()** needs to be called with the desired LIN module base to initialize the LIN with a set of default values and settings. Such settings include putting LIN in "LIN mode" as master, setting up the frame and timing values, and preparing the module for external communication.

LIN has the ability to operate as a SCI module instead of LIN when in "SCI mode". Use the **LIN_enableSCIMode()** function to switch to "SCI mode". The API is divided into three sets of functions: LIN only, SCI only, and both. The SCI-only functions state in their descriptions "SCI mode only" and have "SCI" in their function names such as **LIN_setSCICommMode()**. They can only be used when operating in SCI mode. The LIN-only functions state in their descriptions "LIN mode only" and can only be used when operating in LIN mode. The functions that state in their descriptions "LIN and SCI mode" can be used regardless of which operating mode the module is in.

The code for this module is contained in `driverlib/lin.c`, with `driverlib/lin.h` containing the API declarations for use by applications.

26.2.2 Enumeration Type Documentation

26.2.2.1 enum **LIN_LoopbackType**

The following are defines for the *type* parameter of the LIN_enableExtLoopback() function.

Enumerator

LIN_LOOPBACK_DIGITAL Digital Loopback Mode.

LIN_LOOPBACK_ANALOG Analog Loopback Mode.

26.2.2.2 enum **LIN_AnalogLoopback**

The following are defines for the *path* parameter of the LIN_enableExtLoopback() function.

Enumerator

LIN_ANALOG_LOOP_NONE Default path for digital loopback mode.

LIN_ANALOG_LOOP_TX Analog loopback through transmit pin.

LIN_ANALOG_LOOP_RX Analog loopback through receive pin.

26.2.2.3 enum **LIN_CommMode**

The following are defines for the *mode* parameter of the LIN_setCommMode() function.

Enumerator

LIN_COMM_LIN_USELENGTHVAL Use the length indicated in the LENGTH field of the SCIFORMAT register.

LIN_COMM_LIN_ID4ID5LENCTL Use ID4 and ID5 to convey the length.

26.2.2.4 enum **LIN_SCICommMode**

The following are defines for the *mode* parameter of the LIN_setSCICommMode() function.

Enumerator

LIN_COMM_SCI_IDLELINE Idle-line mode is used.

LIN_COMM_SCI_ADDRBIT Address bit mode is used.

26.2.2.5 enum **LIN_LINMode**

The following are defines for the *mode* parameter of the LIN_setLINMode() function.

Enumerator

LIN_MODE_LIN_SLAVE The node is in slave mode.

LIN_MODE_LIN_MASTER The node is in master mode.

26.2.2.6 enum **LIN_InterruptLine**

The following are defines for the *line* parameter of the LIN_enableGlobalInterrupt(), LIN_disableGlobalInterrupt(), LIN_clearGlobalInterruptStatus(), and LIN_getGlobalInterruptStatus() functions.

Enumerator

LIN_INTERRUPT_LINE0 Interrupt line 0.

LIN_INTERRUPT_LINE1 Interrupt line 1.

26.2.2.7 enum **LIN_MessageFilter**

The following are defines for the *type* parameter of the LIN_setMessageFiltering() function.

Enumerator

LIN_MSG_FILTER_IDBYTE LIN Message ID Byte Filtering.

LIN_MSG_FILTER_IDSLAVE Slave Task ID Byte Filtering.

26.2.2.8 enum **LIN_ChecksumType**

The following are defines for the *type* parameter of the LIN_setChecksumType() function.

Enumerator

LIN_CHECKSUM_CLASSIC Checksum Classic.

LIN_CHECKSUM_ENHANCED Checksum Enhanced.

26.2.2.9 enum **LIN_DebugMode**

The following are defines for the *mode* parameter of the LIN_setDebugSuspendMode() function.

Enumerator

LIN_DEBUG_FROZEN Freeze module during debug.

LIN_DEBUG_COMPLETE Complete Tx/Rx before Freezing.

26.2.2.10 enum **LIN_PinSampleMask**

The following are defines for the *mask* parameter of the LIN_setPinSampleMask() function.

Enumerator

LIN_PINMASK_NONE No Pin Mask.

LIN_PINMASK_CENTER Invert Tx Pin value at T-bit center.

LIN_PINMASK_CENTER_SCLK Invert Tx Pin value at T-bit center + SCLK.

LIN_PINMASK_CENTER_2SCLK Invert Tx Pin value at T-bit center + 2 SCLK.

26.2.2.11 enum **LIN_SCIParityType**

The following are defines for the *parity* parameter of the LIN_enableSCIParity() function.

Enumerator

LIN_SCI_PAR_ODD Odd parity.

LIN_SCI_PAR_EVEN Even parity.

26.2.2.12 enum **LIN_SCIStopBits**

The following are defines for the *number* parameter of the LIN_setSCIStopBits() function.

Enumerator

LIN_SCI_STOP_ONE Use One Stop bit.

LIN_SCI_STOP_TWO Use Two Stop bits.

26.2.3 Function Documentation

26.2.3.1 static void LIN_setLINMode (uint32_t *base*, **LIN_LINMode** *mode*) `[inline]`, `[static]`

Sets the LIN mode

Parameters

base	is the LIN module base address
mode	is the desired mode (slave or master)

In LIN mode only, this function sets the mode of the LIN mode to either slave or master. The *mode* parameter should be passed a value of **LIN_MODE_LIN_SLAVE** or **LIN_MODE_LIN_MASTER** to configure the mode of the LIN module specified by *base*.

Returns

None.

Referenced by LIN_initModule().

26.2.3.2 static void LIN_setMaximumBaudRate (uint32_t *base*, uint32_t *clock*) `[inline]`, `[static]`

Set Maximum Baud Rate Prescaler

Parameters

base	is the LIN module base address
clock	is the device system clock (Hz)

In LIN mode only, this function is used to set the maximum baud rate prescaler used during synchronization phase of a slave module if the ADAPT bit is set. The maximum baud rate prescaler is used by the wakeup and idle timer counters for a constant 4 second expiration time relative to a 20kHz rate.

Note

Use LIN_enableAutomaticBaudrate() to set the ADAPT bit and enable automatic bit rate mod detection.

Returns

None.

Referenced by LIN_initModule().

26.2.3.3 static void LIN_setMessageFiltering (uint32_t *base*, **LIN_MessageFilter** *type*) [inline], [static]

Set Message filtering Type

Parameters

base	is the LIN module base address
type	is the mask filtering comparison type

In LIN mode only, this function sets the message filtering type. The *type* parameter can be one of the following values:

- **LIN_MSG_FILTER_IDBYTE** - Filtering uses LIN message ID Byte
- **LIN_MSG_FILTER_IDSLAVE** - Filtering uses the Slave Task ID Byte

Returns

None.

References LIN_MSG_FILTER_IDBYTE.

Referenced by LIN_initModule().

26.2.3.4 static void LIN_enableParity (uint32_t *base*) [inline], [static]

Enable Parity mode.

Parameters

base	is the LIN module base address

In LIN mode only, this function enables the parity check.

Returns

None.

Referenced by LIN_initModule().

26.2.3.5 static void LIN_disableParity (uint32_t *base*) [inline], [static]

Disable Parity mode.

Parameters

base	is the LIN module base address

In LIN mode only, this function disables the parity check.

Returns

None.

26.2.3.6 static uint16_t LIN_generateParityID (uint16_t *identifier*) [inline], [static]

Generate Parity Identifier

Parameters

identifier	is the LIN header ID byte

In LIN mode only, this function generates the identifier parity bits and appends them to the identifier.

Note

An ID must be generated with parity before header generation in LIN master mode when parity is enabled using the function LIN_enableParity().

Returns

Returns the identifier appended with parity bits.

References LIN_ID0, LIN_ID1, LIN_ID2, LIN_ID3, LIN_ID4, and LIN_ID5.

26.2.3.7 static void LIN_setIDByte (uint32_t *base,* uint16_t *identifier*) [inline], [static]

Set ID Byte

Parameters

base	is the LIN module base address
identifier	is the LIN header ID byte

In LIN mode only, this function sets the message ID byte. In master mode, writing to this ID initiates a header transmission. In slave task, this ID is used for message filtering when HGENCTRL is 0.

Returns

None.

26.2.3.8 static void LIN_setIDSlaveTask (uint32_t *base,* uint16_t *identifier*) [inline], [static]

Set ID-SlaveTask

Parameters

base	is the LIN module base address
identifier	is the Received ID comparison ID

In LIN mode only, this function sets the identifier to which the received ID of an incoming Header will be compared in order to decide whether a RX response, a TX response, or no action is required.

Returns
None.

26.2.3.9 static void LIN_sendWakeupSignal (uint32_t *base*) [inline], [static]

Send LIN wakeup signal

Parameters

base	is the LIN module base address

In LIN mode only, this function sends the LIN wakeup signal to terminate the sleep mode of any LIN node connected to the bus.

Returns
None.

References LIN_WAKEUP_KEY.

26.2.3.10 static void LIN_enterSleep (uint32_t *base*) [inline], [static]

Enter LIN Sleep Mode.

Parameters

base	is the LIN module base address

In LIN mode only, this function puts the LIN module into a low-power, sleep mode. This can also be called to forcefully enter sleep when there is no activity on the bus.

Note
If this function is called while the receiver is actively receiving data and the wakeup interrupt is disabled, then the module will delay sleep mode from being entered until completion of reception.

Returns
None.

26.2.3.11 static void LIN_sendChecksum (uint32_t *base*) [inline], [static]

Send Checksum Byte

Parameters

base	is the LIN module base address

In LIN mode only, this function enables the transmitter with extended frames to send a checkbyte.

Returns
None.

26.2.3.12 static void LIN_triggerChecksumCompare (uint32_t *base*) [inline], [static]

Trigger Checksum Compare

Parameters

base	is the LIN module base address

In LIN mode only, this function enables the receiver for extended frames to trigger a checksum compare.

Returns
None.

Referenced by LIN_initModule().

26.2.3.13 static bool LIN_isTxReady (uint32_t *base*) [inline], [static]

Check Tx buffer ready flag

Parameters

base	is the LIN module base address

In LIN mode only, this function checks to see if the Tx ready flag is set indicating that the Tx buffer(s) is/are ready to get another character.

Returns
Returns **true** if the TX ready flag is set, else returns **false**

26.2.3.14 static void LIN_setFrameLength (uint32_t *base*, uint16_t *length*) [inline], [static]

Set LIN Frame Length

Parameters

base	is the LIN module base address
length	is the number of bytes.

In LIN mode only, this function sets the number of bytes in the response field.

The *length* parameter must be in a range between 1 and 8.

Returns
None.

Referenced by LIN_initModule().

26.2.3.15 static void LIN_setCommMode (uint32_t *base*, **LIN_CommMode** *mode*) [inline], [static]

Set LIN communication mode

Parameters

base	is the LIN module base address
mode	is the selected communication mode

In LIN mode only, this function is used to choose how the length of data is conveyed. This choice relates to the version of LIN being used. The *mode* parameter can have one of two values:

- **LIN_COMM_LIN_USELENGTHVAL** will use the length set with the LIN_setFrameLength() function.
- **LIN_COMM_LIN_ID4ID5LENCTL** will use ID4 and ID5 for length control.

Returns
None.

Referenced by LIN_initModule().

26.2.3.16 static void LIN_setTxMask (uint32_t *base*, uint16_t *mask*) [inline], [static]

Sets the transmit ID mask

Parameters

base	is the LIN module base address
mask	is the mask value to be set

In LIN mode only, this function sets the mask used for filtering an incoming ID message to determine if the TX ID flag should be set.

Returns
None.

Referenced by LIN_initModule().

26.2.3.17 static void LIN_setRxMask (uint32_t *base*, uint16_t *mask*) [inline], [static]

Sets the receive ID mask

Parameters

base	is the LIN module base address
mask	is the mask value to be set

In LIN mode only, this function sets the mask used for filtering an incoming ID message to determine if the ID RX flag should be set.

Returns

None.

Referenced by LIN_initModule().

26.2.3.18 static uint16_t LIN_getTxMask (uint32_t *base*) [inline], [static]

Gets the transmit ID mask

Parameters

base	is the LIN module base address

In LIN mode only, this function gets the mask used for filtering an incoming ID message to determine if the TX ID flag should be set.

Returns

Returns the Transmit ID Mask.

26.2.3.19 static uint16_t LIN_getRxMask (uint32_t *base*) [inline], [static]

Gets the receive ID mask

Parameters

base	is the LIN module base address

In LIN mode only, this function gets the mask used for filtering an incoming ID message to determine if the ID RX flag should be set.

Returns

Returns the Receive ID Mask.

26.2.3.20 static bool LIN_isRxReady (uint32_t *base*) [inline], [static]

Check if Rx data is ready

Parameters

base	is the LIN module base address

In LIN mode only, checks to see if the Rx ready bit is set indicating that a valid message frame has been received.

Returns

Returns **true** if the Rx ready flag is set, else returns **false**.

26.2.3.21 static uint16_t LIN_getRxIdentifier (uint32_t *base*) `[inline]`, `[static]`

Get last received identifier

Parameters

base	is the LIN module base address

In LIN mode only, this function gets the last received identifier.

Returns

Returns the Received Identifier.

26.2.3.22 static bool LIN_isTxMatch (uint32_t *base*) [inline], [static]

Checks for Tx ID Match Received

Parameters

base	is the LIN module base address

In LIN mode only, this function checks if an ID is received with a TX match and no ID-parity error.

Returns

Returns **true** if a valid ID is matched, else returns **false**.

26.2.3.23 static bool LIN_isRxMatch (uint32_t *base*) [inline], [static]

Checks for Rx ID Match Received

Parameters

base	is the LIN module base address

In LIN mode only, this function checks if an ID is received with a RX match and no ID-parity error.

Returns

Returns **true** if a valid ID is matched, else returns **false**.

26.2.3.24 static void LIN_enableInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Enable interrupts

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be enabled.

In LIN mode only, this function enables the interrupts for the specified interrupt sources.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_INT_WAKEUP** - Wakeup

- **LIN_INT_TO** - Time out
- **LIN_INT_TOAWUS** - Time out after wakeup signal
- **LIN_INT_TOA3WUS** - Time out after 3 wakeup signals
- **LIN_INT_TX** - Transmit buffer ready
- **LIN_INT_RX** - Receive buffer ready
- **LIN_INT_ID** - Received matching identifier
- **LIN_INT_PE** - Parity error
- **LIN_INT_OE** - Overrun error
- **LIN_INT_FE** - Framing error
- **LIN_INT_NRE** - No response error
- **LIN_INT_ISFE** - Inconsistent sync field error
- **LIN_INT_CE** - Checksum error
- **LIN_INT_PBE** - Physical bus error
- **LIN_INT_BE** - Bit error

Returns
None.

26.2.3.25 static void LIN_disableInterrupt (uint32_t *base*, uint32_t *intFlags*) [inline], [static]

Disable interrupts

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be disabled.

In LIN mode only, this function disables the interrupts for the specified interrupt sources.

The *intFlags* parameter can be set to the following value to disable all the flag bits:

- **LIN_INT_ALL** - All Interrupts

To disable individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_INT_WAKEUP** - Wakeup
- **LIN_INT_TO** - Time out
- **LIN_INT_TOAWUS** - Time out after wakeup signal
- **LIN_INT_TOA3WUS** - Time out after 3 wakeup signals
- **LIN_INT_TX** - Transmit buffer ready
- **LIN_INT_RX** - Receive buffer ready
- **LIN_INT_ID** - Received matching identifier
- **LIN_INT_PE** - Parity error
- **LIN_INT_OE** - Overrun error
- **LIN_INT_FE** - Framing error

- **LIN_INT_NRE** - No response error
- **LIN_INT_ISFE** - Inconsistent sync field error
- **LIN_INT_CE** - Checksum error
- **LIN_INT_PBE** - Physical bus error
- **LIN_INT_BE** - Bit error

Returns

None.

Referenced by LIN_initModule().

26.2.3.26 static void LIN_clearInterruptStatus (uint32_t *base*, uint32_t *intFlags*) [inline], [static]

Clear interrupt status

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be cleared.

In LIN mode only, this function clears the specified status flags.

The *intFlags* parameter can be set to the following value to clear all the flag bits:

- **LIN_INT_ALL** - All Interrupts

To clear individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_INT_WAKEUP** - Wakeup
- **LIN_INT_TO** - Time out
- **LIN_INT_TOAWUS** - Time out after wakeup signal
- **LIN_INT_TOA3WUS** - Time out after 3 wakeup signals
- **LIN_INT_TX** - Transmit buffer ready
- **LIN_INT_RX** - Receive buffer ready
- **LIN_INT_ID** - Received matching identifier
- **LIN_INT_PE** - Parity error
- **LIN_INT_OE** - Overrun error
- **LIN_INT_FE** - Framing error
- **LIN_INT_NRE** - No response error
- **LIN_INT_ISFE** - Inconsistent sync field error
- **LIN_INT_CE** - Checksum error
- **LIN_INT_PBE** - Physical bus error
- **LIN_INT_BE** - Bit error

Returns

None.

26.2.3.27 static void LIN_setInterruptLevel0 (uint32_t *base,* uint32_t *intFlags*) `[inline]`, `[static]`

Set interrupt level to 0

Parameters

base	is the LIN module base address
intFlags	is the bit mask of interrupt sources to be configured

In LIN mode only, this function sets the specified interrupt sources to level 0.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_INT_WAKEUP** - Wakeup
- **LIN_INT_TO** - Time out
- **LIN_INT_TOAWUS** - Time out after wakeup signal
- **LIN_INT_TOA3WUS** - Time out after 3 wakeup signals
- **LIN_INT_TX** - Transmit buffer ready
- **LIN_INT_RX** - Receive buffer ready
- **LIN_INT_ID** - Received matching identifier
- **LIN_INT_PE** - Parity error
- **LIN_INT_OE** - Overrun error
- **LIN_INT_FE** - Framing error
- **LIN_INT_NRE** - No response error
- **LIN_INT_ISFE** - Inconsistent sync field error
- **LIN_INT_CE** - Checksum error
- **LIN_INT_PBE** - Physical bus error
- **LIN_INT_BE** - Bit error

Returns
None.

26.2.3.28 static void LIN_setInterruptLevel1 (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Set interrupt level to 1

Parameters

base	is the LIN module base address
intFlags	is the bit mask of interrupt sources to be configured

In LIN mode only, this function sets the specified interrupt sources to level 1.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_INT_WAKEUP** - Wakeup

- **LIN_INT_TO** - Time out
- **LIN_INT_TOAWUS** - Time out after wakeup signal
- **LIN_INT_TOA3WUS** - Time out after 3 wakeup signals
- **LIN_INT_TX** - Transmit buffer ready
- **LIN_INT_RX** - Receive buffer ready
- **LIN_INT_ID** - Received matching identifier
- **LIN_INT_PE** - Parity error
- **LIN_INT_OE** - Overrun error
- **LIN_INT_FE** - Framing error
- **LIN_INT_NRE** - No response error
- **LIN_INT_ISFE** - Inconsistent sync field error
- **LIN_INT_CE** - Checksum error
- **LIN_INT_PBE** - Physical bus error
- **LIN_INT_BE** - Bit error

Returns

None.

26.2.3.29 static void LIN_enableModuleErrors (uint32_t *base,* uint32_t *errors*) [inline], [static]

Enable Module Errors for Testing

Parameters

base	is the LIN module base address
errors	is the specified errors to be enabled

In LIN mode only, this function enables the specified errors in the module for testing. The *errors* parameter can be a logical OR-ed result of the following values or **LIN_ALL_ERRORS** can be used to enable all of them:

- **LIN_BIT_ERROR** - Simulates a bit error
- **LIN_BUS_ERROR** - Simulates a physical bus error
- **LIN_CHECKSUM_ERROR** - Simulates a checksum error
- **LIN_ISF_ERROR** - Simulates an inconsistent synch field error

Note

To disable these errors, use the LIN_disableModuleErrors() function.

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.30 static void LIN_disableModuleErrors (uint32_t *base,* uint32_t *errors*) [inline], [static]

Disable Module Errors for Testing

Parameters

base	is the LIN module base address
errors	is the specified errors to be disabled

In LIN mode only, this function disables the specified errors in the module for testing. The *errors* parameter can be a logical OR-ed result of the following values or **LIN_ALL_ERRORS** can be used to disable all of them:

- **LIN_BIT_ERROR** - Simulates a bit error
- **LIN_BUS_ERROR** - Simulates a physical bus error
- **LIN_CHECKSUM_ERROR** - Simulates a checksum error
- **LIN_ISF_ERROR** - Simulates an inconsistent synch field error

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.31 static void LIN_enableAutomaticBaudrate (uint32_t *base*) [inline], [static]

Enable Automatic Baudrate Adjustment

Parameters

base	is the LIN module base address

In LIN mode only, this function enables the automatic baudrate adjustment mode during the detection of the Synch Field.

Note

The baudrate selection register will be updated automatically by a slave node if this mode is enabled.

Returns

None.

26.2.3.32 static void LIN_disableAutomaticBaudrate (uint32_t *base*) [inline], [static]

Disable Automatic Baudrate Adjustment

Parameters

base	is the LIN module base address

In LIN mode only, this function disables the automatic baudrate adjustment mode during the detection of the Synch Field. This results in a fixed baud rate.

Returns

None.

Referenced by LIN_initModule().

26.2.3.33 static void LIN_stopExtendedFrame (uint32_t *base*) [inline], [static]

Stops LIN Extended Frame Communication

Parameters

base	is the LIN module base address

In LIN mode only, this function stops the extended frame communication. Once stopped, the bit is automatically cleared.

Note

This function can only be called during extended frame communication.

Returns

None.

26.2.3.34 static void LIN_setChecksumType (uint32_t *base*, **LIN_ChecksumType** *type*) [inline], [static]

Set Checksum Type

Parameters

base	is the LIN module base address
type	is the checksum type

In LIN mode only, this function sets the checksum type. The *type* parameter can be one of the following two values:

- **LIN_CHECKSUM_CLASSIC** - Checksum Classic
- **LIN_CHECKSUM_ENHANCED** - Checksum Enhanced

Returns

None.

References LIN_CHECKSUM_ENHANCED.

Referenced by LIN_initModule().

26.2.3.35 static void LIN_setSyncFields (uint32_t *base*, uint16_t *syncBreak*, uint16_t *delimiter*) [inline], [static]

Set Sync Break Extend and Delimiter

Parameters

base	is the LIN module base address
syncBreak	is the sync break extend value
delimiter	is the sync delimiter value

In LIN mode only, this function sets the 3-bit sync break extend value and the 2-bit sync delimiter compare value.

The *break* parameter can be a value between 0 to 7. Details:

- **0** - Sync Break has no additional T-bit
- **1** - Sync Break has 1 additional T-bit

- ...
- **7** - Sync Break has 7 additional T-bits

The *delimiter* parameter can be a value between 1 to 4. Details:

- **1** - Delimiter has 1 T-bit
- **2** - Delimiter has 2 T-bits
- **3** - Delimiter has 3 T-bits
- **4** - Delimiter has 4 T-bits

Returns
None.

Referenced by LIN_initModule().

26.2.3.36 static void LIN_enableSCIMode (uint32_t *base*) [inline], [static]

Enable SCI Mode

Parameters

base	is the LIN module base address

This function enables the LIN peripheral to function as a SCI.

Returns
None.

26.2.3.37 static void LIN_disableSCIMode (uint32_t *base*) [inline], [static]

Disable SCI Mode

Parameters

base	is the LIN module base address

This function disables the SCI mode of the LIN peripheral.

Returns
None.

Referenced by LIN_initModule().

26.2.3.38 static void LIN_setSCICommMode (uint32_t *base*, **LIN_SCICommMode** *mode*) [inline], [static]

Set SCI communication mode

Parameters

base	is the LIN module base address
mode	is the selected communication mode

In SCI mode only, this function is used to select between idle-line mode and address-bit mode. The *mode* parameter can have one of the following values:

- **LIN_COMM_SCI_IDLELINE** - Idle-line mode.
- **LIN_COMM_SCI_ADDRBIT** - Address-bit mode.

Returns
None.

26.2.3.39 static void LIN_enableSCIParity (uint32_t *base*, **LIN_SCIParityType** *parity*) [inline], [static]

Enable SCI Parity mode.

Parameters

base	is the LIN module base address
parity	is the SCI parity type

In SCI mode only, this function enables the parity check and sets the parity type. The *parity* parameter can one of the following values:

- **LIN_SCI_PAR_ODD** - Sets Odd parity
- **LIN_SCI_PAR_EVEN** - Sets Even parity

Returns
None.

References LIN_SCI_PAR_ODD.

26.2.3.40 static void LIN_disableSCIParity (uint32_t *base*) [inline], [static]

Disable SCI Parity mode.

Parameters

base	is the LIN module base address

In SCI mode only, this function disables the parity check.

Returns
None.

26.2.3.41 static void LIN_setSCIStopBits (uint32_t *base*, **LIN_SCIStopBits** *number*) [inline], [static]

Set the number of stop bits for SCI

Parameters

base	is the LIN module base address
number	is the number of stop bits

In SCI mode only, this function sets the number of stop bits transmitted. The *number* parameter can be one of the following values:

- **LIN_SCI_STOP_ONE** - Set one stop bit
- **LIN_SCI_STOP_TWO** - Set two stop bits

Returns
None.

References LIN_SCI_STOP_ONE.

26.2.3.42 static void LIN_enableSCISleepMode (uint32_t *base*) [inline], [static]

Enable SCI Sleep mode.

Parameters

base	is the LIN module base address

In SCI mode only, this function enables the receive sleep mode functionality.

Note
The receiver still operates when the sleep mode is enabled, however, RXRDY is updated and SCIRD is loaded with new data only when an address frame is detected.

Returns
None.

26.2.3.43 static void LIN_disableSCISleepMode (uint32_t *base*) [inline], [static]

Disable SCI Sleep mode.

Parameters

base	is the LIN module base address

In SCI mode only, this function disables the receive sleep mode functionality.

Returns
None.

26.2.3.44 static void LIN_enterSCILowPower (uint32_t *base*) [inline], [static]

Enter SCI Local Low-Power Mode

Parameters

base	is the LIN module base address

In SCI mode only, this function enters the SCI local low-power mode.

Note

If this function is called while the receiver is actively receiving data and the wakeup interrupt is disabled, then the module will delay sleep mode from being entered until completion of reception.

Returns

None.

26.2.3.45 static void LIN_exitSCILowPower (uint32_t *base*) [inline], [static]

Exit SCI Local Low-Power Mode

Parameters

base	is the LIN module base address

In SCI mode only, this function exits the SCI local low-power mode.

Returns

None.

26.2.3.46 static void LIN_setSCICharLength (uint32_t *base,* uint16_t *numBits*) [inline], [static]

Set SCI character length

Parameters

base	is the LIN module base address
numBits	is the number of bits per character.

In SCI mode only, this function sets the number of bits per character.

The *numBits* parameter must be in a range between 1 and 8.

Returns

None.

26.2.3.47 static void LIN_setSCIFrameLength (uint32_t *base,* uint16_t *length*) [inline], [static]

Set SCI Frame Length

Parameters

base	is the LIN module base address
length	is the number of characters

In SCI mode only, this function sets the number of characters in the response field.

The *length* parameter must be in a range between 1 and 8.

Returns

None.

26.2.3.48 static bool LIN_isSCIDataAvailable (uint32_t *base*) [inline], [static]

Check if new SCI data is ready to be read

Parameters

base	is the LIN module base address

In SCI mode only, this function checks to see if the Rx ready bit is set indicating that a new data has been received.

Returns

Returns **true** if the Rx ready flag is set, else returns **false**.

Referenced by LIN_readSCICharBlocking().

26.2.3.49 static bool LIN_isSCISpaceAvailable (uint32_t *base*) [inline], [static]

Check if Space is available in SCI Transmit Buffer

Parameters

base	is the LIN module base address

In SCI mode only, this function checks to see if the Tx ready flag is set indicating that the Tx buffer(s) is/are ready to get another character.

Returns

Returns **true** if the TX ready flag is set, else returns **false**

Referenced by LIN_writeSCICharBlocking().

26.2.3.50 static uint16_t LIN_readSCICharNonBlocking (uint32_t *base,* bool *emulation*) [inline], [static]

Reads a SCI character without Blocking

Parameters

base	is the LIN module base address
emulation	sets whether the data is being read by an emulator or not

In SCI mode only, this function gets the byte of data received. The *emulation* parameter can have one of the following values:

- **true** - Emulator is being used, the RXRDY flag won't be cleared
- **false** - Emulator isn't being used, the RXRDY flag will be cleared automatically on read

Note

1. If the SCI receives data that is fewer than 8 bits in length, the data is left-justified and padded with trailing zeros.
2. To determine if new data is available to read, use the function LIN_isSCIDataAvailable().

Returns

Returns the received data.

26.2.3.51 static uint16_t LIN_readSCICharBlocking (uint32_t *base*, bool *emulation*) [inline], [static]

Reads a SCI character with Blocking

Parameters

base	is the LIN module base address
emulation	sets whether the data is being read by an emulator or not

In SCI mode only, this function gets the byte of data received. If new data isn't available, this function will wait until new data arrives. The *emulation* parameter can have one of the following values:

- **true** - Emulator is being used, the RXRDY flag won't be cleared
- **false** - Emulator isn't being used, the RXRDY flag will be cleared automatically on read

Note

If the SCI receives data that is fewer than 8 bits in length, the data is left-justified and padded with trailing zeros.

Returns

Returns the received data.

References LIN_isSCIDataAvailable().

26.2.3.52 static void LIN_writeSCICharNonBlocking (uint32_t *base*, uint16_t *data*) [inline], [static]

Sends a SCI character without blocking

Parameters

base	is the LIN module base address
data	is the byte of data to be transmitted

In SCI mode only, this function sets the byte of data to be transmitted without blocking.

Note

The transmit ready flag gets set when this buffer is ready to be loaded with another byte of data. Use LIN_isSCISpaceAvailable() to determine if space is available to write another character.

Returns

None.

26.2.3.53 static void LIN_writeSCICharBlocking (uint32_t *base,* uint16_t *data*) [inline], [static]

Sends a SCI character with blocking

Parameters

base	is the LIN module base address
data	is the byte of data to be transmitted

In SCI mode only, this function sets the byte of data to be transmitted with blocking functionality. If the buffer isn't ready to get new data written to, this function will wait until space is available.

Returns

None.

References LIN_isSCISpaceAvailable().

26.2.3.54 static void LIN_enableSCIModuleErrors (uint32_t *base,* uint32_t *errors*) [inline], [static]

Enable SCI Module Errors for Testing

Parameters

base	is the LIN module base address
errors	is the specified errors to be enabled

In SCI mode only, this function enables the specified errors in the module for testing. The *errors* parameter can be a logical OR-ed result of the following values or **LIN_SCI_ALL_ERRORS** can be used to enable all of them:

- **LIN_SCI_FRAME_ERROR** - Simulates a frame error
- **LIN_SCI_PARITY_ERROR** - Simulates a parity error
- **LIN_SCI_BREAK_ERROR** - Simulates a break detect error

Note

To disable these errors, use the LIN_disableSCIModuleErrors() function.

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.55 static void LIN_disableSCIModuleErrors (uint32_t *base,* uint32_t *errors*) [inline], [static]

Disable SCI Module Errors for Testing

Parameters

base	is the LIN module base address
errors	is the specified errors to be disabled

In SCI mode only, this function disables the specified errors in the module for testing. The *errors* parameter can be a logical OR-ed result of the following values or **LIN_SCI_ALL_ERRORS** can be used to enable all of them:

- **LIN_SCI_FRAME_ERROR** - Simulates a frame error
- **LIN_SCI_PARITY_ERROR** - Simulates a parity error
- **LIN_SCI_BREAK_ERROR** - Simulates a break detect error

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.56 static void LIN_enableSCIInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Enable SCI interrupts

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be enabled.

In SCI mode only, this function enables the interrupts for the specified interrupt sources.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_SCI_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_SCI_INT_BREAK** - Break Detect
- **LIN_SCI_INT_WAKEUP** - Wakeup
- **LIN_SCI_INT_TX** - Transmit Buffer
- **LIN_SCI_INT_RX** - Receive Buffer
- **LIN_SCI_INT_PARITY** - Parity Error
- **LIN_SCI_INT_OVERRUN** - Overrun Error
- **LIN_SCI_INT_FRAME** - Framing Error

Returns
None.

26.2.3.57 static void LIN_disableSCIInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Disable SCI interrupts

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be disabled.

In SCI mode only, this function disables the interrupts for the specified interrupt sources.

The *intFlags* parameter can be set to the following value to disable all the flag bits:

- **LIN_SCI_INT_ALL** - All Interrupts

To disable individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_SCI_INT_BREAK** - Break Detect
- **LIN_SCI_INT_WAKEUP** - Wakeup
- **LIN_SCI_INT_TX** - Transmit Buffer
- **LIN_SCI_INT_RX** - Receive Buffer
- **LIN_SCI_INT_PARITY** - Parity Error
- **LIN_SCI_INT_OVERRUN** - Overrun Error
- **LIN_SCI_INT_FRAME** - Framing Error

Returns
None.

26.2.3.58 static void LIN_clearSCIInterruptStatus (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Clear SCI interrupt status

Parameters

base	is the LIN module base address
intFlags	is the bit mask of the interrupt sources to be cleared.

In SCI mode only, this function clears the specified status flags.

The *intFlags* parameter can be set to the following value to clear all the flag bits:

- **LIN_SCI_INT_ALL** - All Interrupts

To clear individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_SCI_INT_BREAK** - Break Detect
- **LIN_SCI_INT_WAKEUP** - Wakeup

- **LIN_SCI_INT_TX** - Transmit Buffer
- **LIN_SCI_INT_RX** - Receive Buffer
- **LIN_SCI_INT_PARITY** - Parity Error
- **LIN_SCI_INT_OVERRUN** - Overrun Error
- **LIN_SCI_INT_FRAME** - Framing Error

Returns
None.

26.2.3.59 static void LIN_setSCIInterruptLevel0 (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Set SCI interrupt level to 0

Parameters

base	is the LIN module base address
intFlags	is the bit mask of interrupt sources to be configured

In SCI mode only, this function sets the specified interrupt sources to level 0.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_SCI_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_SCI_INT_BREAK** - Break Detect
- **LIN_SCI_INT_WAKEUP** - Wakeup
- **LIN_SCI_INT_TX** - Transmit Buffer
- **LIN_SCI_INT_RX** - Receive Buffer
- **LIN_SCI_INT_PARITY** - Parity Error
- **LIN_SCI_INT_OVERRUN** - Overrun Error
- **LIN_SCI_INT_FRAME** - Framing Error

Returns
None.

26.2.3.60 static void LIN_setSCIInterruptLevel1 (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Set SCI interrupt level to 1

Parameters

base	is the LIN module base address
intFlags	is the bit mask of interrupt sources to be configured

In SCI mode only, this function sets the specified interrupt sources to level 1.

The *intFlags* parameter can be set to the following value to set all the flag bits:

- **LIN_SCI_INT_ALL** - All Interrupts

To set individual flags, the *intFlags* parameter can be the logical OR of any of the following:

- **LIN_SCI_INT_BREAK** - Break Detect
- **LIN_SCI_INT_WAKEUP** - Wakeup
- **LIN_SCI_INT_TX** - Transmit Buffer
- **LIN_SCI_INT_RX** - Receive Buffer
- **LIN_SCI_INT_PARITY** - Parity Error
- **LIN_SCI_INT_OVERRUN** - Overrun Error
- **LIN_SCI_INT_FRAME** - Framing Error

Returns

None.

26.2.3.61 static bool LIN_isSCIReceiverIdle (uint32_t *base*) `[inline]`, `[static]`

Check if SCI Receiver is Idle

Parameters

base	is the LIN module base address

In SCI mode only, this function checks if the receiver is in an idle state.

Returns

Returns **true** if the state is idle, else returns **false**.

26.2.3.62 static bool LIN_getSCITxFrameType (uint32_t *base*) `[inline]`, `[static]`

Gets the SCI Transmit Frame Type

Parameters

base	is the LIN module base address

In SCI mode only, this function gets the transmit frame type which can be either data or an address.

Returns

Returns **true** if the frame will be an address, and returns **false** if the frame will be data.

26.2.3.63 static bool LIN_getSCIRxFrameType (uint32_t *base*) `[inline]`, `[static]`

Gets the SCI Receiver Frame Type

Parameters

base	is the LIN module base address

In SCI mode only, this function gets the receiver frame type which can be either an address or not an address.

Returns

Returns **true** if the frame is an address, and returns **false** if the frame isn't an address.

26.2.3.64 static bool LIN_isSCIBreakDetected (uint32_t *base*) [inline], [static]

Check if SCI Detected a Break Condition

Parameters

base	is the LIN module base address

In SCI mode only, this function checks if the module detected a break condition on the Rx pin.

Returns

Returns **true** if break detected, else returns **false**.

26.2.3.65 static void LIN_enableModule (uint32_t *base*) [inline], [static]

Enables the LIN module.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function sets the RESET bit of the SCIGCR0 register. Registers in this module are not writable until this has been done. Additionally, the transmit and receive pin control functionality is enabled.

Returns

None.

Referenced by LIN_initModule().

26.2.3.66 static void LIN_disableModule (uint32_t *base*) [inline], [static]

Disable the LIN module.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function clears the RESET bit of the SCIGCR0 register. Registers in this module are not writable when this bit is cleared. Additionally, the transmit and receive pin control functionality is disabled.

Returns

None.

Referenced by LIN_initModule().

26.2.3.67 static void LIN_setBaudRatePrescaler (uint32_t *base,* uint32_t *prescaler,* uint32_t *divider*) [inline], [static]

Set Baud Rate Prescaler

Parameters

base	is the LIN module base address
prescaler	is the 24-bit integer prescaler
divider	is the 4-bit fractional divider

In LIN and SCI mode, this function is used to set the baudrate based on the *prescaler* and *divider* values.

P = Prescaler

M = Fractional Divider

Bitrate = (SYSCLOCK) / ((P + 1 + M/16) $*$ 16)

Returns

None.

Referenced by LIN_initModule().

26.2.3.68 static void LIN_enableDataTransmitter (uint32_t *base*) [inline], [static]

Enable Transmit Data Transfer.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function enables the transfer of data from SCITD or TDy to the transmit shift register.

Returns

None.

Referenced by LIN_initModule().

26.2.3.69 static void LIN_disableDataTransmitter (uint32_t *base*) [inline], [static]

Disable Transmit Data Transfer.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function disables the transfer of data from SCITD or TDy to the transmit shift register.

Returns

None.

26.2.3.70 static void LIN_enableDataReceiver (uint32_t *base*) [inline], [static]

Enable Receive Data Transfer.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function enables the receiver to transfer data from the shift buffer register to the receive buffer or multi-buffer.

Returns

None.

Referenced by LIN_initModule().

26.2.3.71 static void LIN_disableDataReceiver (uint32_t *base*) [inline], [static]

Disable Receive Data Transfer.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function disables the receiver to transfer data from the shift buffer register to the receive buffer or multi-buffer.

Returns

None.

26.2.3.72 static void LIN_performSoftwareReset (uint32_t *base*) [inline], [static]

Perform software reset.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function will reset the LIN state machine and clear all pending flags. It is required to call this function after a wakeup signal has been sent.

To enter the reset state separately, use LIN_enterSoftwareReset(). To come out of reset, use LIN_exitSoftwareReset().

Returns

None.

26.2.3.73 static void LIN_enterSoftwareReset (uint32_t *base*) [inline], [static]

Put LIN into its reset state.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function will reset the LIN state machine and clear all pending flags. It is required to call this function after a wakeup signal has been sent. When in this state, changes to the configuration of this module may be made.

To take LIN out of the reset state and back into the ready state, use LIN_exitSoftwareReset().

Returns

None.

Referenced by LIN_initModule().

26.2.3.74 static void LIN_exitSoftwareReset (uint32_t *base*) [inline], [static]

Put LIN into its ready state.

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function will put LIN into its ready state. Transmission and reception can be done in this state. While in the ready state, configuration of the module should not be changed.

To put the module into its reset state, use LIN_enterSoftwareReset().

Returns

None.

Referenced by LIN_initModule().

26.2.3.75 static bool LIN_isBusBusy (uint32_t *base*) [inline], [static]

Check if Bus is Busy

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function checks if the receiver bus is busy receiving a frame.

Returns

Returns **true** if the bus is busy, else returns **false**.

26.2.3.76 static bool LIN_isTxBufferEmpty (uint32_t *base*) [inline], [static]

Check if the Transmit Buffer is Empty

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function checks if the transmit buffer is empty or not.

Returns

Returns **true** if the Tx buffer is empty, else returns **false**.

26.2.3.77 static void LIN_enableExtLoopback (uint32_t *base*, **LIN_LoopbackType** *loopbackType*, **LIN_AnalogLoopback** *path*) [inline], [static]

Enable External Loopback mode for self test

Parameters

base	is the LIN module base address
loopbackType	is the loopback type (analog or digital)
path	sets the transmit or receive pin to be included in the communication path (Analog loopback mode only)

In LIN and SCI mode, this function enables the external Loopback mode for self test. The *loopbackType* parameter can be one of the following values:

- **LIN_LOOPBACK_DIGITAL** - Digital Loopback
- **LIN_LOOPBACK_ANALOG** - Analog Loopback

The *path* parameter is only applicable in analog loopback mode and can be one of the following values:

- **LIN_ANALOG_LOOP_NONE** - Default option for digital loopback mode
- **LIN_ANALOG_LOOP_TX** - Enables analog loopback through the Tx pin
- **LIN_ANALOG_LOOP_RX** - Enables analog loopback through the Rx pin

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.78 static void LIN_disableExtLoopback (uint32_t *base*) [inline], [static]

Disable External Loopback mode for self test

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function disables the external Loopback mode.

Note

This function also resets the analog loopback communication path to the default transmit pin.

Returns

None.

Referenced by LIN_initModule().

26.2.3.79 static void LIN_enableIntLoopback (uint32_t *base*) [inline], [static]

Enable Internal Loopback mode for self test

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function enables the internal Loopback mode for self test.

Returns

None.

26.2.3.80 static void LIN_disableIntLoopback (uint32_t *base*) [inline], [static]

Disable Internal Loopback mode for self test

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function disables the internal Loopback mode for self test.

Returns

None.

Referenced by LIN_initModule().

26.2.3.81 static uint32_t LIN_getInterruptStatus (uint32_t *base*) [inline], [static]

Get Interrupt Flags Status

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function returns the interrupt status register.

The following flags can be used to mask the value returned:

- **LIN_FLAG_BREAK** - Break Detect Flag (SCI mode only)
- **LIN_FLAG_WAKEUP** - Wake-up Flag
- **LIN_FLAG_IDLE** - Receiver in Idle State (SCI mode only)
- **LIN_FLAG_BUSY** - Busy Flag
- **LIN_FLAG_TO** - Bus Idle Timeout Flag (LIN mode only)
- **LIN_FLAG_TOAWUS** - Timeout after Wakeup Signal (LIN mode only)
- **LIN_FLAG_TOA3WUS** - Timeout after 3 Wakeup Signals (LIN mode only)
- **LIN_FLAG_TXRDY** - Transmitter Buffer Ready Flag
- **LIN_FLAG_RXRDY** - Receiver Buffer Ready Flag
- **LIN_FLAG_TXWAKE** - Transmitter Wakeup Method Select (SCI mode only)
- **LIN_FLAG_TXEMPTY** - Transmitter Empty Flag
- **LIN_FLAG_RXWAKE** - Receiver Wakeup Detect Flag
- **LIN_FLAG_TXID** - Identifier on Transmit Flag (LIN mode only)
- **LIN_FLAG_RXID** - Identifier on Receive Flag (LIN mode only)
- **LIN_FLAG_PE** - Parity Error Flag
- **LIN_FLAG_OE** - Overrun Error Flag
- **LIN_FLAG_FE** - Framing Error Flag
- **LIN_FLAG_NRE** - No-Response Error Flag (LIN mode only)

- **LIN_FLAG_ISFE** - Inconsistent Synch Field Error Flag (LIN mode only)
- **LIN_FLAG_CE** - Checksum Error Flag (LIN mode only)
- **LIN_FLAG_PBE** - Physical Bus Error Flag (LIN mode only)
- **LIN_FLAG_BE** - Bit Error Flag (LIN mode only)

Returns
Returns the status flag register.

26.2.3.82 static uint32_t LIN_getInterruptLevel (uint32_t *base*) [inline], [static]

Get the Interrupt Level

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function gets the interrupt level status for all interrupt sources.

Returns
Returns the value of the interrupt level register.

26.2.3.83 static uint16_t LIN_getInterruptLine0Offset (uint32_t *base*) [inline], [static]

Gets the Interrupt Vector Offset for Line 0

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function gets the offset for interrupt line 0. A read to the specified line register updates its value to the next highest priority pending interrupt in the flag register and clears the flag corresponding to the offset that was read.

Note
The flags for the receive and the transmit interrupts cannot be cleared by reading the corresponding offset vector in this function.

The following are values that can be returned:

- **LIN_VECT_NONE** - No Interrupt
- **LIN_VECT_WAKEUP** - Wakeup
- **LIN_VECT_ISFE** - Inconsistent-sync-field Error
- **LIN_VECT_PE** - Parity Error
- **LIN_VECT_ID** - ID Interrupt
- **LIN_VECT_PBE** - Physical Bus Error
- **LIN_VECT_FE** - Frame Error
- **LIN_VECT_BREAK** - Break detect
- **LIN_VECT_CE** - Checksum Error

- **LIN_VECT_OE** - Overrun Error
- **LIN_VECT_BE** - Bit Error
- **LIN_VECT_RX** - Receive Interrupt
- **LIN_VECT_TX** - Transmit Interrupt
- **LIN_VECT_NRE** - No-response Error
- **LIN_VECT_TOAWUS** - Timeout after wakeup signal
- **LIN_VECT_TOA3WUS** - Timeout after 3 wakeup signals
- **LIN_VECT_TO** - Timeout (Bus Idle)

Returns

Returns the interrupt vector offset for interrupt line 0.

26.2.3.84 static uint16_t LIN_getInterruptLine1Offset (uint32_t *base*) [inline], [static]

Gets the Interrupt Vector Offset for Line 1

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function gets the offset for interrupt line 1. A read to the specified line register updates its value to the next highest priority pending interrupt in the flag register and clears the flag corresponding to the offset that was read.

Note

The flags for the receive and the transmit interrupts cannot be cleared by reading the corresponding offset vector in this function.

The following are values that can be returned:

- **LIN_VECT_NONE** - No Interrupt
- **LIN_VECT_WAKEUP** - Wakeup
- **LIN_VECT_ISFE** - Inconsistent-sync-field Error
- **LIN_VECT_PE** - Parity Error
- **LIN_VECT_ID** - ID Interrupt
- **LIN_VECT_PBE** - Physical Bus Error
- **LIN_VECT_FE** - Frame Error
- **LIN_VECT_BREAK** - Break detect
- **LIN_VECT_CE** - Checksum Error
- **LIN_VECT_OE** - Overrun Error
- **LIN_VECT_BE** - Bit Error
- **LIN_VECT_RX** - Receive Interrupt
- **LIN_VECT_TX** - Transmit Interrupt
- **LIN_VECT_NRE** - No-response Error
- **LIN_VECT_TOAWUS** - Timeout after wakeup signal
- **LIN_VECT_TOA3WUS** - Timeout after 3 wakeup signals
- **LIN_VECT_TO** - Timeout (Bus Idle)

Returns

Returns the interrupt vector offset for interrupt line 1.

26.2.3.85 static void LIN_enableMultibufferMode (uint32_t *base*) [inline], [static]

Enable Multi-buffer Mode

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function enables the multi-buffer mode.

Returns

None.

Referenced by LIN_initModule().

26.2.3.86 static void LIN_disableMultibufferMode (uint32_t *base*) [inline], [static]

Disable Multi-buffer Mode

Parameters

base	is the LIN module base address

In LIN and SCI mode, this function disables the multi-buffer mode.

Returns

None.

26.2.3.87 static void LIN_setTransmitDelay (uint32_t *base,* uint16_t *delay*) [inline], [static]

Set Transmit Pin Delay

Parameters

base	is the LIN module base address
delay	is number of clock delays for the Tx pin (0 to 7)

In LIN and SCI mode, this function sets the delay by which the value on the transmit pin is delayed so that the value on the receive pin is asynchronous.

Note

This is not applicable to the Start bit.

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.88 static void LIN_setPinSampleMask (uint32_t *base*, **LIN_PinSampleMask** *mask*) [inline], [static]

Set Pin Sample Mask

Parameters

base	is the LIN module base address
mask	is the pin sample mask to be set

In LIN and SCI mode, this function sets sample number at which the transmit pin value that is being transmitted will be inverted to verify the receive pin samples correctly with the majority detection circuitry. The *mask* parameter can be one of the following values:

- **LIN_PINMASK_NONE** - No mask
- **LIN_PINMASK_CENTER** - Invert Tx Pin value at T-bit center
- **LIN_PINMASK_CENTER_SCLK** - Invert Tx Pin value at T-bit center + SCLK
- **LIN_PINMASK_CENTER_2SCLK** - Invert Tx Pin value at T-bit center + 2 SCLK

Returns

None.

References LIN_IO_DFT_KEY.

26.2.3.89 static void LIN_setDebugSuspendMode (uint32_t *base,* **LIN_DebugMode** *mode*) [inline], [static]

Set the Debug Suspended Mode

Parameters

base	is the LIN module base address
mode	is the debug mode

In LIN and SCI mode, this function sets how the module operates when the program is suspended and being debugged with an emulator. The *mode* parameter can be one of the following values:

- **LIN_DEBUG_FROZEN** - The module state machine is frozen; transmissions and LIN counters are halted until debug mode is exited.
- **LIN_DEBUG_COMPLETE** - The module continues to operate until the current transmit and receive functions are complete.

Returns

None.

References LIN_DEBUG_FROZEN.

Referenced by LIN_initModule().

26.2.3.90 static void LIN_enableGlobalInterrupt (uint32_t *base,* **LIN_InterruptLine** *line*) [inline], [static]

Enables a LIN global interrupt.

Parameters

base	is the LIN module base address
line	is specified interrupt vector line

In LIN and SCI mode, this function globally enables an interrupt corresponding to a specified interrupt line. The *line* parameter can be one of the following enumerated values:

- **LIN_INTERRUPT_LINE0** - Interrupt Vector Line 0
- **LIN_INTERRUPT_LINE1** - Interrupt Vector Line 1

Returns
None.

26.2.3.91 static void LIN_disableGlobalInterrupt (uint32_t *base,* **LIN_InterruptLine** *line*) `[inline]`, `[static]`

Disables a LIN global interrupt.

Parameters

base	is the LIN module base address
line	is specified interrupt vector line

In LIN and SCI mode, this function globally disables an interrupt corresponding to a specified interrupt line. The *line* parameter can be one of the following enumerated values:

- **LIN_INTERRUPT_LINE0** - Interrupt Vector Line 0
- **LIN_INTERRUPT_LINE1** - Interrupt Vector Line 1

Returns
None.

26.2.3.92 static void LIN_clearGlobalInterruptStatus (uint32_t *base,* **LIN_InterruptLine** *line*) `[inline]`, `[static]`

Clears a LIN global interrupt flag.

Parameters

base	is the LIN module base address
line	is specified interrupt vector line

In LIN and SCI mode, this function clears the global interrupt flag that corresponds to a specified interrupt line. The *line* parameter can be one of the following enumerated values:

- **LIN_INTERRUPT_LINE0** - Interrupt Vector Line 0
- **LIN_INTERRUPT_LINE1** - Interrupt Vector Line 1

Returns
None.

26.2.3.93 static bool LIN_getGlobalInterruptStatus (uint32_t *base*, **LIN_InterruptLine** *line*) [inline], [static]

Returns a LIN global interrupt flag status.

Parameters

base	is the LIN module base address
line	is specified interrupt vector line

In LIN and SCI mode, this function returns the status of a global interrupt flag that corresponds to a specified interrupt line. The *line* parameter can be one of the following enumerated values:

- **LIN_INTERRUPT_LINE0** - Interrupt Vector Line 0
- **LIN_INTERRUPT_LINE1** - Interrupt Vector Line 1

Returns

Returns **true** if the interrupt flag is set. Return **false** if not.

26.2.3.94 void LIN_initModule (uint32_t *base*)

Initializes the LIN Driver

Parameters

base	is the LIN module base address

This function initializes the LIN module.

Returns

None.

References LIN_CHECKSUM_ENHANCED, LIN_COMM_LIN_USELENGTHVAL, LIN_DEBUG_COMPLETE, LIN_disableAutomaticBaudrate(), LIN_disableExtLoopback(), LIN_disableInterrupt(), LIN_disableIntLoopback(), LIN_disableModule(), LIN_disableSCIMode(), LIN_enableDataReceiver(), LIN_enableDataTransmitter(), LIN_enableModule(), LIN_enableMultibufferMode(), LIN_enableParity(), LIN_enterSoftwareReset(), LIN_exitSoftwareReset(), LIN_MODE_LIN_MASTER, LIN_MSG_FILTER_IDSLAVE, LIN_setBaudRatePrescaler(), LIN_setChecksumType(), LIN_setCommMode(), LIN_setDebugSuspendMode(), LIN_setFrameLength(), LIN_setLINMode(), LIN_setMaximumBaudRate(), LIN_setMessageFiltering(), LIN_setRxMask(), LIN_setSyncFields(), LIN_setTxMask(), and LIN_triggerChecksumCompare().

26.2.3.95 void LIN_sendData (uint32_t *base*, uint16_t ∗ *data*)

Send Data

Parameters

base	is the LIN module base address
data	is the pointer to data to send

In LIN mode only, this function sends a block of data pointed to by 'data'. The number of data to transmit must be set with LIN_setFrameLength() before.

Returns

None.

26.2.3.96 void LIN_getData (uint32_t *base,* uint16_t ∗const *data*)

Read received data

Parameters

base	is the LIN module base address
data	is the pointer to the data buffer

In LIN mode only, this function reads a block of bytes and place it into the data buffer pointed to by 'data'.

Returns
None.

27 MemCfg Module

Introduction . . . 506
API Functions . . . 506

27.1 MemCfg Introduction

The MemCfg module provides an API to configure the device's Memory Control Module. The functions that are provided fall into three main categories: RAM section configuration, access violation status and interrupts, and memory error status an interrupts. The RAM section configuration functions can initialize RAM, configure access protection settings, and configure section ownership. The access violation and memory error categories contain functions that can return violation and error status and address information as well as configure interrupts that can be generated as a result of these issues.

27.2 API Functions

Enumerations

- enum MemCfg_CLAMemoryType { MEMCFG_CLA_MEM_DATA, MEMCFG_CLA_MEM_PROGRAM }
- enum MemCfg_LSRAMMasterSel { MEMCFG_LSRAMMASTER_CPU_ONLY, MEMCFG_LSRAMMASTER_CPU_CLA1 }
- enum MemCfg_TestMode { MEMCFG_TEST_FUNCTIONAL, MEMCFG_TEST_WRITE_DATA, MEMCFG_TEST_WRITE_ECC, MEMCFG_TEST_WRITE_PARITY }

Functions

- static void MemCfg_setCLAMemType (uint32_t ramSections, MemCfg_CLAMemoryType claMemType)
- static void MemCfg_enableViolationInterrupt (uint32_t intFlags)
- static void MemCfg_disableViolationInterrupt (uint32_t intFlags)
- static uint32_t MemCfg_getViolationInterruptStatus (void)
- static void MemCfg_forceViolationInterrupt (uint32_t intFlags)
- static void MemCfg_clearViolationInterruptStatus (uint32_t intFlags)
- static void MemCfg_setCorrErrorThreshold (uint32_t threshold)
- static uint32_t MemCfg_getCorrErrorCount (void)
- static void MemCfg_enableCorrErrorInterrupt (uint32_t intFlags)
- static void MemCfg_disableCorrErrorInterrupt (uint32_t intFlags)
- static uint32_t MemCfg_getCorrErrorInterruptStatus (void)
- static void MemCfg_forceCorrErrorInterrupt (uint32_t intFlags)
- static void MemCfg_clearCorrErrorInterruptStatus (uint32_t intFlags)
- static uint32_t MemCfg_getCorrErrorStatus (void)
- static uint32_t MemCfg_getUncorrErrorStatus (void)
- static void MemCfg_forceCorrErrorStatus (uint32_t stsFlags)
- static void MemCfg_forceUncorrErrorStatus (uint32_t stsFlags)

- static void MemCfg_clearCorrErrorStatus (uint32_t stsFlags)
- static void MemCfg_clearUncorrErrorStatus (uint32_t stsFlags)
- void MemCfg_lockConfig (uint32_t ramSections)
- void MemCfg_unlockConfig (uint32_t ramSections)
- void MemCfg_commitConfig (uint32_t ramSections)
- void MemCfg_setProtection (uint32_t ramSection, uint32_t protectMode)
- void MemCfg_setLSRAMMasterSel (uint32_t ramSection, MemCfg_LSRAMMasterSel masterSel)
- void MemCfg_setTestMode (uint32_t ramSection, MemCfg_TestMode testMode)
- void MemCfg_initSections (uint32_t ramSections)
- bool MemCfg_getInitStatus (uint32_t ramSections)
- uint32_t MemCfg_getViolationAddress (uint32_t intFlag)
- uint32_t MemCfg_getCorrErrorAddress (uint32_t stsFlag)
- uint32_t MemCfg_getUncorrErrorAddress (uint32_t stsFlag)

27.2.1 Detailed Description

Many of the functions provided by this API to configure RAM sections' settings will take a RAM section identifier or an OR of several identifiers as a parameter. These are defines with names in the format **MEMCFG_SECT_X**. Take care to read the function description to learn which functions can operate on multiple sections of the same type at a time and which ones can only configure one section at a time. A quick way to check this is to see if the parameter says ramSection or the plural ramSections. Some functions may also be able to take a **MEMCFG_SECT_ALL** value to indicate that all RAM sections should be operated on at the same time. Again, read the function's detailed description to be sure.

The code for this module is contained in `driverlib/memcfg.c`, with `driverlib/memcfg.h` containing the API declarations for use by applications.

27.2.2 Enumeration Type Documentation

27.2.2.1 enum **MemCfg_CLAMemoryType**

Values that can be passed to MemCfg_setCLAMemType() as the *claMemType* parameter.

Enumerator

MEMCFG_CLA_MEM_DATA Section is CLA data memory.

MEMCFG_CLA_MEM_PROGRAM Section is CLA program memory.

27.2.2.2 enum **MemCfg_LSRAMMasterSel**

Values that can be passed to MemCfg_setLSRAMMasterSel() as the *masterSel* parameter.

Enumerator

MEMCFG_LSRAMMASTER_CPU_ONLY CPU is the master of the section.

MEMCFG_LSRAMMASTER_CPU_CLA1 CPU and CLA1 share this section.

27.2.2.3 enum **MemCfg_TestMode**

Values that can be passed to MemCfg_setTestMode() as the *testMode* parameter.

Enumerator

MEMCFG_TEST_FUNCTIONAL Functional mode.

MEMCFG_TEST_WRITE_DATA Writes allowed to data only.

MEMCFG_TEST_WRITE_ECC Writes allowed to ECC only (for DxRAM)

MEMCFG_TEST_WRITE_PARITY Writes allowed to parity only (for LSxRAM, GSxRAM, and MSGxRAM)

27.2.3 Function Documentation

27.2.3.1 static void MemCfg_setCLAMemType (uint32_t *ramSections,* **MemCfg_CLAMemoryType** *claMemType*) [inline], [static]

Sets the CLA memory type of the specified RAM section.

Parameters

ramSections	is the logical OR of the sections to be configured.
claMemType	indicates data memory or program memory.

This function sets the CLA memory type configuration of the RAM section. If the *claMemType* parameter is **MEMCFG_CLA_MEM_DATA**, the RAM section will be configured as CLA data memory. If **MEMCFG_CLA_MEM_PROGRAM**, the RAM section will be configured as CLA program memory.

The *ramSections* parameter is an OR of the following indicators: **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx**.

Note

This API only applies to LSx RAM and has no effect if the CLA isn't master of the memory section.

See Also

MemCfg_setLSRAMMasterSel()

Returns

None.

References MEMCFG_CLA_MEM_PROGRAM.

27.2.3.2 static void MemCfg_enableViolationInterrupt (uint32_t *intFlags*) [inline], [static]

Enables individual RAM access violation interrupt sources.

Parameters

intFlags	is a bit mask of the interrupt sources to be enabled. Can be a logical OR any of the following values: ■ **MEMCFG_NMVIOL_CPUREAD** - Non-master CPU read access ■ **MEMCFG_NMVIOL_CPUWRITE** - Non-master CPU write access ■ **MEMCFG_NMVIOL_CPUFETCH** - Non-master CPU fetch access ■ **MEMCFG_NMVIOL_DMAWRITE** - Non-master DMA write access ■ **MEMCFG_NMVIOL_CLA1READ** - Non-master CLA1 read access ■ **MEMCFG_NMVIOL_CLA1WRITE** - Non-master CLA1 write access ■ **MEMCFG_NMVIOL_CLA1FETCH** - Non-master CLA1 fetch access ■ **MEMCFG_MVIOL_CPUFETCH** - Master CPU fetch access ■ **MEMCFG_MVIOL_CPUWRITE** - Master CPU write access ■ **MEMCFG_MVIOL_DMAWRITE** - Master DMA write access

This function enables the indicated RAM access violation interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns
None.

27.2.3.3 static void MemCfg_disableViolationInterrupt (uint32_t *intFlags*) [inline], [static]

Disables individual RAM access violation interrupt sources.

Parameters

intFlags	is a bit mask of the interrupt sources to be disabled. Can be a logical OR any of the following values: ■ **MEMCFG_NMVIOL_CPUREAD** ■ **MEMCFG_NMVIOL_CPUWRITE** ■ **MEMCFG_NMVIOL_CPUFETCH** ■ **MEMCFG_NMVIOL_DMAWRITE** ■ **MEMCFG_NMVIOL_CLA1READ** ■ **MEMCFG_NMVIOL_CLA1WRITE** ■ **MEMCFG_NMVIOL_CLA1FETCH** ■ **MEMCFG_MVIOL_CPUFETCH** ■ **MEMCFG_MVIOL_CPUWRITE** ■ **MEMCFG_MVIOL_DMAWRITE**

This function disables the indicated RAM access violation interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note
Note that only non-master violations may generate interrupts.

Returns
None.

27.2.3.4 static uint32_t MemCfg_getViolationInterruptStatus (void) [inline], [static]

Gets the current RAM access violation status.

This function returns the RAM access violation status. This function will return flags for both master and non-master access violations although only the non-master flags have the ability to cause the generation of an interrupt.

Returns
Returns the current violation status, enumerated as a bit field of the values:
- **MEMCFG_NMVIOL_CPUREAD** - Non-master CPU read access
- **MEMCFG_NMVIOL_CPUWRITE** - Non-master CPU write access
- **MEMCFG_NMVIOL_CPUFETCH** - Non-master CPU fetch access
- **MEMCFG_NMVIOL_DMAWRITE** - Non-master DMA write access
- **MEMCFG_NMVIOL_CLA1READ** - Non-master CLA1 read access
- **MEMCFG_NMVIOL_CLA1WRITE** - Non-master CLA1 write access
- **MEMCFG_NMVIOL_CLA1FETCH** - Non-master CLA1 fetch access
- **MEMCFG_MVIOL_CPUFETCH** - Master CPU fetch access
- **MEMCFG_MVIOL_CPUWRITE** - Master CPU write access
- **MEMCFG_MVIOL_DMAWRITE** - Master DMA write access

27.2.3.5 static void MemCfg_forceViolationInterrupt (uint32_t *intFlags*) [inline], [static]

Sets the RAM access violation status.

Parameters

intFlags	is a bit mask of the access violation flags to be set. Can be a logical OR any of the following values: ■ **MEMCFG_NMVIOL_CPUREAD** ■ **MEMCFG_NMVIOL_CPUWRITE** ■ **MEMCFG_NMVIOL_CPUFETCH** ■ **MEMCFG_NMVIOL_DMAWRITE** ■ **MEMCFG_NMVIOL_CLA1READ** ■ **MEMCFG_NMVIOL_CLA1WRITE** ■ **MEMCFG_NMVIOL_CLA1FETCH** ■ **MEMCFG_MVIOL_CPUFETCH** ■ **MEMCFG_MVIOL_CPUWRITE** ■ **MEMCFG_MVIOL_DMAWRITE**

This function sets the RAM access violation status. This function will set flags for both master and non-master access violations, and an interrupt will be generated if it is enabled.

Returns

None.

27.2.3.6 static void MemCfg_clearViolationInterruptStatus (uint32_t *intFlags*) [inline], [static]

Clears RAM access violation flags.

Parameters

intFlags	is a bit mask of the access violation flags to be cleared. Can be a logical OR any of the following values: ■ **MEMCFG_NMVIOL_CPUREAD** ■ **MEMCFG_NMVIOL_CPUWRITE** ■ **MEMCFG_NMVIOL_CPUFETCH** ■ **MEMCFG_NMVIOL_DMAWRITE** ■ **MEMCFG_NMVIOL_CLA1READ** ■ **MEMCFG_NMVIOL_CLA1WRITE** ■ **MEMCFG_NMVIOL_CLA1FETCH** ■ **MEMCFG_MVIOL_CPUFETCH** ■ **MEMCFG_MVIOL_CPUWRITE** ■ **MEMCFG_MVIOL_DMAWRITE**

Returns

None.

27.2.3.7 static void MemCfg_setCorrErrorThreshold (uint32_t *threshold*) `[inline]`, `[static]`

Sets the correctable error threshold value.

Parameters

threshold	is the correctable error threshold.

This value sets the error-count threshold at which a correctable error interrupt is generated. That is when the error count register reaches the value specified by the *threshold* parameter, an interrupt is generated if it is enabled.

Returns

None.

27.2.3.8 static uint32_t MemCfg_getCorrErrorCount (void) [inline], [static]

Gets the correctable error count.

Returns

Returns the number of correctable error have occurred.

27.2.3.9 static void MemCfg_enableCorrErrorInterrupt (uint32_t *intFlags*) [inline], [static]

Enables individual RAM correctable error interrupt sources.

Parameters

intFlags	is a bit mask of the interrupt sources to be enabled. Can take the value **MEMCFG_CERR_CPUREAD** only. Other values are reserved.

This function enables the indicated RAM correctable error interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note

Note that only correctable errors may generate interrupts.

Returns

None.

27.2.3.10 static void MemCfg_disableCorrErrorInterrupt (uint32_t *intFlags*) [inline], [static]

Disables individual RAM correctable error interrupt sources.

Parameters

intFlags	is a bit mask of the interrupt sources to be disabled. Can take the value **MEMCFG_CERR_CPUREAD** only. Other values are reserved.

This function disables the indicated RAM correctable error interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note
Note that only correctable errors may generate interrupts.

Returns
None.

27.2.3.11 static uint32_t MemCfg_getCorrErrorInterruptStatus (void) [inline], [static]

Gets the current RAM correctable error interrupt status.

Returns
Returns the current error interrupt status. Will return a value of **MEMCFG_CERR_CPUREAD** if an interrupt has been generated. If not, the function will return 0.

27.2.3.12 static void MemCfg_forceCorrErrorInterrupt (uint32_t *intFlags*) [inline], [static]

Sets the RAM correctable error interrupt status.

Parameters

intFlags	is a bit mask of the interrupt sources to be set. Can take the value **MEMCFG_CERR_CPUREAD** only. Other values are reserved.

This function sets the correctable error interrupt flag.

Note
Note that only correctable errors may generate interrupts.

Returns
None.

27.2.3.13 static void MemCfg_clearCorrErrorInterruptStatus (uint32_t *intFlags*) [inline], [static]

Clears the RAM correctable error interrupt status.

Parameters

intFlags	is a bit mask of the interrupt sources to be cleared. Can take the value **MEMCFG_CERR_CPUREAD** only. Other values are reserved.

This function clears the correctable error interrupt flag.

Note
Note that only correctable errors may generate interrupts.

Returns
None.

27.2.3.14 static uint32_t MemCfg_getCorrErrorStatus (void) [inline], [static]

Gets the current correctable RAM error status.

Returns
Returns the current error status, enumerated as a bit field of **MEMCFG_CERR_CPUREAD**, **MEMCFG_CERR_DMAREAD**, or **MEMCFG_CERR_CLA1READ**

27.2.3.15 static uint32_t MemCfg_getUncorrErrorStatus (void) [inline], [static]

Gets the current uncorrectable RAM error status.

Returns
Returns the current error status, enumerated as a bit field of **MEMCFG_UCERR_CPUREAD**, **MEMCFG_UCERR_DMAREAD**, or **MEMCFG_UCERR_CLA1READ**.

27.2.3.16 static void MemCfg_forceCorrErrorStatus (uint32_t *stsFlags*) [inline], [static]

Sets the specified correctable RAM error status flag.

Parameters

stsFlags	is a bit mask of the error sources. This parameter can be any of the following values: **MEMCFG_CERR_CPUREAD**, **MEMCFG_CERR_DMAREAD**, or **MEMCFG_CERR_CLA1READ**.

This function sets the specified correctable RAM error status flag.

Returns
None.

27.2.3.17 static void MemCfg_forceUncorrErrorStatus (uint32_t *stsFlags*) [inline], [static]

Sets the specified uncorrectable RAM error status flag.

Parameters

stsFlags	is a bit mask of the error sources. This parameter can be any of the following values: **MEMCFG_UCERR_CPUREAD**, **MEMCFG_UCERR_DMAREAD**, or **MEMCFG_UCERR_CLA1READ**.

This function sets the specified uncorrectable RAM error status flag.

Returns
None.

27.2.3.18 static void MemCfg_clearCorrErrorStatus (uint32_t *stsFlags*) [inline], [static]

Clears correctable RAM error flags.

Parameters

stsFlags	is a bit mask of the status flags to be cleared. This parameter can be any of the **MEMCFG_CERR_CPUREAD**, **MEMCFG_CERR_DMAREAD**, or **MEMCFG_CERR_CLA1READ** values.

This function clears the specified correctable RAM error flags.

Returns
None.

27.2.3.19 static void MemCfg_clearUncorrErrorStatus (uint32_t *stsFlags*) [inline], [static]

Clears uncorrectable RAM error flags.

Parameters

stsFlags	is a bit mask of the status flags to be cleared. This parameter can be any of the **MEMCFG_UCERR_CPUREAD**, **MEMCFG_UCERR_DMAREAD**, or **MEMCFG_UCERR_CLA1READ** values.

This function clears the specified uncorrectable RAM error flags.

Returns
None.

27.2.3.20 void MemCfg_lockConfig (uint32_t *ramSections*)

Locks the writes to the configuration of specified RAM sections.

Parameters

ramSections	is the logical OR of the sections to be configured.

This function locks writes to the access protection and master select configuration of a RAM section. That means calling MemCfg_setProtection() or MemCfg_setLSRAMMasterSel() for a locked RAM section will have no effect until MemCfg_unlockConfig() is called.

The *ramSections* parameter is an OR of one of the following sets of indicators:

- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx** or **MEMCFG_SECT_LSX_ALL**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx** or **MEMCFG_SECT_GSX_ALL**
- **OR** use **MEMCFG_SECT_ALL** to configure all possible sections.

Returns

None.

27.2.3.21 void MemCfg_unlockConfig (uint32_t *ramSections*)

Unlocks the writes to the configuration of a RAM section.

Parameters

ramSections	is the logical OR of the sections to be configured.

This function unlocks writes to the access protection and master select configuration of a RAM section that has been locked using MemCfg_lockConfig().

The *ramSections* parameter is an OR of one of the following sets of indicators:

- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx** or **MEMCFG_SECT_LSX_ALL**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx** or **MEMCFG_SECT_GSX_ALL**
- **OR** use **MEMCFG_SECT_ALL** to configure all possible sections.

Returns

None.

27.2.3.22 void MemCfg_commitConfig (uint32_t *ramSections*)

Permanently locks writes to the configuration of a RAM section.

Parameters

ramSections	is the logical OR of the sections to be configured.

This function permanently locks writes to the access protection and master select configuration of a RAM section. That means calling MemCfg_setProtection() or MemCfg_setLSRAMMasterSel() for a locked RAM section will have no effect. To lock the configuration in a nonpermanent way, use MemCfg_lockConfig().

The *ramSections* parameter is an OR of one of the following sets of indicators:

- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx** or **MEMCFG_SECT_LSX_ALL**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx** or **MEMCFG_SECT_GSX_ALL**
- **OR** use **MEMCFG_SECT_ALL** to configure all possible sections.

Returns

None.

27.2.3.23 void MemCfg_setProtection (uint32_t *ramSection,* uint32_t *protectMode*)

Sets the access protection mode of a single RAM section.

Parameters

ramSection	is the RAM section to be configured.
protectMode	is the logical OR of the settings to be applied.

This function sets the access protection mode of the specified RAM section. The mode is passed into the *protectMode* parameter as the logical OR of the following values:

- **MEMCFG_PROT_ALLOWCPUFETCH** or **MEMCFG_PROT_BLOCKCPUFETCH** - CPU fetch
- **MEMCFG_PROT_ALLOWCPUWRITE** or **MEMCFG_PROT_BLOCKCPUWRITE** - CPU write
- **MEMCFG_PROT_ALLOWDMAWRITE** or **MEMCFG_PROT_BLOCKDMAWRITE** - DMA write

The *ramSection* parameter is one of the following indicators:

- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx**

This function will have no effect if the associated registers have been locked by MemCfg_lockConfig() or MemCfg_commitConfig() or if the memory is configured as CLA program memory.

Returns

None.

27.2.3.24 void MemCfg_setLSRAMMasterSel (uint32_t *ramSection,* **MemCfg_LSRAMMasterSel** *masterSel*)

Sets the master of the specified RAM section.

Parameters

ramSection	is the RAM section to be configured.
masterSel	is the sharing selection.

This function sets the master select configuration of the RAM section. If the *masterSel* parameter is **MEMCFG_LSRAMMASTER_CPU_ONLY**, the RAM section passed into the *ramSection* parameter will be dedicated to the CPU. If **MEMCFG_LSRAMMASTER_CPU_CLA1**, the memory section will be shared between the CPU and the CLA.

The *ramSection* parameter should be a value from **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx**.

This function will have no effect if the associated registers have been locked by MemCfg_lockConfig() or MemCfg_commitConfig().

Note

This API only applies to LSx RAM.

Returns

None.

27.2.3.25 void MemCfg_setTestMode (uint32_t *ramSection*, **MemCfg_TestMode** *testMode*)

Sets the test mode of the specified RAM section.

Parameters

ramSection	is the RAM section to be configured.
testMode	is the test mode selected.

This function sets the test mode configuration of the RAM section. The *testMode* parameter can take one of the following values:

- **MEMCFG_TEST_FUNCTIONAL**
- **MEMCFG_TEST_WRITE_DATA**
- **MEMCFG_TEST_WRITE_ECC** (DxRAM) or MEMCFG_TEST_WRITE_PARITY (LSx, GSx, or MSGxRAM)

The *ramSection* parameter is one of the following indicators:

- **MEMCFG_SECT_M0** or **MEMCFG_SECT_M1**
- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx**
- **MEMCFG_SECT_MSGCPUTOCLA1** or **MEMCFG_SECT_MSGCLA1TOCPU**

Returns

None.

27.2.3.26 void MemCfg_initSections (uint32_t *ramSections*)

Starts the initialization the specified RAM sections.

Parameters

ramSections	is the logical OR of the sections to be initialized.

This function starts the initialization of the specified RAM sections. Use MemCfg_getInitStatus() to check if the initialization is done.

The *ramSections* parameter is an OR of one of the following sets of indicators:

- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx** or **MEMCFG_SECT_LSX_ALL**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx** or **MEMCFG_SECT_GSX_ALL**
- **MEMCFG_SECT_MSGCPUTOCLA1** and **MEMCFG_SECT_MSGCLA1TOCPU** or **MEMCFG_SECT_MSGX_ALL**
- **OR** use **MEMCFG_SECT_ALL** to configure all possible sections.

Returns

None.

27.2.3.27 bool MemCfg_getInitStatus (uint32_t *ramSections*)

Get the status of initialized RAM sections.

Parameters

ramSections	is the logical OR of the sections to be checked.

This function gets the initialization status of the RAM sections specified by the *ramSections* parameter.

The *ramSections* parameter is an OR of one of the following sets of indicators:

- **MEMCFG_SECT_M0** and **MEMCFG_SECT_M1** or **MEMCFG_SECT_DX_ALL**
- **MEMCFG_SECT_LS0** through **MEMCFG_SECT_LSx** or **MEMCFG_SECT_LSX_ALL**
- **MEMCFG_SECT_GS0** through **MEMCFG_SECT_GSx** or **MEMCFG_SECT_GSX_ALL**
- **MEMCFG_SECT_MSGCPUTOCLA1** and **MEMCFG_SECT_MSGCLA1TOCPU** or **MEMCFG_SECT_MSGX_ALL**
- **OR** use **MEMCFG_SECT_ALL** to get status of all possible sections.

Note

Use MemCfg_initSections() to start the initialization.

Returns

Returns **true** if all the sections specified by *ramSections* have been initialized and **false** if not.

27.2.3.28 uint32_t MemCfg_getViolationAddress (uint32_t *intFlag*)

Get the violation address associated with a intFlag.

Parameters

intFlag	is the type of access violation as indicated by ONE of these values: ■ **MEMCFG_NMVIOL_CPUREAD** ■ **MEMCFG_NMVIOL_CPUWRITE** ■ **MEMCFG_NMVIOL_CPUFETCH** ■ **MEMCFG_NMVIOL_DMAWRITE** ■ **MEMCFG_NMVIOL_CLA1READ** ■ **MEMCFG_NMVIOL_CLA1WRITE** ■ **MEMCFG_NMVIOL_CLA1FETCH** ■ **MEMCFG_MVIOL_CPUFETCH** ■ **MEMCFG_MVIOL_CPUWRITE** ■ **MEMCFG_MVIOL_DMAWRITE**

Returns

Returns the violation address associated with the *intFlag*.

27.2.3.29 uint32_t MemCfg_getCorrErrorAddress (uint32_t *stsFlag*)

Get the correctable error address associated with a stsFlag.

Parameters

stsFlag	is the type of error to which the returned address will correspond. Can currently take the value **MEMCFG_CERR_CPUREAD** only. Other values are reserved.

Returns

Returns the error address associated with the stsFlag.

27.2.3.30 uint32_t MemCfg_getUncorrErrorAddress (uint32_t *stsFlag*)

Get the uncorrectable error address associated with a stsFlag.

Parameters

stsFlag	is the type of error to which the returned address will correspond. It may be passed one of these values: **MEMCFG_UCERR_CPUREAD**, **MEMCFG_UCERR_DMAREAD**, or **MEMCFG_UCERR_CLA1READ** values.

Returns

Returns the error address associated with the stsFlag.

28 PGA Module

Introduction 522
API Functions 522

28.1 PGA Introduction

The Programmable Gain Amplifier (PGA) API provides a set of functions for configuring and using the PGA module. The functions provided allow the user to setup and configure the PGA by providing access to the PGA wrapper registers.

28.2 API Functions

Macros

- #define PGA_REGISTER_PGACTL
- #define PGA_REGISTER_GAIN3TRIM
- #define PGA_REGISTER_GAIN6TRIM
- #define PGA_REGISTER_GAIN12TRIM
- #define PGA_REGISTER_GAIN24TRIM

Enumerations

- enum PGA_GainValue { PGA_GAIN_3, PGA_GAIN_6, PGA_GAIN_12, PGA_GAIN_24 }
- enum PGA_LowPassResistorValue {
  PGA_LOW_PASS_FILTER_DISABLED, PGA_LOW_PASS_FILTER_RESISTOR_200_OHM,
  PGA_LOW_PASS_FILTER_RESISTOR_160_OHM,
  PGA_LOW_PASS_FILTER_RESISTOR_130_OHM,
  PGA_LOW_PASS_FILTER_RESISTOR_100_OHM,
  PGA_LOW_PASS_FILTER_RESISTOR_80_OHM,
  PGA_LOW_PASS_FILTER_RESISTOR_50_OHM }

Functions

- static void PGA_enable (uint32_t base)
- static void PGA_disable (uint32_t base)
- static void PGA_setGain (uint32_t base, PGA_GainValue gainValue)
- static void PGA_setFilterResistor (uint32_t base, PGA_LowPassResistorValue resistorValue)
- static uint16_t PGA_getPGARevision (uint32_t base)
- static uint16_t PGA_getPGAType (uint32_t base)
- static void PGA_lockRegisters (uint32_t base, uint16_t registerType)

28.2.1 Detailed Description

The code for this module is contained in `driverlib/pga.c`, with `driverlib/pga.h` containing the API declarations for use by applications.

28.2.2 Macro Definition Documentation

28.2.2.1 #define PGA_REGISTER_PGACTL

PGA Register PGACTL

28.2.2.2 #define PGA_REGISTER_GAIN3TRIM

PGA Register GAIN3TRIM

28.2.2.3 #define PGA_REGISTER_GAIN6TRIM

PGA Register GAIN6TRIM

28.2.2.4 #define PGA_REGISTER_GAIN12TRIM

PGA Register GAIN12TRIM

28.2.2.5 #define PGA_REGISTER_GAIN24TRIM

PGA Register GAIN24TRIM

28.2.3 Enumeration Type Documentation

28.2.3.1 enum **PGA_GainValue**

Values that can be passed to PGA_setGain() as the *gainValue* parameter.

Enumerator

PGA_GAIN_3 PGA gain value of 3.
PGA_GAIN_6 PGA gain value of 6.
PGA_GAIN_12 PGA gain value of 12.
PGA_GAIN_24 PGA gain value of 24.

28.2.3.2 enum **PGA_LowPassResistorValue**

Values that can be passed to PGA_setFilterResistor() as the *resistorValue* parameter.

Enumerator

PGA_LOW_PASS_FILTER_DISABLED Low pass filter disabled (open circuit)
PGA_LOW_PASS_FILTER_RESISTOR_200_OHM Resistor value of 200 Ohm.
PGA_LOW_PASS_FILTER_RESISTOR_160_OHM Resistor value of 160 Ohm.
PGA_LOW_PASS_FILTER_RESISTOR_130_OHM Resistor value of 130 Ohm.
PGA_LOW_PASS_FILTER_RESISTOR_100_OHM Resistor value of 100 Ohm.
PGA_LOW_PASS_FILTER_RESISTOR_80_OHM Resistor value of 80 Ohm.
PGA_LOW_PASS_FILTER_RESISTOR_50_OHM Resistor value of 50 Ohm.

28.2.4 Function Documentation

28.2.4.1 static void PGA_enable (uint32_t *base*) `[inline], [static]`

Enables PGA.

Parameters

base	is the base address of the PGA module.

This function enables the PGA module.

Returns

None.

28.2.4.2 static void PGA_disable (uint32_t *base*) `[inline], [static]`

Disables PGA.

Parameters

base	is the base address of the PGA module.

This function disables the PGA module.

Returns

None.

28.2.4.3 static void PGA_setGain (uint32_t *base,* **PGA_GainValue** *gainValue*) `[inline], [static]`

Sets PGA gain value

Parameters

base	is the base address of the PGA module.
gainValue	is the PGA gain value.

This function sets the PGA gain value.

Returns
None.

28.2.4.4 static void PGA_setFilterResistor (uint32_t *base*, **PGA_LowPassResistorValue** *resistorValue*) [inline], [static]

Sets PGA output filter resistor value

Parameters

base	is the base address of the PGA module.
resistorValue	is the PGA output resistor value.

This function sets the resistor value for the PGA output low pass RC filter. The resistance for the RC low pass filter is provided within the microprocessor and is determined by the value of resistorValue. The capacitor, however, has to be connected outside the microprocessor.

Note: Setting a value of PGA_LOW_PASS_FILTER_RESISTOR_0_OHM will disable the internal resistance value.

Returns
None.

28.2.4.5 static uint16_t PGA_getPGARevision (uint32_t *base*) [inline], [static]

Returns the PGA revision number.

Parameters

base	is the base address of the PGA module.

This function returns the PGA revision number.

Returns
Returns PGA revision.

28.2.4.6 static uint16_t PGA_getPGAType (uint32_t *base*) [inline], [static]

Returns the PGA Type.

Parameters

base	is the base address of the PGA module.

This function returns the PGA Type number.

Returns
Returns PGA type.

28.2.4.7 static void PGA_lockRegisters (uint32_t *base,* uint16_t *registerType*) [inline], [static]

Locks PGA registers.

Parameters

base	is the base address of the PGA module.
registerType	is the PGA register to be locked.

This function locks the PGA registers specified by registerType. Valid values for registerType are: PGA_REGISTER_PGACTL, PGA_REGISTER_GAINxTRIM, where x is 3, 6, 12 or 24.

Returns
None.

29 PMBus Module

Introduction . . . 527
PMBus Message Types . . . 534
API Functions . . . 534

29.1 PMBus Introduction

The PMBus peripheral module supports both master and slave features and is compliant to the PMBus protocol v1.0. It supports the following transactions:

1. Quick Command
2. Send Byte
3. Write Byte
4. Write Word
5. Block Write
6. Receive Byte
7. Alert Response
8. Read Byte
9. Read Word
10. Block Read
11. Block Write/ Read/ Process Call
12. Group Command
13. Extended Read/Write Byte/Word

The **PMBus Communications Stack** is available as a separate library with support for the slave mode of operation and a state machine to handle the transactions mentioned above.

The PMBus module can also be placed into I2C mode, and the existing functions in driverlib can be used to communicate on an I2C bus. The I2C mode is compliant to the Philips specification with one exception – in the event of an arbitration lost the module does not automatically switch from master to slave mode. The user must poll the **ARB_LOST** bit in the status register and then place the module in slave mode before it is able to receive any data from the winning master.

The code for this module is contained in `driverlib/pmbus.c`, with `driverlib/pmbus.h` containing the API declarations for use by applications.

29.2 PMBus Message Types

This section describes the different transactions (message types) that are recognized, and supported by the PMBus module. The messages are depicted from a slave mode perspective.

The primary signals in determining the type of message are

- DATA_READY

- EOM
- DATA_REQUEST
- RD_BYTE_COUNT

The following abbreviations are used in the descriptions of the transactions,

S
The start signal on the bus
ADDR
The address of the slave device
Rd/R
The read bit asserted after the slave address is put on the bus
Wr/W
The write bit asserted after the slave address is put on the bus
A
Acknowledgment
NA
NACK or No Acknowledgment
P
The stop signal on the bus
Sr
Repeated Start
PEC
Packet Error Check byte

Each transaction (message) description will have an image of the message format; Fig. 29.1 describes the convention used,

Figure 29.1: Message Format Legend

29.2.1 Quick Command

When a Quick Command is received, the **EOM (End-of-Message)** status bit is set, and the **RD_BYTE_COUNT (Received Byte Count)** field is 0.

The Slave manually ACKs the transaction by writing to the PMBACK register.

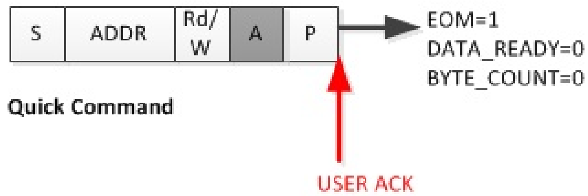

Figure 29.2: Quick Command

29.2.2 Send Byte

When a Send Byte is received, the **DATA_READY** and **EOM (End-of-Message)** status bits are set, and the **RD_BYTE_COUNT (Received Byte Count)** field is 2, indicating two bytes were received, the data byte and the PEC.

The Slave reads the data and manually ACKs the message by writing to the PMBACK register.

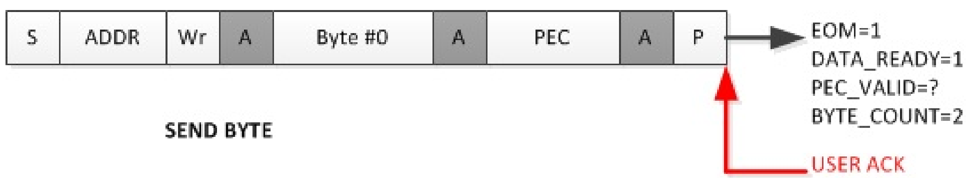

Figure 29.3: Send Byte

29.2.3 Write Byte

The Write Byte is identical to Send Byte, with the exception that **RD_BYTE_COUNT (Received Byte Count)** is now 3, that is, a command byte, a data byte and the PEC.

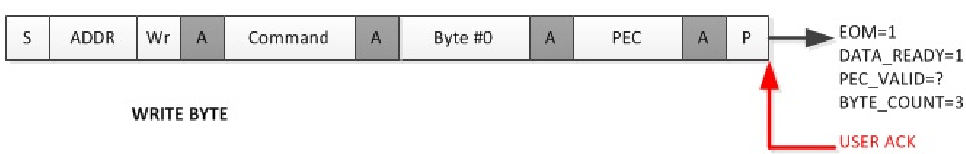

Figure 29.4: Write Byte

29.2.4 Write Word

The Write Word is identical to Send Byte, with the exception that **RD_BYTE_COUNT (Received Byte Count)** is 4, that is, a command byte, 2 data bytes and the PEC.

Figure 29.5: Write Word

29.2.5 Block Write

The Block Write is issued when the master has to transfer more than 2 data bytes (up to a maximum of 255 bytes). The master will transmit a command, a count (how many bytes it intends to send), followed by the bytes, ending with the PEC.

For every 4 bytes the slave receives, **DATA_READY** is asserted and **RD_BYTE_COUNT** is 4; no End-of-Message (EOM) is received at this point. The slave must read the receive buffer and manually ACK reception of 4 bytes before the master can proceed sending the next 4 bytes. On the very last transmission **DATA_READY** and **EOM** are asserted, indicating the end of transmission. The slave must read the receive buffer (which has 1 to 4 bytes depending on the initial count) and manually ACK the transaction.

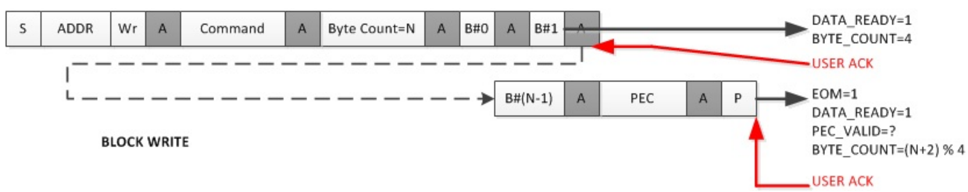

Figure 29.6: Block Write

29.2.6 Receive Byte

The master initiates a Receive Byte by putting the slave's address on the bus followed by a read bit. The slave will automatically ACK its address, load its transmit buffer, and transmit a byte and its PEC.

If there is no error in the transmission the master will **NACK** the PEC indicating the end of the transaction. Both the **NACK** and **EOM** status bits are asserted at this point.

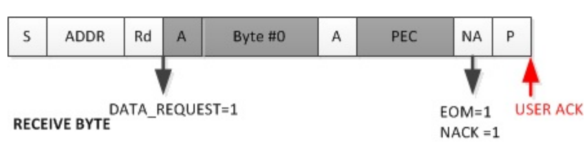

Figure 29.7: Receive Byte

29.2.7 Alert Response

A special variant of the Receive Byte is the **Alert Response** transactions where the slave device pulls the **ALERT** line low; the master must respond with the **ALERT RESPONSE ADDRESS** and a read. The alerting slave will respond by transmitting its own address as shown in Fig. 29.8.

When the master puts the Alert Response Address on the line with a read, the alerting slave hardware will automatically respond with its address, without software intervention.
NOTE: THE 7 BIT DEVICE ADDRESS PROVIDED BY THE SLAVE TRANSMIT DEVICE IS PLACED IN THE 7 MOST SIGNIFICANT BITS OF THE BYTE. THE EIGHTH BIT CAN BE A ZERO OR ONE.

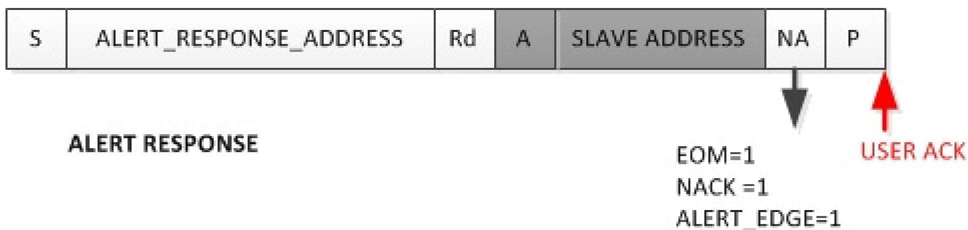

Figure 29.8: Alert Responsed

29.2.8 Read Byte

The master initiates a Read Byte by putting the slave's address on the bus followed by a write bit. The master issues a Read Byte command followed by a repeated start with the slave address and the read bit. When the repeated start is issued on the bus, the **DATA_READY** bit is asserted at the slave end with a **RD_BYTE_COUNT** of 1. At the read bit the **DATA_REQUEST** bit is asserted; the slave responds by transmitting a single byte followed by the PEC. If there is no error in the transmission, the master will **NACK** the PEC, indicating the end of the transaction. Both the **NACK** and **EOM** status bits are asserted at this point.

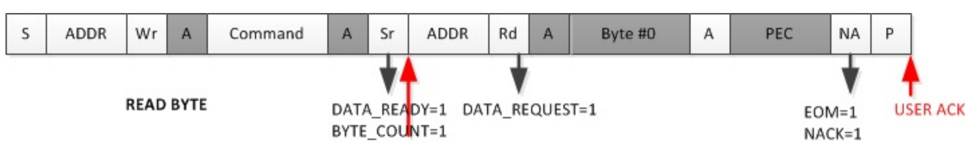

Figure 29.9: Read Byte

29.2.9 Read Word

Read Word is similar to Read Byte with the exception that the slave responds to the repeated start (read bit) by transmitting two bytes followed by the PEC.

;

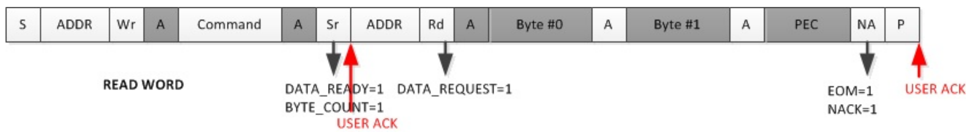

Figure 29.10: Read Word

29.2.10 Block Read

If the master transmits a Block Read command, the slave responds by sending more than 2 bytes (up to a maximum of 255 bytes). The transaction, including the status bit assertions, is similar to the read word command. The first byte sent by the slave is always the byte count, that is, the number of bytes it intends to transmit. It then follows this with the data bytes. For every 4 bytes sent by the slave (and acknowledged by the master) the **DATA_REQUEST** bit is asserted, requesting the slave to send the next set of bytes. The transaction is terminated by the master by issuing a **NACK** on the bus; both the **NACK** and **EOM** status bits are asserted at the slave end at this point.

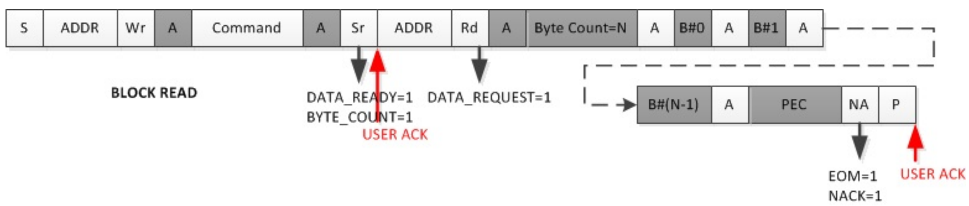

Figure 29.11: Block Read

29.2.11 Block Write/ Read/ Process Call

This is basically a block write followed by a block read. The key points to note here are the byte counts on the block write and block read must be the same, and a single PEC is sent at the end of the block read.

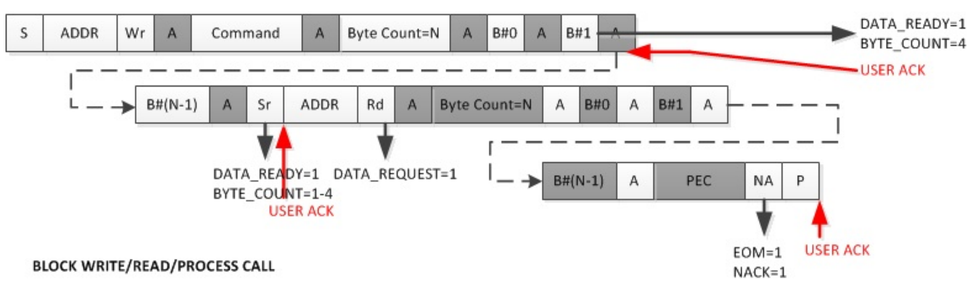

Figure 29.12: Block Write/ Read/ Process Call

NOTE: A PROCESS CALL, A WRITE WORD FOLLOWED BY A READ WORD, FALLS UNDER THE PURVIEW OF THIS TRANSACTION IN THE STATE MACHINE HANDLER

29.2.12 Group Command

The Master writes to a group of slaves in a single transaction. It does this by putting each slave's address (with a write) followed by a command, two bytes, and a PEC on the bus after a repeated start (the exception is the first slave address which follows the start). A slave device will acknowledge its address on the bus, and its state machine will respond when the **DATA_READY** is asserted on the next repeated start (or on a stop, if the slave in question is the last to be addressed).

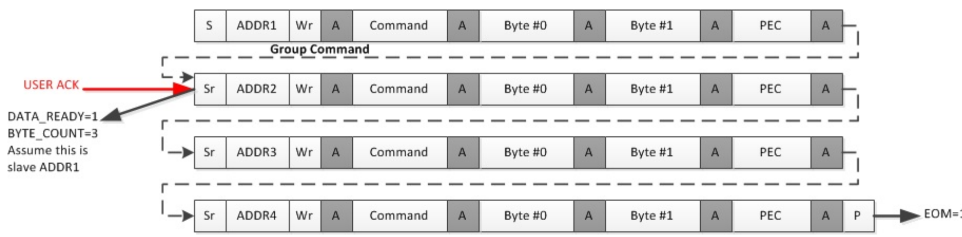

Figure 29.13: Group Command

29.2.13 Extended Command

The extended command is supported for four transaction types

1. Extended Read Byte
2. Extended Read Word
3. Extended Write Byte
4. Extended Write Word

These commands are similar to their non-extended counterparts, with the exception that the command is preceded by the extension byte (0xFE or 0xFF). The master issues a repeated start with the slave address and the read (for a read transaction) or write (for a write transaction) bit asserted.

NOTE: THIS BEHAVIOR CONFORMS TO V1.0 OF THE PMBUS PROTOCOL; V1.2 OF THE PROTOCOL DOES NOT REQUIRE A REPEATED START (AND SLAVE ADDRESS) AFTER THE EXTENSION AND COMMAND BYTES ARE SENT

At this point the slave sees the **DATA_READY** signal asserted and a **RD_BYTE_COUNT** of 2. It must check the first byte for the extension code before acknowledging. If the transaction is a write the master proceeds; an extended write byte involves 4 bytes: the extension code, the command byte, a data byte, and finally the PEC, whereas a write word transaction involves an additional data byte, making the total 5 bytes. If the transaction is a read, the slave must transfer 1 (read byte) or 2 (read words) bytes depending on the command received, followed by the PEC.

NOTE: THE PEC IS CALCULATED ON THE SLAVE ADDRESS (WITH WRITE BIT ASSERTED), THE EXTENSION, COMMAND BYTE, SECOND SLAVE ADDRESS (AND EITHER READ/WRITE BIT DEPENDING ON THE TRANSACTION), AND FINALLY THE DATA BYTE(S).

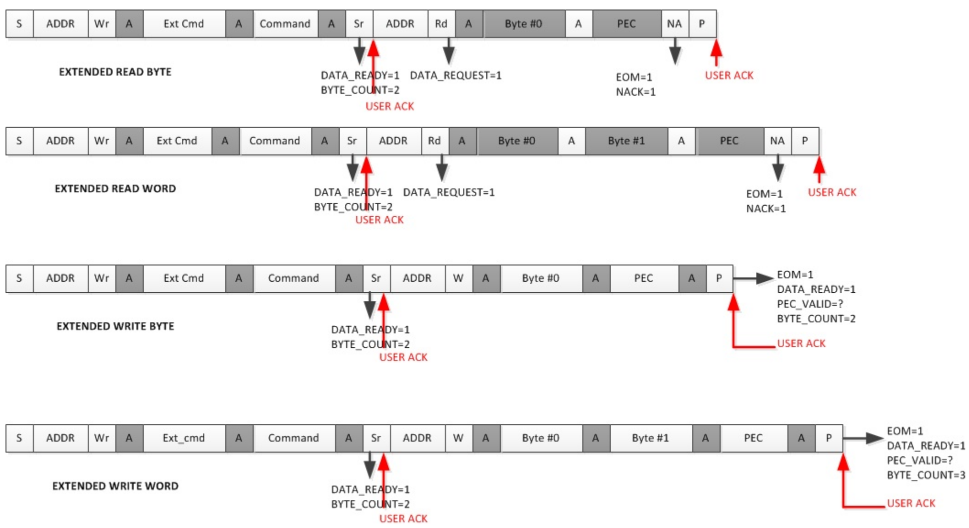

Figure 29.14: Extended Commands

The code for this module is contained in `driverlib/pmbus.c`, with `driverlib/pmbus.h` containing the API declarations for use by applications.

29.3 API Functions

Macros

- #define PMBUS_SYS_FREQ_MIN
- #define PMBUS_SYS_FREQ_MAX
- #define PMBUS_MODULE_FREQ_MAX
- #define PMBUS_MODULE_FREQ_MIN

Enumerations

- enum PMBus_Transaction { PMBUS_TRANSACTION_NONE, PMBUS_TRANSACTION_QUICKCOMMAND, PMBUS_TRANSACTION_WRITEBYTE, PMBUS_TRANSACTION_READBYTE, PMBUS_TRANSACTION_SENDBYTE, PMBUS_TRANSACTION_RECEIVEBYTE, PMBUS_TRANSACTION_BLOCKWRITE, PMBUS_TRANSACTION_BLOCKREAD, PMBUS_TRANSACTION_WRITEWORD, PMBUS_TRANSACTION_READWORD, PMBUS_TRANSACTION_BLOCKWRPC }
- enum PMBus_ClockMode { PMBUS_CLOCKMODE_STANDARD, PMBUS_CLOCKMODE_FAST, PMBUS_CLOCKMODE_FASTPLUS }
- enum PMBus_accessType { PMBUS_ACCESSTYPE_WRITE, PMBUS_ACCESSTYPE_READ }
- enum PMBus_intEdge { PMBUS_INTEDGE_FALLING, PMBUS_INTEDGE_RISING }

Functions

- static void PMBus_disableModule (uint32_t base)
- static void PMBus_enableModule (uint32_t base)
- static void PMBus_enableInterrupt (uint32_t base, uint32_t intFlags)
- static void PMBus_disableInterrupt (uint32_t base, uint32_t intFlags)
- static bool PMBus_isBusBusy (uint32_t status)
- static bool PMBus_isPECValid (uint32_t status)
- static void PMBus_enableI2CMode (uint32_t base)
- static void PMBus_disableI2CMode (uint32_t base)
- static uint32_t PMBus_getStatus (uint32_t base)
- static void PMBus_ackTransaction (uint32_t base)
- static void PMBus_nackTransaction (uint32_t base)
- static void PMBus_assertAlertLine (uint32_t base)
- static void PMBus_deassertAlertLine (uint32_t base)
- static void PMBus_configMaster (uint32_t base, uint16_t slaveAddress, uint16_t byteCount, uint32_t configWord)
- static uint16_t PMBus_getOwnAddress (uint32_t base)
- static PMBus_accessType PMBus_getCurrentAccessType (uint32_t base)
- static void PMBus_setCtrlIntEdge (uint32_t base, PMBus_intEdge intEdge)
- static void PMBus_setClkLowTimeoutIntEdge (uint32_t base, PMBus_intEdge intEdge)
- void PMBus_initSlaveMode (uint32_t base, uint16_t address, uint16_t mask)
- void PMBus_configSlave (uint32_t base, uint32_t configWord)
- uint32_t PMBus_getInterruptStatus (uint32_t base)
- uint16_t PMBus_getData (uint32_t base, uint16_t ∗buffer, uint32_t status)
- void PMBus_putSlaveData (uint32_t base, uint16_t ∗buffer, uint16_t nBytes, bool txPEC)
- void PMBus_ackAddress (uint32_t base, uint32_t address, uint32_t status, uint16_t ∗buffer)
- void PMBus_ackCommand (uint32_t base, uint32_t command, uint32_t status, uint16_t ∗buffer)
- void PMBus_generateCRCTable (uint16_t ∗crcTable)
- bool PMBus_verifyPEC (uint32_t base, uint16_t ∗buffer, const uint16_t ∗crcTable, uint16_t byteCount, uint16_t pec)
- void PMBus_initMasterMode (uint32_t base)
- void PMBus_putMasterData (uint32_t base, uint16_t ∗buffer, uint16_t nBytes)
- uint32_t PMBus_configModuleClock (uint32_t base, uint32_t moduleFrequency, uint32_t sysFrequency)
- bool PMBus_configBusClock (uint32_t base, PMBus_ClockMode mode, uint32_t moduleFrequency)

29.3.1 Detailed Description

The API functions provide for basic configurability of the PMBus registers; they do not directly support the different transaction types mentioned in the introduction. For example, if attempting to perform a send byte as Master, the user must first set the PMBus in master mode (PMBus_initMasterMode()), then configure the master control register (with PMBus_configMaster()), and then load a single byte into the transmission register using PMBus_putMasterData().

Similarly, in slave mode the user would set the PMBus module in slave mode (with PMBus_initSlaveMode()), configure the slave (PMBus_configSlave()), and then wait for the send byte event. The user checks for this event by polling the status register (PMBus_getStatus()) bits, EOM (End-of-Message) and DATA_READY, and then checking if the RD_BYTE_COUNT is either 1 (no Packet Error Check) or 2 (PEC is enabled). The **PMBus Communications Stack** implements a state machine to handle these transactions (in slave mode only) and the user is referred to that package for examples and further documentation.

In Master mode the user must immediately follow (or precede) PMBus_configMaster() with PMBus_putMasterData(). The act of configuring the master control register will trigger a transaction on the bus; the data in the transmit registers will be put on the bus (depending on how many bytes were set to be in the transfer).

The API does not have separate functions to handle the I2C mode of operation. Similar to PMBus mode of operation, the user must configure the master (or slave) control registers using PMBus_configMaster() or PMBus_configSlave() and write (or read) data using PMBus_putMasterData(), PMBus_putSlaveData(), PMBus_getMasterData(), and PMBus_getSlaveData().

The act of reading the status register using the PMBus_getStatus() clears the content of the status registers. If using an interrupts based approach, it is advisable to read the contents of the status register once at the beginning of the Interrupt Service Routine (ISR).

29.3.2 Enumeration Type Documentation

29.3.2.1 enum **PMBus_Transaction**

Transaction Descriptor

Defines the transaction type, used in the command object and passed to PMBus_configTransfer()

Enumerator

PMBUS_TRANSACTION_NONE No Transaction.
PMBUS_TRANSACTION_QUICKCOMMAND Quick Command.
PMBUS_TRANSACTION_WRITEBYTE Write single byte.
PMBUS_TRANSACTION_READBYTE Read single byte.
PMBUS_TRANSACTION_SENDBYTE Send Byte.
PMBUS_TRANSACTION_RECEIVEBYTE Receive Byte.
PMBUS_TRANSACTION_BLOCKWRITE Block Write (up to 255 bytes)
PMBUS_TRANSACTION_BLOCKREAD Block Read (up to 255 bytes)
PMBUS_TRANSACTION_WRITEWORD Write word.
PMBUS_TRANSACTION_READWORD Read word.
PMBUS_TRANSACTION_BLOCKWRPC Block write, then process call.

29.3.2.2 enum **PMBus_ClockMode**

Clock Mode Descriptor

Used in PMBus_configBusClock() to set up the bus speed. There are three possible modes of operation:

1. Standard Mode 100 kHz
2. Fast Mode 400 kHz
3. Fast Mode Plus 1 MHz

Enumerator

PMBUS_CLOCKMODE_STANDARD Standard mode 100 kHz.

PMBUS_CLOCKMODE_FAST Fast Mode 400 kHz.
PMBUS_CLOCKMODE_FASTPLUS Fast Mode plus 1 MHz.

29.3.2.3 enum **PMBus_accessType**

Access Type Descriptor

Used in PMBus_getCurrentAccessType() to determine if the device, in slave mode, was accessed with read or write enabled.

Enumerator

PMBUS_ACCESSTYPE_WRITE Slave last address for write transaction.
PMBUS_ACCESSTYPE_READ Slave last address for read transaction.

29.3.2.4 enum **PMBus_intEdge**

Interrupt Edge Descriptor

Used in PMBus_setCtrlIntEdge() and PMBus_setClkLowTimeoutIntEdge() to set the edge, falling or rising, that triggers an interrupt

Enumerator

PMBUS_INTEDGE_FALLING Interrupt generated on falling edge.
PMBUS_INTEDGE_RISING Interrupt generated on rising edge.

29.3.3 Function Documentation

29.3.3.1 static void PMBus_disableModule (uint32_t *base*) [inline], [static]

Disables the PMBus module.

Parameters

base	is the base address of the PMBus instance used.

This function resets the internal state machine of the PMBus module and holds it in that state

Returns

None.

29.3.3.2 static void PMBus_enableModule (uint32_t *base*) [inline], [static]

Enables the PMBus module.

Parameters

base	is the base address of the PMBus instance used.

This function enables operation of the PMBus module by removing it from the reset state

Returns
None.

29.3.3.3 static void PMBus_enableInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Enables PMBus interrupt sources.

Parameters

base	is the base address of the PMBus instance used.
intFlags	is the bit mask of the interrupt sources to be enabled.

This function enables the indicated PMBus interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The *intFlags* parameter is the logical OR of any of the following:

- **PMBUS_INT_BUS_FREE** - Bus Free Interrupt
- **PMBUS_INT_CLK_LOW_TIMEOUT** - Clock Low Time-out Interrupt
- **PMBUS_INT_DATA_READY** - Data Ready Interrupt
- **PMBUS_INT_DATA_REQUEST** - Data Request Interrupt
- **PMBUS_INT_SLAVE_ADDR_READY** - Slave Address Ready Interrupt
- **PMBUS_INT_EOM** - End of Message Interrupt
- **PMBUS_INT_ALERT** - Alert Detection Interrupt
- **PMBUS_INT_CONTROL** - Control Detection Interrupt
- **PMBUS_INT_LOST_ARB** - Lost Arbitration Interrupt
- **PMBUS_INT_CLK_HIGH_DETECT** - Clock High Detection Interrupt
- **PMBUS_INT_ALL** - all PMBus interrupts

Returns
None.

29.3.3.4 static void PMBus_disableInterrupt (uint32_t *base,* uint32_t *intFlags*) [inline], [static]

Disables PMBus interrupt sources.

Parameters

base	is the base address of the PMBus instance used.

intFlags	is the bit mask of the interrupt sources to be disabled.

This function disables the indicated PMBus interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt. Disabled sources have no effect on the processor.

The *intFlags* parameter has the same definition as the *intFlags* parameter to PMBus_enableInterrupt().

Returns

None.

29.3.3.5 static bool PMBus_isBusBusy (uint32_t *status*) [inline], [static]

Indicates whether or not the PMBus bus is busy.

Parameters

status	the value of the status register (PMBUS_O_PMBSTS)

This function returns an indication of whether or not the PMBus bus is busy

Note

The status register is cleared each time it is read, therefore, it should be read once at the beginning of an interrupt service routine using PMBus_getInterruptStatus() and saved to a temporary variable for further processing.

Returns

Returns **true** if the PMBus bus is busy; otherwise, returns **false**.

29.3.3.6 static bool PMBus_isPECValid (uint32_t *status*) [inline], [static]

Indicates whether or not the PEC is valid

Parameters

status	the value of the status register (PMBUS_O_PMBSTS)

This function returns an indication of whether or not the received PEC was valid

Note

The status register is cleared each time it is read, therefore, it should be read once at the beginning of an interrupt service routine using PMBus_getStatus() and saved to a temporary variable for further processing.

Returns

Returns **true** if the PEC is valid; otherwise, returns **false**.

29.3.3.7 static void PMBus_enableI2CMode (uint32_t *base*) [inline], [static]

Enable I2C mode

Parameters

base	is the base address of the PMBus instance used.

Set the PMBus module to work in I2C mode

Returns
None.

29.3.3.8 static void PMBus_disableI2CMode (uint32_t *base*) [inline], [static]

Disable I2C mode

Parameters

base	is the base address of the PMBus instance used.

Set the PMBus module to work in PMBus mode

Returns
None.

29.3.3.9 static uint32_t PMBus_getStatus (uint32_t *base*) [inline], [static]

Read the status register

Parameters

base	is the base address of the PMBus instance used.

Returns
Contents of the status register.

29.3.3.10 static void PMBus_ackTransaction (uint32_t *base*) [inline], [static]

Acknowledge the transaction by writing to the PMBACK register

Parameters

base	is the base address of the PMBus instance used.

Returns
None.

29.3.3.11 static void PMBus_nackTransaction (uint32_t *base*) [inline], [static]

Nack the transaction by writing to the PMBACK register

Parameters

base	is the base address of the PMBus instance used.

Returns
None.

29.3.3.12 static void PMBus_assertAlertLine (uint32_t *base*) [inline], [static]

Alert the master by asserting the ALERT line

Parameters

base	is the base address of the PMBus instance used.

A slave PMBus can alert the master by pulling the alert line low. This triggers an Alert Response from the master, where the master issues the **Alert Response Address** on the bus with a read bit and the alerting slave is required to reply with its address.

Note
The alerting device should be in slave mode.

Returns
None.

29.3.3.13 static void PMBus_deassertAlertLine (uint32_t *base*) [inline], [static]

De-assert the alert line

Parameters

base	is the base address of the PMBus instance used.

Returns
None.

29.3.3.14 static void PMBus_configMaster (uint32_t *base,* uint16_t *slaveAddress,* uint16_t *byteCount,* uint32_t *configWord*) [inline], [static]

Configure the PMBus operation in Master Mode.

Parameters

base	is the base address of the PMBus instance used.
slaveAddress	address of the Slave device
byteCount	number of bytes transmitted (or read) in the message (up to 255)

configWord	can be a combination of the following 1. PMBUS_MASTER_ENABLE_PRC_CALL 2. PMBUS_MASTER_ENABLE_GRP_CMD 3. PMBUS_MASTER_ENABLE_PEC 4. PMBUS_MASTER_ENABLE_EXT_CMD 5. PMBUS_MASTER_ENABLE_CMD 6. PMBUS_MASTER_ENABLE_READ

Assuming the PMBus module is set to Master mode this function will configure the PMBMC register. It overwrites the contents of the PMBMC register.

Note

1. Writing to the PMBMC register initiates a message on the bus once the bus is free. In the event of a write the TXBUF must be loaded prior to configuration, or very quickly after configuration, before the module starts the bus clock.
2. If the user does not specify an option for example, PMBUS_MASTER_ENABLE_READ, the code will write a 0 (a write) in its bit field.
3. setting byteCount to 0U (on a write) triggers a quick command; there is no need to precede this command with the PMBus_putMasterData()
4. If transmitting with a non-zero byteCount the user must precede this with the PMBus_putMasterData(), supplying it with the location of the data and the number of bytes (<= 4). For block transmissions the user will have to call PMBus_putMasterData(), PMBus_configMaster() and then continue calling PMBus_putMasterData() transmitting 4 (or less for the final transmit) bytes at a time till all the data bytes are transmitted.
5. If receiving the user must follow up with the PMBus_getData(), supplying it with the location of an empty buffer and the status byte
6. In cases where the master must transmit for example, process call, the user must call PMBus_putMasterData(), then configure the master to transmit the command and two bytes, then call PMBus_getData() to read two bytes from the slave. The master module need not be reconfigured between write and read whenever a repeated start is involved in the transaction

Returns

None.

29.3.3.15 static uint16_t PMBus_getOwnAddress (uint32_t *base*) [inline], [static]

Get the address that the PMBus module will respond to (in slave mode)

Parameters

base	is the base address of the PMBus instance used.
ownAddress	address that the module will respond to.

This function will query the PMBUS_O_PMBHSA register, this will be the address of the module when used in Slave Mode.

Returns

Address of the PMBus device (in slave mode).

29.3.3.16 static **PMBus_accessType** PMBus_getCurrentAccessType (uint32_t *base*) [inline], [static]

Determine the current access (read/write) type

Parameters

base	is the base address of the PMBus instance used.

This function will query the PMBUS_O_PMBHSA register, to determine if the current access type was a read or write access. This bit is relevant only when the PMBus module is addressed as a slave.

Returns

an enum of the type PMBus_accessType which specifies if the device, in slave mode, was addressed for a read or write operation

29.3.3.17 static void PMBus_setCtrlIntEdge (uint32_t *base*, **PMBus_intEdge** *intEdge*) [inline], [static]

Sets the triggering edge of the Control Interrupt

Parameters

base	is the base address of the PMBus instance used.
intEdge	interrupt to trigger on rising or falling edge

Returns

None.

References PMBUS_INTEDGE_FALLING.

29.3.3.18 static void PMBus_setClkLowTimeoutIntEdge (uint32_t *base*, **PMBus_intEdge** *intEdge*) [inline], [static]

Sets the triggering edge of the Clock Low Time-out Interrupt

Parameters

base	is the base address of the PMBus instance used.
intEdge	interrupt to trigger on rising or falling edge

Returns

None.

References PMBUS_INTEDGE_FALLING.

29.3.3.19 void PMBus_initSlaveMode (uint32_t *base*, uint16_t *address*, uint16_t *mask*)

Initializes the PMBus to Slave Mode.

Parameters

base	is the base address of the PMBus instance used.
address	Slave address
mask	Slave address mask - Used in address detection, the slave mask enables acknowledgement of multiple device addresses by the slave. Writing a '0' to a bit within the slave mask enables the corresponding bit in the slave address to be either '1' or '0' and still allow for a match. Writing a '0' to all bits in the mask enables the PMBus Interface to acknowledge any device address. Upon power-up, the slave mask defaults to 7Fh, indicating the slave will only acknowledge the address programmed into the Slave Address (Bits 6-0). Set to PMBUS_DISABLE_SLAVE_ADDRESS_MASK if you do not wish to have a mask

This function sets up the PMBus in slave mode and also configures the slave address for the PMBus module

Returns

None.

29.3.3.20 void PMBus_configSlave (uint32_t *base,* uint32_t *configWord*)

Configure the PMBus operation in Slave Mode.

Parameters

base	is the base address of the PMBus instance used.
configWord	can be a combination of the following 1. PMBUS_SLAVE_ENABLE_MANUAL_ACK 2. PMBUS_SLAVE_ENABLE_PEC_PROCESSING 3. PMBUS_SLAVE_ENABLE_MANUAL_CMD_ACK 4. PMBUS_SLAVE_DISABLE_ADDRESS_MASK 5. PMBUS_SLAVE_AUTO_ACK_1_BYTES 6. PMBUS_SLAVE_AUTO_ACK_2_BYTES 7. PMBUS_SLAVE_AUTO_ACK_3_BYTES 8. PMBUS_SLAVE_AUTO_ACK_4_BYTES

Assuming the PMBus module is set to slave mode, this function will configure the PMBSC register. It overwrites the contents of the PMBSC register, with the exception of the address, slave mask, TXPEC and byte count bit fields.

Note

If the user does not specify an option, for example, PMBUS_SLAVE_ENABLE_PEC_PROCESSING, the code will write a 0 (a write) in its bit field.

Returns

None.

29.3.3.21 uint32_t PMBus_getInterruptStatus (uint32_t *base*)

Gets the current PMBus interrupt status.

Parameters

base	is the base address of the PMBus instance used.

This function returns the interrupt status for the PMBus module.

Returns

The current interrupt status, as a bit field of

- **PMBUS_INTSRC_BUS_FREE**
- **PMBUS_INTSRC_CLK_LOW_TIMEOUT**
- **PMBUS_INTSRC_DATA_READY**
- **PMBUS_INTSRC_DATA_REQUEST**
- **PMBUS_INTSRC_SLAVE_ADDR_READY**
- **PMBUS_INTSRC_EOM**
- **PMBUS_INTSRC_ALERT**
- **PMBUS_INTSRC_CONTROL**
- **PMBUS_INTSRC_LOST_ARB**
- **PMBUS_INTSRC_CLK_HIGH_DETECT**

29.3.3.22 uint16_t PMBus_getData (uint32_t *base,* uint16_t ∗ *buffer,* uint32_t *status*)

Read the receive buffer (Slave or Master mode)

Parameters

base	is the base address of the PMBus instance used.
buffer	pointer to the message buffer where the received bytes will be written to
status	the value of the status register (PMBUS_O_PMBSTS)

This function can read up to 4 bytes in the receive buffer.

Note

1. The status register is cleared each time it is read, therefore, it should be read once at the beginning of an interrupt service routine using PMBus_getStatus() and saved to a temporary variable for further processing.
2. The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

Returns the number of byte(s) received by the PMBus in the array pointed to by buffer.

29.3.3.23 void PMBus_putSlaveData (uint32_t *base,* uint16_t ∗ *buffer,* uint16_t *nBytes,* bool *txPEC*)

write to the transmit buffer (Slave mode)

Parameters

base	is the base address of the PMBus instance used.
buffer	pointer to the message buffer where the transmit bytes are stored
nBytes	number of transmit bytes, up to 4
txPEC	1 transmit PEC at end of message, 0 no PEC

This function can write up to 4 bytes in the transmit buffer.

Note

1. The user must check the UNIT_BUSY bit before attempting a transmission.
2. The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

None.

29.3.3.24 void PMBus_ackAddress (uint32_t *base,* uint32_t *address,* uint32_t *status,* uint16_t ∗ *buffer*)

Manual acknowledgement of the slave address

Parameters

base	is the base address of the PMBus instance used.
address	address of the slave
status	the value of the status register (PMBUS_O_PMBSTS)
buffer	pointer to a buffer to store the received data

This function will read the address that was put on the bus, compare it with address passed to this function and then acknowledge on a match (or nack on mismatch). For this function to work, SLAVE_ADDR_READY bit in PBINTM must be enabled. This function checks the SLAVE_ADDR_READY bit in the status register before acknowledging so it would be preferable to use this function in an interrupt handler that responds to the SLAVE_ADDR_READY interrupt.

Note

1. The status register is cleared each time it is read, therefore, it should be read once at the beginning of an interrupt service routine using PMBus_getStatus() and saved to a temporary variable for further processing.
2. The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

None.

29.3.3.25 void PMBus_ackCommand (uint32_t *base,* uint32_t *command,* uint32_t *status,* uint16_t ∗ *buffer*)

Manual acknowledgement of a command

Parameters

base	is the base address of the PMBus instance used.
command	command to manually acknowledge - it can be any of the commands listed in this header file. All commands have the common prefix **PMBUS_CMD**.
status	contents of the status register PMBUS_O_PMBSTS
buffer	pointer to a buffer to store the received data

This function will read the command that was put on the bus, compare it with command passed to this function and then acknowledge on a match (or nack on mismatch). For this function to work, DATA_READY bit in PBINTM must be enabled. This function checks the DATA_READY bit in the status register before acknowledging so it would be preferable to use this function in an interrupt handler that responds to the DATA_READY interrupt.

Note

1. The status register is cleared each time it is read, therefore, it should be read once at the beginning of an interrupt service routine using PMBus_getStatus() and saved to a temporary variable for further processing.
2. The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

None.

29.3.3.26 void PMBus_generateCRCTable (uint16_t ∗ *crcTable*)

Generate a CRC table at run time

Parameters

base	is the base address of the PMBus instance used.
crcTable	points to the CRC8 Table (must be size 256)

This function generates a CRC lookup table to run a CRC on the received data. The table is generated from the polynomial x^8 + x^2 + x^1 + 1 (0x7 - leading 1 is implicit)

Returns

None.

29.3.3.27 bool PMBus_verifyPEC (uint32_t *base,* uint16_t ∗ *buffer,* const uint16_t ∗ *crcTable,* uint16_t *byteCount,* uint16_t *pec*)

Run a CRC on the received data and check against the received PEC to validate the integrity of the data

Parameters

base	is the base address of the PMBus instance used.
buffer	points to the received message crcTable points to the CRC8 Table
byteCount	size of the message, does not include the PEC byte
pec	is the received PEC to check against

This function uses a CRC lookup table to run a CRC on the received data. The table was generated from the polynomial x^8 + x^2 + x^1 + 1 (0x7 - leading 1 is implicit)

Note

The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

true if the calculated CRC is equal to the PEC, **false** otherwise.

29.3.3.28 void PMBus_initMasterMode (uint32_t *base*)

Initializes the PMBus to Master Mode.

Parameters

base	is the base address of the PMBus instance used.

This function sets up the PMBus in master mode.

Returns

None.

29.3.3.29 void PMBus_putMasterData (uint32_t *base,* uint16_t ∗ *buffer,* uint16_t *nBytes*)

write to the transmit buffer (Master mode)

Parameters

base	is the base address of the PMBus instance used.
buffer	pointer to the message buffer where the transmit bytes are stored
nBytes	number of transmit bytes, up to 255

This function can write up to 255 bytes in the transmit buffer.

Note

1. The user must check the UNIT_BUSY bit before attempting the first transmission.
2. The buffer should be at least 4 words long; anything smaller will lead to the possibility of memory overrun when a transaction of 4 bytes happens.

Returns

None.

29.3.3.30 uint32_t PMBus_configModuleClock (uint32_t *base,* uint32_t *moduleFrequency,* uint32_t *sysFrequency*)

Configure the PMBus module clock

Parameters

base	is the base address of the PMBus instance used.
moduleFrequency	desired module frequency; can range from PMBUS_MODULE_FREQ_MIN Hz to PMBUS_MODULE_FREQ_MAX Hz. Please input the frequency in Hz, e.g. 312500 for 312.4 kHz etc.
sysFrequency	Frequency of the system clock (input to PMBus). The values may range anywhere from PMBUS_SYS_FREQ_MIN Hz to PMBUS_SYS_FREQ_MAX Hz. Please input the frequency in Hz, e.g. 100000000 for 100 MHz etc.

The frequency to the PMBus module may not exceed PMBUS_MODULE_FREQ_MAX Hz, the appropriate clock divider is chosen to bring the module clock to the desired frequency - this value is then returned by the function. In the event that the desired bus frequency is unattainable, the clock divider is set to the maximum possible value

Returns

module frequency calculated from the system frequency and clock divider.

References PMBUS_MODULE_FREQ_MAX, PMBUS_MODULE_FREQ_MIN, PMBUS_SYS_FREQ_MAX, and PMBUS_SYS_FREQ_MIN.

29.3.3.31 bool PMBus_configBusClock (uint32_t *base,* **PMBus_ClockMode** *mode,* uint32_t *moduleFrequency*)

Configure the bus clock by overriding the default settings

Parameters

base	is the base address of the PMBus instance used.
mode	is the operating mode for the PMBus, can be ■ Standard Mode ■ Fast Mode ■ Fast Mode Plus
moduleFrequency	desired module frequency; can range from PMBUS_MODULE_FREQ_MIN Hz to PMBUS_MODULE_FREQ_MAX Hz. Please input the frequency in Hz, e.g. 312500 for 312.4 kHz etc.

The frequency to the PMBus module may not exceed PMBUS_MODULE_FREQ_MAX Hz.

Note

1. The module comes out of reset with preprogrammed values that allow it to work in standard mode with a module clock of 10MHz. The module clock is set to 10MHz at power cycle, therefore, the user does not have to call this function unless they wish to change the operating frequency of the module clock from the default 10 MHz.
2. As per PMBus Standard 'PMBus_Specification_Part_I_Rev_1-0_20100906' the maximum bus Speed is 400 kHz hence FASTPLUS mode does not apply to PMBus but rather to the I2C mode of the PMBus module.

Returns

true for successful override, **false** on failure.

References PMBUS_CLOCKMODE_FAST, PMBUS_CLOCKMODE_FASTPLUS, PMBUS_CLOCKMODE_STANDARD, PMBUS_MODULE_FREQ_MAX, and PMBUS_MODULE_FREQ_MIN.

30 SCI Module

Introduction 552
API Functions 552

30.1 SCI Introduction

The SCI driver provides functions which can configure the data word length, baud rate, parity, and stop bits of the SCI communication. It can also be used to perform an autobaud lock, enable or disable loopback mode, enable the FIFO enhancement, configure interrupts, and send and receive data. If FIFO enhancement is enabled, the application must use the provided FIFO read and write functions to guarantee proper execution.

30.2 API Functions

Macros

- #define SCI_INT_RXERR
- #define SCI_INT_RXRDY_BRKDT
- #define SCI_INT_TXRDY
- #define SCI_INT_TXFF
- #define SCI_INT_RXFF
- #define SCI_INT_FE
- #define SCI_INT_OE
- #define SCI_INT_PE
- #define SCI_CONFIG_WLEN_MASK
- #define SCI_CONFIG_WLEN_8
- #define SCI_CONFIG_WLEN_7
- #define SCI_CONFIG_WLEN_6
- #define SCI_CONFIG_WLEN_5
- #define SCI_CONFIG_WLEN_4
- #define SCI_CONFIG_WLEN_3
- #define SCI_CONFIG_WLEN_2
- #define SCI_CONFIG_WLEN_1
- #define SCI_CONFIG_STOP_MASK
- #define SCI_CONFIG_STOP_ONE
- #define SCI_CONFIG_STOP_TWO
- #define SCI_CONFIG_PAR_MASK
- #define SCI_RXSTATUS_WAKE
- #define SCI_RXSTATUS_PARITY
- #define SCI_RXSTATUS_OVERRUN
- #define SCI_RXSTATUS_FRAMING
- #define SCI_RXSTATUS_BREAK
- #define SCI_RXSTATUS_READY
- #define SCI_RXSTATUS_ERROR

Enumerations

- enum SCI_ParityType { SCI_CONFIG_PAR_NONE, SCI_CONFIG_PAR_EVEN, SCI_CONFIG_PAR_ODD }
- enum SCI_TxFIFOLevel { SCI_FIFO_TX0, SCI_FIFO_TX1, SCI_FIFO_TX2, SCI_FIFO_TX3, SCI_FIFO_TX4, SCI_FIFO_TX5, SCI_FIFO_TX6, SCI_FIFO_TX7, SCI_FIFO_TX8, SCI_FIFO_TX9, SCI_FIFO_TX10, SCI_FIFO_TX11, SCI_FIFO_TX12, SCI_FIFO_TX13, SCI_FIFO_TX14, SCI_FIFO_TX15, SCI_FIFO_TX16 }
- enum SCI_RxFIFOLevel { SCI_FIFO_RX0, SCI_FIFO_RX1, SCI_FIFO_RX2, SCI_FIFO_RX3, SCI_FIFO_RX4, SCI_FIFO_RX5, SCI_FIFO_RX6, SCI_FIFO_RX7, SCI_FIFO_RX8, SCI_FIFO_RX9, SCI_FIFO_RX10, SCI_FIFO_RX11, SCI_FIFO_RX12, SCI_FIFO_RX13, SCI_FIFO_RX14, SCI_FIFO_RX15, SCI_FIFO_RX16 }

Functions

- static void SCI_setParityMode (uint32_t base, SCI_ParityType parity)
- static SCI_ParityType SCI_getParityMode (uint32_t base)
- static void SCI_lockAutobaud (uint32_t base)
- static void SCI_setFIFOInterruptLevel (uint32_t base, SCI_TxFIFOLevel txLevel, SCI_RxFIFOLevel rxLevel)
- static void SCI_getFIFOInterruptLevel (uint32_t base, SCI_TxFIFOLevel ∗txLevel, SCI_RxFIFOLevel ∗rxLevel)
- static void SCI_getConfig (uint32_t base, uint32_t lspclkHz, uint32_t ∗baud, uint32_t ∗config)
- static void SCI_enableModule (uint32_t base)
- static void SCI_disableModule (uint32_t base)
- static void SCI_enableFIFO (uint32_t base)
- static void SCI_disableFIFO (uint32_t base)
- static bool SCI_isFIFOEnabled (uint32_t base)
- static void SCI_resetRxFIFO (uint32_t base)
- static void SCI_resetTxFIFO (uint32_t base)
- static void SCI_resetChannels (uint32_t base)
- static bool SCI_isDataAvailableNonFIFO (uint32_t base)
- static bool SCI_isSpaceAvailableNonFIFO (uint32_t base)
- static SCI_TxFIFOLevel SCI_getTxFIFOStatus (uint32_t base)
- static SCI_RxFIFOLevel SCI_getRxFIFOStatus (uint32_t base)
- static bool SCI_isTransmitterBusy (uint32_t base)
- static void SCI_writeCharBlockingFIFO (uint32_t base, uint16_t data)
- static void SCI_writeCharBlockingNonFIFO (uint32_t base, uint16_t data)
- static void SCI_writeCharNonBlocking (uint32_t base, uint16_t data)
- static uint16_t SCI_readCharBlockingFIFO (uint32_t base)
- static uint16_t SCI_readCharBlockingNonFIFO (uint32_t base)
- static uint16_t SCI_readCharNonBlocking (uint32_t base)
- static uint16_t SCI_getRxStatus (uint32_t base)
- static void SCI_performSoftwareReset (uint32_t base)
- static void SCI_enableLoopback (uint32_t base)
- static void SCI_disableLoopback (uint32_t base)
- static bool SCI_getOverflowStatus (uint32_t base)
- static void SCI_clearOverflowStatus (uint32_t base)
- void SCI_setConfig (uint32_t base, uint32_t lspclkHz, uint32_t baud, uint32_t config)
- void SCI_writeCharArray (uint32_t base, const uint16_t ∗const array, uint16_t length)
- void SCI_readCharArray (uint32_t base, uint16_t ∗const array, uint16_t length)

- void SCI_enableInterrupt (uint32_t base, uint32_t intFlags)
- void SCI_disableInterrupt (uint32_t base, uint32_t intFlags)
- uint32_t SCI_getInterruptStatus (uint32_t base)
- void SCI_clearInterruptStatus (uint32_t base, uint32_t intFlags)

30.2.1 Detailed Description

The code for this module is contained in `driverlib/sci.c`, with `driverlib/sci.h` containing the API declarations for use by applications.

30.2.2 Enumeration Type Documentation

30.2.2.1 enum **SCI_ParityType**

Values that can be used with SCI_setParityMode() and SCI_getParityMode() to describe the parity of the SCI communication.

Enumerator

SCI_CONFIG_PAR_NONE No parity.
SCI_CONFIG_PAR_EVEN Even parity.
SCI_CONFIG_PAR_ODD Odd parity.

30.2.2.2 enum **SCI_TxFIFOLevel**

Values that can be passed to SCI_setFIFOInterruptLevel() as the txLevel parameter and returned by SCI_getFIFOInteruptLevel() and SCI_getTxFIFOStatus().

Enumerator

SCI_FIFO_TX0 Transmit interrupt empty.
SCI_FIFO_TX1 Transmit interrupt 1/16 full.
SCI_FIFO_TX2 Transmit interrupt 2/16 full.
SCI_FIFO_TX3 Transmit interrupt 3/16 full.
SCI_FIFO_TX4 Transmit interrupt 4/16 full.
SCI_FIFO_TX5 Transmit interrupt 5/16 full.
SCI_FIFO_TX6 Transmit interrupt 6/16 full.
SCI_FIFO_TX7 Transmit interrupt 7/16 full.
SCI_FIFO_TX8 Transmit interrupt 8/16 full.
SCI_FIFO_TX9 Transmit interrupt 9/16 full.
SCI_FIFO_TX10 Transmit interrupt 10/16 full.
SCI_FIFO_TX11 Transmit interrupt 11/16 full.
SCI_FIFO_TX12 Transmit interrupt 12/16 full.
SCI_FIFO_TX13 Transmit interrupt 13/16 full.
SCI_FIFO_TX14 Transmit interrupt 14/16 full.
SCI_FIFO_TX15 Transmit interrupt 15/16 full.
SCI_FIFO_TX16 Transmit interrupt full.

30.2.2.3 enum **SCI_RxFIFOLevel**

Values that can be passed to SCI_setFIFOInterruptLevel() as the rxLevel parameter and returned by SCI_getFIFOInterruptLevel() and SCI_getRxFIFOStatus().

Enumerator

SCI_FIFO_RX0 Receive interrupt empty.
SCI_FIFO_RX1 Receive interrupt 1/16 full.
SCI_FIFO_RX2 Receive interrupt 2/16 full.
SCI_FIFO_RX3 Receive interrupt 3/16 full.
SCI_FIFO_RX4 Receive interrupt 4/16 full.
SCI_FIFO_RX5 Receive interrupt 5/16 full.
SCI_FIFO_RX6 Receive interrupt 6/16 full.
SCI_FIFO_RX7 Receive interrupt 7/16 full.
SCI_FIFO_RX8 Receive interrupt 8/16 full.
SCI_FIFO_RX9 Receive interrupt 9/16 full.
SCI_FIFO_RX10 Receive interrupt 10/16 full.
SCI_FIFO_RX11 Receive interrupt 11/16 full.
SCI_FIFO_RX12 Receive interrupt 12/16 full.
SCI_FIFO_RX13 Receive interrupt 13/16 full.
SCI_FIFO_RX14 Receive interrupt 14/16 full.
SCI_FIFO_RX15 Receive interrupt 15/16 full.
SCI_FIFO_RX16 Receive interrupt full.

30.2.3 Function Documentation

30.2.3.1 static void SCI_setParityMode (uint32_t *base*, **SCI_ParityType** *parity*) `[inline]`, `[static]`

Sets the type of parity.

Parameters

base	is the base address of the SCI port.
parity	specifies the type of parity to use.

Sets the type of parity to use for transmitting and expect when receiving. The *parity* parameter must be one of the following: **SCI_CONFIG_PAR_NONE**, **SCI_CONFIG_PAR_EVEN**, **SCI_CONFIG_PAR_ODD**.

Returns

None.

References SCI_CONFIG_PAR_MASK.

30.2.3.2 static **SCI_ParityType** SCI_getParityMode (uint32_t *base*) `[inline]`, `[static]`

Gets the type of parity currently being used.

Parameters

base	is the base address of the SCI port.

This function gets the type of parity used for transmitting data and expected when receiving data.

Returns

Returns the current parity settings, specified as one of the following: **SCI_CONFIG_PAR_NONE**, **SCI_CONFIG_PAR_EVEN**, **SCI_CONFIG_PAR_ODD**.

References SCI_CONFIG_PAR_MASK.

30.2.3.3 static void SCI_lockAutobaud (uint32_t *base*) [inline], [static]

Locks Autobaud.

Parameters

base	is the base address of the SCI port.

This function performs an autobaud lock for the SCI.

Returns

None.

30.2.3.4 static void SCI_setFIFOInterruptLevel (uint32_t *base*, **SCI_TxFIFOLevel** *txLevel*, **SCI_RxFIFOLevel** *rxLevel*) [inline], [static]

Sets the FIFO interrupt level at which interrupts are generated.

Parameters

base	is the base address of the SCI port.
txLevel	is the transmit FIFO interrupt level, specified as one of the following: **SCI_FIFO_TX0**, **SCI_FIFO_TX1**, **SCI_FIFO_TX2**, . . . or **SCI_FIFO_TX15**.
rxLevel	is the receive FIFO interrupt level, specified as one of the following **SCI_FIFO_RX0**, **SCI_FIFO_RX1**, **SCI_FIFO_RX2**, ... or **SCI_FIFO_RX15**.

This function sets the FIFO level at which transmit and receive interrupts are generated.

Returns

None.

30.2.3.5 static void SCI_getFIFOInterruptLevel (uint32_t *base*, **SCI_TxFIFOLevel** ∗ *txLevel*, **SCI_RxFIFOLevel** ∗ *rxLevel*) [inline], [static]

Gets the FIFO interrupt level at which interrupts are generated.

Parameters

base	is the base address of the SCI port.
txLevel	is a pointer to storage for the transmit FIFO interrupt level, returned as one of the following: **SCI_FIFO_TX0**, **SCI_FIFO_TX1**, **SCI_FIFO_TX2**, ... or **SCI_FIFO_TX15**.
rxLevel	is a pointer to storage for the receive FIFO interrupt level, returned as one of the following: **SCI_FIFO_RX0**, **SCI_FIFO_RX1**, **SCI_FIFO_RX2**, ... or **SCI_FIFO_RX15**.

This function gets the FIFO level at which transmit and receive interrupts are generated.

Returns

None.

30.2.3.6 static void SCI_getConfig (uint32_t *base,* uint32_t *lspclkHz,* uint32_t ∗ *baud,* uint32_t ∗ *config*) [inline], [static]

Gets the current configuration of a SCI.

Parameters

base	is the base address of the SCI port.
lspclkHz	is the rate of the clock supplied to the SCI module. This is the LSPCLK.
baud	is a pointer to storage for the baud rate.
config	is a pointer to storage for the data format.

The baud rate and data format for the SCI is determined, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an “official” baud rate. The data format returned in *config* is enumerated the same as the *config* parameter of SCI_setConfig().

The peripheral clock is the low speed peripheral clock. This will be the value returned by SysCtl_getLowSeedClock(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtl_getLowSpeedClock()).

Returns

None.

References SCI_CONFIG_PAR_MASK, SCI_CONFIG_STOP_MASK, and SCI_CONFIG_WLEN_MASK.

30.2.3.7 static void SCI_enableModule (uint32_t *base*) [inline], [static]

Enables transmitting and receiving.

Parameters

base	is the base address of the SCI port.

Enables SCI by taking SCI out of the software reset. Sets the TXENA, and RXENA bits which enables transmit and receive.

Returns

None.

Referenced by SCI_setConfig().

30.2.3.8 static void SCI_disableModule (uint32_t *base*) [inline], [static]

Disables transmitting and receiving.

Parameters

base	is the base address of the SCI port.

Clears the SCIEN, TXE, and RXE bits. The user should ensure that all the data has been sent before disable the module during transmission.

Returns

None.

Referenced by SCI_setConfig().

30.2.3.9 static void SCI_enableFIFO (uint32_t *base*) [inline], [static]

Enables the transmit and receive FIFOs.

Parameters

base	is the base address of the SCI port.

This functions enables the transmit and receive FIFOs in the SCI.

Returns

None.

30.2.3.10 static void SCI_disableFIFO (uint32_t *base*) [inline], [static]

Disables the transmit and receive FIFOs.

Parameters

base	is the base address of the SCI port.

This functions disables the transmit and receive FIFOs in the SCI.

Returns

None.

30.2.3.11 static bool SCI_isFIFOEnabled (uint32_t *base*) [inline], [static]

Determines if the FIFO enhancement is enabled.

Parameters

base	is the base address of the SCI port.

This function returns a flag indicating whether or not the FIFO enhancement is enabled.

Returns

Returns **true** if the FIFO enhancement is enabled or **false** if the FIFO enhancement is disabled.

Referenced by SCI_isTransmitterBusy(), SCI_readCharArray(), and SCI_writeCharArray().

30.2.3.12 static void SCI_resetRxFIFO (uint32_t *base*) [inline], [static]

Resets the receive FIFO.

Parameters

base	is the base address of the SCI port.

This functions resets the receive FIFO of the SCI.

Returns

None.

30.2.3.13 static void SCI_resetTxFIFO (uint32_t *base*) `[inline]`, `[static]`

Resets the transmit FIFO.

Parameters

base	is the base address of the SCI port.

This functions resets the transmit FIFO of the SCI.

Returns

None.

30.2.3.14 static void SCI_resetChannels (uint32_t *base*) `[inline]`, `[static]`

Resets the SCI Transmit and Receive Channels

Parameters

base	is the base address of the SCI port.

This functions resets transmit and receive channels in the SCI.

Returns

None.

30.2.3.15 static bool SCI_isDataAvailableNonFIFO (uint32_t *base*) `[inline]`, `[static]`

Determines if there are any characters in the receive buffer when the FIFO enhancement is not enabled.

Parameters

base	is the base address of the SCI port.

This function returns a flag indicating whether or not there is data available in the receive buffer.

Returns

Returns **true** if there is data in the receive buffer or **false** if there is no data in the receive buffer.

Referenced by SCI_readCharArray(), and SCI_readCharBlockingNonFIFO().

30.2.3.16 static bool SCI_isSpaceAvailableNonFIFO (uint32_t *base*) [inline], [static]

Determines if there is any space in the transmit buffer when the FIFO enhancement is not enabled.

Parameters

base	is the base address of the SCI port.

This function returns a flag indicating whether or not there is space available in the transmit buffer when not using the FIFO enhancement.

Returns

Returns **true** if there is space available in the transmit buffer or **false** if there is no space available in the transmit buffer.

Referenced by SCI_writeCharArray(), and SCI_writeCharBlockingNonFIFO().

30.2.3.17 static **SCI_TxFIFOLevel** SCI_getTxFIFOStatus (uint32_t *base*) [inline], [static]

Get the transmit FIFO status

Parameters

base	is the base address of the SCI port.

This functions gets the current number of words in the transmit FIFO.

Returns

Returns the current number of words in the transmit FIFO specified as one of the following: **SCI_FIFO_TX0**, **SCI_FIFO_TX1**, **SCI_FIFO_TX2**, **SCI_FIFO_TX3 SCI_FIFO_TX4**, ..., or **SCI_FIFO_TX16**

Referenced by SCI_writeCharArray(), and SCI_writeCharBlockingFIFO().

30.2.3.18 static **SCI_RxFIFOLevel** SCI_getRxFIFOStatus (uint32_t *base*) [inline], [static]

Get the receive FIFO status

Parameters

base	is the base address of the SCI port.

This functions gets the current number of words in the receive FIFO.

Returns

Returns the current number of words in the receive FIFO specified as one of the following: **SCI_FIFO_RX0**, **SCI_FIFO_RX1**, **SCI_FIFO_RX2**, **SCI_FIFO_RX3 SCI_FIFO_RX4**, ..., or **SCI_FIFO_RX16**

Referenced by SCI_readCharArray(), and SCI_readCharBlockingFIFO().

30.2.3.19 static bool SCI_isTransmitterBusy (uint32_t *base*) [inline], [static]

Determines whether the SCI transmitter is busy or not.

Parameters

base	is the base address of the SCI port.

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware when the FIFO is not enabled. When the FIFO is enabled, this function allows the caller to determine whether there is any data in the FIFO.

Without the FIFO enabled, if **false** is returned, the transmit buffer and shift registers are empty and the transmitter is not busy. With the FIFO enabled, if **false** is returned, the FIFO is empty. This does not necessarily mean that the transmitter is not busy. The empty FIFO does not reflect the status of the transmitter shift register. The FIFO may be empty while the transmitter is still transmitting data.

Returns

Returns **true** if the SCI is transmitting or **false** if transmissions are complete.

References SCI_isFIFOEnabled().

30.2.3.20 static void SCI_writeCharBlockingFIFO (uint32_t *base*, uint16_t *data*) [inline], [static]

Waits to send a character from the specified port when the FIFO enhancement is enabled.

Parameters

base	is the base address of the SCI port.
data	is the character to be transmitted.

Sends the character *data* to the transmit buffer for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning. *data* is a uint16_t but only 8 bits are written to the SCI port. SCI only transmits 8 bit characters.

Returns

None.

References SCI_FIFO_TX15, and SCI_getTxFIFOStatus().

30.2.3.21 static void SCI_writeCharBlockingNonFIFO (uint32_t *base*, uint16_t *data*) [inline], [static]

Waits to send a character from the specified port.

Parameters

base	is the base address of the SCI port.
data	is the character to be transmitted.

Sends the character *data* to the transmit buffer for the specified port. If there is no space available in the transmit buffer, or the transmit FIFO if it is enabled, this function waits until there is space available before returning. *data* is a uint16_t but only 8 bits are written to the SCI port. SCI only transmits 8 bit characters.

Returns

None.

References SCI_isSpaceAvailableNonFIFO().

30.2.3.22 static void SCI_writeCharNonBlocking (uint32_t *base*, uint16_t *data*) [inline], [static]

Sends a character to the specified port.

Parameters

base	is the base address of the SCI port.
data	is the character to be transmitted.

Writes the character *data* to the transmit buffer for the specified port. This function does not block and only writes to the transmit buffer. The user should use SCI_isSpaceAvailableNonFIFO() or SCI_getTxFIFOStatus() to determine if the transmit buffer or FIFO have space available. *data* is a uint16_t but only 8 bits are written to the SCI port. SCI only transmits 8 bit characters.

This function replaces the original SCICharNonBlockingPut() API and performs the same actions. A macro is provided in `sci.h` to map the original API to this API.

Returns

None.

30.2.3.23 static uint16_t SCI_readCharBlockingFIFO (uint32_t *base*) [inline], [static]

Waits for a character from the specified port when the FIFO enhancement is enabled.

Parameters

base	is the base address of the SCI port.

Gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

Returns

Returns the character read from the specified port as *uint16_t*.

References SCI_FIFO_RX0, and SCI_getRxFIFOStatus().

30.2.3.24 static uint16_t SCI_readCharBlockingNonFIFO (uint32_t *base*) [inline], [static]

Waits for a character from the specified port when the FIFO enhancement is not enabled.

Parameters

base	is the base address of the SCI port.

Gets a character from the receive buffer for the specified port. If there is no characters available, this function waits until a character is received before returning.

Returns

Returns the character read from the specified port as *uint16_t*.

References SCI_isDataAvailableNonFIFO().

30.2.3.25 static uint16_t SCI_readCharNonBlocking (uint32_t *base*) [inline], [static]

Receives a character from the specified port.

Parameters

base	is the base address of the SCI port.

Gets a character from the receive buffer for the specified port. This function does not block and only reads the receive buffer. The user should use SCI_isDataAvailableNonFIFO() or SCI_getRxFIFOStatus() to determine if the receive buffer or FIFO have data available.

This function replaces the original SCICharNonBlockingGet() API and performs the same actions. A macro is provided in `sci.h` to map the original API to this API.

Returns

Returns *uin16_t* which is read from the receive buffer.

30.2.3.26 static uint16_t SCI_getRxStatus (uint32_t *base*) [inline], [static]

Gets current receiver status flags.

Parameters

base	is the base address of the SCI port.

This function returns the current receiver status flags. The returned error flags are equivalent to the error bits returned via the previous reading or receiving of a character with the exception that the overrun error is set immediately the overrun occurs rather than when a character is next read.

Returns

Returns a bitwise OR combination of the receiver status flags, **SCI_RXSTATUS_WAKE**, **SCI_RXSTATUS_PARITY**, **SCI_RXSTATUS_OVERRUN**, **SCI_RXSTATUS_FRAMING**, **SCI_RXSTATUS_BREAK**, **SCI_RXSTATUS_READY**, and **SCI_RXSTATUS_ERROR**.

30.2.3.27 static void SCI_performSoftwareReset (uint32_t *base*) [inline], [static]

Performs a software reset of the SCI and Clears all reported receiver status flags.

Parameters

base	is the base address of the SCI port.

This function performs a software reset of the SCI port. It affects the operating flags of the SCI, but it neither affects the configuration bits nor restores the reset values.

Returns

None.

Referenced by SCI_clearInterruptStatus().

30.2.3.28 static void SCI_enableLoopback (uint32_t *base*) [inline], [static]

Enables Loop Back Test Mode

Parameters

base	is the base address of the SCI port.

Enables the loop back test mode where the Tx pin is internally connected to the Rx pin.

Returns

None.

30.2.3.29 static void SCI_disableLoopback (uint32_t *base*) [inline], [static]

Disables Loop Back Test Mode

Parameters

base	is the base address of the SCI port.

Disables the loop back test mode where the Tx pin is no longer internally connected to the Rx pin.

Returns

None.

30.2.3.30 static bool SCI_getOverflowStatus (uint32_t *base*) [inline], [static]

Get the receive FIFO Overflow flag status

Parameters

base	is the base address of the SCI port.

This functions gets the receive FIFO overflow flag status.

Returns

Returns **true** if overflow has occurred, else returned **false** if an overflow hasn't occurred.

30.2.3.31 static void SCI_clearOverflowStatus (uint32_t *base*) [inline], [static]

Clear the receive FIFO Overflow flag status

Parameters

base	is the base address of the SCI port.

This functions clears the receive FIFO overflow flag status.

Returns

None.

30.2.3.32 void SCI_setConfig (uint32_t *base,* uint32_t *lspclkHz,* uint32_t *baud,* uint32_t *config*)

Sets the configuration of a SCI.

Parameters

base	is the base address of the SCI port.
lspclkHz	is the rate of the clock supplied to the SCI module. This is the LSPCLK.
baud	is the desired baud rate.
config	is the data format for the port (number of data bits, number of stop bits, and parity).

This function configures the SCI for operation in the specified data format. The baud rate is provided in the *baud* parameter and the data format in the *config* parameter.

The *config* parameter is the bitwise OR of three values: the number of data bits, the number of stop bits, and the parity. **SCI_CONFIG_WLEN_8**, **SCI_CONFIG_WLEN_7**, **SCI_CONFIG_WLEN_6**, **SCI_CONFIG_WLEN_5**, **SCI_CONFIG_WLEN_4**, **SCI_CONFIG_WLEN_3**, **SCI_CONFIG_WLEN_2**, and **SCI_CONFIG_WLEN_1**. Select from eight to one data bits per byte (respectively). **SCI_CONFIG_STOP_ONE** and **SCI_CONFIG_STOP_TWO** select one or two stop bits (respectively). **SCI_CONFIG_PAR_NONE**, **SCI_CONFIG_PAR_EVEN**, **SCI_CONFIG_PAR_ODD**, select the parity mode (no parity bit, even parity bit, odd parity bit respectively).

The peripheral clock is the low speed peripheral clock. This will be the value returned by SysCtl_getLowSpeedClock(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtl_getLowSpeedClock()).

Returns

None.

References SCI_CONFIG_PAR_MASK, SCI_CONFIG_STOP_MASK, SCI_CONFIG_WLEN_MASK, SCI_disableModule(), and SCI_enableModule().

30.2.3.33 void SCI_writeCharArray (uint32_t *base,* const uint16_t ∗const *array,* uint16_t *length*)

Waits to send an array of characters from the specified port.

Parameters

base	is the base address of the SCI port.
array	is the address of the array of characters to be transmitted. It is pointer to the array of characters to be transmitted.
length	is the length of the array, or number of characters in the array to be transmitted.

Sends the number of characters specified by *length*, starting at the address *array*, out of the transmit buffer for the specified port. If there is no space available in the transmit buffer, or the transmit FIFO if it is enabled, this function waits until there is space available and *length* number of characters are transmitted before returning. *array* is a pointer to uint16_ts but only the least significant 8 bits are written to the SCI port. SCI only transmits 8 bit characters.

Returns

None.

References SCI_FIFO_TX15, SCI_getTxFIFOStatus(), SCI_isFIFOEnabled(), and SCI_isSpaceAvailableNonFIFO().

30.2.3.34 void SCI_readCharArray (uint32_t *base,* uint16_t ∗const *array,* uint16_t *length*)

Waits to receive an array of characters from the specified port.

Parameters

base	is the base address of the SCI port.
array	is the address of the array of characters to be received. It is a pointer to the array of characters to be received.
length	is the length of the array, or number of characters in the array to be received.

Receives an array of characters from the receive buffer for the specified port, and stores them as an array of characters starting at address *array*. This function waits until the *length* number of characters are received before returning.

Returns

None.

References SCI_FIFO_RX0, SCI_getRxFIFOStatus(), SCI_isDataAvailableNonFIFO(), and SCI_isFIFOEnabled().

30.2.3.35 void SCI_enableInterrupt (uint32_t *base,* uint32_t *intFlags*)

Enables individual SCI interrupt sources.

Parameters

base	is the base address of the SCI port.
intFlags	is the bit mask of the interrupt sources to be enabled.

Enables the indicated SCI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *intFlags* parameter is the bitwise OR of any of the following:

- **SCI_INT_RXERR** - RXERR Interrupt
- **SCI_INT_RXRDY_BRKDT** - RXRDY/BRKDT Interrupt

- **SCI_INT_TXRDY** - TXRDY Interrupt
- **SCI_INT_TXFF** - TX FIFO Level Interrupt
- **SCI_INT_RXFF** - RX FIFO Level Interrupt
- **SCI_INT_FE** - Frame Error
- **SCI_INT_OE** - Overrun Error
- **SCI_INT_PE** - Parity Error

Returns

None.

References SCI_INT_RXERR, SCI_INT_RXFF, SCI_INT_RXRDY_BRKDT, SCI_INT_TXFF, and SCI_INT_TXRDY.

30.2.3.36 void SCI_disableInterrupt (uint32_t *base*, uint32_t *intFlags*)

Disables individual SCI interrupt sources.

Parameters

base	is the base address of the SCI port.
intFlags	is the bit mask of the interrupt sources to be disabled.

Disables the indicated SCI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *intFlags* parameter has the same definition as the *intFlags* parameter to SCI_enableInterrupt().

Returns

None.

References SCI_INT_RXERR, SCI_INT_RXFF, SCI_INT_RXRDY_BRKDT, SCI_INT_TXFF, and SCI_INT_TXRDY.

30.2.3.37 uint32_t SCI_getInterruptStatus (uint32_t *base*)

Gets the current interrupt status.

Parameters

base	is the base address of the SCI port.

Returns

Returns the current interrupt status, enumerated as a bit field of values described in SCI_enableInterrupt().

References SCI_INT_FE, SCI_INT_OE, SCI_INT_PE, SCI_INT_RXERR, SCI_INT_RXFF, SCI_INT_RXRDY_BRKDT, SCI_INT_TXFF, and SCI_INT_TXRDY.

30.2.3.38 void SCI_clearInterruptStatus (uint32_t *base,* uint32_t *intFlags*)

Clears SCI interrupt sources.

Parameters

base	is the base address of the SCI port.
intFlags	is a bit mask of the interrupt sources to be cleared.

The specified SCI interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being recognized again immediately upon exit.

The *intFlags* parameter has the same definition as the *intFlags* parameter to SCI_enableInterrupt().

Returns

None.

References SCI_INT_FE, SCI_INT_OE, SCI_INT_PE, SCI_INT_RXERR, SCI_INT_RXFF, SCI_INT_RXRDY_BRKDT, SCI_INT_TXFF, and SCI_performSoftwareReset().

31 SDFM Module

Introduction 573
API Functions 573

31.1 SDFM Introduction

The Sigma-Delta Filter Module (SDFM) API provides a set of functions for configuring and using the SDFM module. The functions provided allow the user to setup and configure the Input data type to SDFM filters, the Primary (data) and Secondary (comparator) filters, Data FIFO, the PWM - SDFM sync signals, comparator threshold values and interrupt sources. Functions are also provided to read the filter data and the status of the SDFM module components.

Note that the Secondary (comparator) Filter configuration APIs have the "Comp" key word embedded to represent access to the Comparator sub-module. For example the function SDFM_setComparatorFilterType() sets the comparator filter type while SDFM_setFilterType() sets the primary filter type.

APIs providing higher level abstraction are also available in the sdfm.c source file. These APIs can be used to configure the Comparator, Data Filter and the Data filter FIFO.

31.2 API Functions

Macros

- #define SDFM_GET_LOW_THRESHOLD(C)
- #define SDFM_GET_HIGH_THRESHOLD(C)
- #define SDFM_SET_OSR(X)
- #define SDFM_SHIFT_VALUE(X)
- #define SDFM_THRESHOLD(H, L)
- #define SDFM_SET_FIFO_LEVEL(X)
- #define SDFM_SET_ZERO_CROSS_THRESH_VALUE(X)
- #define SDFM_FILTER_DISABLE
- #define SDFM_MODULATOR_FAILURE_INTERRUPT
- #define SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT
- #define SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT
- #define SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT
- #define SDFM_FIFO_INTERRUPT
- #define SDFM_FIFO_OVERFLOW_INTERRUPT
- #define SDFM_MASTER_INTERRUPT_FLAG
- #define SDFM_FILTER_1_HIGH_THRESHOLD_FLAG
- #define SDFM_FILTER_1_LOW_THRESHOLD_FLAG
- #define SDFM_FILTER_2_HIGH_THRESHOLD_FLAG
- #define SDFM_FILTER_2_LOW_THRESHOLD_FLAG
- #define SDFM_FILTER_3_HIGH_THRESHOLD_FLAG
- #define SDFM_FILTER_3_LOW_THRESHOLD_FLAG
- #define SDFM_FILTER_4_HIGH_THRESHOLD_FLAG
- #define SDFM_FILTER_4_LOW_THRESHOLD_FLAG
- #define SDFM_FILTER_1_MOD_FAILED_FLAG
- #define SDFM_FILTER_2_MOD_FAILED_FLAG

- #define SDFM_FILTER_3_MOD_FAILED_FLAG
- #define SDFM_FILTER_4_MOD_FAILED_FLAG
- #define SDFM_FILTER_1_NEW_DATA_FLAG
- #define SDFM_FILTER_2_NEW_DATA_FLAG
- #define SDFM_FILTER_3_NEW_DATA_FLAG
- #define SDFM_FILTER_4_NEW_DATA_FLAG
- #define SDFM_FILTER_1_FIFO_OVERFLOW_FLAG
- #define SDFM_FILTER_2_FIFO_OVERFLOW_FLAG
- #define SDFM_FILTER_3_FIFO_OVERFLOW_FLAG
- #define SDFM_FILTER_4_FIFO_OVERFLOW_FLAG
- #define SDFM_FILTER_1_FIFO_INTERRUPT_FLAG
- #define SDFM_FILTER_2_FIFO_INTERRUPT_FLAG
- #define SDFM_FILTER_3_FIFO_INTERRUPT_FLAG
- #define SDFM_FILTER_4_FIFO_INTERRUPT_FLAG

Enumerations

- enum SDFM_OutputThresholdStatus { SDFM_OUTPUT_WITHIN_THRESHOLD, SDFM_OUTPUT_ABOVE_THRESHOLD, SDFM_OUTPUT_BELOW_THRESHOLD }
- enum SDFM_FilterNumber { SDFM_FILTER_1, SDFM_FILTER_2, SDFM_FILTER_3, SDFM_FILTER_4 }
- enum SDFM_FilterType { SDFM_FILTER_SINC_FAST, SDFM_FILTER_SINC_1, SDFM_FILTER_SINC_2, SDFM_FILTER_SINC_3 }
- enum SDFM_ModulatorClockMode { SDFM_MODULATOR_CLK_EQUAL_DATA_RATE, SDFM_MODULATOR_CLK_HALF_DATA_RATE, SDFM_MODULATOR_CLK_OFF, SDFM_MODULATOR_CLK_DOUBLE_DATA_RATE }
- enum SDFM_OutputDataFormat { SDFM_DATA_FORMAT_16_BIT, SDFM_DATA_FORMAT_32_BIT }
- enum SDFM_DataReadyInterruptSource { SDFM_DATA_READY_SOURCE_DIRECT, SDFM_DATA_READY_SOURCE_FIFO }
- enum SDFM_PWMSyncSource { SDFM_SYNC_PWM1_SOCA, SDFM_SYNC_PWM1_SOCB, SDFM_SYNC_PWM2_SOCA, SDFM_SYNC_PWM2_SOCB, SDFM_SYNC_PWM3_SOCA, SDFM_SYNC_PWM3_SOCB, SDFM_SYNC_PWM4_SOCA, SDFM_SYNC_PWM4_SOCB, SDFM_SYNC_PWM5_SOCA, SDFM_SYNC_PWM5_SOCB, SDFM_SYNC_PWM6_SOCA, SDFM_SYNC_PWM6_SOCB, SDFM_SYNC_PWM7_SOCA, SDFM_SYNC_PWM7_SOCB, SDFM_SYNC_PWM8_SOCA, SDFM_SYNC_PWM8_SOCB }
- enum SDFM_FIFOClearSyncMode { SDFM_FIFO_NOT_CLEARED_ON_SYNC, SDFM_FIFO_CLEARED_ON_SYNC }
- enum SDFM_WaitForSyncClearMode { SDFM_MANUAL_CLEAR_WAIT_FOR_SYNC, SDFM_AUTO_CLEAR_WAIT_FOR_SYNC }

Functions

- static void SDFM_enableExternalReset (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_disableExternalReset (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_enableFilter (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_disableFilter (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_enableFIFOBuffer (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_disableFIFOBuffer (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getZeroCrossTripStatus (uint32_t base, SDFM_FilterNumber filterNumber)

- static void SDFM_clearZeroCrossTripStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_enableComparator (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_disableComparator (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_setFilterType (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_FilterType filterType)
- static void SDFM_setFilterOverSamplingRatio (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t overSamplingRatio)
- static void SDFM_setupModulatorClock (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_ModulatorClockMode clockMode)
- static void SDFM_setOutputDataFormat (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_OutputDataFormat dataFormat)
- static void SDFM_setDataShiftValue (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t shiftValue)
- static void SDFM_setCompFilterHighThreshold (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t highThreshold)
- static void SDFM_setCompFilterLowThreshold (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t lowThreshold)
- static void SDFM_setCompFilterZeroCrossThreshold (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t zeroCrossThreshold)
- static void SDFM_enableZeroCrossEdgeDetect (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_disableZeroCrossEdgeDetect (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_enableInterrupt (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t intFlags)
- static void SDFM_disableInterrupt (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t intFlags)
- static void SDFM_setComparatorFilterType (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_FilterType filterType)
- static void SDFM_setCompFilterOverSamplingRatio (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t overSamplingRatio)
- static uint32_t SDFM_getFilterData (uint32_t base, SDFM_FilterNumber filterNumber)
- static SDFM_OutputThresholdStatus SDFM_getThresholdStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getModulatorStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getNewFilterDataStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getFIFOOverflowStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getFIFOISRStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static bool SDFM_getIsrStatus (uint32_t base)
- static void SDFM_clearInterruptFlag (uint32_t base, uint32_t flag)
- static void SDFM_enableMasterInterrupt (uint32_t base)
- static void SDFM_disableMasterInterrupt (uint32_t base)
- static void SDFM_enableMasterFilter (uint32_t base)
- static void SDFM_disableMasterFilter (uint32_t base)
- static uint16_t SDFM_getFIFODataCount (uint32_t base, SDFM_FilterNumber filterNumber)
- static uint16_t SDFM_getComparatorSincData (uint32_t base, SDFM_FilterNumber filterNumber)
- static uint32_t SDFM_getFIFOData (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_setFIFOInterruptLevel (uint32_t base, SDFM_FilterNumber filterNumber, uint16_t fifoLevel)
- static void SDFM_setDataReadyInterruptSource (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_DataReadyInterruptSource dataReadySource)
- static bool SDFM_getWaitForSyncStatus (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_clearWaitForSyncFlag (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_enableWaitForSync (uint32_t base, SDFM_FilterNumber filterNumber)

- static void SDFM_disableWaitForSync (uint32_t base, SDFM_FilterNumber filterNumber)
- static void SDFM_setPWMSyncSource (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_PWMSyncSource syncSource)
- static void SDFM_setFIFOClearOnSyncMode (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_FIFOClearSyncMode fifoClearSyncMode)
- static void SDFM_setWaitForSyncClearMode (uint32_t base, SDFM_FilterNumber filterNumber, SDFM_WaitForSyncClearMode syncClearMode)
- void SDFM_configComparator (uint32_t base, uint16_t config1, uint32_t config2, uint16_t config3)
- void SDFM_configDataFilter (uint32_t base, uint16_t config1, uint16_t config2)
- void SDFM_configZeroCrossComparator (uint32_t base, uint16_t config1, uint16_t config2)
- void SDFM_configDataFilterFIFO (uint32_t base, uint16_t config1, uint16_t config2)

31.2.1 Detailed Description

The code for this module is contained in `driverlib/sdfm.c`, with `driverlib/sdfm.h` containing the API declarations for use by applications.

31.2.2 Macro Definition Documentation

31.2.2.1 #define SDFM_GET_LOW_THRESHOLD(*C*)

Macro to get the low threshold

Referenced by SDFM_configComparator().

31.2.2.2 #define SDFM_GET_HIGH_THRESHOLD(*C*)

Macro to get the high threshold

Referenced by SDFM_configComparator().

31.2.2.3 #define SDFM_SET_OSR(*X*)

Macro to convert comparator over sampling ratio to acceptable bit location

31.2.2.4 #define SDFM_SHIFT_VALUE(*X*)

Macro to convert the data shift bit values to acceptable bit location

31.2.2.5 #define SDFM_THRESHOLD(*H*, *L*)

Macro to combine high threshold and low threshold values

31.2.2.6 #define SDFM_SET_FIFO_LEVEL(*X*)

Macro to set the FIFO level to acceptable bit location

31.2.2.7 #define SDFM_SET_ZERO_CROSS_THRESH_VALUE(*X*)

Macro to set and enable the zero cross threshold value.

31.2.2.8 #define SDFM_FILTER_DISABLE

Macros to enable or disable filter.

31.2.2.9 #define SDFM_MODULATOR_FAILURE_INTERRUPT

Interrupt is generated if Modulator fails.

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.10 #define SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT

Interrupt on Comparator low-level threshold.

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.11 #define SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT

Interrupt on Comparator high-level threshold.

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.12 #define SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT

Interrupt on Acknowledge flag

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.13 #define SDFM_FIFO_INTERRUPT

Interrupt on FIFO level

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.14 #define SDFM_FIFO_OVERFLOW_INTERRUPT

Interrupt on FIFO overflow

Referenced by SDFM_disableInterrupt(), and SDFM_enableInterrupt().

31.2.2.15 #define SDFM_MASTER_INTERRUPT_FLAG

Master interrupt flag

31.2.2.16 #define SDFM_FILTER_1_HIGH_THRESHOLD_FLAG

Filter 1 high -level threshold flag

31.2.2.17 #define SDFM_FILTER_1_LOW_THRESHOLD_FLAG

Filter 1 low -level threshold flag

31.2.2.18 #define SDFM_FILTER_2_HIGH_THRESHOLD_FLAG

Filter 2 high -level threshold flag

31.2.2.19 #define SDFM_FILTER_2_LOW_THRESHOLD_FLAG

Filter 2 low -level threshold flag

31.2.2.20 #define SDFM_FILTER_3_HIGH_THRESHOLD_FLAG

Filter 3 high -level threshold flag

31.2.2.21 #define SDFM_FILTER_3_LOW_THRESHOLD_FLAG

Filter 3 low -level threshold flag

31.2.2.22 #define SDFM_FILTER_4_HIGH_THRESHOLD_FLAG

Filter 4 high -level threshold flag

31.2.2.23 #define SDFM_FILTER_4_LOW_THRESHOLD_FLAG

Filter 4 low -level threshold flag

31.2.2.24 #define SDFM_FILTER_1_MOD_FAILED_FLAG

Filter 1 modulator failed flag

31.2.2.25 #define SDFM_FILTER_2_MOD_FAILED_FLAG

Filter 2 modulator failed flag

31.2.2.26 #define SDFM_FILTER_3_MOD_FAILED_FLAG

Filter 3 modulator failed flag

31.2.2.27 #define SDFM_FILTER_4_MOD_FAILED_FLAG

Filter 4 modulator failed flag

31.2.2.28 #define SDFM_FILTER_1_NEW_DATA_FLAG

Filter 1 new data flag

31.2.2.29 #define SDFM_FILTER_2_NEW_DATA_FLAG

Filter 2 new data flag

31.2.2.30 #define SDFM_FILTER_3_NEW_DATA_FLAG

Filter 3 new data flag

31.2.2.31 #define SDFM_FILTER_4_NEW_DATA_FLAG

Filter 4 new data flag

31.2.2.32 #define SDFM_FILTER_1_FIFO_OVERFLOW_FLAG

Filter 1 FIFO overflow flag

31.2.2.33 #define SDFM_FILTER_2_FIFO_OVERFLOW_FLAG

Filter 2 FIFO overflow flag

31.2.2.34 #define SDFM_FILTER_3_FIFO_OVERFLOW_FLAG

Filter 3 FIFO overflow flag

31.2.2.35 #define SDFM_FILTER_4_FIFO_OVERFLOW_FLAG

Filter 4 FIFO overflow flag

31.2.2.36 #define SDFM_FILTER_1_FIFO_INTERRUPT_FLAG

Filter 1 FIFO overflow flag

31.2.2.37 #define SDFM_FILTER_2_FIFO_INTERRUPT_FLAG

Filter 2 FIFO overflow flag

31.2.2.38 #define SDFM_FILTER_3_FIFO_INTERRUPT_FLAG

Filter 3 FIFO overflow flag

31.2.2.39 #define SDFM_FILTER_4_FIFO_INTERRUPT_FLAG

Filter 4 FIFO overflow flag

31.2.3 Enumeration Type Documentation

31.2.3.1 enum **SDFM_OutputThresholdStatus**

Values that can be returned from SDFM_getThresholdStatus()

Enumerator

SDFM_OUTPUT_WITHIN_THRESHOLD SDFM output is within threshold.
SDFM_OUTPUT_ABOVE_THRESHOLD SDFM output is above threshold.
SDFM_OUTPUT_BELOW_THRESHOLD SDFM output is below threshold.

31.2.3.2 enum **SDFM_FilterNumber**

Values that can be passed to all functions as the *filterNumber* parameter.

Enumerator

SDFM_FILTER_1 Digital filter 1.
SDFM_FILTER_2 Digital filter 2.
SDFM_FILTER_3 Digital filter 3.
SDFM_FILTER_4 Digital filter 4.

31.2.3.3 enum **SDFM_FilterType**

Values that can be passed to SDFM_setFilterType(), SDFM_setComparatorFilterType() as the *filterType* parameter.

Enumerator

SDFM_FILTER_SINC_FAST Digital filter with SincFast structure.

SDFM_FILTER_SINC_1 Digital filter with Sinc1 structure.

SDFM_FILTER_SINC_2 Digital filter with Sinc3 structure.

SDFM_FILTER_SINC_3 Digital filter with Sinc4 structure.

31.2.3.4 enum **SDFM_ModulatorClockMode**

Values that can be passed to SDFM_setupModulatorClock(), as the *clockMode* parameter.

Enumerator

SDFM_MODULATOR_CLK_EQUAL_DATA_RATE Modulator clock is identical to the data rate.

SDFM_MODULATOR_CLK_HALF_DATA_RATE Modulator clock is half the data rate.

SDFM_MODULATOR_CLK_OFF Modulator clock is off. Data is Manchester coded.

SDFM_MODULATOR_CLK_DOUBLE_DATA_RATE Modulator clock is double the data rate.

31.2.3.5 enum **SDFM_OutputDataFormat**

Values that can be passed to SDFM_setOutputDataFormat(), as the *dataFormat* parameter.

Enumerator

SDFM_DATA_FORMAT_16_BIT Filter output is in 16 bits 2's complement format.

SDFM_DATA_FORMAT_32_BIT Filter output is in 32 bits 2's complement format.

31.2.3.6 enum **SDFM_DataReadyInterruptSource**

Values that can be passed to SDFM_setDataReadyInterruptSource(), as the *dataReadySource* parameter.

Enumerator

SDFM_DATA_READY_SOURCE_DIRECT Data ready interrupt source is direct (non -FIFO).

SDFM_DATA_READY_SOURCE_FIFO Data ready interrupt source is FIFO.

31.2.3.7 enum **SDFM_PWMSyncSource**

Values that can be passed to SDFM_setPWMSyncSource(), as the *syncSource* parameter.

Enumerator

SDFM_SYNC_PWM1_SOCA SDFM sync source is PWM1 SOCA.

SDFM_SYNC_PWM1_SOCB SDFM sync source is PWM1 SOCB.
SDFM_SYNC_PWM2_SOCA SDFM sync source is PWM2 SOCA.
SDFM_SYNC_PWM2_SOCB SDFM sync source is PWM2 SOCB.
SDFM_SYNC_PWM3_SOCA SDFM sync source is PWM3 SOCA.
SDFM_SYNC_PWM3_SOCB SDFM sync source is PWM3 SOCB.
SDFM_SYNC_PWM4_SOCA SDFM sync source is PWM4 SOCA.
SDFM_SYNC_PWM4_SOCB SDFM sync source is PWM4 SOCB.
SDFM_SYNC_PWM5_SOCA SDFM sync source is PWM5 SOCA.
SDFM_SYNC_PWM5_SOCB SDFM sync source is PWM5 SOCB.
SDFM_SYNC_PWM6_SOCA SDFM sync source is PWM6 SOCA.
SDFM_SYNC_PWM6_SOCB SDFM sync source is PWM6 SOCB.
SDFM_SYNC_PWM7_SOCA SDFM sync source is PWM7 SOCA.
SDFM_SYNC_PWM7_SOCB SDFM sync source is PWM7 SOCB.
SDFM_SYNC_PWM8_SOCA SDFM sync source is PWM8 SOCA.
SDFM_SYNC_PWM8_SOCB SDFM sync source is PWM8 SOCB.

31.2.3.8 enum **SDFM_FIFOClearSyncMode**

Values that can be passed to SDFM_setFIFOClearOnSyncMode(), as the *fifoClearSyncMode* parameter.

Enumerator

SDFM_FIFO_NOT_CLEARED_ON_SYNC SDFM FIFO buffer is not cleared on Sync signal.

SDFM_FIFO_CLEARED_ON_SYNC SDFM FIFO buffer is cleared on Sync signal.

31.2.3.9 enum **SDFM_WaitForSyncClearMode**

Values that can be passed to SDFM_setWaitForSyncClearMode(), as the *syncClearMode* parameter.

Enumerator

SDFM_MANUAL_CLEAR_WAIT_FOR_SYNC Wait for sync cleared using software.
SDFM_AUTO_CLEAR_WAIT_FOR_SYNC Wait for sync cleared automatically.

31.2.4 Function Documentation

31.2.4.1 static void SDFM_enableExternalReset (uint32_t *base,* **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable external reset

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables data filter to be reset by an external source (PWM compare output).

Returns
None.

31.2.4.2 static void SDFM_disableExternalReset (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable external reset

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function disables data filter from being reset by an external source (PWM compare output).

Returns
None.

31.2.4.3 static void SDFM_enableFilter (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable filter

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables the filter specified by the *filterNumber* variable.

Returns
None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.4 static void SDFM_disableFilter (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable filter

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function disables the filter specified by the *filterNumber* variable.

Returns
None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.5 static void SDFM_enableFIFOBuffer (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable FIFO buffer

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables the filter FIFO buffer specified by the *filterNumber* variable.

Returns
None.

Referenced by SDFM_configDataFilterFIFO().

31.2.4.6 static void SDFM_disableFIFOBuffer (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable FIFO buffer

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function disables the filter FIFO buffer specified by the *filterNumber* variable.

Returns
None.

31.2.4.7 static bool SDFM_getZeroCrossTripStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Return the Zero Cross Trip status

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the Zero Cross Trip status for the filter specified by filterNumber variable.

Returns

true if Comparator filter output >= High-level threshold (Z) **false** if Comparator filter output < High-level threshold (Z)

31.2.4.8 static void SDFM_clearZeroCrossTripStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Clear the Zero Cross Trip status

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function clears the Zero Cross Trip status for the filter specified by filterNumber variable.

Returns

None.

31.2.4.9 static void SDFM_enableComparator (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable Comparator.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables the Comparator for the selected filter.

Returns

None.

31.2.4.10 static void SDFM_disableComparator (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable Comparator.

Parameters

base	is the base address of the SDFM module

filterNumber	is the filter number.

This function disables the Comparator for the selected filter.

Returns
None.

31.2.4.11 static void SDFM_setFilterType (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_FilterType** *filterType*) [inline], [static]

Set filter type.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
filterType	is the filter type or structure.

This function sets the filter type or structure to be used as specified by filterType for the selected filter number as specified by filterNumber.

Returns
None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.12 static void SDFM_setFilterOverSamplingRatio (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *overSamplingRatio*) [inline], [static]

Set data filter over sampling ratio.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
overSamplingRatio	is the data filter over sampling ratio.

This function sets the filter oversampling ratio for the filter specified by the filterNumber variable.Valid values for the variable overSamplingRatio are 0 to 255 inclusive. The actual oversampling ratio will be this value plus one.

Returns
None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.13 static void SDFM_setupModulatorClock (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_ModulatorClockMode** *clockMode*) [inline], [static]

Set modulator clock mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
clockMode	is the modulator clock mode.

This function sets the modulator clock mode specified by clockMode for the filter specified by filterNumber.

Returns

None.

31.2.4.14 static void SDFM_setOutputDataFormat (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_OutputDataFormat** *dataFormat*) [inline], [static]

Set the output data format

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
dataFormat	is the output data format.

This function sets the output data format for the filter specified by filterNumber.

Returns

None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.15 static void SDFM_setDataShiftValue (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *shiftValue*) [inline], [static]

Set data shift value.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
shiftValue	is the data shift value.

This function sets the shift value for the 16 bit 2's complement data format. The valid maximum value for shiftValue is 31.

Note: Use this function with 16 bit 2's complement data format only.

Returns
None.

Referenced by SDFM_configDataFilter(), and SDFM_configDataFilterFIFO().

31.2.4.16 static void SDFM_setCompFilterHighThreshold (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *highThreshold*) [inline], [static]

Set Filter output high-level threshold.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
highThreshold	is the high-level threshold.

This function sets the unsigned high-level threshold value for the Comparator filter output. If the output value of the filter exceeds highThreshold and interrupt generation is enabled, an interrupt will be issued.

Returns

None.

Referenced by SDFM_configComparator().

31.2.4.17 static void SDFM_setCompFilterLowThreshold (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *lowThreshold*) [inline], [static]

Set Filter output low-level threshold.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
lowThreshold	is the low-level threshold.

This function sets the unsigned low-level threshold value for the Comparator filter output. If the output value of the filter gets below lowThreshold and interrupt generation is enabled, an interrupt will be issued.

Returns

None.

Referenced by SDFM_configComparator().

31.2.4.18 static void SDFM_setCompFilterZeroCrossThreshold (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *zeroCrossThreshold*) [inline], [static]

Set Filter output zero-cross threshold.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
zero-CrossThreshold	is the zero-cross threshold.

This function sets the unsigned zero-cross threshold value for the Comparator filter output.

Returns

None.

Referenced by SDFM_configComparator(), and SDFM_configZeroCrossComparator().

31.2.4.19 static void SDFM_enableZeroCrossEdgeDetect (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable zero-cross Edge detect mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables Zero Cross Edge detection.

Returns

None.

Referenced by SDFM_configComparator().

31.2.4.20 static void SDFM_disableZeroCrossEdgeDetect (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable zero-cross Edge detect mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function disables Zero Cross Edge detection.

Returns

None.

31.2.4.21 static void SDFM_enableInterrupt (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *intFlags*) [inline], [static]

Enable SDFM interrupts.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
intFlags	is the interrupt source.

This function enables the low threshold , high threshold or modulator failure interrupt as determined by intFlags for the filter specified by filterNumber. Valid values for intFlags are: SDFM_MODULATOR_FAILURE_INTERRUPT , SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT, SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT, SDFM_FIFO_INTERRUPT, SDFM_FIFO_OVERFLOW_INTERRUPT, SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT

Returns

None.

References SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT, SDFM_FIFO_INTERRUPT, SDFM_FIFO_OVERFLOW_INTERRUPT, SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT, SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT, and SDFM_MODULATOR_FAILURE_INTERRUPT.

31.2.4.22 static void SDFM_disableInterrupt (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *intFlags*) [inline], [static]

Disable SDFM interrupts.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
intFlags	is the interrupt source.

This function disables the low threshold , high threshold or modulator failure interrupt as determined by intFlags for the filter specified by filterNumber. Valid values for intFlags are: SDFM_MODULATOR_FAILURE_INTERRUPT , SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT, SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT, SDFM_FIFO_INTERRUPT, SDFM_FIFO_OVERFLOW_INTERRUPT, SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT

Returns

None.

References SDFM_DATA_FILTER_ACKNOWLEDGE_INTERRUPT, SDFM_FIFO_INTERRUPT, SDFM_FIFO_OVERFLOW_INTERRUPT, SDFM_HIGH_LEVEL_THRESHOLD_INTERRUPT, SDFM_LOW_LEVEL_THRESHOLD_INTERRUPT, and SDFM_MODULATOR_FAILURE_INTERRUPT.

31.2.4.23 static void SDFM_setComparatorFilterType (uint32_t *base,* **SDFM_FilterNumber** *filterNumber,* **SDFM_FilterType** *filterType*) [inline], [static]

Set the comparator filter type.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
filterType	is the comparator filter type or structure.

This function sets the Comparator filter type or structure to be used as specified by filterType for the selected filter number as specified by filterNumber.

Returns

None.

Referenced by SDFM_configComparator(), and SDFM_configZeroCrossComparator().

31.2.4.24 static void SDFM_setCompFilterOverSamplingRatio (uint32_t *base,* **SDFM_FilterNumber** *filterNumber,* uint16_t *overSamplingRatio*) [inline], [static]

Set Comparator filter over sampling ratio.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
overSamplingRatio	is the comparator filter over sampling ration.

This function sets the comparator filter oversampling ratio for the filter specified by the filterNumber.Valid values for the variable overSamplingRatio are 0 to 31 inclusive. The actual oversampling ratio will be this value plus one.

Returns

None.

Referenced by SDFM_configComparator(), and SDFM_configZeroCrossComparator().

31.2.4.25 static uint32_t SDFM_getFilterData (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Get the filter data output.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the latest data filter output. Depending on the filter data output format selected, the valid value will be the lower 16 bits or the whole 32 bits of the returned value.

Returns

Returns the latest data filter output.

31.2.4.26 static **SDFM_OutputThresholdStatus** SDFM_getThresholdStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Get the Comparator threshold status.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the Comparator output threshold status for the given filterNumber.

Returns

Returns the following status flags.

- **SDFM_OUTPUT_WITHIN_THRESHOLD** if the output is within the specified threshold.
- **SDFM_OUTPUT_ABOVE_THRESHOLD** if the output is above the high threshold
- **SDFM_OUTPUT_BELOW_THRESHOLD** if the output is below the low threshold.

31.2.4.27 static bool SDFM_getModulatorStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Get the Modulator status.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the Modulator status.

Returns

Returns true if the Modulator is operating normally Returns false if the Modulator has failed

31.2.4.28 static bool SDFM_getNewFilterDataStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Check if new Filter data is available.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns new filter data status.

Returns

Returns **true** if new filter data is available Returns **false** if no new filter data is available

31.2.4.29 static bool SDFM_getFIFOOverflowStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Check if FIFO buffer is overflowed.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the status of the FIFO buffer overflow for the given filter value.

Returns

Returns **true** if FIFO buffer is overflowed Returns **false** if FIFO buffer is not overflowed

31.2.4.30 static bool SDFM_getFIFOISRStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Check FIFO buffer interrupt status.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the status of the FIFO buffer interrupt for the given filter.

Returns

Returns **true** if FIFO buffer interrupt has occurred. Returns **false** if FIFO buffer interrupt has not occurred.

31.2.4.31 static bool SDFM_getIsrStatus (uint32_t *base*) [inline], [static]

Get pending interrupt.

Parameters

base	is the base address of the SDFM module

This function returns any pending interrupt status.

Returns

Returns **true** if there is a pending interrupt. Returns **false** if no interrupt is pending.

31.2.4.32 static void SDFM_clearInterruptFlag (uint32_t *base*, uint32_t *flag*) [inline], [static]

Clear pending flags.

Parameters

base	is the base address of the SDFM module
flag	is the SDFM status

This function clears the specified pending interrupt flag. Valid values are SDFM_MASTER_INTERRUPT_FLAG, SDFM_FILTER_1_NEW_DATA_FLAG, SDFM_FILTER_2_NEW_DATA_FLAG, SDFM_FILTER_3_NEW_DATA_FLAG, SDFM_FILTER_4_NEW_DATA_FLAG, SDFM_FILTER_1_MOD_FAILED_FLAG, SDFM_FILTER_2_MOD_FAILED_FLAG, SDFM_FILTER_3_MOD_FAILED_FLAG, SDFM_FILTER_4_MOD_FAILED_FLAG, SDFM_FILTER_1_HIGH_THRESHOLD_FLAG, SDFM_FILTER_1_LOW_THRESHOLD_FLAG, SDFM_FILTER_2_HIGH_THRESHOLD_FLAG, SDFM_FILTER_2_LOW_THRESHOLD_FLAG, SDFM_FILTER_3_HIGH_THRESHOLD_FLAG, SDFM_FILTER_3_LOW_THRESHOLD_FLAG, SDFM_FILTER_4_HIGH_THRESHOLD_FLAG, SDFM_FILTER_4_LOW_THRESHOLD_FLAG, SDFM_FILTER_1_FIFO_OVERFLOW_FLAG, SDFM_FILTER_2_FIFO_OVERFLOW_FLAG, SDFM_FILTER_3_FIFO_OVERFLOW_FLAG SDFM_FILTER_4_FIFO_OVERFLOW_FLAG, SDFM_FILTER_1_FIFO_INTERRUPT_FLAG, SDFM_FILTER_2_FIFO_INTERRUPT_FLAG, SDFM_FILTER_3_FIFO_INTERRUPT_FLAG SDFM_FILTER_4_FIFO_INTERRUPT_FLAG or any combination of the above flags.

Returns

None

31.2.4.33 static void SDFM_enableMasterInterrupt (uint32_t *base*) [inline], [static]

Enable master interrupt.

Parameters

base	is the base address of the SDFM module

This function enables the master SDFM interrupt.

Returns

None

31.2.4.34 static void SDFM_disableMasterInterrupt (uint32_t *base*) [inline], [static]

Disable master interrupt.

Parameters

base	is the base address of the SDFM module

This function disables the master SDFM interrupt.

Returns

None

31.2.4.35 static void SDFM_enableMasterFilter (uint32_t *base*) [inline], [static]

Enable master interrupt.

Parameters

base	is the base address of the SDFM module

This function enables master filter.

Returns

None

31.2.4.36 static void SDFM_disableMasterFilter (uint32_t *base*) [inline], [static]

Disable master filter.

Parameters

base	is the base address of the SDFM module

This function disables master filter.

Returns

None

31.2.4.37 static uint16_t SDFM_getFIFODataCount (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Return the FIFO data count

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the FIFO data count.

Returns

Returns the number of data words available in FIFO buffer.

31.2.4.38 static uint16_t SDFM_getComparatorSincData (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) `[inline]`, `[static]`

Return the Comparator sinc filter data

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the Comparator sinc filter data output.

Returns

Returns the Comparator sinc filter data output.

31.2.4.39 static uint32_t SDFM_getFIFOData (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) `[inline]`, `[static]`

Return the FIFO data

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the latest FIFO data.

Returns

Returns the latest FIFO data.

Note

Discard the upper 16 bits if the output data format is 16bits.

31.2.4.40 static void SDFM_setFIFOInterruptLevel (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, uint16_t *fifoLevel*) `[inline]`, `[static]`

Set the FIFO interrupt level.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
fifoLevel	is the FIFO interrupt level.

This function sets the FIFO interrupt level. Interrupt is generated when the FIFO buffer word count gets to or exceeds the value of *fifoLevel*. Maximum value for *fifoLevel* is 16.

Returns

None.

Referenced by SDFM_configDataFilterFIFO().

31.2.4.41 static void SDFM_setDataReadyInterruptSource (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_DataReadyInterruptSource** *dataReadySource*) [inline], [static]

Set data ready interrupt source.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
dataReadySource	is the data ready interrupt source.

This function sets the data ready interrupt source. Valid values for *dataReadySource:*

- SDFM_DATA_READY_SOURCE_DIRECT - Direct data ready
- SDFM_DATA_READY_SOURCE_FIFO - FIFO data ready.

Returns

None.

31.2.4.42 static bool SDFM_getWaitForSyncStatus (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Get the wait-for-sync event status.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function returns the Wait-for-Sync event status.

Returns

Returns true if sync event has occurred. Returns false if sync event has not occurred.

31.2.4.43 static void SDFM_clearWaitForSyncFlag (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Clear the Wait-for-sync event status.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function clears the Wait-for-sync event status.

Returns

None.

31.2.4.44 static void SDFM_enableWaitForSync (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Enable wait for sync mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function enables the wait for sync mode. Data to FIFO will be written only after PWM sync event.

Returns

None.

31.2.4.45 static void SDFM_disableWaitForSync (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*) [inline], [static]

Disable wait for sync mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.

This function disables the wait for sync mode. Data to FIFO will be written every Data ready event.

Returns

None.

31.2.4.46 static void SDFM_setPWMSyncSource (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_PWMSyncSource** *syncSource*) [inline], [static]

Set the PWM sync mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
syncSource	is the PWM sync source.

This function sets the PWM sync source for the specific SDFM filter. Valid values for syncSource are SDFM_SYNC_PWMx_CMPy. Where x ranges from 1 to 8 Representing PWM1 to PWM8 respectively and y ranges from A to D representing PWM comparators A to D.

Returns
None.

31.2.4.47 static void SDFM_setFIFOClearOnSyncMode (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_FIFOClearSyncMode** *fifoClearSyncMode*) [inline], [static]

Set FIFO clear on sync mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
fifoClearSyncMode	is the FIFO clear on sync mode.

This function sets the FIFO clear mode for the specified filter when a sync happens depending on the value of fifoClearSyncMode. Valid values for fifoClearSyncMode are:

- SDFM_FIFO_NOT_CLEARED_ON_SYNC - FIFO is not cleared on sync.
- SDFM_FIFO_CLEARED_ON_SYNC - FIFO is cleared on sync.

Returns
None.

31.2.4.48 static void SDFM_setWaitForSyncClearMode (uint32_t *base*, **SDFM_FilterNumber** *filterNumber*, **SDFM_WaitForSyncClearMode** *syncClearMode*) [inline], [static]

Set Wait-for-sync clear mode.

Parameters

base	is the base address of the SDFM module
filterNumber	is the filter number.
syncClearMode	is the wait-for-sync clear mode.

This function sets the Wait-For-sync clear mode depending on the value of syncClearMode. Valid values for syncClearMode are:

- SDFM_MANUAL_CLEAR_WAIT_FOR_SYNC - Wait-for-sync flag is cleared by invoking SDFM_clearWaitForSyncFlag().
- SDFM_AUTO_CLEAR_WAIT_FOR_SYNC - Wait-for-sync flag is cleared automatically on FIFO interrupt.

Returns
None.

31.2.4.49 void SDFM_configComparator (uint32_t *base,* uint16_t *config1,* uint32_t *config2,* uint16_t *config3*)

Configure SDFM comparator high and low thresholds

Parameters

base	is the base address of the SDFM module
config1	is the filter number, filter type and over sampling ratio.
config2	is high-level and low-level threshold values.
config3	is the zero-cross threshold value.

This function configures the comparator filter threshold values based on configurations config1 and config2.

The config1 parameter is the logical OR of the filter number, filter type and oversampling ratio. The bit definitions for config1 are as follow:

- config1.[3:0] filter number
- config1.[7:4] filter type
- config1.[15:8] Over sampling Ratio Valid values for filter number and filter type are defined in SDFM_FilterNumber and SDFM_FilterType enumerations respectively. SDFM_SET_OSR(X) macro can be used to set the value of the oversampling ratio , which ranges [1, 32] inclusive, in the appropriate bit location. For example the value (SDFM_FILTER_1 | SDFM_FILTER_SINC_2 | SDFM_SET_OSR(16)) will select Filter 1, SINC 2 type with an oversampling ratio of 16.

The config2 parameter is the logical OR of the filter high and low threshold values. The bit definitions for config2 are as follow:

- config2.[15:0] low threshold
- config2.[31:16] high threshold The upper 16 bits define the high threshold and the lower 16 bits define the low threshold. SDFM_THRESHOLD(H, L) can be used to combine the high and low thresholds.

The config3 parameter is the logical OR of the zero cross threshold enable flag and the zero-cross threshold value. The bit definitions for config3 are as follow:

- config3.[15] - Enable or disable zero cross threshold. Valid values are 1 or 0 to enable or disable the zero cross threshold respectively. -config3.[14:0] - Zero Cross Threshold value. The SDFM_SET_ZERO_CROSS_THRESH_VALUE(X) macro can be used to specify the zero-cross threshold value and OR the 1 to enable it.

Returns

None.

References SDFM_enableZeroCrossEdgeDetect(), SDFM_GET_HIGH_THRESHOLD, SDFM_GET_LOW_THRESHOLD, SDFM_setComparatorFilterType(), SDFM_setCompFilterHighThreshold(), SDFM_setCompFilterLowThreshold(), SDFM_setCompFilterOverSamplingRatio(), and SDFM_setCompFilterZeroCrossThreshold().

31.2.4.50 void SDFM_configDataFilter (uint32_t *base,* uint16_t *config1,* uint16_t *config2*)

Configure SDFM data filter

Parameters

base	is the base address of the SDFM module
config1	is the filter number, filter type and over sampling ratio configuration.
config2	is filter switch, data representation and data shift values configuration.

This function configures the data filter based on configurations config1 and config2.

The config1 parameter is the logical OR of the filter number, filter type and oversampling ratio. The bit definitions for config1 are as follow:

- config1.[3:0] Filter number
- config1.[7:4] Filter type
- config1.[15:8] Over sampling Ratio Valid values for filter number and filter type are defined in SDFM_FilterNumber and SDFM_FilterType enumerations respectively. SDFM_SET_OSR(X) macro can be used to set the value of the oversampling ratio , which ranges [1, 256] inclusive , in the appropriate bit location for config1. For example the value (SDFM_FILTER_2 | SDFM_FILTER_SINC_3 | SDFM_SET_OSR(64)) will select Filter 2 , SINC 3 type with an oversampling ratio of 64.

The config2 parameter is the logical OR of data representation, filter switch, and data shift values The bit definitions for config2 are as follow:

- config2.[0] Data representation
- config2.[1] Filter switch
- config2.[15:2] Shift values Valid values for data representation are given in SDFM_OutputDataFormat enumeration. SDFM_FILTER_DISABLE or SDFM_FILTER_ENABLE will define the filter switch values.SDFM_SHIFT_VALUE(X) macro can be used to set the value of the data shift value, which ranges [0, 31] inclusive, in the appropriate bit location for config2. The shift value is valid only in SDFM_DATA_FORMAT_16_BIT data representation format.

Returns

None.

References SDFM_DATA_FORMAT_16_BIT, SDFM_disableFilter(), SDFM_enableFilter(), SDFM_setDataShiftValue(), SDFM_setFilterOverSamplingRatio(), SDFM_setFilterType(), and SDFM_setOutputDataFormat().

31.2.4.51 void SDFM_configZeroCrossComparator (uint32_t *base,* uint16_t *config1,* uint16_t *config2*)

Configure SDFM comparator Zero Cross threshold

Parameters

base	is the base address of the SDFM module
config1	is the filter number, filter type and over sampling ratio.
config2	is the zero cross threshold value.

This function configures the comparator filter zero cross threshold values based on configurations config1 and config2.

The config1 parameter is the logical OR of the filter number, filter type and oversampling ratio. The bit definitions for config1 are as follow:

- config1.[3:0] filter number
- config1.[7:4] filter type
- config1.[15:8] Over sampling Ratio Valid values for filter number and filter type are defined in SDFM_FilterNumber and SDFM_FilterType enumerations respectively. SDFM_SET_OSR(X) macro can be used to set the value of the oversampling ratio , which ranges [1, 32] inclusive, in the appropriate bit location. For example the value (SDFM_FILTER_1 | SDFM_FILTER_SINC_2 | SDFM_SET_OSR(16)) will select Filter 1 , SINC 2 type with an oversampling ratio of 16.

The config2 parameter is the value of the zero cross threshold. The maximum acceptable value is 32767.

Returns

None.

References SDFM_setComparatorFilterType(), SDFM_setCompFilterOverSamplingRatio(), and SDFM_setCompFilterZeroCrossThreshold().

31.2.4.52 void SDFM_configDataFilterFIFO (uint32_t *base,* uint16_t *config1,* uint16_t *config2*)

Configure SDFM data filter FIFO

Parameters

base	is the base address of the SDFM module
config1	is the filter number, filter type and over sampling ratio configuration.
config2	is filter switch, data representation and data shift values and FIFO level configuration.

This function enables and configures the data filter FIFO based on configurations config1 and config2.

The config1 parameter is the logical OR of the filter number, filter type and oversampling ratio. The bit definitions for config1 are as follow:

- config1.[3:0] filter number
- config1.[7:4] filter type
- config1.[15:8] Over sampling Ratio Valid values for filter number and filter type are defined in SDFM_FilterNumber and SDFM_FilterType enumerations respectively. SDFM_SET_OSR(X) macro can be used to set the value of the oversampling ratio , which ranges [1, 256] inclusive , in the appropriate bit location for config1. For example the value (SDFM_FILTER_2 | SDFM_FILTER_SINC_3 | SDFM_SET_OSR(64)) will select Filter 2 , SINC 3 type with an oversampling ratio of 64.

The config2 parameter is the logical OR of data representation, filter switch, data shift value, and FIFO level The bit definitions for config2 are as follow:

- config2.[0] Data representation
- config2.[1] filter switch.
- config2.[6:2] shift values.
- config2.[15:7] FIFO level Valid values for data representation are given in SDFM_OutputDataFormat enumeration. SDFM_FILTER_DISABLE or

SDFM_FILTER_ENABLE will define the filter switch values.SDFM_SHIFT_VALUE(X) macro can be used to set the value of the data shift value, which ranges [0, 31] inclusive, in the appropriate bit location for config2. The value of FIFO level ranges [1, 16] inclusive. The macro SDFM_SET_FIFO_LEVEL(X) can be used to set the value of the FIFO level.

Returns

None.

References SDFM_DATA_FORMAT_16_BIT, SDFM_disableFilter(), SDFM_enableFIFOBuffer(), SDFM_enableFilter(), SDFM_setDataShiftValue(), SDFM_setFIFOInterruptLevel(), SDFM_setFilterOverSamplingRatio(), SDFM_setFilterType(), and SDFM_setOutputDataFormat().

32 SPI Module

Introduction 609
API Functions 609

32.1 SPI Introduction

The serial peripheral interface (SPI) API provides a set of functions to configure the device's SPI module. Functions are provided to initialize the module, to send and receive data, to obtain status information, and to manage interrupts. Both master and slave modes are supported.

32.2 API Functions

Enumerations

- enum SPI_TransferProtocol { SPI_PROT_POL0PHA0, SPI_PROT_POL0PHA1, SPI_PROT_POL1PHA0, SPI_PROT_POL1PHA1 }
- enum SPI_Mode { SPI_MODE_SLAVE, SPI_MODE_MASTER, SPI_MODE_SLAVE_OD, SPI_MODE_MASTER_OD }
- enum SPI_TxFIFOLevel { SPI_FIFO_TXEMPTY, SPI_FIFO_TX0, SPI_FIFO_TX1, SPI_FIFO_TX2, SPI_FIFO_TX3, SPI_FIFO_TX4, SPI_FIFO_TX5, SPI_FIFO_TX6, SPI_FIFO_TX7, SPI_FIFO_TX8, SPI_FIFO_TX9, SPI_FIFO_TX10, SPI_FIFO_TX11, SPI_FIFO_TX12, SPI_FIFO_TX13, SPI_FIFO_TX14, SPI_FIFO_TX15, SPI_FIFO_TX16, SPI_FIFO_TXFULL }
- enum SPI_RxFIFOLevel { SPI_FIFO_RXEMPTY, SPI_FIFO_RX0, SPI_FIFO_RX1, SPI_FIFO_RX2, SPI_FIFO_RX3, SPI_FIFO_RX4, SPI_FIFO_RX5, SPI_FIFO_RX6, SPI_FIFO_RX7, SPI_FIFO_RX8, SPI_FIFO_RX9, SPI_FIFO_RX10, SPI_FIFO_RX11, SPI_FIFO_RX12, SPI_FIFO_RX13, SPI_FIFO_RX14, SPI_FIFO_RX15, SPI_FIFO_RX16, SPI_FIFO_RXFULL, SPI_FIFO_RXDEFAULT }
- enum SPI_EmulationMode { SPI_EMULATION_STOP_MIDWAY, SPI_EMULATION_FREE_RUN, SPI_EMULATION_STOP_AFTER_TRANSMIT }
- enum SPI_STEPolarity { SPI_STE_ACTIVE_LOW, SPI_STE_ACTIVE_HIGH }

Functions

- static void SPI_enableModule (uint32_t base)
- static void SPI_disableModule (uint32_t base)
- static void SPI_enableFIFO (uint32_t base)
- static void SPI_disableFIFO (uint32_t base)
- static void SPI_resetTxFIFO (uint32_t base)
- static void SPI_resetRxFIFO (uint32_t base)
- static void SPI_setFIFOInterruptLevel (uint32_t base, SPI_TxFIFOLevel txLevel, SPI_RxFIFOLevel rxLevel)
- static void SPI_getFIFOInterruptLevel (uint32_t base, SPI_TxFIFOLevel ∗txLevel, SPI_RxFIFOLevel ∗rxLevel)

- static SPI_TxFIFOLevel SPI_getTxFIFOStatus (uint32_t base)
- static SPI_RxFIFOLevel SPI_getRxFIFOStatus (uint32_t base)
- static bool SPI_isBusy (uint32_t base)
- static void SPI_writeDataNonBlocking (uint32_t base, uint16_t data)
- static uint16_t SPI_readDataNonBlocking (uint32_t base)
- static void SPI_writeDataBlockingFIFO (uint32_t base, uint16_t data)
- static uint16_t SPI_readDataBlockingFIFO (uint32_t base)
- static void SPI_writeDataBlockingNonFIFO (uint32_t base, uint16_t data)
- static uint16_t SPI_readDataBlockingNonFIFO (uint32_t base)
- static void SPI_enableTriWire (uint32_t base)
- static void SPI_disableTriWire (uint32_t base)
- static void SPI_enableLoopback (uint32_t base)
- static void SPI_disableLoopback (uint32_t base)
- static void SPI_setSTESignalPolarity (uint32_t base, SPI_STEPolarity polarity)
- static void SPI_enableHighSpeedMode (uint32_t base)
- static void SPI_disableHighSpeedMode (uint32_t base)
- static void SPI_setEmulationMode (uint32_t base, SPI_EmulationMode mode)
- void SPI_setConfig (uint32_t base, uint32_t lspclkHz, SPI_TransferProtocol protocol, SPI_Mode mode, uint32_t bitRate, uint16_t dataWidth)
- void SPI_setBaudRate (uint32_t base, uint32_t lspclkHz, uint32_t bitRate)
- void SPI_enableInterrupt (uint32_t base, uint32_t intFlags)
- void SPI_disableInterrupt (uint32_t base, uint32_t intFlags)
- uint32_t SPI_getInterruptStatus (uint32_t base)
- void SPI_clearInterruptStatus (uint32_t base, uint32_t intFlags)

32.2.1 Detailed Description

Before initializing the SPI module, the user first must put the module into the reset state by calling SPI_disableModule(). The next call should be to SPI_setConfig() to set properties like master or slave mode, bit rate of the SPI clock signal, data width, and the number of bits per frame.

The next step is to do any any FIFO or interrupt configuration. FIFOs are configured using SPI_enableFIFO() and SPI_disableFIFO() and SPI_setFIFOInterruptLevel() if interrupts are desired. The functions SPI_enableInterrupt(), SPI_disableInterrupt(), SPI_clearInterruptStatus(), and SPI_getInterruptStatus() are for management of interrupts. Note that the SPI module uses separate interrupt lines for its receive and transmit interrupts when in FIFO mode, but only the "receive" interrupt line when not in FIFO mode.

When configuration is complete, SPI_enableModule() should be called to enable the operation of the module.

To transmit data, there are a few options. SPI_writeDataNonBlocking() will simply write the specified data to the transmit buffer and return. It is left up to the user to check beforehand that the module is ready for a new piece of data to be written to the buffer. This means checking the buffer-full flag is not set or, if in FIFO mode, checking how full the FIFO is using SPI_getTxFIFOStatus() when in FIFO mode. The other option is to use one of the two functions SPI_writeDataBlockingNonFIFO() and SPI_writeDataBlockingFIFO() that will wait in a while-loop for the module to be ready.

When receiving data, again, there are a few options. SPI_readDataNonBlocking() will immediately return the contents of the receive buffer. The user should check that there is in fact data ready by checking the buffer-full flag or, if in FIFO mode, checking how full the FIFO is using SPI_getRxFIFOStatus(). SPI_readDataBlockingNonFIFO() and SPI_readDataBlockingFIFO(), however, will wait in a while-loop for data to become available.

The code for this module is contained in `driverlib/spi.c`, with `driverlib/spi.h` containing the API declarations for use by applications.

32.2.2 Enumeration Type Documentation

32.2.2.1 enum **SPI_TransferProtocol**

Values that can be passed to SPI_setConfig() as the *protocol* parameter.

Enumerator

SPI_PROT_POL0PHA0 Mode 0. Polarity 0, phase 0. Rising edge without delay.
SPI_PROT_POL0PHA1 Mode 1. Polarity 0, phase 1. Rising edge with delay.
SPI_PROT_POL1PHA0 Mode 2. Polarity 1, phase 0. Falling edge without delay.
SPI_PROT_POL1PHA1 Mode 3. Polarity 1, phase 1. Falling edge with delay.

32.2.2.2 enum **SPI_Mode**

Values that can be passed to SPI_setConfig() as the *mode* parameter.

Enumerator

SPI_MODE_SLAVE SPI slave.
SPI_MODE_MASTER SPI master.
SPI_MODE_SLAVE_OD SPI slave w/ output (TALK) disabled.
SPI_MODE_MASTER_OD SPI master w/ output (TALK) disabled.

32.2.2.3 enum **SPI_TxFIFOLevel**

Values that can be passed to SPI_setFIFOInterruptLevel() as the *txLevel* parameter, returned by SPI_getFIFOInterruptLevel() in the *txLevel* parameter, and returned by SPI_getTxFIFOStatus().

Enumerator

SPI_FIFO_TXEMPTY Transmit FIFO empty.
SPI_FIFO_TX0 Transmit FIFO empty.
SPI_FIFO_TX1 Transmit FIFO 1/16 full.
SPI_FIFO_TX2 Transmit FIFO 2/16 full.
SPI_FIFO_TX3 Transmit FIFO 3/16 full.
SPI_FIFO_TX4 Transmit FIFO 4/16 full.
SPI_FIFO_TX5 Transmit FIFO 5/16 full.
SPI_FIFO_TX6 Transmit FIFO 6/16 full.
SPI_FIFO_TX7 Transmit FIFO 7/16 full.
SPI_FIFO_TX8 Transmit FIFO 8/16 full.
SPI_FIFO_TX9 Transmit FIFO 9/16 full.
SPI_FIFO_TX10 Transmit FIFO 10/16 full.
SPI_FIFO_TX11 Transmit FIFO 11/16 full.
SPI_FIFO_TX12 Transmit FIFO 12/16 full.
SPI_FIFO_TX13 Transmit FIFO 13/16 full.
SPI_FIFO_TX14 Transmit FIFO 14/16 full.
SPI_FIFO_TX15 Transmit FIFO 15/16 full.
SPI_FIFO_TX16 Transmit FIFO full.
SPI_FIFO_TXFULL Transmit FIFO full.

32.2.2.4 enum **SPI_RxFIFOLevel**

Values that can be passed to SPI_setFIFOInterruptLevel() as the *rxLevel* parameter, returned by SPI_getFIFOInterruptLevel() in the *rxLevel* parameter, and returned by SPI_getRxFIFOStatus().

Enumerator

SPI_FIFO_RXEMPTY Receive FIFO empty.
SPI_FIFO_RX0 Receive FIFO empty.
SPI_FIFO_RX1 Receive FIFO 1/16 full.
SPI_FIFO_RX2 Receive FIFO 2/16 full.
SPI_FIFO_RX3 Receive FIFO 3/16 full.
SPI_FIFO_RX4 Receive FIFO 4/16 full.
SPI_FIFO_RX5 Receive FIFO 5/16 full.
SPI_FIFO_RX6 Receive FIFO 6/16 full.
SPI_FIFO_RX7 Receive FIFO 7/16 full.
SPI_FIFO_RX8 Receive FIFO 8/16 full.
SPI_FIFO_RX9 Receive FIFO 9/16 full.
SPI_FIFO_RX10 Receive FIFO 10/16 full.
SPI_FIFO_RX11 Receive FIFO 11/16 full.
SPI_FIFO_RX12 Receive FIFO 12/16 full.
SPI_FIFO_RX13 Receive FIFO 13/16 full.
SPI_FIFO_RX14 Receive FIFO 14/16 full.
SPI_FIFO_RX15 Receive FIFO 15/16 full.
SPI_FIFO_RX16 Receive FIFO full.
SPI_FIFO_RXFULL Receive FIFO full.
SPI_FIFO_RXDEFAULT To prevent interrupt at reset.

32.2.2.5 enum **SPI_EmulationMode**

Values that can be passed to SPI_setEmulationMode() as the *mode* parameter.

Enumerator

SPI_EMULATION_STOP_MIDWAY Transmission stops after midway in the bit stream.
SPI_EMULATION_FREE_RUN Continue SPI operation regardless.
SPI_EMULATION_STOP_AFTER_TRANSMIT Transmission will stop after a started transmission completes.

32.2.2.6 enum **SPI_STEPolarity**

Values that can be passed to SPI_setSTESignalPolarity() as the *polarity* parameter.

Enumerator

SPI_STE_ACTIVE_LOW SPISTE is active low (normal)
SPI_STE_ACTIVE_HIGH SPISTE is active high (inverted)

32.2.3 Function Documentation

32.2.3.1 static void SPI_enableModule (uint32_t *base*) [inline], [static]

Enables the serial peripheral interface.

Parameters

base	specifies the SPI module base address.

This function enables operation of the serial peripheral interface. The serial peripheral interface must be configured before it is enabled.

Returns

None.

32.2.3.2 static void SPI_disableModule (uint32_t *base*) `[inline]`, `[static]`

Disables the serial peripheral interface.

Parameters

base	specifies the SPI module base address.

This function disables operation of the serial peripheral interface. Call this function before doing any configuration.

Returns

None.

32.2.3.3 static void SPI_enableFIFO (uint32_t *base*) `[inline]`, `[static]`

Enables the transmit and receive FIFOs.

Parameters

base	is the base address of the SPI port.

This functions enables the transmit and receive FIFOs in the SPI.

Returns

None.

32.2.3.4 static void SPI_disableFIFO (uint32_t *base*) `[inline]`, `[static]`

Disables the transmit and receive FIFOs.

Parameters

base	is the base address of the SPI port.

This functions disables the transmit and receive FIFOs in the SPI.

Returns

None.

32.2.3.5 static void SPI_resetTxFIFO (uint32_t *base*) `[inline]`, `[static]`

Resets the transmit FIFO.

Parameters

base	is the base address of the SPI port.

This function resets the transmit FIFO, setting the FIFO pointer back to zero.

Returns
None.

32.2.3.6 static void SPI_resetRxFIFO (uint32_t *base*) [inline], [static]

Resets the receive FIFO.

Parameters

base	is the base address of the SPI port.

This function resets the receive FIFO, setting the FIFO pointer back to zero.

Returns
None.

32.2.3.7 static void SPI_setFIFOInterruptLevel (uint32_t *base,* **SPI_TxFIFOLevel** *txLevel,* **SPI_RxFIFOLevel** *rxLevel*) [inline], [static]

Sets the FIFO level at which interrupts are generated.

Parameters

base	is the base address of the SPI port.
txLevel	is the transmit FIFO interrupt level, specified as **SPI_FIFO_TX0**, **SPI_FIFO_TX1**, **SPI_FIFO_TX2**, . . . or **SPI_FIFO_TX16**.
rxLevel	is the receive FIFO interrupt level, specified as **SPI_FIFO_RX0**, **SPI_FIFO_RX1**, **SPI_FIFO_RX2**, . . . or **SPI_FIFO_RX16**.

This function sets the FIFO level at which transmit and receive interrupts are generated.

Returns
None.

32.2.3.8 static void SPI_getFIFOInterruptLevel (uint32_t *base,* **SPI_TxFIFOLevel** ∗ *txLevel,* **SPI_RxFIFOLevel** ∗ *rxLevel*) [inline], [static]

Gets the FIFO level at which interrupts are generated.

Parameters

base	is the base address of the SPI port.

txLevel	is a pointer to storage for the transmit FIFO level, returned as one of **SPI_FIFO_TX0**, **SPI_FIFO_TX1**, **SPI_FIFO_TX2**, . . . or **SPI_FIFO_TX16**.
rxLevel	is a pointer to storage for the receive FIFO level, returned as one of **SPI_FIFO_RX0**, **SPI_FIFO_RX1**, **SPI_FIFO_RX2**, . . . or **SPI_FIFO_RX16**.

This function gets the FIFO level at which transmit and receive interrupts are generated.

Returns

None.

32.2.3.9 static **SPI_TxFIFOLevel** SPI_getTxFIFOStatus (uint32_t *base*) `[inline]`, `[static]`

Get the transmit FIFO status

Parameters

base	is the base address of the SPI port.

This function gets the current number of words in the transmit FIFO.

Returns

Returns the current number of words in the transmit FIFO specified as one of the following: **SPI_FIFO_TX0**, **SPI_FIFO_TX1**, **SPI_FIFO_TX2**, **SPI_FIFO_TX3**, ..., or **SPI_FIFO_TX16**

Referenced by SPI_writeDataBlockingFIFO().

32.2.3.10 static **SPI_RxFIFOLevel** SPI_getRxFIFOStatus (uint32_t *base*) `[inline]`, `[static]`

Get the receive FIFO status

Parameters

base	is the base address of the SPI port.

This function gets the current number of words in the receive FIFO.

Returns

Returns the current number of words in the receive FIFO specified as one of the following: **SPI_FIFO_RX0**, **SPI_FIFO_RX1**, **SPI_FIFO_RX2**, **SPI_FIFO_RX3**, ..., or **SPI_FIFO_RX16**

Referenced by SPI_readDataBlockingFIFO().

32.2.3.11 static bool SPI_isBusy (uint32_t *base*) `[inline]`, `[static]`

Determines whether the SPI transmitter is busy or not.

Parameters

base	is the base address of the SPI port.

This function allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the hardware shift register. This function is only valid when operating in FIFO mode.

Returns

Returns **true** if the SPI is transmitting or **false** if all transmissions are complete.

32.2.3.12 static void SPI_writeDataNonBlocking (uint32_t *base,* uint16_t *data*) [inline], [static]

Puts a data element into the SPI transmit buffer.

Parameters

base	specifies the SPI module base address.
data	is the left-justified data to be transmitted over SPI.

This function places the supplied data into the transmit buffer of the specified SPI module.

Note

The data being sent must be left-justified in *data*. The lower 16 - N bits will be discarded where N is the data width selected in SPI_setConfig(). For example, if configured for a 6-bit data width, the lower 10 bits of data will be discarded.

Returns

None.

32.2.3.13 static uint16_t SPI_readDataNonBlocking (uint32_t *base*) [inline], [static]

Gets a data element from the SPI receive buffer.

Parameters

base	specifies the SPI module base address.

This function gets received data from the receive buffer of the specified SPI module and returns it.

Note

Only the lower N bits of the value written to *data* contain valid data, where N is the data width as configured by SPI_setConfig(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *data* contain valid data.

Returns

Returns the word of data read from the SPI receive buffer.

32.2.3.14 static void SPI_writeDataBlockingFIFO (uint32_t *base*, uint16_t *data*) [inline], [static]

Waits for space in the FIFO and then puts data into the transmit buffer.

Parameters

base	specifies the SPI module base address.
data	is the left-justified data to be transmitted over SPI.

This function places the supplied data into the transmit buffer of the specified SPI module once space is available in the transmit FIFO. This function should only be used when the FIFO is enabled.

Note

The data being sent must be left-justified in *data*. The lower 16 - N bits will be discarded where N is the data width selected in SPI_setConfig(). For example, if configured for a 6-bit data width, the lower 10 bits of data will be discarded.

Returns

None.

References SPI_FIFO_TXFULL, and SPI_getTxFIFOStatus().

32.2.3.15 static uint16_t SPI_readDataBlockingFIFO (uint32_t *base*) [inline], [static]

Waits for data in the FIFO and then reads it from the receive buffer.

Parameters

base	specifies the SPI module base address.

This function waits until there is data in the receive FIFO and then reads received data from the receive buffer. This function should only be used when FIFO mode is enabled.

Note

Only the lower N bits of the value written to *data* contain valid data, where N is the data width as configured by SPI_setConfig(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *data* contain valid data.

Returns

Returns the word of data read from the SPI receive buffer.

References SPI_FIFO_RXEMPTY, and SPI_getRxFIFOStatus().

32.2.3.16 static void SPI_writeDataBlockingNonFIFO (uint32_t *base,* uint16_t *data*) [inline], [static]

Waits for the transmit buffer to empty and then writes data to it.

Parameters

base	specifies the SPI module base address.

data	is the left-justified data to be transmitted over SPI.

This function places the supplied data into the transmit buffer of the specified SPI module once it is empty. This function should not be used when FIFO mode is enabled.

Note

The data being sent must be left-justified in *data*. The lower 16 - N bits will be discarded where N is the data width selected in SPI_setConfig(). For example, if configured for a 6-bit data width, the lower 10 bits of data will be discarded.

Returns

None.

32.2.3.17 static uint16_t SPI_readDataBlockingNonFIFO (uint32_t *base*) [inline], [static]

Waits for data to be received and then reads it from the buffer.

Parameters

base	specifies the SPI module base address.

This function waits for data to be received and then reads it from the receive buffer of the specified SPI module. This function should not be used when FIFO mode is enabled.

Note

Only the lower N bits of the value written to *data* contain valid data, where N is the data width as configured by SPI_setConfig(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *data* contain valid data.

Returns

Returns the word of data read from the SPI receive buffer.

32.2.3.18 static void SPI_enableTriWire (uint32_t *base*) [inline], [static]

Enables SPI 3-wire mode.

Parameters

base	is the base address of the SPI port.

This function enables 3-wire mode. When in master mode, this allows SPISIMO to become SPIMOMI and SPISOMI to become free for non-SPI use. When in slave mode, SPISOMI because the SPISISO pin and SPISIMO is free for non-SPI use.

Returns

None.

32.2.3.19 static void SPI_disableTriWire (uint32_t *base*) [inline], [static]

Disables SPI 3-wire mode.

Parameters

base	is the base address of the SPI port.

This function disables 3-wire mode. SPI will operate in normal 4-wire mode.

Returns

None.

32.2.3.20 static void SPI_enableLoopback (uint32_t *base*) [inline], [static]

Enables SPI loopback mode.

Parameters

base	is the base address of the SPI port.

This function enables loopback mode. This mode is only valid during master mode and is helpful during device testing as it internally connects SIMO and SOMI.

Returns

None.

32.2.3.21 static void SPI_disableLoopback (uint32_t *base*) [inline], [static]

Disables SPI loopback mode.

Parameters

base	is the base address of the SPI port.

This function disables loopback mode. Loopback mode is disabled by default after reset.

Returns

None.

32.2.3.22 static void SPI_setSTESignalPolarity (uint32_t *base*, **SPI_STEPolarity** *polarity*) [inline], [static]

Set the slave select (SPISTE) signal polarity.

Parameters

base	is the base address of the SPI port.
polarity	is the SPISTE signal polarity.

This function sets the polarity of the slave select (SPISTE) signal. The two modes to choose from for the *polarity* parameter are **SPI_STE_ACTIVE_LOW** for active-low polarity (typical) and **SPI_STE_ACTIVE_HIGH** for active-high polarity (considered inverted).

Note

This has no effect on the STE signal when in master mode. It is only applicable to slave mode.

Returns
None.

32.2.3.23 static void SPI_enableHighSpeedMode (uint32_t *base*) [inline], [static]

Enables SPI high speed mode.

Parameters

base	is the base address of the SPI port.

This function enables high speed mode.

Returns
None.

32.2.3.24 static void SPI_disableHighSpeedMode (uint32_t *base*) [inline], [static]

Disables SPI high speed mode.

Parameters

base	is the base address of the SPI port.

This function disables high speed mode. High speed mode is disabled by default after reset.

Returns
None.

32.2.3.25 static void SPI_setEmulationMode (uint32_t *base,* **SPI_EmulationMode** *mode*) [inline], [static]

Sets SPI emulation mode.

Parameters

base	is the base address of the SPI port.
mode	is the emulation mode.

This function sets the behavior of the SPI operation when an emulation suspend occurs. The *mode* parameter can be one of the following:

- **SPI_EMULATION_STOP_MIDWAY** - Transmission stops midway through the bit stream. The rest of the bits will be transmitting after the suspend is deasserted.
- **SPI_EMULATION_STOP_AFTER_TRANSMIT** - If the suspend occurs before the first SPICLK pulse, the transmission will not start. If it occurs later, the transmission will be completed.
- **SPI_EMULATION_FREE_RUN** - SPI operation continues regardless of a the suspend.

Returns
None.

32.2.3.26 void SPI_setConfig (uint32_t *base,* uint32_t *lspclkHz,* **SPI_TransferProtocol** *protocol,* **SPI_Mode** *mode,* uint32_t *bitRate,* uint16_t *dataWidth*)

Configures the serial peripheral interface.

Parameters

base	specifies the SPI module base address.
lspclkHz	is the rate of the clock supplied to the SPI module (LSPCLK) in Hz.
protocol	specifies the data transfer protocol.
mode	specifies the mode of operation.
bitRate	specifies the clock rate in Hz.
dataWidth	specifies number of bits transferred per frame.

This function configures the serial peripheral interface. It sets the SPI protocol, mode of operation, bit rate, and data width.

The *protocol* parameter defines the data frame format. The *protocol* parameter can be one of the following values: **SPI_PROT_POL0PHA0**, **SPI_PROT_POL0PHA1**, **SPI_PROT_POL1PHA0**, or **SPI_PROT_POL1PHA1**. These frame formats encode the following polarity and phase configurations:

```
Polarity Phase       Mode
  0        0    SPI_PROT_POL0PHA0
  0        1    SPI_PROT_POL0PHA1
  1        0    SPI_PROT_POL1PHA0
  1        1    SPI_PROT_POL1PHA1
```

The *mode* parameter defines the operating mode of the SPI module. The SPI module can operate as a master or slave; the SPI can also be be configured to disable output on its serial output line. The *mode* parameter can be one of the following values: **SPI_MODE_MASTER**, **SPI_MODE_SLAVE**, **SPI_MODE_MASTER_OD** or **SPI_MODE_SLAVE_OD** ("OD" indicates "output disabled").

The *bitRate* parameter defines the bit rate for the SPI. This bit rate must satisfy the following clock ratio criteria:

- *bitRate* can be no greater than lspclkHz divided by 4.
- *lspclkHz* / *bitRate* cannot be greater than 128.

The *dataWidth* parameter defines the width of the data transfers and can be a value between 1 and 16, inclusive.

The peripheral clock is the low speed peripheral clock. This value is returned by SysCtl_getLowSpeedClock(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtl_getLowSpeedClock()).

Note

SPI operation should be disabled via SPI_disableModule() before any changes to its configuration.

Returns

None.

32.2.3.27 void SPI_setBaudRate (uint32_t *base,* uint32_t *lspclkHz,* uint32_t *bitRate*)

Configures the baud rate of the serial peripheral interface.

Parameters

base	specifies the SPI module base address.
lspclkHz	is the rate of the clock supplied to the SPI module (LSPCLK) in Hz.
bitRate	specifies the clock rate in Hz.

This function configures the SPI baud rate. The *bitRate* parameter defines the bit rate for the SPI. This bit rate must satisfy the following clock ratio criteria:

- *bitRate* can be no greater than *lspclkHz* divided by 4.
- *lspclkHz* / *bitRate* cannot be greater than 128.

The peripheral clock is the low speed peripheral clock. This value is returned by SysCtl_getLowSpeedClock(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtl_getLowSpeedClock()).

Note

SPI_setConfig() also sets the baud rate. Use SPI_setBaudRate() if you wish to configure it separately from protocol and mode.

Returns

None.

32.2.3.28 void SPI_enableInterrupt (uint32_t *base,* uint32_t *intFlags*)

Enables individual SPI interrupt sources.

Parameters

base	specifies the SPI module base address.
intFlags	is a bit mask of the interrupt sources to be enabled.

This function enables the indicated SPI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The *intFlags* parameter can be any of the following values:

- **SPI_INT_RX_OVERRUN** - Receive overrun interrupt
- **SPI_INT_RX_DATA_TX_EMPTY** - Data received, transmit empty
- **SPI_INT_RXFF** (also enables **SPI_INT_RXFF_OVERFLOW**) - RX FIFO level interrupt (and RX FIFO overflow)
- **SPI_INT_TXFF** - TX FIFO level interrupt

Note

SPI_INT_RX_OVERRUN, **SPI_INT_RX_DATA_TX_EMPTY**, **SPI_INT_RXFF_OVERFLOW**, and **SPI_INT_RXFF** are associated with **SPIRXINT**; **SPI_INT_TXFF** is associated with **SPITXINT**.

Returns

None.

32.2.3.29 void SPI_disableInterrupt (uint32_t *base,* uint32_t *intFlags*)

Disables individual SPI interrupt sources.

Parameters

base	specifies the SPI module base address.
intFlags	is a bit mask of the interrupt sources to be disabled.

This function disables the indicated SPI interrupt sources. The *intFlags* parameter can be any of the following values:

- **SPI_INT_RX_OVERRUN**
- **SPI_INT_RX_DATA_TX_EMPTY**
- **SPI_INT_RXFF** (also disables **SPI_INT_RXFF_OVERFLOW**)
- **SPI_INT_TXFF**

Note

SPI_INT_RX_OVERRUN, **SPI_INT_RX_DATA_TX_EMPTY**, **SPI_INT_RXFF_OVERFLOW**, and **SPI_INT_RXFF** are associated with **SPIRXINT**; **SPI_INT_TXFF** is associated with **SPITXINT**.

Returns

None.

32.2.3.30 uint32_t SPI_getInterruptStatus (uint32_t *base*)

Gets the current interrupt status.

Parameters

base	specifies the SPI module base address.

This function returns the interrupt status for the SPI module.

Returns

The current interrupt status, enumerated as a bit field of the following values:

- **SPI_INT_RX_OVERRUN** - Receive overrun interrupt
- **SPI_INT_RX_DATA_TX_EMPTY** - Data received, transmit empty
- **SPI_INT_RXFF** - RX FIFO level interrupt
- **SPI_INT_RXFF_OVERFLOW** - RX FIFO overflow
- **SPI_INT_TXFF** - TX FIFO level interrupt

32.2.3.31 void SPI_clearInterruptStatus (uint32_t *base,* uint32_t *intFlags*)

Clears SPI interrupt sources.

Parameters

base	specifies the SPI module base address.

intFlags	is a bit mask of the interrupt sources to be cleared.

This function clears the specified SPI interrupt sources so that they no longer assert. This function must be called in the interrupt handler to keep the interrupts from being triggered again immediately upon exit. The *intFlags* parameter can consist of a bit field of the following values:

- **SPI_INT_RX_OVERRUN**
- **SPI_INT_RX_DATA_TX_EMPTY**
- **SPI_INT_RXFF**
- **SPI_INT_RXFF_OVERFLOW**
- **SPI_INT_TXFF**

Note

SPI_INT_RX_DATA_TX_EMPTY is cleared by a read of the receive receive buffer, so it usually doesn't need to be cleared using this function.
Also note that **SPI_INT_RX_OVERRUN**, **SPI_INT_RX_DATA_TX_EMPTY**, **SPI_INT_RXFF_OVERFLOW**, and **SPI_INT_RXFF** are associated with **SPIRXINT**; **SPI_INT_TXFF** is associated with **SPITXINT**.

Returns

None.

33 SysCtl Module

Introduction . . . 629
API Functions . . . 629

33.1 SysCtl Introduction

System Control (SysCtl) determines the overall operation of the device. The API provides functions to configure the clocking of the device, the set of peripherals that are enabled, the windowed watchdog, the NMI watchdog, and low-power modes. It also provides functions to handle and obtain information about resets and missing clock detection failures.

33.2 API Functions

Macros

- #define SYSCTL_SYSDIV(x)
- #define SYSCTL_IMULT(x)

Enumerations

- enum SysCtl_PeripheralPCLOCKCR {
SYSCTL_PERIPH_CLK_CLA1, SYSCTL_PERIPH_CLK_DMA,
SYSCTL_PERIPH_CLK_TIMER0, SYSCTL_PERIPH_CLK_TIMER1,
SYSCTL_PERIPH_CLK_TIMER2, SYSCTL_PERIPH_CLK_HRPWM,
SYSCTL_PERIPH_CLK_TBCLKSYNC, SYSCTL_PERIPH_CLK_EPWM1,
SYSCTL_PERIPH_CLK_EPWM2, SYSCTL_PERIPH_CLK_EPWM3,
SYSCTL_PERIPH_CLK_EPWM4, SYSCTL_PERIPH_CLK_EPWM5,
SYSCTL_PERIPH_CLK_EPWM6, SYSCTL_PERIPH_CLK_EPWM7,
SYSCTL_PERIPH_CLK_EPWM8, SYSCTL_PERIPH_CLK_ECAP1,
SYSCTL_PERIPH_CLK_ECAP2, SYSCTL_PERIPH_CLK_ECAP3,
SYSCTL_PERIPH_CLK_ECAP4, SYSCTL_PERIPH_CLK_ECAP5,
SYSCTL_PERIPH_CLK_ECAP6, SYSCTL_PERIPH_CLK_ECAP7,
SYSCTL_PERIPH_CLK_EQEP1, SYSCTL_PERIPH_CLK_EQEP2,
SYSCTL_PERIPH_CLK_SD1, SYSCTL_PERIPH_CLK_SCIA,
SYSCTL_PERIPH_CLK_SCIB, SYSCTL_PERIPH_CLK_SPIA,
SYSCTL_PERIPH_CLK_SPIB, SYSCTL_PERIPH_CLK_I2CA,
SYSCTL_PERIPH_CLK_CANA, SYSCTL_PERIPH_CLK_CANB,
SYSCTL_PERIPH_CLK_ADCA, SYSCTL_PERIPH_CLK_ADCB,
SYSCTL_PERIPH_CLK_ADCC, SYSCTL_PERIPH_CLK_CMPSS1,
SYSCTL_PERIPH_CLK_CMPSS2, SYSCTL_PERIPH_CLK_CMPSS3,
SYSCTL_PERIPH_CLK_CMPSS4, SYSCTL_PERIPH_CLK_CMPSS5,
SYSCTL_PERIPH_CLK_CMPSS6, SYSCTL_PERIPH_CLK_CMPSS7,
SYSCTL_PERIPH_CLK_PGA1, SYSCTL_PERIPH_CLK_PGA2,
SYSCTL_PERIPH_CLK_PGA3, SYSCTL_PERIPH_CLK_PGA4,

SYSCTL_PERIPH_CLK_PGA5, SYSCTL_PERIPH_CLK_PGA6, SYSCTL_PERIPH_CLK_PGA7, SYSCTL_PERIPH_CLK_DACA, SYSCTL_PERIPH_CLK_DACB, SYSCTL_PERIPH_CLK_FSITXA, SYSCTL_PERIPH_CLK_FSIRXA, SYSCTL_PERIPH_CLK_LINA, SYSCTL_PERIPH_CLK_PMBUSA, SYSCTL_PERIPH_CLK_DCC0 }

- enum SysCtl_PeripheralSOFTPRES { SYSCTL_PERIPH_RES_CLA1, SYSCTL_PERIPH_RES_EPWM1, SYSCTL_PERIPH_RES_EPWM2, SYSCTL_PERIPH_RES_EPWM3, SYSCTL_PERIPH_RES_EPWM4, SYSCTL_PERIPH_RES_EPWM5, SYSCTL_PERIPH_RES_EPWM6, SYSCTL_PERIPH_RES_EPWM7, SYSCTL_PERIPH_RES_EPWM8, SYSCTL_PERIPH_RES_ECAP1, SYSCTL_PERIPH_RES_ECAP2, SYSCTL_PERIPH_RES_ECAP3, SYSCTL_PERIPH_RES_ECAP4, SYSCTL_PERIPH_RES_ECAP5, SYSCTL_PERIPH_RES_ECAP6, SYSCTL_PERIPH_RES_ECAP7, SYSCTL_PERIPH_RES_EQEP1, SYSCTL_PERIPH_RES_EQEP2, SYSCTL_PERIPH_RES_SD1, SYSCTL_PERIPH_RES_SCIA, SYSCTL_PERIPH_RES_SCIB, SYSCTL_PERIPH_RES_SPIA, SYSCTL_PERIPH_RES_SPIB, SYSCTL_PERIPH_RES_I2CA, SYSCTL_PERIPH_RES_CANA, SYSCTL_PERIPH_RES_CANB, SYSCTL_PERIPH_RES_ADCA, SYSCTL_PERIPH_RES_ADCB, SYSCTL_PERIPH_RES_ADCC, SYSCTL_PERIPH_RES_CMPSS1, SYSCTL_PERIPH_RES_CMPSS2, SYSCTL_PERIPH_RES_CMPSS3, SYSCTL_PERIPH_RES_CMPSS4, SYSCTL_PERIPH_RES_CMPSS5, SYSCTL_PERIPH_RES_CMPSS6, SYSCTL_PERIPH_RES_CMPSS7, SYSCTL_PERIPH_RES_PGA1, SYSCTL_PERIPH_RES_PGA2, SYSCTL_PERIPH_RES_PGA3, SYSCTL_PERIPH_RES_PGA4, SYSCTL_PERIPH_RES_PGA5, SYSCTL_PERIPH_RES_PGA6, SYSCTL_PERIPH_RES_PGA7, SYSCTL_PERIPH_RES_DACA, SYSCTL_PERIPH_RES_DACB, SYSCTL_PERIPH_RES_FSITXA, SYSCTL_PERIPH_RES_FSIRXA, SYSCTL_PERIPH_RES_LINA, SYSCTL_PERIPH_RES_PMBUSA }
- enum SysCtl_WDPredivider { SYSCTL_WD_PREDIV_2, SYSCTL_WD_PREDIV_4, SYSCTL_WD_PREDIV_8, SYSCTL_WD_PREDIV_16, SYSCTL_WD_PREDIV_32, SYSCTL_WD_PREDIV_64, SYSCTL_WD_PREDIV_128, SYSCTL_WD_PREDIV_256, SYSCTL_WD_PREDIV_512, SYSCTL_WD_PREDIV_1024, SYSCTL_WD_PREDIV_2048, SYSCTL_WD_PREDIV_4096 }
- enum SysCtl_WDPrescaler { SYSCTL_WD_PRESCALE_1, SYSCTL_WD_PRESCALE_2, SYSCTL_WD_PRESCALE_4, SYSCTL_WD_PRESCALE_8, SYSCTL_WD_PRESCALE_16, SYSCTL_WD_PRESCALE_32, SYSCTL_WD_PRESCALE_64 }
- enum SysCtl_WDMode { SYSCTL_WD_MODE_RESET, SYSCTL_WD_MODE_INTERRUPT }
- enum SysCtl_LSPCLKPrescaler { SYSCTL_LSPCLK_PRESCALE_1, SYSCTL_LSPCLK_PRESCALE_2, SYSCTL_LSPCLK_PRESCALE_4, SYSCTL_LSPCLK_PRESCALE_6, SYSCTL_LSPCLK_PRESCALE_8, SYSCTL_LSPCLK_PRESCALE_10, SYSCTL_LSPCLK_PRESCALE_12, SYSCTL_LSPCLK_PRESCALE_14 }
- enum SysCtl_AccessPeripheral { SYSCTL_ACCESS_ADCA, SYSCTL_ACCESS_ADCB, SYSCTL_ACCESS_ADCC,

SYSCTL_ACCESS_CMPSS1,
SYSCTL_ACCESS_CMPSS2, SYSCTL_ACCESS_CMPSS3, SYSCTL_ACCESS_CMPSS4,
SYSCTL_ACCESS_CMPSS5,
SYSCTL_ACCESS_CMPSS6, SYSCTL_ACCESS_CMPSS7, SYSCTL_ACCESS_DACA,
SYSCTL_ACCESS_DACB,
SYSCTL_ACCESS_PGA1, SYSCTL_ACCESS_PGA2, SYSCTL_ACCESS_PGA3,
SYSCTL_ACCESS_PGA4,
SYSCTL_ACCESS_PGA5, SYSCTL_ACCESS_PGA6, SYSCTL_ACCESS_PGA7,
SYSCTL_ACCESS_EPWM1,
SYSCTL_ACCESS_EPWM2, SYSCTL_ACCESS_EPWM3, SYSCTL_ACCESS_EPWM4,
SYSCTL_ACCESS_EPWM5,
SYSCTL_ACCESS_EPWM6, SYSCTL_ACCESS_EPWM7, SYSCTL_ACCESS_EPWM8,
SYSCTL_ACCESS_EQEP1,
SYSCTL_ACCESS_EQEP2, SYSCTL_ACCESS_ECAP1, SYSCTL_ACCESS_ECAP2,
SYSCTL_ACCESS_ECAP3,
SYSCTL_ACCESS_ECAP4, SYSCTL_ACCESS_ECAP5, SYSCTL_ACCESS_ECAP6,
SYSCTL_ACCESS_ECAP7,
SYSCTL_ACCESS_SDFM1, SYSCTL_ACCESS_CLA1PROMCRC,
SYSCTL_ACCESS_SPIA, SYSCTL_ACCESS_SPIB,
SYSCTL_ACCESS_PMBUS_A, SYSCTL_ACCESS_LIN_A, SYSCTL_ACCESS_CANA,
SYSCTL_ACCESS_CANB,
SYSCTL_ACCESS_FSIATX, SYSCTL_ACCESS_FSIARX, SYSCTL_ACCESS_HRPWM_A }

- enum SysCtl_AccessMaster { SYSCTL_ACCESS_CPU1, SYSCTL_ACCESS_CLA1, SYSCTL_ACCESS_DMA1 }
- enum SysCtl_AccessPermission { SYSCTL_ACCESS_FULL, SYSCTL_ACCESS_PROTECTED, SYSCTL_ACCESS_NONE }
- enum SysCtl_ClockOut {
  SYSCTL_CLOCKOUT_PLLSYS, SYSCTL_CLOCKOUT_PLLRAW,
  SYSCTL_CLOCKOUT_SYSCLK, SYSCTL_CLOCKOUT_INTOSC1,
  SYSCTL_CLOCKOUT_INTOSC2, SYSCTL_CLOCKOUT_XTALOSC }
- enum SysCtl_ExternalOscMode { SYSCTL_XTALMODE_CRYSTAL, SYSCTL_XTALMODE_SINGLE }
- enum SysCtl_SyncInput {
  SYSCTL_SYNC_IN_EPWM1, SYSCTL_SYNC_IN_EPWM4, SYSCTL_SYNC_IN_EPWM7,
  SYSCTL_SYNC_IN_ECAP1,
  SYSCTL_SYNC_IN_ECAP4, SYSCTL_SYNC_IN_ECAP6 }
- enum SysCtl_SyncInputSource {
  SYSCTL_SYNC_IN_SRC_EPWM1SYNCOUT,
  SYSCTL_SYNC_IN_SRC_EPWM4SYNCOUT,
  SYSCTL_SYNC_IN_SRC_EPWM7SYNCOUT,
  SYSCTL_SYNC_IN_SRC_ECAP1SYNCOUT,
  SYSCTL_SYNC_IN_SRC_EXTSYNCIN1, SYSCTL_SYNC_IN_SRC_EXTSYNCIN2,
  SYSCTL_SYNC_IN_SRC_ECAP4SYNCOUT }
- enum SysCtl_SyncOutputSource { SYSCTL_SYNC_OUT_SRC_EPWM1SYNCOUT,
  SYSCTL_SYNC_OUT_SRC_EPWM4SYNCOUT,
  SYSCTL_SYNC_OUT_SRC_EPWM7SYNCOUT }
- enum SysCtl_DeviceParametric {
  SYSCTL_DEVICE_QUAL, SYSCTL_DEVICE_PINCOUNT, SYSCTL_DEVICE_INSTASPIN,
  SYSCTL_DEVICE_FLASH,
  SYSCTL_DEVICE_FAMILY, SYSCTL_DEVICE_PARTNO, SYSCTL_DEVICE_CLASSID }

Functions

- static void SysCtl_resetPeripheral (SysCtl_PeripheralSOFTPRES peripheral)
- static void SysCtl_enablePeripheral (SysCtl_PeripheralPCLOCKCR peripheral)
- static void SysCtl_disablePeripheral (SysCtl_PeripheralPCLOCKCR peripheral)
- static void SysCtl_resetDevice (void)
- static uint32_t SysCtl_getResetCause (void)
- static void SysCtl_clearResetCause (uint32_t rstCauses)
- static void SysCtl_setLowSpeedClock (SysCtl_LSPCLKPrescaler prescaler)
- static void SysCtl_selectClockOutSource (SysCtl_ClockOut source)
- static void SysCtl_setExternalOscMode (SysCtl_ExternalOscMode mode)
- static uint16_t SysCtl_getExternalOscCounterValue (void)
- static void SysCtl_clearExternalOscCounterValue (void)
- static void SysCtl_turnOnOsc (uint32_t oscSource)
- static void SysCtl_turnOffOsc (uint32_t oscSource)
- static void SysCtl_enterIdleMode (void)
- static void SysCtl_enterHaltMode (void)
- static void SysCtl_enableLPMWakeupPin (uint32_t pin)
- static void SysCtl_disableLPMWakeupPin (uint32_t pin)
- static void SysCtl_enableWatchdogInHalt (void)
- static void SysCtl_disableWatchdogInHalt (void)
- static void SysCtl_setWatchdogMode (SysCtl_WDMode mode)
- static bool SysCtl_isWatchdogInterruptActive (void)
- static void SysCtl_disableWatchdog (void)
- static void SysCtl_enableWatchdog (void)
- static void SysCtl_serviceWatchdog (void)
- static void SysCtl_setWatchdogPredivider (SysCtl_WDPredivider predivider)
- static void SysCtl_setWatchdogPrescaler (SysCtl_WDPrescaler prescaler)
- static uint16_t SysCtl_getWatchdogCounterValue (void)
- static bool SysCtl_getWatchdogResetStatus (void)
- static void SysCtl_clearWatchdogResetStatus (void)
- static void SysCtl_setWatchdogWindowValue (uint16_t value)
- static bool SysCtl_getNMIStatus (void)
- static uint32_t SysCtl_getNMIFlagStatus (void)
- static bool SysCtl_isNMIFlagSet (uint32_t nmiFlags)
- static void SysCtl_clearNMIStatus (uint32_t nmiFlags)
- static void SysCtl_clearAllNMIFlags (void)
- static void SysCtl_forceNMIFlags (uint32_t nmiFlags)
- static uint16_t SysCtl_getNMIWatchdogCounter (void)
- static void SysCtl_setNMIWatchdogPeriod (uint16_t wdPeriod)
- static uint16_t SysCtl_getNMIWatchdogPeriod (void)
- static uint32_t SysCtl_getNMIShadowFlagStatus (void)
- static bool SysCtl_isNMIShadowFlagSet (uint32_t nmiFlags)
- static void SysCtl_enableMCD (void)
- static void SysCtl_disableMCD (void)
- static bool SysCtl_isMCDClockFailureDetected (void)
- static void SysCtl_resetMCD (void)
- static void SysCtl_connectMCDClockSource (void)
- static void SysCtl_disconnectMCDClockSource (void)
- static void SysCtl_lockAccessControlRegs (void)
- static void SysCtl_setPeripheralAccessControl (SysCtl_AccessPeripheral peripheral, SysCtl_AccessMaster master, SysCtl_AccessPermission permission)
- static uint32_t SysCtl_getPeripheralAccessControl (SysCtl_AccessPeripheral peripheral, SysCtl_AccessMaster master)
- static void SysCtl_setSyncInputConfig (SysCtl_SyncInput syncInput, SysCtl_SyncInputSource syncSrc)
- static void SysCtl_setSyncOutputConfig (SysCtl_SyncOutputSource syncSrc)
- static void SysCtl_enableExtADCSOCSource (uint32_t adcsocSrc)

- static void SysCtl_disableExtADCSOCSource (uint32_t adcsocSrc)
- static void SysCtl_lockExtADCSOCSelect (void)
- static void SysCtl_lockSyncSelect (void)
- static uint32_t SysCtl_getDeviceRevision (void)
- void SysCtl_delay (uint32_t count)
- uint32_t SysCtl_getClock (uint32_t clockInHz)
- bool SysCtl_setClock (uint32_t config)
- bool SysCtl_isPLLValid (uint32_t oscSource, uint32_t pllMult)
- void SysCtl_selectXTAL (void)
- void SysCtl_selectXTALSingleEnded (void)
- void SysCtl_selectOscSource (uint32_t oscSource)
- uint32_t SysCtl_getLowSpeedClock (uint32_t clockInHz)
- uint16_t SysCtl_getDeviceParametric (SysCtl_DeviceParametric parametric)

33.2.1 Detailed Description

Many of the functions provided by the SysCtl API are related to device clocking. The most important of these functions is SysCtl_setClock() which will configure which oscillator is to be used, configure the PLL, and configure the system clock divider. SysCtl_getClock() is a complementary function to this one that will, given the frequency of the oscillator source used, read back the configuration of the PLL and clock divider and calculate the system clock frequency. A similar pair of functions is provided for the low-speed peripheral clock, SysCtl_setLowSpeedClock() and SysCtl_getLowSpeedClock().

The ability to enable (turn on the module clock), disable (gate off the module clock), and perform a software reset on most of the peripherals on a device is provided by SysCtl_enablePeripheral(), SysCtl_disablePeripheral(), and SysCtl_resetPeripheral() respectively.

The device's windowed watchdog is enabled and disabled by SysCtl_enableWatchdog() and SysCtl_disableWatchdog() respectively. The watchdog can be serviced by SysCtl_serviceWatchdog(). Several functions are also provided to configure the watchdog's clock and windowed functionality.

This section will give further details of these functions and each of the others used for the configuration of SysCtl.

The code for this module is contained in `driverlib/sysctl.c`, with `driverlib/sysctl.h` containing the API declarations for use by applications.

33.2.2 Macro Definition Documentation

33.2.2.1 #define SYSCTL_SYSDIV(*x*)

Macro to format system clock divider value. x must be 1 or even values up to 126.

33.2.2.2 #define SYSCTL_IMULT(*x*)

Macro to format integer multiplier value. x is a number from 1 to 127.

33.2.3 Enumeration Type Documentation

33.2.3.1 enum **SysCtl_PeripheralPCLOCKCR**

The following are values that can be passed to SysCtl_enablePeripheral() and SysCtl_disablePeripheral() as the *peripheral* parameter.

Enumerator

SYSCTL_PERIPH_CLK_CLA1 CLA1 clock.
SYSCTL_PERIPH_CLK_DMA DMA clock.
SYSCTL_PERIPH_CLK_TIMER0 CPUTIMER0 clock.
SYSCTL_PERIPH_CLK_TIMER1 CPUTIMER1 clock.
SYSCTL_PERIPH_CLK_TIMER2 CPUTIMER2 clock.
SYSCTL_PERIPH_CLK_HRPWM HRPWM clock.
SYSCTL_PERIPH_CLK_TBCLKSYNC ePWM time base clock sync
SYSCTL_PERIPH_CLK_EPWM1 ePWM1 clock
SYSCTL_PERIPH_CLK_EPWM2 ePWM2 clock
SYSCTL_PERIPH_CLK_EPWM3 ePWM3 clock
SYSCTL_PERIPH_CLK_EPWM4 ePWM4 clock
SYSCTL_PERIPH_CLK_EPWM5 ePWM5 clock
SYSCTL_PERIPH_CLK_EPWM6 ePWM6 clock
SYSCTL_PERIPH_CLK_EPWM7 ePWM7 clock
SYSCTL_PERIPH_CLK_EPWM8 ePWM8 clock
SYSCTL_PERIPH_CLK_ECAP1 eCAP1 clock
SYSCTL_PERIPH_CLK_ECAP2 eCAP2 clock
SYSCTL_PERIPH_CLK_ECAP3 eCAP3 clock
SYSCTL_PERIPH_CLK_ECAP4 eCAP4 clock
SYSCTL_PERIPH_CLK_ECAP5 eCAP5 clock
SYSCTL_PERIPH_CLK_ECAP6 eCAP6 clock
SYSCTL_PERIPH_CLK_ECAP7 eCAP7 clock
SYSCTL_PERIPH_CLK_EQEP1 eQEP1 clock
SYSCTL_PERIPH_CLK_EQEP2 eQEP2 clock
SYSCTL_PERIPH_CLK_SD1 SDFM1 clock.
SYSCTL_PERIPH_CLK_SCIA SCIA clock.
SYSCTL_PERIPH_CLK_SCIB SCIB clock.
SYSCTL_PERIPH_CLK_SPIA SPIA clock.
SYSCTL_PERIPH_CLK_SPIB SPIB clock.
SYSCTL_PERIPH_CLK_I2CA I2CA clock.
SYSCTL_PERIPH_CLK_CANA CANA clock.
SYSCTL_PERIPH_CLK_CANB CANB clock.
SYSCTL_PERIPH_CLK_ADCA ADCA clock.
SYSCTL_PERIPH_CLK_ADCB ADCB clock.
SYSCTL_PERIPH_CLK_ADCC ADCC clock.
SYSCTL_PERIPH_CLK_CMPSS1 CMPSS1 clock.
SYSCTL_PERIPH_CLK_CMPSS2 CMPSS2 clock.
SYSCTL_PERIPH_CLK_CMPSS3 CMPSS3 clock.

SYSCTL_PERIPH_CLK_CMPSS4 CMPSS4 clock.
SYSCTL_PERIPH_CLK_CMPSS5 CMPSS5 clock.
SYSCTL_PERIPH_CLK_CMPSS6 CMPSS6 clock.
SYSCTL_PERIPH_CLK_CMPSS7 CMPSS7 clock.
SYSCTL_PERIPH_CLK_PGA1 PGA1 clock.
SYSCTL_PERIPH_CLK_PGA2 PGA2 clock.
SYSCTL_PERIPH_CLK_PGA3 PGA3 clock.
SYSCTL_PERIPH_CLK_PGA4 PGA4 clock.
SYSCTL_PERIPH_CLK_PGA5 PGA5 clock.
SYSCTL_PERIPH_CLK_PGA6 PGA6 clock.
SYSCTL_PERIPH_CLK_PGA7 PGA7 clock.
SYSCTL_PERIPH_CLK_DACA DACA clock.
SYSCTL_PERIPH_CLK_DACB DACB clock.
SYSCTL_PERIPH_CLK_FSITXA FSITXA clock.
SYSCTL_PERIPH_CLK_FSIRXA FSIRXA clock.
SYSCTL_PERIPH_CLK_LINA LINA clock.
SYSCTL_PERIPH_CLK_PMBUSA PMBusA clock.
SYSCTL_PERIPH_CLK_DCC0 DCC0 clock.

33.2.3.2 enum **SysCtl_PeripheralSOFTPRES**

The following are values that can be passed to SysCtl_resetPeripheral() as the *peripheral* parameter.

Enumerator

SYSCTL_PERIPH_RES_CLA1 Reset CLA1.
SYSCTL_PERIPH_RES_EPWM1 Reset ePWM1.
SYSCTL_PERIPH_RES_EPWM2 Reset ePWM2.
SYSCTL_PERIPH_RES_EPWM3 Reset ePWM3.
SYSCTL_PERIPH_RES_EPWM4 Reset ePWM4.
SYSCTL_PERIPH_RES_EPWM5 Reset ePWM5.
SYSCTL_PERIPH_RES_EPWM6 Reset ePWM6.
SYSCTL_PERIPH_RES_EPWM7 Reset ePWM7.
SYSCTL_PERIPH_RES_EPWM8 Reset ePWM8.
SYSCTL_PERIPH_RES_ECAP1 Reset eCAP1.
SYSCTL_PERIPH_RES_ECAP2 Reset eCAP2.
SYSCTL_PERIPH_RES_ECAP3 Reset eCAP3.
SYSCTL_PERIPH_RES_ECAP4 Reset eCAP4.
SYSCTL_PERIPH_RES_ECAP5 Reset eCAP5.
SYSCTL_PERIPH_RES_ECAP6 Reset eCAP6.
SYSCTL_PERIPH_RES_ECAP7 Reset eCAP7.
SYSCTL_PERIPH_RES_EQEP1 Reset eQEP1.
SYSCTL_PERIPH_RES_EQEP2 Reset eQEP2.
SYSCTL_PERIPH_RES_SD1 Reset SDFM1.
SYSCTL_PERIPH_RES_SCIA Reset SCIA.
SYSCTL_PERIPH_RES_SCIB Reset SCIB.

SYSCTL_PERIPH_RES_SPIA Reset SPIA.
SYSCTL_PERIPH_RES_SPIB Reset SPIB.
SYSCTL_PERIPH_RES_I2CA Reset I2CA.
SYSCTL_PERIPH_RES_CANA Reset CANA.
SYSCTL_PERIPH_RES_CANB Reset CANB.
SYSCTL_PERIPH_RES_ADCA Reset ADCA.
SYSCTL_PERIPH_RES_ADCB Reset ADCB.
SYSCTL_PERIPH_RES_ADCC Reset ADCC.
SYSCTL_PERIPH_RES_CMPSS1 Reset CMPSS1.
SYSCTL_PERIPH_RES_CMPSS2 Reset CMPSS2.
SYSCTL_PERIPH_RES_CMPSS3 Reset CMPSS3.
SYSCTL_PERIPH_RES_CMPSS4 Reset CMPSS4.
SYSCTL_PERIPH_RES_CMPSS5 Reset CMPSS5.
SYSCTL_PERIPH_RES_CMPSS6 Reset CMPSS6.
SYSCTL_PERIPH_RES_CMPSS7 Reset CMPSS7.
SYSCTL_PERIPH_RES_PGA1 Reset PGA1.
SYSCTL_PERIPH_RES_PGA2 Reset PGA2.
SYSCTL_PERIPH_RES_PGA3 Reset PGA3.
SYSCTL_PERIPH_RES_PGA4 Reset PGA4.
SYSCTL_PERIPH_RES_PGA5 Reset PGA5.
SYSCTL_PERIPH_RES_PGA6 Reset PGA6.
SYSCTL_PERIPH_RES_PGA7 Reset PGA7.
SYSCTL_PERIPH_RES_DACA Reset DACA.
SYSCTL_PERIPH_RES_DACB Reset DACB.
SYSCTL_PERIPH_RES_FSITXA Reset FSITXA.
SYSCTL_PERIPH_RES_FSIRXA Reset FSIRXA.
SYSCTL_PERIPH_RES_LINA Reset LINA.
SYSCTL_PERIPH_RES_PMBUSA Reset PMBusA.

33.2.3.3 enum **SysCtl_WDPredivider**

The following are values that can be passed to SysCtl_setWatchdogPredivider() as the *predivider* parameter.

Enumerator

SYSCTL_WD_PREDIV_2 PREDIVCLK = INTOSC1 / 2.
SYSCTL_WD_PREDIV_4 PREDIVCLK = INTOSC1 / 4.
SYSCTL_WD_PREDIV_8 PREDIVCLK = INTOSC1 / 8.
SYSCTL_WD_PREDIV_16 PREDIVCLK = INTOSC1 / 16.
SYSCTL_WD_PREDIV_32 PREDIVCLK = INTOSC1 / 32.
SYSCTL_WD_PREDIV_64 PREDIVCLK = INTOSC1 / 64.
SYSCTL_WD_PREDIV_128 PREDIVCLK = INTOSC1 / 128.
SYSCTL_WD_PREDIV_256 PREDIVCLK = INTOSC1 / 256.
SYSCTL_WD_PREDIV_512 PREDIVCLK = INTOSC1 / 512.
SYSCTL_WD_PREDIV_1024 PREDIVCLK = INTOSC1 / 1024.
SYSCTL_WD_PREDIV_2048 PREDIVCLK = INTOSC1 / 2048.
SYSCTL_WD_PREDIV_4096 PREDIVCLK = INTOSC1 / 4096.

33.2.3.4 enum **SysCtl_WDPrescaler**

The following are values that can be passed to SysCtl_setWatchdogPrescaler() as the *prescaler* parameter.

Enumerator

SYSCTL_WD_PRESCALE_1 WDCLK = PREDIVCLK / 1.
SYSCTL_WD_PRESCALE_2 WDCLK = PREDIVCLK / 2.
SYSCTL_WD_PRESCALE_4 WDCLK = PREDIVCLK / 4.
SYSCTL_WD_PRESCALE_8 WDCLK = PREDIVCLK / 8.
SYSCTL_WD_PRESCALE_16 WDCLK = PREDIVCLK / 16.
SYSCTL_WD_PRESCALE_32 WDCLK = PREDIVCLK / 32.
SYSCTL_WD_PRESCALE_64 WDCLK = PREDIVCLK / 64.

33.2.3.5 enum **SysCtl_WDMode**

The following are values that can be passed to SysCtl_setWatchdogMode() as the *prescaler* parameter.

Enumerator

SYSCTL_WD_MODE_RESET Watchdog can generate a reset signal.
SYSCTL_WD_MODE_INTERRUPT Watchdog can generate an interrupt signal; reset signal is disabled.

33.2.3.6 enum **SysCtl_LSPCLKPrescaler**

The following are values that can be passed to SysCtl_setLowSpeedClock() as the *prescaler* parameter.

Enumerator

SYSCTL_LSPCLK_PRESCALE_1 LSPCLK = SYSCLK / 1.
SYSCTL_LSPCLK_PRESCALE_2 LSPCLK = SYSCLK / 2.
SYSCTL_LSPCLK_PRESCALE_4 LSPCLK = SYSCLK / 4 (default)
SYSCTL_LSPCLK_PRESCALE_6 LSPCLK = SYSCLK / 6.
SYSCTL_LSPCLK_PRESCALE_8 LSPCLK = SYSCLK / 8.
SYSCTL_LSPCLK_PRESCALE_10 LSPCLK = SYSCLK / 10.
SYSCTL_LSPCLK_PRESCALE_12 LSPCLK = SYSCLK / 12.
SYSCTL_LSPCLK_PRESCALE_14 LSPCLK = SYSCLK / 14.

33.2.3.7 enum **SysCtl_AccessPeripheral**

The following are values that can be passed to SysCtl_setPeripheralAccessControl() and SysCtl_getPeripheralAccessControl() as the *peripheral* parameter.

Enumerator

SYSCTL_ACCESS_ADCA ADCA access.
SYSCTL_ACCESS_ADCB ADCB access.

SYSCTL_ACCESS_ADCC ADCC access.
SYSCTL_ACCESS_CMPSS1 CMPSS1 access.
SYSCTL_ACCESS_CMPSS2 CMPSS2 access.
SYSCTL_ACCESS_CMPSS3 CMPSS3 access.
SYSCTL_ACCESS_CMPSS4 CMPSS4 access.
SYSCTL_ACCESS_CMPSS5 CMPSS5 access.
SYSCTL_ACCESS_CMPSS6 CMPSS6 access.
SYSCTL_ACCESS_CMPSS7 CMPSS7 access.
SYSCTL_ACCESS_DACA DACA access.
SYSCTL_ACCESS_DACB DACB access.
SYSCTL_ACCESS_PGA1 PGA1 access.
SYSCTL_ACCESS_PGA2 PGA2 access.
SYSCTL_ACCESS_PGA3 PGA3 access.
SYSCTL_ACCESS_PGA4 PGA4 access.
SYSCTL_ACCESS_PGA5 PGA5 access.
SYSCTL_ACCESS_PGA6 PGA6 access.
SYSCTL_ACCESS_PGA7 PGA7 access.
SYSCTL_ACCESS_EPWM1 ePWM1 access
SYSCTL_ACCESS_EPWM2 ePWM2 access
SYSCTL_ACCESS_EPWM3 ePWM3 access
SYSCTL_ACCESS_EPWM4 ePWM4 access
SYSCTL_ACCESS_EPWM5 ePWM5 access
SYSCTL_ACCESS_EPWM6 ePWM6 access
SYSCTL_ACCESS_EPWM7 ePWM7 access
SYSCTL_ACCESS_EPWM8 ePWM8 access
SYSCTL_ACCESS_EQEP1 eQEP1 access
SYSCTL_ACCESS_EQEP2 eQEP2 access
SYSCTL_ACCESS_ECAP1 eCAP1 access
SYSCTL_ACCESS_ECAP2 eCAP2 access
SYSCTL_ACCESS_ECAP3 eCAP3 access
SYSCTL_ACCESS_ECAP4 eCAP4 access
SYSCTL_ACCESS_ECAP5 eCAP5 access
SYSCTL_ACCESS_ECAP6 eCAP6 access
SYSCTL_ACCESS_ECAP7 eCAP7 access
SYSCTL_ACCESS_SDFM1 SDFM1 access.
SYSCTL_ACCESS_CLA1PROMCRC CLA1PROMCRC access.
SYSCTL_ACCESS_SPIA SPIA access.
SYSCTL_ACCESS_SPIB SPIB access.
SYSCTL_ACCESS_PMBUS_A PMBusA access.
SYSCTL_ACCESS_LIN_A LIN access.
SYSCTL_ACCESS_CANA CANA access.
SYSCTL_ACCESS_CANB CANB access.
SYSCTL_ACCESS_FSIATX FSITXA access.
SYSCTL_ACCESS_FSIARX FSIRXA access.
SYSCTL_ACCESS_HRPWM_A HRPWM access.

33.2.3.8 enum **SysCtl_AccessMaster**

The following are values that can be passed to SysCtl_setPeripheralAccessControl() and SysCtl_getPeripheralAccessControl() as the *master* parameter.

Enumerator

SYSCTL_ACCESS_CPU1 CPU access to the peripheral.

SYSCTL_ACCESS_CLA1 CLA1 access to the peripheral.

SYSCTL_ACCESS_DMA1 DMA access to the peripheral.

33.2.3.9 enum **SysCtl_AccessPermission**

The following are values that can be passed to SysCtl_setPeripheralAccessControl() as the *permission* parameter.

Enumerator

SYSCTL_ACCESS_FULL Full Access for both read and write.

SYSCTL_ACCESS_PROTECTED Protected RD access such that FIFOs. Clear on read, registers are not changed and no write access.

SYSCTL_ACCESS_NONE No read or write access.

33.2.3.10 enum **SysCtl_ClockOut**

The following are values that can be passed to SysCtl_selectClockOutSource() as the *source* parameter.

Enumerator

SYSCTL_CLOCKOUT_PLLSYS PLL System Clock.

SYSCTL_CLOCKOUT_PLLRAW PLL Raw Clock.

SYSCTL_CLOCKOUT_SYSCLK CPU System Clock.

SYSCTL_CLOCKOUT_INTOSC1 Internal Oscillator 1.

SYSCTL_CLOCKOUT_INTOSC2 Internal Oscillator 2.

SYSCTL_CLOCKOUT_XTALOSC External Oscillator.

33.2.3.11 enum **SysCtl_ExternalOscMode**

The following are values that can be passed to SysCtl_setExternalOscMode() as the *mode* parameter.

Enumerator

SYSCTL_XTALMODE_CRYSTAL XTAL Oscillator Crystal Mode.

SYSCTL_XTALMODE_SINGLE XTAL Oscillator Single-Ended Mode.

33.2.3.12 enum **SysCtl_SyncInput**

The following values define the *syncInput* parameter for SysCtl_setSyncInputConfig().

Enumerator

SYSCTL_SYNC_IN_EPWM1 Sync input to ePWM 1.
SYSCTL_SYNC_IN_EPWM4 Sync input to ePWM 4.
SYSCTL_SYNC_IN_EPWM7 Sync input to ePWM 7.
SYSCTL_SYNC_IN_ECAP1 Sync input to eCAP 1.
SYSCTL_SYNC_IN_ECAP4 Sync input to eCAP 4.
SYSCTL_SYNC_IN_ECAP6 Sync input to eCAP 6.

33.2.3.13 enum **SysCtl_SyncInputSource**

The following values define the *syncSrc* parameter for SysCtl_setSyncInputConfig(). Note that some of these are only valid for certain values of *syncInput*. See device technical reference manual for info on time-base counter synchronization for details.

Enumerator

SYSCTL_SYNC_IN_SRC_EPWM1SYNCOUT EPWM1SYNCOUT.
SYSCTL_SYNC_IN_SRC_EPWM4SYNCOUT EPWM4SYNCOUT.
SYSCTL_SYNC_IN_SRC_EPWM7SYNCOUT EPWM7SYNCOUT.
SYSCTL_SYNC_IN_SRC_ECAP1SYNCOUT ECAP1SYNCOUT.
SYSCTL_SYNC_IN_SRC_EXTSYNCIN1 EXTSYNCIN1–Valid for all values of syncInput.
SYSCTL_SYNC_IN_SRC_EXTSYNCIN2 EXTSYNCIN2–Valid for all values of syncInput except EPWM1.
SYSCTL_SYNC_IN_SRC_ECAP4SYNCOUT ECAP4SYNCOUT.

33.2.3.14 enum **SysCtl_SyncOutputSource**

The following values define the *syncSrc* parameter for SysCtl_setSyncOutputConfig().

Enumerator

SYSCTL_SYNC_OUT_SRC_EPWM1SYNCOUT EPWM1SYNCOUT –> EXTSYNCOUT.
SYSCTL_SYNC_OUT_SRC_EPWM4SYNCOUT EPWM4SYNCOUT –> EXTSYNCOUT.
SYSCTL_SYNC_OUT_SRC_EPWM7SYNCOUT EPWM7SYNCOUT –> EXTSYNCOUT.

33.2.3.15 enum **SysCtl_DeviceParametric**

The following values define the *parametric* parameter for SysCtl_getDeviceParametric().

Enumerator

SYSCTL_DEVICE_QUAL Device Qualification Status.
SYSCTL_DEVICE_PINCOUNT Device Pin Count.
SYSCTL_DEVICE_INSTASPIN Device InstaSPIN Feature Set.
SYSCTL_DEVICE_FLASH Device Flash size (KB)

SYSCTL_DEVICE_FAMILY Device Family.
SYSCTL_DEVICE_PARTNO Device Part Number.
SYSCTL_DEVICE_CLASSID Device Class ID.

33.2.4 Function Documentation

33.2.4.1 static void SysCtl_resetPeripheral (**SysCtl_PeripheralSOFTPRES** *peripheral*) [inline], [static]

Resets a peripheral

Parameters

peripheral	is the peripheral to reset.

This function uses the SOFTPRESx registers to reset a specified peripheral. Module registers will be returned to their reset states.

Note

This includes registers containing trim values.

Returns

None.

33.2.4.2 static void SysCtl_enablePeripheral (**SysCtl_PeripheralPCLOCKCR** *peripheral*) [inline], [static]

Enables a peripheral.

Parameters

peripheral	is the peripheral to enable.

Peripherals are enabled with this function. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

Returns

None.

Referenced by SysCtl_isPLLValid().

33.2.4.3 static void SysCtl_disablePeripheral (**SysCtl_PeripheralPCLOCKCR** *peripheral*) [inline], [static]

Disables a peripheral.

Parameters

peripheral	is the peripheral to disable.

Peripherals are disabled with this function. Once disabled, they will not operate or respond to register reads/writes.

Returns

None.

33.2.4.4 static void SysCtl_resetDevice (void) `[inline]`, `[static]`

Resets the device.

This function performs a watchdog reset of the device.

Returns

This function does not return.

33.2.4.5 static uint32_t SysCtl_getResetCause (void) `[inline]`, `[static]`

Gets the reason for a reset.

This function will return the reason(s) for a reset. Since the reset reasons are sticky until either cleared by software or an external reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason will be a logical OR of

- **SYSCTL_CAUSE_POR** - Power-on reset
- **SYSCTL_CAUSE_XRS** - External reset pin
- **SYSCTL_CAUSE_WDRS** - Watchdog reset
- **SYSCTL_CAUSE_NMIWDRS** - NMI watchdog reset
- **SYSCTL_CAUSE_SCCRESET** - SCCRESETn reset from DCSM

Note

If you re-purpose the reserved boot ROM RAM, the POR and XRS reset statuses won't be accurate.

Returns

Returns the reason(s) for a reset.

33.2.4.6 static void SysCtl_clearResetCause (uint32_t *rstCauses*) `[inline]`, `[static]`

Clears reset reasons.

Parameters

rstCauses	are the reset causes to be cleared; must be a logical OR of **SYSCTL_CAUSE_POR**, **SYSCTL_CAUSE_XRS**, **SYSCTL_CAUSE_WDRS**, **SYSCTL_CAUSE_NMIWDRS**, and/or **SYSCTL_CAUSE_SCCRESET**.

This function clears the specified sticky reset reasons. Once cleared, another reset for the same reason can be detected, and a reset for a different reason can be distinguished (instead of having two reset causes set). If the reset reason is used by an application, all reset causes should be cleared after they are retrieved with SysCtl_getResetCause().

Note

Some reset causes are cleared by the boot ROM.

Returns

None.

33.2.4.7 static void SysCtl_setLowSpeedClock (**SysCtl_LSPCLKPrescaler** *prescaler*) [inline], [static]

Sets the low speed peripheral clock rate prescaler.

Parameters

prescaler	is the LSPCLK rate relative to SYSCLK

This function configures the clock rate of the low speed peripherals. The *prescaler* parameter is the value by which the SYSCLK rate is divided to get the LSPCLK rate. For example, a *prescaler* of **SYSCTL_LSPCLK_PRESCALE_4** will result in a LSPCLK rate that is a quarter of the SYSCLK rate.

Returns

None.

33.2.4.8 static void SysCtl_selectClockOutSource (**SysCtl_ClockOut** *source*) [inline], [static]

Selects a clock source to mux to an external GPIO pin (XCLKOUT).

Parameters

source	is the internal clock source to be configured.

This function configures the specified clock source to be muxed to an external clock out (XCLKOUT) GPIO pin. The *source* parameter may take a value of one of the following values:

- **SYSCTL_CLOCKOUT_PLLSYS**
- **SYSCTL_CLOCKOUT_PLLRAW**
- **SYSCTL_CLOCKOUT_SYSCLK**
- **SYSCTL_CLOCKOUT_INTOSC1**
- **SYSCTL_CLOCKOUT_INTOSC2**
- **SYSCTL_CLOCKOUT_XTALOSC**

Returns
None.

33.2.4.9 static void SysCtl_setExternalOscMode (**SysCtl_ExternalOscMode** *mode*) [inline], [static]

Set the external oscillator mode.

Parameters

mode	is the external oscillator mode to be configured.

This function sets the external oscillator mode specified by the *mode* parameter which may take one of two values:

- **SYSCTL_XTALMODE_CRYSTAL** - Crystal Mode
- **SYSCTL_XTALMODE_SINGLE** - Single-Ended Mode

Note
The external oscillator must be powered off before this configuration can be performed.

Returns
None.

References SYSCTL_XTALMODE_CRYSTAL, and SYSCTL_XTALMODE_SINGLE.

33.2.4.10 static uint16_t SysCtl_getExternalOscCounterValue (void) [inline], [static]

Gets the external oscillator counter value.

This function returns the X1 clock counter value. When the return value reaches 0x3FF, it freezes. Before switching from INTOSC2 to an external oscillator (XTAL), an application should call this function to make sure the counter is saturated.

Returns
Returns the value of the 10-bit X1 clock counter.

33.2.4.11 static void SysCtl_clearExternalOscCounterValue (void) [inline], [static]

Clears the external oscillator counter value.

Returns
None.

33.2.4.12 static void SysCtl_turnOnOsc (uint32_t *oscSource*) [inline], [static]

Turns on the specified oscillator sources.

Parameters

oscSource	is the oscillator source to be configured.

This function turns on the oscillator specified by the *oscSource* parameter which may take a value of **SYSCTL_OSCSRC_OSC2** or **SYSCTL_OSCSRC_XTAL**.

Note

SYSCTL_OSCSRC_OSC1 is not a valid value for *oscSource*.

Returns

None.

33.2.4.13 static void SysCtl_turnOffOsc (uint32_t *oscSource*) [inline], [static]

Turns off the specified oscillator sources.

Parameters

oscSource	is the oscillator source to be configured.

This function turns off the oscillator specified by the *oscSource* parameter which may take a value of **SYSCTL_OSCSRC_OSC2** or **SYSCTL_OSCSRC_XTAL**.

Note

SYSCTL_OSCSRC_OSC1 is not a valid value for *oscSource*.

Returns

None.

33.2.4.14 static void SysCtl_enterIdleMode (void) [inline], [static]

Enters IDLE mode.

This function puts the device into IDLE mode. The CPU clock is gated while all peripheral clocks are left running. Any enabled interrupt will wake the CPU up from IDLE mode.

Returns

None.

33.2.4.15 static void SysCtl_enterHaltMode (void) [inline], [static]

Enters HALT mode.

This function puts the device into HALT mode. This will gate almost all systems and clocks and allows for the power-down of oscillators and analog blocks. The watchdog may be left clocked to produce a reset. See SysCtl_enableWatchdogInHalt() to enable this. GPIOs should be configured to wake the CPU subsystem. See SysCtl_enableLPMWakeupPin().

The CPU will receive an interrupt (WAKEINT) on wakeup.

Returns

None.

33.2.4.16 static void SysCtl_enableLPMWakeupPin (uint32_t *pin*) `[inline]`, `[static]`

Enables a pin to wake up the device from HALT.

Parameters

pin	is the identifying number of the pin.

This function connects a pin to the LPM circuit, allowing an event on the pin to wake up the device when when it is in HALT mode.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*. Only GPIOs 0 through 63 are capable of being connected to the LPM circuit.

Returns

None.

33.2.4.17 static void SysCtl_disableLPMWakeupPin (uint32_t *pin*) `[inline]`, `[static]`

Disables a pin to wake up the device from HALT.

Parameters

pin	is the identifying number of the pin.

This function disconnects a pin to the LPM circuit, disallowing an event on the pin to wake up the device when when it is in HALT mode.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*. Only GPIOs 0 through 63 are valid.

Returns

None.

33.2.4.18 static void SysCtl_enableWatchdogInHalt (void) `[inline]`, `[static]`

Enable the watchdog to run while in HALT mode.

This function configures the watchdog to continue to run while in HALT mode. Additionally, INTOSC1 and INTOSC2 are not powered down when the system enters HALT mode. By default the watchdog is gated when the system enters HALT.

Returns

None.

33.2.4.19 static void SysCtl_disableWatchdogInHalt (void) [inline], [static]

Disable the watchdog from running while in HALT mode.

This function gates the watchdog when the system enters HALT mode. INTOSC1 and INTOSC2 will be powered down. This is the default behavior of the device.

Returns
None.

33.2.4.20 static void SysCtl_setWatchdogMode (**SysCtl_WDMode** *mode*) [inline], [static]

Configures whether the watchdog generates a reset or an interrupt signal.

Parameters

mode	is a flag to select the watchdog mode.

This function configures the action taken when the watchdog counter reaches its maximum value. When the *mode* parameter is **SYSCTL_WD_MODE_INTERRUPT**, the watchdog is enabled to generate a watchdog interrupt signal and disables the generation of a reset signal. This will allow the watchdog module to wake up the device from IDLE.

When the *mode* parameter is **SYSCTL_WD_MODE_RESET**, the watchdog will be put into reset mode and generation of a watchdog interrupt signal will be disabled. This is how the watchdog is configured by default.

Note
Check the status of the watchdog interrupt using SysCtl_isWatchdogInterruptActive() before calling this function. If the interrupt is still active, switching from interrupt mode to reset mode will immediately reset the device.

Returns
None.

References SYSCTL_WD_MODE_INTERRUPT.

33.2.4.21 static bool SysCtl_isWatchdogInterruptActive (void) [inline], [static]

Gets the status of the watchdog interrupt signal.

This function returns the status of the watchdog interrupt signal. If the interrupt is active, this function will return **true**. If **false**, the interrupt is NOT active.

Note
Make sure to call this function to ensure that the interrupt is not active before making any changes to the configuration of the watchdog to prevent any unexpected behavior. For instance, switching from interrupt mode to reset mode while the interrupt is active will immediately reset the device.

Returns
true if the interrupt is active and **false** if it is not.

33.2.4.22 static void SysCtl_disableWatchdog (void) [inline], [static]

Disables the watchdog.

This function disables the watchdog timer. Note that the watchdog timer is enabled on reset.

Returns
None.

33.2.4.23 static void SysCtl_enableWatchdog (void) [inline], [static]

Enables the watchdog.

This function enables the watchdog timer. Note that the watchdog timer is enabled on reset.

Returns
None.

33.2.4.24 static void SysCtl_serviceWatchdog (void) [inline], [static]

Services the watchdog.

This function resets the watchdog.

Returns
None.

33.2.4.25 static void SysCtl_setWatchdogPredivider (**SysCtl_WDPredivider** *predivider*) [inline], [static]

Sets up watchdog clock (WDCLK) pre-divider.

Parameters

predivider	is the value that configures the pre-divider.

This function sets up the watchdog clock (WDCLK) pre-divider. There are two dividers that scale INTOSC1 to WDCLK. The *predivider* parameter divides INTOSC1 down to PREDIVCLK and the prescaler (set by the SysCtl_setWatchdogPrescaler() function) divides PREDIVCLK down to WDCLK.

Returns
None.

33.2.4.26 static void SysCtl_setWatchdogPrescaler (**SysCtl_WDPrescaler** *prescaler*) `[inline]`, `[static]`

Sets up watchdog clock (WDCLK) prescaler.

Parameters

prescaler	is the value that configures the watchdog clock relative to the value from the pre-divider.

This function sets up the watchdog clock (WDCLK) prescaler. There are two dividers that scale INTOSC1 to WDCLK. The predivider (set with the SysCtl_setWatchdogPredivider() function) divides INTOSC1 down to PREDIVCLK and the *prescaler* parameter divides PREDIVCLK down to WDCLK.

Returns
None.

33.2.4.27 static uint16_t SysCtl_getWatchdogCounterValue (void) [inline], [static]

Gets the watchdog counter value.

Returns
Returns the current value of the 8-bit watchdog counter. If this count value overflows, a watchdog output pulse is generated.

33.2.4.28 static bool SysCtl_getWatchdogResetStatus (void) [inline], [static]

Gets the watchdog reset status.

This function returns the watchdog reset status. If this function returns **true**, that indicates that a watchdog reset generated the last reset condition. Otherwise, it was an external device or power-up reset condition.

Returns
Returns **true** if the watchdog generated the last reset condition.

33.2.4.29 static void SysCtl_clearWatchdogResetStatus (void) [inline], [static]

Clears the watchdog reset status.

This function clears the watchdog reset status. To check if it was set first, see SysCtl_getWatchdogResetStatus().

Returns
None.

33.2.4.30 static void SysCtl_setWatchdogWindowValue (uint16_t *value*) [inline], [static]

Set the minimum threshold value for windowed watchdog

Parameters

value	is the value to set the window threshold

This function sets the minimum threshold value used to define the lower limit of the windowed watchdog functionality.

Returns

None.

33.2.4.31 static bool SysCtl_getNMIStatus (void) [inline], [static]

Read NMI interrupts.

Read the current state of NMI interrupt.

Returns

true if NMI interrupt is triggered, **false** if not.

33.2.4.32 static uint32_t SysCtl_getNMIFlagStatus (void) [inline], [static]

Read NMI Flags.

Read the current state of individual NMI interrupts

Returns

Value of NMIFLG register. These defines are provided to decode the value:

- **SYSCTL_NMI_NMIINT** - Non-maskable interrupt
- **SYSCTL_NMI_CLOCKFAIL** - Clock Failure
- **SYSCTL_NMI_RAMUNCERR** - Uncorrectable RAM error
- **SYSCTL_NMI_FLUNCERR** - Uncorrectable Flash error
- **SYSCTL_NMI_PIEVECTERR** - PIE Vector Fetch Error
- **SYSCTL_NMI_SWERR** - SW Error Force NMI Flag

Referenced by SysCtl_clearAllNMIFlags().

33.2.4.33 static bool SysCtl_isNMIFlagSet (uint32_t *nmiFlags*) [inline], [static]

Check if the individual NMI interrupts are set.

Parameters

nmiFlags	Bit mask of the NMI interrupts that user wants to clear. The bit format of this parameter is same as of the NMIFLG register. These defines are provided: ■ **SYSCTL_NMI_NMIINT** - Non-maskable interrupt ■ **SYSCTL_NMI_CLOCKFAIL** - Clock Failure ■ **SYSCTL_NMI_RAMUNCERR** - Uncorrectable RAM error ■ **SYSCTL_NMI_FLUNCERR** - Uncorrectable Flash error ■ **SYSCTL_NMI_PIEVECTERR** - PIE Vector Fetch Error ■ **SYSCTL_NMI_SWERR** - SW Error Force NMI Flag

Check if interrupt flags corresponding to the passed in bit mask are asserted.

Returns

true if any of the NMI asked for in the parameter bit mask is set. **false** if none of the NMI requested in the parameter bit mask are set.

33.2.4.34 static void SysCtl_clearNMIStatus (uint32_t *nmiFlags*) `[inline]`, `[static]`

Function to clear individual NMI interrupts.

Parameters

nmiFlags	Bit mask of the NMI interrupts that user wants to clear. The bit format of this parameter is same as of the NMIFLG register. These defines are provided: ■ **SYSCTL_NMI_CLOCKFAIL** ■ **SYSCTL_NMI_RAMUNCERR** ■ **SYSCTL_NMI_FLUNCERR** ■ **SYSCTL_NMI_PIEVECTERR** ■ **SYSCTL_NMI_SWERR**

Clear NMI interrupt flags that correspond with the passed in bit mask.

Note: The NMI Interrupt flag is always cleared by default and therefore doesn't have to be included in the bit mask.

Returns

None.

33.2.4.35 static void SysCtl_clearAllNMIFlags (void) `[inline]`, `[static]`

Clear all the NMI Flags that are currently set.

Returns

None.

References SysCtl_getNMIFlagStatus().

33.2.4.36 static void SysCtl_forceNMIFlags (uint32_t *nmiFlags*) [inline], [static]

Function to force individual NMI interrupt fail flags

Parameters

nmiFlags	Bit mask of the NMI interrupts that user wants to clear. The bit format of this parameter is same as of the NMIFLG register. These defines are provided: ■ **SYSCTL_NMI_CLOCKFAIL** ■ **SYSCTL_NMI_RAMUNCERR** ■ **SYSCTL_NMI_FLUNCERR** ■ **SYSCTL_NMI_PIEVECTERR** ■ **SYSCTL_NMI_SWERR**

Returns
None.

33.2.4.37 static uint16_t SysCtl_getNMIWatchdogCounter (void) [inline], [static]

Gets the NMI watchdog counter value.

Note: The counter is clocked at the SYSCLKOUT rate.

Returns
Returns the NMI watchdog counter register's current value.

33.2.4.38 static void SysCtl_setNMIWatchdogPeriod (uint16_t *wdPeriod*) [inline], [static]

Sets the NMI watchdog period value.

Parameters

wdPeriod	is the 16-bit value at which a reset is generated.

This function writes to the NMI watchdog period register that holds the value to which the NMI watchdog counter is compared. When the two registers match, a reset is generated. By default, the period is 0xFFFF.

Note
If a value smaller than the current counter value is passed into the *wdPeriod* parameter, a NMIRSn will be forced.

Returns
None.

33.2.4.39 static uint16_t SysCtl_getNMIWatchdogPeriod (void) [inline], [static]

Gets the NMI watchdog period value.

Returns
Returns the NMI watchdog period register's current value.

33.2.4.40 static uint32_t SysCtl_getNMIShadowFlagStatus (void) [inline], [static]

Read NMI Shadow Flags.

Read the current state of individual NMI interrupts

Returns
Value of NMISHDFLG register. These defines are provided to decode the value:
- **SYSCTL_NMI_NMIINT** - Non-maskable interrupt
- **SYSCTL_NMI_CLOCKFAIL** - Clock Failure
- **SYSCTL_NMI_RAMUNCERR** - Uncorrectable RAM error
- **SYSCTL_NMI_FLUNCERR** - Uncorrectable Flash error
- **SYSCTL_NMI_PIEVECTERR** - PIE Vector Fetch Error
- **SYSCTL_NMI_SWERR** - SW Error Force NMI Flag

33.2.4.41 static bool SysCtl_isNMIShadowFlagSet (uint32_t *nmiFlags*) [inline], [static]

Check if the individual NMI shadow flags are set.

Parameters

nmiFlags	Bit mask of the NMI interrupts that user wants to clear. The bit format of this parameter is same as of the NMIFLG register. These defines are provided: ■ **SYSCTL_NMI_NMIINT** ■ **SYSCTL_NMI_CLOCKFAIL** ■ **SYSCTL_NMI_RAMUNCERR** ■ **SYSCTL_NMI_FLUNCERR** ■ **SYSCTL_NMI_PIEVECTERR** ■ **SYSCTL_NMI_SWERR**

Check if interrupt flags corresponding to the passed in bit mask are asserted.

Returns
true if any of the NMI asked for in the parameter bit mask is set. **false** if none of the NMI requested in the parameter bit mask are set.

33.2.4.42 static void SysCtl_enableMCD (void) [inline], [static]

Enable the missing clock detection (MCD) Logic

Returns
None.

33.2.4.43 static void SysCtl_disableMCD (void) [inline], [static]

Disable the missing clock detection (MCD) Logic

Returns
None.

33.2.4.44 static bool SysCtl_isMCDClockFailureDetected (void) [inline], [static]

Get the missing clock detection Failure Status

Note
A failure means the oscillator clock is missing

Returns
Returns **true** if a failure is detected or **false** if a failure isn't detected

Referenced by SysCtl_getClock(), SysCtl_selectXTAL(), SysCtl_selectXTALSingleEnded(), and SysCtl_setClock().

33.2.4.45 static void SysCtl_resetMCD (void) [inline], [static]

Reset the missing clock detection logic after clock failure

Returns
None.

Referenced by SysCtl_selectXTAL().

33.2.4.46 static void SysCtl_connectMCDClockSource (void) [inline], [static]

Re-connect missing clock detection clock source to stop simulating clock failure

Returns
None.

33.2.4.47 static void SysCtl_disconnectMCDClockSource (void) [inline], [static]

Disconnect missing clock detection clock source to simulate clock failure. This is for testing the MCD functionality.

Returns
None.

33.2.4.48 static void SysCtl_lockAccessControlRegs (void) [inline], [static]

Lock the Access Control Registers

This function locks the access control registers and puts them in a read-only state.

Note
Only a reset can unlock the access control registers.

Returns
None.

33.2.4.49 static void SysCtl_setPeripheralAccessControl (**SysCtl_AccessPeripheral** *peripheral,* **SysCtl_AccessMaster** *master,* **SysCtl_AccessPermission** *permission*) [inline], [static]

Set the peripheral access control permissions

Parameters

peripheral	is the selected peripheral
master	is the selected master (CPU1, CLA1, or DMA1)
permission	is the selected access permissions

This function sets the specified peripheral access control permissions for the the specified master (CPU1, CLA1, or DMA1)

The *peripheral* parameter can have one enumerated value in the format of **SYSCTL_ACCESS_X** where X is the name of the peripheral instance to be configured such as **SYSCTL_ACCESS_ADCA**.

The *master* parameter can have one the following enumerated values:

- **SYSCTL_ACCESS_CPU1** - CPU1 Master
- **SYSCTL_ACCESS_CLA1** - CLA1 Master
- **SYSCTL_ACCESS_DMA1** - DMA1 Master

The *permission* parameter can have one the following enumerated values:

- **SYSCTL_ACCESS_FULL** - Full Access for both read and write
- **SYSCTL_ACCESS_PROTECTED** - Protected read access such that FIFOs, clear on read registers are not changed, and no write access
- **SYSCTL_ACCESS_NONE** - No read or write access

Returns
None.

33.2.4.50 static uint32_t SysCtl_getPeripheralAccessControl (**SysCtl_AccessPeripheral** *peripheral,* **SysCtl_AccessMaster** *master*) [inline], [static]

Get the peripheral access control permissions

Parameters

peripheral	is the selected peripheral
master	is the selected master (CPU1, CLA1, or DMA1)

This function gets the specified peripheral access control permissions for the the specified master (CPU1, CLA1, or DMA1)

The *peripheral* parameter can have one enumerated value in the format of **SYSCTL_ACCESS_X** where X is the name of the peripheral instance to be configured such as **SYSCTL_ACCESS_ADCA**.

The *master* parameter can have one the following enumerated values:

- **SYSCTL_ACCESS_CPU1** - CPU1 Master
- **SYSCTL_ACCESS_CLA1** - CLA1 Master
- **SYSCTL_ACCESS_DMA1** - DMA1 Master

Returns

Returns one of the following enumerated permission values:
- **SYSCTL_ACCESS_FULL** - Full Access for both read and write
- **SYSCTL_ACCESS_PROTECTED** - Protected read access such that FIFOs, clear on read registers are not changed, and no write access
- **SYSCTL_ACCESS_NONE** - No read or write access

33.2.4.51 static void SysCtl_setSyncInputConfig (**SysCtl_SyncInput** *syncInput,* **SysCtl_SyncInputSource** *syncSrc*) [inline], [static]

Configures the sync input source for the ePWM and eCAP signals.

Parameters

syncInput	is the sync input being configured
syncSrc	is sync input source selection.

This function configures the sync input source for the ePWM and eCAP modules. The *syncInput* parameter is the sync input being configured. It should be passed a value of **SYSCTL_SYNC_IN_XXXX**, where XXXX is the ePWM or eCAP instance the sync signal is entering.

The *syncSrc* parameter is the sync signal selected as the source of the sync input. It should be passed a value of **SYSCTL_SYNC_IN_SRC_XXXX**, XXXX is a sync signal coming from an ePWM, eCAP or external sync output. where For example, a *syncInput* value of **SYSCTL_SYNC_IN_ECAP1** and a *syncSrc* value of **SYSCTL_SYNC_IN_SRC_EPWM1SYNCOUT** will make the EPWM1SYNCOUT signal drive eCAP1's SYNCIN signal.

Note that some *syncSrc* values are only valid for certain values of *syncInput*. See device technical reference manual for details on time-base counter synchronization.

Returns

None.

References SYSCTL_SYNC_IN_EPWM1.

33.2.4.52 static void SysCtl_setSyncOutputConfig (**SysCtl_SyncOutputSource** *syncSrc*) [inline], [static]

Configures the sync output source.

Parameters

syncSrc	is sync output source selection.

This function configures the sync output source from the ePWM modules. The *syncSrc* parameter is a value **SYSCTL_SYNC_OUT_SRC_XXXX**, where XXXX is a sync signal coming from an ePWM such as SYSCTL_SYNC_OUT_SRC_EPWM1SYNCOUT

Returns
None.

33.2.4.53 static void SysCtl_enableExtADCSOCSource (uint32_t *adcsocSrc*) [inline], [static]

Enables ePWM SOC signals to drive an external (off-chip) ADCSOC signal.

Parameters

adcsocSrc	is a bit field of the selected signals to be enabled

This function configures which ePWM SOC signals are enabled as a source for either ADCSOCAO or ADCSOCBO. The *adcsocSrc* parameter takes a logical OR of **SYSCTL_ADCSOC_SRC_PWMxSOCA**/**B** values that correspond to different signals.

Returns
None.

33.2.4.54 static void SysCtl_disableExtADCSOCSource (uint32_t *adcsocSrc*) [inline], [static]

Disables ePWM SOC signals from driving an external ADCSOC signal.

Parameters

adcsocSrc	is a bit field of the selected signals to be disabled

This function configures which ePWM SOC signals are disabled as a source for either ADCSOCAO or ADCSOCBO. The *adcsocSrc* parameter takes a logical OR of **SYSCTL_ADCSOC_SRC_PWMxSOCA**/**B** values that correspond to different signals.

Returns
None.

33.2.4.55 static void SysCtl_lockExtADCSOCSelect (void) [inline], [static]

Locks the SOC Select of the Trig X-BAR.

This function locks the external ADC SOC select of the Trig X-BAR.

Returns
None.

33.2.4.56 static void SysCtl_lockSyncSelect (void) [inline], [static]

Locks the Sync Select of the Trig X-BAR.

This function locks Sync Input and Output Select of the Trig X-BAR.

Returns
None.

33.2.4.57 static uint32_t SysCtl_getDeviceRevision (void) [inline], [static]

Get the Device Silicon Revision ID

This function returns the silicon revision ID for the device.

Returns
Returns the silicon revision ID value.

33.2.4.58 void SysCtl_delay (uint32_t *count*)

Delays for a fixed number of cycles.

Parameters

count	is the number of delay loop iterations to perform.

This function generates a constant length delay using assembly code. The loop takes 5 cycles per iteration plus 9 cycles of overhead.

Note
If count is equal to zero, the loop will underflow and run for a very long time.

Returns
None.

Referenced by CAN_initModule(), and SysCtl_setClock().

33.2.4.59 uint32_t SysCtl_getClock (uint32_t *clockInHz*)

Calculates the system clock frequency (SYSCLK).

Parameters

clockInHz	is the frequency of the oscillator clock source (OSCCLK).

This function determines the frequency of the system clock based on the frequency of the oscillator clock source (from *clockInHz*) and the PLL and clock divider configuration registers.

Returns

Returns the system clock frequency. If a missing clock is detected, the function will return the INTOSC1 frequency. This needs to be corrected and cleared (see SysCtl_resetMCD()) before trying to call this function again.

References SysCtl_isMCDClockFailureDetected().

Referenced by SysCtl_getLowSpeedClock().

33.2.4.60 bool SysCtl_setClock (uint32_t *config*)

Configures the clocking of the device.

Parameters

config	is the required configuration of the device clocking.

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The *config* parameter is the OR of several different values, many of which are grouped into sets where only one can be chosen.

- The system clock divider is chosen with the macro **SYSCTL_SYSDIV(x)** where x is either 1 or an even value up to 126.
- The use of the PLL is chosen with either **SYSCTL_PLL_ENABLE** or **SYSCTL_PLL_DISABLE**.
- The integer multiplier is chosen **SYSCTL_IMULT(x)** where x is a value from 1 to 127.
- The fractional multiplier is chosen with either **SYSCTL_FMULT_0**, **SYSCTL_FMULT_1_4**, **SYSCTL_FMULT_1_2**, or **SYSCTL_FMULT_3_4**.
- The oscillator source chosen with **SYSCTL_OSCSRC_OSC2**, **SYSCTL_OSCSRC_XTAL**, **SYSCTL_OSCSRC_XTAL_SE** or **SYSCTL_OSCSRC_OSC1**.

This function uses the DCC to check that the PLLRAWCLK is running at the expected rate. If you are using the DCC, you must back up its configuration before calling this function and restore it afterward.

Note

See your device errata for more details about locking the PLL.

Returns

Returns **false** if a missing clock error is detected. This needs to be cleared (see SysCtl_resetMCD()) before trying to call this function again. Also, returns **false** if the PLLRAWCLK is not running and its expected rate after **SYSCTL_PLL_RETRIES** retries. Otherwise, returns **true**.

References SysCtl_delay(), SysCtl_isMCDClockFailureDetected(), SysCtl_isPLLValid(), and SysCtl_selectOscSource().

33.2.4.61 bool SysCtl_isPLLValid (uint32_t *oscSource*, uint32_t *pllMult*)

Validates PLL Raw Clock Frequency (PLLRAWCLK)

Parameters

oscSource	is the Clock Source for the PLL that is also used for DCC
pllMult	has the PLL Multiplier Register configuration which include integer and fractional multiplier used to configure the DCC Counter1 clock

This function uses DCC module to validate the PLL clock frequency. It uses oscSource as a reference clock for DCC, and PLL is used as clock under test. As long as the Counter0 (running of oscSource) & Counter1 (running of PLL) expire at the same time, DCC will not generate an Error. This function gives 100 attempts for PLL to lock and make sure frequency is as expected.

Note

This function does not validate if PLL output frequency (PLLRAWCLK) is within the operating range as per the datasheet.

- The *oscSource* parameter is the oscillator source chosen with **SYSCTL_OSCSRC_OSC2**, **SYSCTL_OSCSRC_XTAL**, **SYSCTL_OSCSRC_XTAL_SE** or **SYSCTL_OSCSRC_OSC1**.
- The *pllMult* parameter is a bitwise OR of **SYSCTL_IMULT(x)** where x is a value from 1 to 127 and one of the following fractional values: **SYSCTL_FMULT_0**, **SYSCTL_FMULT_1_4**, **SYSCTL_FMULT_1_2**, or **SYSCTL_FMULT_3_4**.

Returns

Returns **true** if the DCCSTATUS error flag is not set. Otherwise, returns **false**.

References DCC_COUNT0SRC_INTOSC1, DCC_COUNT0SRC_INTOSC2, DCC_COUNT0SRC_XTAL, DCC_COUNT1SRC_PLL, DCC_disableErrorSignal(), DCC_disableModule(), DCC_enableDoneSignal(), DCC_enableErrorSignal(), DCC_enableModule(), DCC_enableSingleShotMode(), DCC_MODE_COUNTER_ZERO, DCC_setCounter0ClkSource(), DCC_setCounter1ClkSource(), DCC_setCounterSeeds(), SysCtl_enablePeripheral(), and SYSCTL_PERIPH_CLK_DCC0.

Referenced by SysCtl_setClock().

33.2.4.62 void SysCtl_selectXTAL (void)

Configures the external oscillator for the clocking of the device.

This function configures the external oscillator (XTAL) to be used for the clocking of the device in crystal mode. It follows the procedure to turn on the oscillator, wait for it to power up, and select it as the source of the system clock.

Please note that this function blocks while it waits for the XTAL to power up. If the XTAL does not manage to power up properly, the function will loop for a long time. It is recommended that you modify this function to add an appropriate timeout and error-handling procedure.

Returns

None.

References SysCtl_isMCDClockFailureDetected(), and SysCtl_resetMCD().

Referenced by SysCtl_selectOscSource().

33.2.4.63 void SysCtl_selectXTALSingleEnded (void)

Configures the external oscillator for the clocking of the device in single-ended mode.

This function configures the external oscillator (XTAL) to be used for the clocking of the device in single-ended mode. It follows the procedure to turn on the oscillator, wait for it to power up, and select it as the source of the system clock.

Please note that this function blocks while it waits for the XTAL to power up. If the XTAL does not manage to power up properly, the function will loop for a long time. It is recommended that you modify this function to add an appropriate timeout and error-handling procedure.

Returns
None.

References SysCtl_isMCDClockFailureDetected().

Referenced by SysCtl_selectOscSource().

33.2.4.64 void SysCtl_selectOscSource (uint32_t *oscSource*)

Selects the oscillator to be used for the clocking of the device.

Parameters

oscSource	is the oscillator source to be configured.

This function configures the oscillator to be used in the clocking of the device. The *oscSource* parameter may take a value of **SYSCTL_OSCSRC_OSC2**, **SYSCTL_OSCSRC_XTAL**, **SYSCTL_OSCSRC_XTAL_SE**, or **SYSCTL_OSCSRC_OSC1**.

See Also
SysCtl_turnOnOsc()

Returns
None.

References SysCtl_selectXTAL(), and SysCtl_selectXTALSingleEnded().

Referenced by SysCtl_setClock().

33.2.4.65 uint32_t SysCtl_getLowSpeedClock (uint32_t *clockInHz*)

Calculates the low-speed peripheral clock frequency (LSPCLK).

Parameters

clockInHz	is the frequency of the oscillator clock source (OSCCLK).

This function determines the frequency of the low-speed peripheral clock based on the frequency of the oscillator clock source (from *clockInHz*) and the PLL and clock divider configuration registers.

Returns

Returns the low-speed peripheral clock frequency.

References SysCtl_getClock().

33.2.4.66 uint16_t SysCtl_getDeviceParametric (**SysCtl_DeviceParametric** *parametric*)

Get the device part parametric value

Parameters

parametric	is the requested device parametric value

This function gets the device part parametric value.

The *parametric* parameter can have one the following enumerated values:

- **SYSCTL_DEVICE_QUAL** - Device Qualification Status
- **SYSCTL_DEVICE_PINCOUNT** - Device Pin Count
- **SYSCTL_DEVICE_INSTASPIN** - Device InstaSPIN Feature Set
- **SYSCTL_DEVICE_FLASH** - Device Flash size (KB)
- **SYSCTL_DEVICE_FAMILY** - Device Family
- **SYSCTL_DEVICE_PARTNO** - Device Part Number
- **SYSCTL_DEVICE_CLASSID** - Device Class ID

Returns

Returns the specified parametric value.

References SYSCTL_DEVICE_CLASSID, SYSCTL_DEVICE_FAMILY, SYSCTL_DEVICE_FLASH, SYSCTL_DEVICE_INSTASPIN, SYSCTL_DEVICE_PARTNO, SYSCTL_DEVICE_PINCOUNT, and SYSCTL_DEVICE_QUAL.

34 Version Module

Introduction 666
API Functions 666

34.1 Version Introduction

The version driver provides a function which can be used to check the version number of the driverlib.lib that is in use.

34.2 API Functions

Macros

- #define VERSION_NUMBER

Functions

- uint32_t Version_getLibVersion (void)

34.2.1 Detailed Description

The code for this module is contained in `driverlib/version.c`, with `driverlib/version.h` containing the API declarations for use by applications.

34.2.2 Macro Definition Documentation

34.2.2.1 #define VERSION_NUMBER

Version number to be returned by Version_getLibVersion()

Referenced by Version_getLibVersion().

34.2.3 Function Documentation

34.2.3.1 uint32_t Version_getLibVersion (void)

Returns the driverlib version number

This function can be used to check the version number of the driverlib.lib that is in use. The version number will take the format x.xx.xx.xx, so for example, if the function returns 2100200, the driverlib version being used is 2.10.02.00.

Returns

Returns an integer value indicating the driverlib version.

References VERSION_NUMBER.

35 X-BAR Module

Introduction 668
API Functions 668

35.1 X-BAR Introduction

The crossbar or X-BAR API is a set of functions to configure the three X-BARs on the device–the Input X-BAR, the Output X-BAR, and the ePWM X-BAR. The X-BARs route both signals from pins and internal signals from IP blocks to a degree beyond what is possible with GPIO muxing alone. Functions are provided by the API to configure the various muxes, enable and disable signals, and lock in the configurations selected.

35.2 API Functions

Enumerations

- enum XBAR_OutputNum {
  XBAR_OUTPUT1, XBAR_OUTPUT2, XBAR_OUTPUT3, XBAR_OUTPUT4,
  XBAR_OUTPUT5, XBAR_OUTPUT6, XBAR_OUTPUT7, XBAR_OUTPUT8 }
- enum XBAR_TripNum {
  XBAR_TRIP4, XBAR_TRIP5, XBAR_TRIP7, XBAR_TRIP8,
  XBAR_TRIP9, XBAR_TRIP10, XBAR_TRIP11, XBAR_TRIP12 }
- enum XBAR_InputNum {
  XBAR_INPUT1, XBAR_INPUT2, XBAR_INPUT3, XBAR_INPUT4,
  XBAR_INPUT5, XBAR_INPUT6, XBAR_INPUT7, XBAR_INPUT8,
  XBAR_INPUT9, XBAR_INPUT10, XBAR_INPUT11, XBAR_INPUT12,
  XBAR_INPUT13, XBAR_INPUT14, XBAR_INPUT15, XBAR_INPUT16 }

Functions

- static void XBAR_enableOutputMux (XBAR_OutputNum output, uint32_t muxes)
- static void XBAR_disableOutputMux (XBAR_OutputNum output, uint32_t muxes)
- static void XBAR_setOutputLatchMode (XBAR_OutputNum output, bool enable)
- static bool XBAR_getOutputLatchStatus (XBAR_OutputNum output)
- static void XBAR_clearOutputLatch (XBAR_OutputNum output)
- static void XBAR_forceOutputLatch (XBAR_OutputNum output)
- static void XBAR_invertOutputSignal (XBAR_OutputNum output, bool invert)
- static void XBAR_enableEPWMMux (XBAR_TripNum trip, uint32_t muxes)
- static void XBAR_disableEPWMMux (XBAR_TripNum trip, uint32_t muxes)
- static void XBAR_invertEPWMSignal (XBAR_TripNum trip, bool invert)
- static void XBAR_setInputPin (XBAR_InputNum input, uint16_t pin)
- static void XBAR_lockInput (XBAR_InputNum input)
- static void XBAR_lockOutput (void)
- static void XBAR_lockEPWM (void)
- void XBAR_setOutputMuxConfig (XBAR_OutputNum output, XBAR_OutputMuxConfig muxConfig)

- void XBAR_setEPWMMuxConfig (XBAR_TripNum trip, XBAR_EPWMMuxConfig muxConfig)
- bool XBAR_getInputFlagStatus (XBAR_InputFlag inputFlag)
- void XBAR_clearInputFlag (XBAR_InputFlag inputFlag)

35.2.1 Detailed Description

The functions used to configure the ePWM and the Output X-BAR are identifiable as their names will either contain the word EPWM or Output. Both of these X-BARs have multiple output signals that have 32 associated muxes. The select signal of these muxes is configured using the XBAR_setEPWMMuxConfig() and XBAR_setOutputMuxConfig() functions. Each of these mux signals can be enabled and disabled before they are logically OR'd together to arrive at the output signal using XBAR_enableOutputMux() and XBAR_disableOutputMux() and XBAR_enableEPWMMux() and XBAR_disableEPWMMux().

The functions XBAR_getInputFlagStatus() and XBAR_clearInputFlag(), despite their names, are not related to the Input X-BAR. They provide a way to get and clear the status of the signals that are inputs to the ePWM and Output X-BARs. Since these two X-BARs share nearly all of their inputs, they share this set of flags.

The Input X-BAR takes a signal of a GPIO and routes it to an IP block destination. This pin can be selected for each input using the XBAR_setInputPin() function. Note that the descriptions for the values of the XBAR_InputNum enumerated type provide a list of the possible destinations for each input.

The code for this module is contained in `driverlib/xbar.c`, with `driverlib/xbar.h` containing the API declarations for use by applications.

35.2.2 Enumeration Type Documentation

35.2.2.1 enum **XBAR_OutputNum**

The following values define the *output* parameter for XBAR_setOutputMuxConfig(), XBAR_enableOutputMux(), and XBAR_disableOutputMux().

Enumerator

XBAR_OUTPUT1 OUTPUT1 of the Output X-BAR.
XBAR_OUTPUT2 OUTPUT2 of the Output X-BAR.
XBAR_OUTPUT3 OUTPUT3 of the Output X-BAR.
XBAR_OUTPUT4 OUTPUT4 of the Output X-BAR.
XBAR_OUTPUT5 OUTPUT5 of the Output X-BAR.
XBAR_OUTPUT6 OUTPUT6 of the Output X-BAR.
XBAR_OUTPUT7 OUTPUT7 of the Output X-BAR.
XBAR_OUTPUT8 OUTPUT8 of the Output X-BAR.

35.2.2.2 enum **XBAR_TripNum**

The following values define the *trip* parameter for XBAR_setEPWMMuxConfig(), XBAR_enableEPWMMux(), and XBAR_disableEPWMMux().

Enumerator

XBAR_TRIP4 TRIP4 of the ePWM X-BAR.
XBAR_TRIP5 TRIP5 of the ePWM X-BAR.
XBAR_TRIP7 TRIP7 of the ePWM X-BAR.
XBAR_TRIP8 TRIP8 of the ePWM X-BAR.
XBAR_TRIP9 TRIP9 of the ePWM X-BAR.
XBAR_TRIP10 TRIP10 of the ePWM X-BAR.
XBAR_TRIP11 TRIP11 of the ePWM X-BAR.
XBAR_TRIP12 TRIP12 of the ePWM X-BAR.

35.2.2.3 enum **XBAR_InputNum**

The following values define the *input* parameter for XBAR_setInputPin().

Enumerator

XBAR_INPUT1 ePWM[TZ1], ePWM[TRIP1], X-BARs, eCAPs
XBAR_INPUT2 ePWM[TZ2], ePWM[TRIP2], X-BARs, eCAPs
XBAR_INPUT3 ePWM[TZ3], ePWM[TRIP3], X-BARs, eCAPs
XBAR_INPUT4 ADC wrappers, X-BARs, XINT1, eCAPs.
XBAR_INPUT5 EXTSYNCIN1, X-BARs, XINT2, eCAPs.
XBAR_INPUT6 EXTSYNCIN2, ePWM[TRIP6], X-BARs, XINT3, eCAPs.
XBAR_INPUT7 X-BARs, eCAPs.
XBAR_INPUT8 X-BARs, eCAPs.
XBAR_INPUT9 X-BARs, eCAPs.
XBAR_INPUT10 X-BARs, eCAPs.
XBAR_INPUT11 X-BARs, eCAPs.
XBAR_INPUT12 X-BARs, eCAPs.
XBAR_INPUT13 XINT4, X-BARs, eCAPs.
XBAR_INPUT14 XINT5, X-BARs, eCAPs.
XBAR_INPUT15 eCAPs
XBAR_INPUT16 eCAPs

35.2.3 Function Documentation

35.2.3.1 static void XBAR_enableOutputMux (**XBAR_OutputNum** *output,* uint32_t *muxes*) [inline], [static]

Enables the Output X-BAR mux values to be passed to the output signal.

Parameters

output	is the X-BAR output being configured.

muxes	is a bit field of the muxes to be enabled.

This function enables the mux values to be passed to the X-BAR output signal. The *output* parameter is a value **XBAR_OUTPUTy** where y is the output number between 1 and 8 inclusive.

The *muxes* parameter is a bit field of the muxes being enabled where bit 0 represents mux 0, bit 1 represents mux 1 and so on. Defines are provided in the form of **XBAR_MUXnn** that can be OR'd together to enable several muxes on an output at the same time. For example, passing this function (**XBAR_MUX04** | **XBAR_MUX10**) would enable muxes 4 and 10.

Returns
None.

35.2.3.2 static void XBAR_disableOutputMux (**XBAR_OutputNum** *output,* uint32_t *muxes*) [inline], [static]

Disables the Output X-BAR mux values from being passed to the output.

Parameters

output	is the X-BAR output being configured.
muxes	is a bit field of the muxes to be disabled.

This function disables the mux values from being passed to the X-BAR output signal. The *output* parameter is a value **XBAR_OUTPUTy** where y is the output number between 1 and 8 inclusive.

The *muxes* parameter is a bit field of the muxes being disabled where bit 0 represents mux 0, bit 1 represents mux 1 and so on. Defines are provided in the form of **XBAR_MUXnn** that can be OR'd together to disable several muxes on an output at the same time. For example, passing this function (**XBAR_MUX04** | **XBAR_MUX10**) would disable muxes 4 and 10.

Returns
None.

35.2.3.3 static void XBAR_setOutputLatchMode (**XBAR_OutputNum** *output,* bool *enable*) [inline], [static]

Enables or disables the output latch to drive the selected output.

Parameters

output	is the X-BAR output being configured.
enable	is a flag that determines whether or not the latch is selected to drive the X-BAR output.

This function sets the Output X-BAR output signal latch mode. If the *enable* parameter is **true**, the output specified by *output* will be driven by the output latch.

Returns
None.

35.2.3.4 static bool XBAR_getOutputLatchStatus (**XBAR_OutputNum** *output*) [inline], [static]

Returns the status of the output latch

Parameters

output	is the X-BAR output being checked.

Returns

Returns **true** if the output corresponding to *output* was triggered. If not, it will return **false**.

35.2.3.5 static void XBAR_clearOutputLatch (**XBAR_OutputNum** *output*) `[inline]`, `[static]`

Clears the output latch for the specified output.

Parameters

output	is the X-BAR output being configured.

This function clears the Output X-BAR output latch. The output to be configured is specified by the *output* parameter.

Returns

None.

35.2.3.6 static void XBAR_forceOutputLatch (**XBAR_OutputNum** *output*) `[inline]`, `[static]`

Forces the output latch for the specified output.

Parameters

output	is the X-BAR output being configured.

This function forces the Output X-BAR output latch. The output to be configured is specified by the *output* parameter.

Returns

None.

35.2.3.7 static void XBAR_invertOutputSignal (**XBAR_OutputNum** *output,* bool *invert*) `[inline]`, `[static]`

Configures the polarity of an Output X-BAR output.

Parameters

output	is the X-BAR output being configured.
invert	is a flag that determines whether the output is active-high or active-low.

This function inverts the Output X-BAR signal if the *invert* parameter is **true**. If *invert* is **false**, the signal will be passed as is. The *output* parameter is a value **XBAR_OUTPUTy** where y is the output number between 1 and 8 inclusive.

Returns
None.

35.2.3.8 static void XBAR_enableEPWMMux (**XBAR_TripNum** *trip,* uint32_t *muxes*) [inline], [static]

Enables the ePWM X-BAR mux values to be passed to an ePWM module.

Parameters

trip	is the X-BAR output being configured.
muxes	is a bit field of the muxes to be enabled.

This function enables the mux values to be passed to the X-BAR trip signal. The *trip* parameter is a value **XBAR_TRIPy** where y is the number of the trip signal on the ePWM.

The *muxes* parameter is a bit field of the muxes being enabled where bit 0 represents mux 0, bit 1 represents mux 1 and so on. Defines are provided in the form of **XBAR_MUXnn** that can be logically OR'd together to enable several muxes on an output at the same time.

Returns
None.

35.2.3.9 static void XBAR_disableEPWMMux (**XBAR_TripNum** *trip,* uint32_t *muxes*) [inline], [static]

Disables the ePWM X-BAR mux values to be passed to an ePWM module.

Parameters

trip	is the X-BAR output being configured.
muxes	is a bit field of the muxes to be disabled.

This function disables the mux values to be passed to the X-BAR trip signal. The *trip* parameter is a value **XBAR_TRIPy** where y is the number of the trip signal on the ePWM.

The *muxes* parameter is a bit field of the muxes being disabled where bit 0 represents mux 0, bit 1 represents mux 1 and so on. Defines are provided in the form of **XBAR_MUXnn** that can be logically OR'd together to disable several muxes on an output at the same time.

Returns
None.

35.2.3.10 static void XBAR_invertEPWMSignal (**XBAR_TripNum** *trip,* bool *invert*) [inline], [static]

Configures the polarity of an ePWM X-BAR output.

Parameters

trip	is the X-BAR output being configured.
invert	is a flag that determines whether the output is active-high or active-low.

This function inverts the ePWM X-BAR trip signal if the *invert* parameter is **true**. If *invert* is **false**, the signal will be passed as is. The *trip* parameter is a value **XBAR_TRIPy** where y is the number of the trip signal on the ePWM X-BAR that is being configured.

Returns
None.

35.2.3.11 static void XBAR_setInputPin (**XBAR_InputNum** *input,* uint16_t *pin*) `[inline]`, `[static]`

Sets the GPIO pin for an Input X-BAR input.

Parameters

input	is the X-BAR input being configured.
pin	is the identifying number of the pin.

This function configures which GPIO is assigned to an Input X-BAR input. The *input* parameter is a value in the form of a define **XBAR_INPUTy** where y is a the input number for the Input X-BAR.

The pin is specified by its numerical value. For example, GPIO34 is specified by passing 34 as *pin*.

Returns
None.

Referenced by GPIO_setInterruptPin().

35.2.3.12 static void XBAR_lockInput (**XBAR_InputNum** *input*) `[inline]`, `[static]`

Locks an input to the Input X-BAR.

Parameters

input	is an input to the Input X-BAR.

This function locks the specific input on the Input X-BAR.

Returns
None.

35.2.3.13 static void XBAR_lockOutput (void) `[inline]`, `[static]`

Locks the Output X-BAR.

This function locks the Output X-BAR.

Returns
None.

35.2.3.14 static void XBAR_lockEPWM (void) `[inline]`, `[static]`

Locks the ePWM X-BAR.

This function locks the ePWM X-BAR.

Returns
None.

35.2.3.15 void XBAR_setOutputMuxConfig (**XBAR_OutputNum** *output*, XBAR_OutputMuxConfig *muxConfig*)

Configures the Output X-BAR mux that determines the signals passed to an output.

Parameters

output	is the X-BAR output being configured.
muxConfig	is mux configuration that specifies the signal.

This function configures an Output X-BAR mux. This determines which signal(s) should be passed through the X-BAR to a GPIO. The *output* parameter is a value **XBAR_OUTPUTy** where y is a the output number between 1 and 8 inclusive.

The *muxConfig* parameter is the mux configuration value that specifies which signal will be passed from the mux. The values have the format of **XBAR_OUT_MUXnn_xx** where the 'xx' is the signal and nn is the mux number (00 through 11). The possible values are found in `xbar.h`

This function may be called for each mux of an output and their values will be logically OR'd before being passed to the output signal. This means that this function may be called, for example, with the argument **XBAR_OUT_MUX00_ECAP1_OUT** and then with the argument **XBAR_OUT_MUX01_INPUTXBAR1**, resulting in the values of MUX00 and MUX03 being logically OR'd if both are enabled. Calling the function twice for the same mux on the output will result in the configuration in the second call overwriting the first.

Returns
None.

35.2.3.16 void XBAR_setEPWMMuxConfig (**XBAR_TripNum** *trip*, XBAR_EPWMMuxConfig *muxConfig*)

Configures the ePWM X-BAR mux that determines the signals passed to an ePWM module.

Parameters

trip	is the X-BAR output being configured.
muxConfig	is mux configuration that specifies the signal.

This function configures an ePWM X-BAR mux. This determines which signal(s) should be passed through the X-BAR to an ePWM module. The *trip* parameter is a value **XBAR_TRIPy** where y is a the number of the trip signal on the ePWM.

The *muxConfig* parameter is the mux configuration value that specifies which signal will be passed from the mux. The values have the format of **XBAR_EPWM_MUXnn_xx** where the 'xx' is the signal and nn is the mux number (0 through 31). The possible values are found in `xbar.h`

This function may be called for each mux of an output and their values will be logically OR'd before being passed to the trip signal. This means that this function may be called, for example, with the argument **XBAR_EPWM_MUX00_ECAP1_OUT** and then with the argument **XBAR_EPWM_MUX01_INPUTXBAR1**, resulting in the values of MUX00 and MUX03 being logically OR'd if both are enabled. Calling the function twice for the same mux on the output will result in the configuration in the second call overwriting the first.

Returns
None.

35.2.3.17 bool XBAR_getInputFlagStatus (XBAR_InputFlag *inputFlag*)

Returns the status of the input latch.

Parameters

inputFlag	is the X-BAR input latch being checked. Values are in the format of /b XBAR_INPUT_FLG_XXXX where "XXXX" is name of the signal.

Returns
Returns **true** if the X-BAR input corresponding to the *inputFlag* has been triggered. If not, it will return **false**.

35.2.3.18 void XBAR_clearInputFlag (XBAR_InputFlag *inputFlag*)

Clears the input latch for the specified input latch.

Parameters

inputFlag	is the X-BAR input latch being cleared.

This function clears the Input X-BAR input latch. The input latch to be cleared is specified by the *inputFlag* parameter.

Returns
None.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless