MSP432® Peripheral Driver Library

USER'S GUIDE

Copyright

Texas Instruments
13532 N. Central Expressway MS3810
Dallas, TX 75243
www.ti.com/

Revision Information

This is version 4.10.02.00 of this document, last updated on Mon Oct 30 00:57:56 -05 2017.

Table of Contents

1 DriverLib Introduction

1.1 What DriverLib is

The Texas Instruments MSP432 Driver Library (DriverLib) is a set of fully functional APIs used to configure, control, and manipulate the hardware peripherals of the MSP432 platform. In addition to being able to control the MSP432 peripherals, DriverLib also gives the user the ability to use common ARM peripherals such as the Interrupt (NVIC) and Memory Protection Unit (MPU) as well as MSP430 peripherals such as the eUSCI Serial peripherals and Watchdog Timer (WDT).

DriverLib for MSP432 Series has been tested and compiled under a variety of different toolchains. Subsequently, for each toolchain a specific debugger was used for testing validation. Below is a list that contains the supported toolchain and corresponding hardware debugger used.

- **Texas Instruments Code Composer Studio 6.1** (XDS100v3)
- **IAR Embedded Workbench for ARM 7.30** (SEGGER J-LINK)
- **GNU C Compiler 4.8 (gcc)** (SEGGER J-LINK)
- **Keil Embedded Development Tools for ARM 5.13** (KEIL U-LINK Pro)

The DriverLib is meant to provide a "software" layer to the programmer in order to facilitate higher level of programming compared to direct register accesses. Nearly every aspect of a MSP432 device can be configured and driven using the DriverLib APIs. By using the high level software APIs provided by DriverLib, users can create powerful and intuitive code which is highly portable between not only devices within the MSP432 platform, but between different families in the MSP430/MSP432 platforms.

Writing code in DriverLib will make user code more legible and easier to share among a group. For example, examine the following pair of code snippets. Both sets of code set MCLK to be sourced from VLO with a divider of four:

Traditional Register Access

```
CSKEY  =  0x695A;
CSCTL1 |= SELM_1 | DIVM_2;
CSKEY  =  0;
```

DriverLib Equivalent

```
CS_initClockSignal(CS_MCLK,  CS_VLOCLK_SELECT,  CS_CLOCK_DIVIDER_32);
```

As can be seen, the DriverLib API is readable, sensible, and easy to program for the software engineer. Additionally, DriverLib APIs for other platforms such as MSP430 will use very similar (if not identical) APIs giving code written with DriverLib APIs a boost in portability.

1.2 What DriverLib is not

The Driver Library is not meant to provide a layer of intelligence on the level of a user application. It is meant to be an aid to the programmer to be part of the larger solution- not the solution itself.

Interrupt handlers are also not included with the DriverLib APIs. APIs to manage/enable/disable interrupts are included, however the actual authoring of the interrupt service routine is left up to the programmer. For reference, A typical interrupt handler that takes advantage of DriverLib APIs can be seen in the following code snippet:

```
void port6_isr(void)
{
   uint32_t status = GPIO_getEnabledInterruptStatus(GPIO_PORT_P6);

   GPIO_clearInterruptFlag(GPIO_PORT_P6,  status);

   if (status & GPIO_PIN7)
   {
       if (powerStates[curPowerState] == PCM_LPM3)
       {
           curPowerState = 0;
       }
       stateChange = true;
   }
}
```

1.3 Cross Module Considerations

Each DriverLib module will, for the most part, only interact and configure the module that it is designed for. Any cross-module interaction is left up to the user. For example, when changing power modes to a low frequency mode with the PCM module, the user will have to ensure that the proper frequency requirements are configured with the CS module (low frequency requires that the system frequency be no greater that 128Khz).

Calling the following API alone while MCLK is greater that 128Khz will result in a system error:

```
PCM_setPowerState(PCM_AM_LF_VCORE1);
```

This is because the DriverLib module will not account for the overall system frequency of the system. Instead, similar APIs to the following must be called in conjunction:

```
CS_setReferenceOscillatorFrequency(CS_REFO_128KHZ);
CS_initClockSignal(CS_MCLK,  CS_REFOCLK_SELECT,  CS_CLOCK_DIVIDER_1);
PCM_setPowerState(PCM_AM_LF_VCORE1);
```

Cross-module considerations such as these must be taken when programming with DriverLib APIs as DriverLib was not designed to account for high level system requirements.

1.4 DriverLib in ROM

With all MSP432 devices, a copy of DriverLib is included within the device's ROM space. This allows programmers to take advantage of using high level APIs without having to worry about additional memory overhead of a flash library. In addition to a more optimized execution, the user can drastically cut down the memory footprint requirement of their application when using the software Driver Libraries available in ROM.

Accessing Driver Library APIs in ROM is as easy as including the rom.h header file, and then replacing normal API calls with a *ROM_* prefix. For example, take the following API from the pcm.c module that changes the power state to PCM_AM_DCDC_VCORE1:

```
PCM_setPowerState(PCM_AM_DCDC_VCORE1);
```

After including the rom.h file, all that would have to be done to switch to the ROM equivalent of the API would be add the ROM_ prefix to the API:

```
ROM_PCM_setPowerState(PCM_AM_DCDC_VCORE1);
```

While the majority of DriverLib APIs are available in ROM, due to architectural limitations some APIs are omitted from being included in ROM. In addition, if any bug fixes were added to the API after the device ROM was programmed, it is desirable to use the flash version of the API. An "intelligence" has been created to account for this problem. If the user includes the rom_map.h header file and uses the *MAP_* prefix in front of the API, the header file will automatically use preprocessor macros to decide whether to use a ROM or flash version of the API.

```
MAP_PCM_setPowerState(PCM_AM_DCDC_VCORE1);
```

1.5 MSP430 Legacy APIs

Since the MSP432 platform is built with many modules from Texas Instruments' MSP430 platform, many shared modules exist between MSP430 and MSP432. For this reason, a "compatibility" layer is provided to provide between the MSP430 Driver Library and the MSP430 Driver Library. The following modules are shared between MSP432 and MSP430:

- AES256
- COMP_E
- CRC32
- GPIO
- EUSCI_A_SPI (SPI)
- EUSCI_A_UART (UART)
- EUSCI_B_I2C (I2C)
- EUSCI_B_SPI (SPI)
- PMAP
- REF_A
- RTC_C

- TIMER_A
- WDT_A

To use these legacy APIs, no additional work is needed. All that is needed is to include the header file of the module you want to use and both the old and the new APIs will be available. For example, for WDT_A:

```
#include <wdt_a.h>
```

By including this header file, the user is granted access to all of the legacy DriverLib APIs from MSP430 Driver Library verbatim. For additional documentation on the MSP430 implementation of DriverLib, please refer to the `MSP430Ware Website`.

Many of the APIs were simplified and refactored for the MSP432 version of Driver Library. For example, to halt the watchdog module for a 5xx MSP430 device, the following API is used:

```
WDT_A_hold(WDT_A_BASE);
```

For MSP432 Driver Library, this same API has been simplified to the following API:

```
WDT_A_holdTimer();
```

Note that while many Driver Library APIs are shared between MSP430 and MSP432, there are a few underlying differences between the two architectures. Interrupts, for example, are a bit difference on MSP432 compared to MSP430 due to integration with ARM's interrupt controller (the NVIC). While each module will still have individual status (IFG), enable/disable, and clear bits, interrupt service routines now have to be associated with the ARM NVIC before usage.

1.6 Quick Start

Getting started using DriverLib for MSP432 Series is very simple regardless of the chosen development environment.

An empty "skeleton" project is provided in the examples directory of the SDK release. This project includes links to the DriverLib library as well as everything that is needed for the programmer to immediately start writing a DriverLib application. A user can import this project in CCS using the TI Resource Explorer, or open the workspace with IAR Embedded Workbench for ARM or KEIL uVision 5. All of the include paths and compiler options are set up to allow the user to seamlessly start development on their MSP432 DriverLib application.

The GNU compiler tools for ARM are fully supported by the MSP432 Series DriverLib. While no IDE in specific is supported, Makefiles are provided for both the library and all of the code examples. Vector table definitions that are compatible with the GCC compiler are also provided for code examples in the startup_gcc.c file for each individual code example. For the GNU tools, separate header files are included in the inc directory of the root installation of DriverLib. These header files are the latest that are available at the time of DriverLib release, however newer header files may be downloaded as a part of the CCS installation.

2 14-Bit Analog-to-Digital Converter (ADC14)

2.1 Module Operation

The ADC14 module for Driver Library is designed to allow the user to make simple analog to digital conversions as well make complex and simultaneous conversions across multiple channels.

2.2 Conversion Modes

With Single Conversion Mode, the user will sample only a single ADC channel which will be stored in a single ADC memory location. This is the most basic ADC sample/convert mode and allows for very simple measurements on a single channel. To configure single sample mode, only a single destination is configured for the sample/conversion result. The following is a code snippet for configuring/initializing the ADC module in single conversion mode as well as kicking off the start of conversion/sampling.

```
/* Initializing ADC (MCLK/1/4) */
MAP_ADC14_enableModule();
MAP_ADC14_initModule(ADC_CLOCKSOURCE_MCLK,  ADC_PREDIVIDER_1,  ADC_DIVIDER_4,
        0);

/* Configuring GPIOs (5.5 A0) */
MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P5,  GPIO_PIN5,
GPIO_TERTIARY_MODULE_FUNCTION);

/* Configuring ADC Memory */
MAP_ADC14_configureSingleSampleMode(ADC_MEM0,  true);
MAP_ADC14_configureConversionMemory(ADC_MEM0,  ADC_VREFPOS_AVCC_VREFNEG_VSS,
ADC_INPUT_A0,  false);

/* Configuring Sample Timer */
MAP_ADC14_enableSampleTimer(ADC_MANUAL_ITERATION);

/* Enabling/Toggling Conversion */
MAP_ADC14_enableConversion();
MAP_ADC14_toggleConversionTrigger();
```

When using single sample mode, only one memory location will be written for a conversion/sample cycle. To access the result of this conversion, the ADC14_getResult API is used with the corresponding memory location specified. This is usually done within the interrupt service routine of the ADC module.

```
/* ADC Interrupt Handler. This handler is called whenever there is a conversion
 * that is finished for ADC_MEM0.
 */
void ADC14_IRQHandler(void)
{
    uint64_t status = MAP_ADC14_getEnabledInterruptStatus();
    MAP_ADC14_clearInterruptFlag(status);

    if (ADC_INT0 & status)
    {
        curADCResult = MAP_ADC14_getResult(ADC_MEM0);
        normalizedADCRes = (curADCResult * 3.3) / 16384;

        MAP_ADC14_toggleConversionTrigger();
    }
}
```

The ADC14 APIs also support the setup/configuration of multiple conversion mode. With multiple conversion mode, multiple ADC channels are sampled and stored in multiple ADC memory addresses in a single sweep. This is particularly useful when the user wants to take a sample of multiple channels over a period of time (also known as scan mode). The ADC14_getMultiSequenceResult function is used to populate the given array pointer with the result over a wide memory arrange (setup with ADC14_configureMultiSequenceMode).

2.3 Repeat Modes

When configuring the ADC module to use multiple or single sample/conversion mode, a boolean argument is provided to signal whether the DriverLib ADC module should work in "repeat" mode. With repeat mode, once a conversion/sample is completed and read by the API, a new conversion happens until the user manually stops conversion using the ADC14_toggleConversionTrigger command. Repeat mode is useful when the user wants to continuously sample an ADC channel over an extended period of time.

When repeat mode is specified to be false, whenever a conversion/sample is finished and read from the result register, the module will stop conversion until called by the ADC14_toggleConversionTrigger function.

2.4 Conversion of Results

When reading a result of an ADC14 conversion, it is important to note that the result will be relevant to the current resolution of the ADC14 device. For example, say the ADC14 module is setup with a 14-bit resolution and a positive reference of 2.5v (and a negative of 0v). In this case, if the conversion result of 16383 would signify a value of 2.5v (if in unsigned) mode. A snippet of code that converts the conversion result in the ADC register to a real life value can be seen in the following:

```
/* Converts the ADC result (14-bit) to a float with respect to a 3.3v reference
 */
static float convertToFloat(uint16_t result)
{
    int32_t temp;

        if(0x8000 & result)
        {
            temp = (result >> 2) | 0xFFFFC000;
            return ((temp * 3.3f) / 8191);
        }
        else
            return ((result >> 2)*3.3f) / 8191;
}
```

It is important to note that when using floating point arithmetic, it is important to enable the devices FPU (if available) to save CPU cycles and energy consumption.

2.5 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the ADC14 module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the ADC14 module in single sample mode. For a set of more detailed code examples, please refer to the code examples in the examples/ directory of the SDK release:

```
    /* Initializing ADC (MCLK/1/1) */
    ADC14_enableModule();
    ADC14_initModule(ADC_CLOCKSOURCE_MCLK,  ADC_PREDIVIDER_1,  ADC_DIVIDER_1,
            0);

    /* Configuring ADC Memory (ADC_MEM0 A0/A1) in repeat mode
     * with use of external references */
    ADC14_configureSingleSampleMode(ADC_MEM0,  true);
    ADC14_configureConversionMemory(ADC_MEM0,
      ADC_VREFPOS_EXTPOS_VREFNEG_EXTNEG,
            ADC_INPUT_A0,  false);

    /* Setting up GPIO pins as analog inputs (and references) */
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P5,
            GPIO_PIN7 | GPIO_PIN6 | GPIO_PIN5 | GPIO_PIN4,  GPIO_TERTIARY_MODULE_FUNCTION);

    /* Enabling sample timer in auto iteration mode and interrupts*/
    ADC14_enableSampleTimer(ADC_AUTOMATIC_ITERATION);
    ADC14_enableInterrupt(ADC_INT0);

    /* Enabling Interrupts */
    Interrupt_enableInterrupt(INT_ADC14);
```

```
Interrupt_enableMaster();

/* Triggering the start of the sample */
ADC14_enableConversion();
ADC14_toggleConversionTrigger();
```

2.6 Definitions

Functions

- void ADC14_clearInterruptFlag (uint_fast64_t mask)
- bool ADC14_configureConversionMemory (uint32_t memorySelect, uint32_t refSelect, uint32_t channelSelect, bool differntialMode)
- bool ADC14_configureMultiSequenceMode (uint32_t memoryStart, uint32_t memoryEnd, bool repeatMode)
- bool ADC14_configureSingleSampleMode (uint32_t memoryDestination, bool repeatMode)
- bool ADC14_disableComparatorWindow (uint32_t memorySelect)
- void ADC14_disableConversion (void)
- void ADC14_disableInterrupt (uint_fast64_t mask)
- bool ADC14_disableModule (void)
- bool ADC14_disableReferenceBurst (void)
- bool ADC14_disableSampleTimer (void)
- bool ADC14_enableComparatorWindow (uint32_t memorySelect, uint32_t windowSelect)
- bool ADC14_enableConversion (void)
- void ADC14_enableInterrupt (uint_fast64_t mask)
- void ADC14_enableModule (void)
- bool ADC14_enableReferenceBurst (void)
- bool ADC14_enableSampleTimer (uint32_t multiSampleConvert)
- uint_fast64_t ADC14_getEnabledInterruptStatus (void)
- uint_fast64_t ADC14_getInterruptStatus (void)
- void ADC14_getMultiSequenceResult (uint16_t *res)
- uint_fast32_t ADC14_getResolution (void)
- uint_fast16_t ADC14_getResult (uint32_t memorySelect)
- void ADC14_getResultArray (uint32_t memoryStart, uint32_t memoryEnd, uint16_t *res)
- bool ADC14_initModule (uint32_t clockSource, uint32_t clockPredivider, uint32_t clockDivider, uint32_t internalChannelMask)
- bool ADC14_isBusy (void)
- void ADC14_registerInterrupt (void(*intHandler)(void))
- bool ADC14_setComparatorWindowValue (uint32_t window, int16_t low, int16_t high)
- bool ADC14_setPowerMode (uint32_t powerMode)
- void ADC14_setResolution (uint32_t resolution)
- bool ADC14_setResultFormat (uint32_t resultFormat)
- bool ADC14_setSampleHoldTime (uint32_t firstPulseWidth, uint32_t secondPulseWidth)
- bool ADC14_setSampleHoldTrigger (uint32_t source, bool invertSignal)
- bool ADC14_toggleConversionTrigger (void)
- void ADC14_unregisterInterrupt (void)

2.6.1 Detailed Description

The code for this module is contained in `driverlib/adc14.c`, with `driverlib/adc14.h` containing the API declarations for use by applications.

2.6.2 Function Documentation

2.6.2.1 void ADC14_clearInterruptFlag (uint_fast64_t *mask*)

Clears the indicated ADCC interrupt sources.

Parameters

mask	is the bit mask of interrupts to clear. The ADC_INT0 through ADC_INT31 parameters correspond to a completion event of the corresponding memory location. For example, when the ADC_MEM0 location finishes a conversion cycle, the ADC_INT0 interrupt will be set. Valid values are a bitwise OR of the following values: ■ **ADC_INT0** through ADC_INT31 ■ **ADC_IN_INT** - Interrupt enable for a conversion in the result register is either greater than the ADCLO or lower than the ADCHI threshold. ■ **ADC_LO_INT** - Interrupt enable for the falling short of the lower limit interrupt of the window comparator for the result register. ■ **ADC_HI_INT** - Interrupt enable for the exceeding the upper limit of the window comparator for the result register. ■ **ADC_OV_INT** - Interrupt enable for a conversion that is about to save to a memory buffer that has not been read out yet. ■ **ADC_TOV_INT** -Interrupt enable for a conversion that is about to start before the previous conversion has been completed. ■ **ADC_RDY_INT** -Interrupt enable for the local buffered reference ready signal.

Returns
NONE

2.6.2.2 bool ADC14_configureConversionMemory (uint32_t *memorySelect,* uint32_t *refSelect,* uint32_t *channelSelect,* bool *differntialMode*)

Configures an individual memory location for the ADC module.

Parameters

memorySelect	is the individual ADC memory location to configure. If multiple memory locations want to be configured with the same configuration, this value can be logically ORed together with other values. ■ **ADC_MEM0** through **ADC_MEM31**
refSelect	is the voltage reference to use for the selected memory spot. Possible values include: ■ **ADC_VREFPOS_AVCC_VREFNEG_VSS** [DEFAULT] ■ **ADC_VREFPOS_INTBUF_VREFNEG_VSS** ■ **ADC_VREFPOS_EXTPOS_VREFNEG_EXTNEG** ■ **ADC_VREFPOS_EXTBUF_VREFNEG_EXTNEG**
channelSelect	selects the channel to be used for ADC sampling. Note if differential mode is enabled, the value sampled will be equal to the difference between the corresponding even/odd memory locations. Possible values are: ■ **ADC_INPUT_A0** through **ADC_INPUT_A31**
differntialMode	selects if the channel selected by the channelSelect will be configured in differential mode. If this parameter is given as true, the configured channel will be paired with its neighbor in differential mode. for example, if channel A0 or A1 is selected, the channel configured will be the difference between A0 and A1. If A2 or A3 are selected, the channel configured will be the difference between A2 and A3 (and so on). Users can enter true or false, or one of the following values: ■ ADC_NONDIFFERENTIAL_INPUTS ■ ADC_DIFFERENTIAL_INPUTS

Returns

false if setting fails due to an in progress conversion

2.6.2.3 bool ADC14_configureMultiSequenceMode (uint32_t *memoryStart,* uint32_t *memoryEnd,* bool *repeatMode*)

Configures the ADC module to use a multiple memory sample scheme. This means that multiple samples will consecutively take place and be stored in multiple memory locations. The first sample/conversion will be placed in memoryStart, while the last sample will be stored in memoryEnd. Each memory location should be configured individually using the ADC14_configureConversionMemory function.

The ADC module can be started in "repeat" mode which will cause the ADC module to resume sampling once the initial sample/conversion set is executed. For multi-sample mode, this means that the sampling of the entire memory provided.

Parameters

memoryStart	Memory location to store first sample/conversion value. Possible values include: ■ **ADC_MEM0** through **ADC_MEM31**
memoryEnd	Memory location to store last sample. Possible values include: ■ **ADC_MEM0** through **ADC_MEM31**
repeatMode	Specifies whether or not to repeat the conversion/sample cycle after the first round of sample/conversions. Valid values are true or false.

Returns

false if setting fails due to an in progress conversion

2.6.2.4 bool ADC14_configureSingleSampleMode (uint32_t *memoryDestination,* bool *repeatMode*)

Configures the ADC module to use a a single ADC memory location for sampling/conversion. This is used when only one channel might be needed for conversion, or where using a multiple sampling scheme is not important.

The ADC module can be started in "repeat" mode which will cause the ADC module to resume sampling once the initial sample/conversion set is executed. In single sample mode, this will cause the ADC module to continuously sample into the memory destination provided.

Parameters

memoryDestination	Memory location to store sample/conversion value. Possible values include: ■ **ADC_MEM0** through **ADC_MEM31**
repeatMode	Specifies whether or not to repeat the conversion/sample cycle after the first round of sample/conversions

Returns

false if setting fails due to an in progress conversion

2.6.2.5 bool ADC14_disableComparatorWindow (uint32_t *memorySelect*)

Disables the comparator window on the specified memory channels

Parameters

memorySelect	is the mask of memory locations to disable the comparator window for. This can be a bitwise OR of the following values: ■ **ADC_MEM0** through **ADC_MEM31**

Returns
false if setting fails due to an in progress conversion

2.6.2.6 void ADC14_disableConversion (void)

Halts conversion conversion of the ADC module. Note that the software bit for triggering conversions will also be cleared with this function.

If multi-sequence conversion mode was enabled, the position of the last completed conversion can be retrieved using ADCLastConversionMemoryGet

Returns
none

2.6.2.7 void ADC14_disableInterrupt (uint_fast64_t *mask*)

Disables the indicated ADCC interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The ADC_INT0 through ADC_INT31 parameters correspond to a completion event of the corresponding memory location. For example, when the ADC_MEM0 location finishes a conversion cycle, the ADC_INT0 interrupt will be set.

Parameters

mask	is the bit mask of interrupts to disable. Valid values are a bitwise OR of the following values: ■ **ADC_INT0** through ADC_INT31 ■ **ADC_IN_INT** - Interrupt enable for a conversion in the result register is either greater than the ADCLO or lower than the ADCHI threshold. ■ **ADC_LO_INT** - Interrupt enable for the falling short of the lower limit interrupt of the window comparator for the result register. ■ **ADC_HI_INT** - Interrupt enable for the exceeding the upper limit of the window comparator for the result register. ■ **ADC_OV_INT** - Interrupt enable for a conversion that is about to save to a memory buffer that has not been read out yet. ■ **ADC_TOV_INT** -Interrupt enable for a conversion that is about to start before the previous conversion has been completed. ■ **ADC_RDY_INT** -Interrupt enable for the local buffered reference ready signal.

Returns
NONE

2.6.2.8 bool ADC14_disableModule (void)

Disables the ADC block.

This will disable operation of the ADC block.

Returns
false if user is trying to disable during active conversion

2.6.2.9 bool ADC14_disableReferenceBurst (void)

Disables the "on-demand" activity of the voltage reference register.

Returns
false if setting fails due to an in progress conversion

2.6.2.10 bool ADC14_disableSampleTimer (void)

Disables SAMPCON from being sourced from the sample timer.

Returns
false if the initialization fails due to an in progress conversion

2.6.2.11 bool ADC14_enableComparatorWindow (uint32_t *memorySelect,* uint32_t *windowSelect*)

Enables the specified mask of memory channels to use the specified comparator window. THe ADCC module has two different comparator windows that can be set with this function.

Parameters

memorySelect	is the mask of memory locations to enable the comparator window for. This can be a bitwise OR of the following values: ■ **ADC_MEM0** through **ADC_MEM31**
windowSelect	Memory location to store sample/conversion value. Possible values include: **ADCOMP_WINDOW0** [DEFAULT] **ADCOMP_WINDOW1**

Returns
false if setting fails due to an in progress conversion

2.6.2.12 bool ADC14_enableConversion (void)

Enables conversion of ADC data. Note that this only enables conversion. To trigger the conversion, you will have to call the ADC14_toggleConversionTrigger or use the source trigger configured in ADC14_setSampleHoldTrigger.

Returns
false if setting fails due to an in progress conversion

2.6.2.13 void ADC14_enableInterrupt (uint_fast64_t *mask*)

Enables the indicated ADCC interrupt sources. The ADC_INT0 through ADC_INT31 parameters correspond to a completion event of the corresponding memory location. For example, when the ADC_MEM0 location finishes a conversion cycle, the ADC_INT0 interrupt will be set.

Parameters

mask	is the bit mask of interrupts to enable. Valid values are a bitwise OR of the following values: ■ **ADC_INT0** through ADC_INT31 ■ **ADC_IN_INT** - Interrupt enable for a conversion in the result register is either greater than the ADCLO or lower than the ADCHI threshold. ■ **ADC_LO_INT** - Interrupt enable for the falling short of the lower limit interrupt of the window comparator for the result register. ■ **ADC_HI_INT** - Interrupt enable for the exceeding the upper limit of the window comparator for the result register. ■ **ADC_OV_INT** - Interrupt enable for a conversion that is about to save to a memory buffer that has not been read out yet. ■ **ADC_TOV_INT** -Interrupt enable for a conversion that is about to start before the previous conversion has been completed. ■ **ADC_RDY_INT** -Interrupt enable for the local buffered reference ready signal.

Returns
NONE

2.6.2.14 void ADC14_enableModule (void)

Enables the ADC block.

This will enable operation of the ADC block.

Returns
none.

2.6.2.15 bool ADC14_enableReferenceBurst (void)

Enables the "on-demand" activity of the voltage reference register. If this setting is enabled, the internal voltage reference buffer will only be updated during a sample or conversion cycle. This is used to optimize power consumption.

Returns
false if setting fails due to an in progress conversion

2.6.2.16 bool ADC14_enableSampleTimer (uint32_t *multiSampleConvert*)

Enables SAMPCON to be sourced from the sampling timer and to configures multi sample and conversion mode.

Parameters

multiSample-Convert	- Switches between manual and automatic iteration when using the sample timer. Valid values are: ■ **ADC_MANUAL_ITERATION** The user will have to manually set the SHI signal (usually by ADC14_toggleConversionTrigger) at the end of each sample/conversion cycle. ■ **ADC_AUTOMATIC_ITERATION** After one sample/convert is finished, the ADC module will automatically continue on to the next sample

Returns

false if the initialization fails due to an in progress conversion

2.6.2.17 uint_fast64_t ADC14_getEnabledInterruptStatus (void)

Returns the status of a the ADC interrupt register masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR. The ADC_INT0 through ADC_INT31 parameters correspond to a completion event of the corresponding memory location. For example, when the ADC_MEM0 location finishes a conversion cycle, the ADC_INT0

Returns

The interrupt status. Value is a bitwise OR of the following values:

- **ADC_INT0** through ADC_INT31
- **ADC_IN_INT** - Interrupt enable for a conversion in the result register is either greater than the ADCLO or lower than the ADCHI threshold.
- **ADC_LO_INT** - Interrupt enable for the falling short of the lower limit interrupt of the window comparator for the result register.
- **ADC_HI_INT** - Interrupt enable for the exceeding the upper limit of the window comparator for the result register.
- **ADC_OV_INT** - Interrupt enable for a conversion that is about to save to a memory buffer that has not been read out yet.
- **ADC_TOV_INT** -Interrupt enable for a conversion that is about to start before the previous conversion has been completed.
- **ADC_RDY_INT** -Interrupt enable for the local buffered reference ready signal.

References ADC14_getInterruptStatus().

2.6.2.18 uint_fast64_t ADC14_getInterruptStatus (void)

Returns the status of a the ADC interrupt register. The ADC_INT0 through ADC_INT31 parameters correspond to a completion event of the corresponding memory location. For example, when the ADC_MEM0 location finishes a conversion cycle, the ADC_INT0 interrupt will be set.

Returns

The interrupt status. Value is a bitwise OR of the following values:

- **ADC_INT0** through ADC_INT31
- **ADC_IN_INT** - Interrupt enable for a conversion in the result register is either greater than the ADCLO or lower than the ADCHI threshold.
- **ADC_LO_INT** - Interrupt enable for the falling short of the lower limit interrupt of the window comparator for the result register.
- **ADC_HI_INT** - Interrupt enable for the exceeding the upper limit of the window comparator for the result register.
- **ADC_OV_INT** - Interrupt enable for a conversion that is about to save to a memory buffer that has not been read out yet.
- **ADC_TOV_INT** -Interrupt enable for a conversion that is about to start before the previous conversion has been completed.
- **ADC_RDY_INT** -Interrupt enable for the local buffered reference ready signal.

Referenced by ADC14_getEnabledInterruptStatus().

2.6.2.19 void ADC14_getMultiSequenceResult (uint16_t * *res*)

Returns the conversion results of the currently configured multi-sequence conversion. If a multi-sequence conversion has not happened, this value is unreliable. Note that it is up to the user to verify the integrity of and proper size of the array being passed. If there are 16 multi-sequence results, and an array with only 4 elements allocated is passed, invalid memory settings will occur

Parameters

res	conversion result of the last multi-sequence sample in an array of unsigned 16-bit integers

Returns

None

2.6.2.20 uint_fast32_t ADC14_getResolution (void)

Gets the resolution of the ADC module.

Returns

Resolution of the ADC module

- **ADC_8BIT** (10 clock cycle conversion time)
- **ADC_10BIT** (12 clock cycle conversion time)
- **ADC_12BIT** (14 clock cycle conversion time)
- **ADC_14BIT** (16 clock cycle conversion time)

2.6.2.21 uint_fast16_t ADC14_getResult (uint32_t *memorySelect*)

Returns the conversion result for the specified memory channel in the format assigned by the ADC14_setResultFormat (unsigned binary by default) function.

Parameters

memorySelect	is the memory location to get the conversion result. Valid values are: ■ **ADC_MEM0** through **ADC_MEM31**

Returns
conversion result of specified memory channel

2.6.2.22 void ADC14_getResultArray (uint32_t *memoryStart,* uint32_t *memoryEnd,* uint16_t ∗ *res*)

Returns the conversion results of the specified ADC memory locations. Note that it is up to the user to verify the integrity of and proper size of the array being passed. If there are 16 multi-sequence results, and an array with only 4 elements allocated is passed, invalid memory settings will occur. This function is inclusive.

Parameters

memoryStart	is the memory location to get the conversion result. Valid values are: ■ **ADC_MEM0** through **ADC_MEM31**
memoryEnd	is the memory location to get the conversion result. Valid values are: ■ **ADC_MEM0** through **ADC_MEM31**
res	conversion result of the last multi-sequence sample in an array of unsigned 16-bit integers

Returns
None

2.6.2.23 bool ADC14_initModule (uint32_t *clockSource,* uint32_t *clockPredivider,* uint32_t *clockDivider,* uint32_t *internalChannelMask*)

Initializes the ADC module and sets up the clock system divider/pre-divider. This initialization function will also configure the internal/external signal mapping.

Note

A call to this function while active ADC conversion is happening is an invalid case and will result in a false value being returned.

Parameters

clockSource	The clock source to use for the ADC module. ■ **ADC_CLOCKSOURCE_ADCOSC** [DEFAULT] ■ **ADC_CLOCKSOURCE_SYSOSC** ■ **ADC_CLOCKSOURCE_ACLK** ■ **ADC_CLOCKSOURCE_MCLK** ■ **ADC_CLOCKSOURCE_SMCLK** ■ **ADC_CLOCKSOURCE_HSMCLK**
clockPredivider	Divides the given clock source before feeding it into the main clock divider. Valid values are: ■ **ADC_PREDIVIDER_1** [DEFAULT] ■ **ADC_PREDIVIDER_4** ■ **ADC_PREDIVIDER_32** ■ **ADC_PREDIVIDER_64**
clockDivider	Divides the pre-divided clock source Valid values are ■ **ADC_DIVIDER_1** [Default value] ■ **ADC_DIVIDER_2** ■ **ADC_DIVIDER_3** ■ **ADC_DIVIDER_4** ■ **ADC_DIVIDER_5** ■ **ADC_DIVIDER_6** ■ **ADC_DIVIDER_7** ■ **ADC_DIVIDER_8**

internalChannelMask	Configures the internal/external pin mappings for the ADC modules. This setting determines if the given ADC channel or component is mapped to an external pin (default), or routed to an internal component. This parameter is a bit mask where a logical high value will switch the component to the internal routing. For a list of internal routings, please refer to the device specific data sheet. Valid values are a logical OR of the following values: ■ **ADC_MAPINTCH3** ■ **ADC_MAPINTCH2** ■ **ADC_MAPINTCH1** ■ **ADC_MAPINTCH0** ■ **ADC_TEMPSENSEMAP** ■ **ADC_BATTMAP** ■ ADC_NOROUTE If internalChannelMask is not desired, pass ADC_NOROUTE in lieu of this parameter.

Returns
false if the initialization fails due to an in progress conversion

2.6.2.24 bool ADC14_isBusy (void)

Returns a boolean value that tells if a conversion/sample is in progress

Returns
true if conversion is active, false otherwise

2.6.2.25 void ADC14_registerInterrupt (void(∗)(void) *intHandler*)

Registers an interrupt handler for the ADC interrupt.

Parameters

intHandler	is a pointer to the function to be called when the ADC interrupt occurs.

This function registers the handler to be called when an ADC interrupt occurs. This function enables the global interrupt in the interrupt controller; specific ADC14 interrupts must be enabled via ADC14_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via ADC14_clearInterruptFlag().

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

2.6.2.26 bool ADC14_setComparatorWindowValue (uint32_t *window,* int16_t *low,* int16_t *high*)

Sets the lower and upper limits of the specified window comparator. Note that this function will truncate values based of the resolution/data format configured. If the ADC is operating in 10-bit mode, and a 12-bit value is passed into this function the most significant 2 bits will be truncated.

The parameters provided to this function for the upper and lower threshold depend on the current resolution for the ADC. For example, if configured in 12-bit mode, a 12-bit resolution is the maximum that can be provided for the window. If in 2's complement mode, Bit 15 is used as the MSB.

Parameters

window	Memory location to store sample/conversion value. Possible values include: **ADC_COMP_WINDOW0** [DEFAULT] **ADC_COMP_WINDOW1**
low	is the lower limit of the window comparator
high	is the upper limit of the window comparator

Returns

false if setting fails due to an in progress conversion

2.6.2.27 bool ADC14_setPowerMode (uint32_t *powerMode*)

Sets the power mode of the ADC module. A more aggressive power mode will restrict the number of samples per second for sampling while optimizing power consumption. Ideally, if power consumption is a concern, this value should be set to the most restrictive setting that satisfies your sampling requirement.

Parameters

adcPowerMode	is the power mode to set. Valid values are: ■ **ADC_UNRESTRICTED_POWER_MODE** (no restriction) ■ **ADC_LOW_POWER_MODE** (500ksps restriction) ■ **ADC_ULTRA_LOW_POWER_MODE** (200ksps restriction) ■ **ADC_EXTREME_LOW_POWER_MODE** (50ksps restriction)

Returns

false if setting fails due to an in progress conversion

2.6.2.28 void ADC14_setResolution (uint32_t *resolution*)

Sets the resolution of the ADC module. The default resolution is 12-bit, however for power consumption concerns this can be limited to a lower resolution

Parameters

resolution	Resolution of the ADC module ■ **ADC_8BIT** (10 clock cycle conversion time) ■ **ADC_10BIT** (12 clock cycle conversion time) ■ **ADC_12BIT** (14 clock cycle conversion time) ■ **ADC_14BIT** (16 clock cycle conversion time)[DEFAULT]

Returns
none

2.6.2.29 bool ADC14_setResultFormat (uint32_t *resultFormat*)

Switches between a binary unsigned data format and a signed 2's complement data format.

Parameters

resultFormat	Format for result to conversion results. Possible values include: **ADC_UNSIGNED_BINARY** [DEFAULT] **ADC_SIGNED_BINARY**

Returns
false if setting fails due to an in progress conversion

2.6.2.30 bool ADC14_setSampleHoldTime (uint32_t *firstPulseWidth,* uint32_t *secondPulseWidth*)

Sets the sample/hold time for the specified memory register range. The duration of time required for a sample differs depending on the user's hardware configuration.

There are two values in the ADCC module. The first value controls ADC memory locations ADC_MEMORY_0 through ADC_MEMORY_7 and ADC_MEMORY_24 through ADC_MEMORY_31, while the second value controls memory locations ADC_MEMORY_8 through ADC_MEMORY_23.

Parameters

firstPulseWidth	Pulse width of the first pulse in ADCCLK cycles Possible values must be one of the following: ■ **ADC_PULSE_WIDTH_4** [DEFAULT] ■ **ADC_PULSE_WIDTH_8** ■ **ADC_PULSE_WIDTH_16** ■ **ADC_PULSE_WIDTH_32** ■ **ADC_PULSE_WIDTH_64** ■ **ADC_PULSE_WIDTH_96** ■ **ADC_PULSE_WIDTH_128** ■ **ADC_PULSE_WIDTH_192**
second-PulseWidth	Pulse width of the second pulse in ADCCLK cycles. Possible values must be one of the following: ■ **ADC_PULSE_WIDTH_4** [DEFAULT] ■ **ADC_PULSE_WIDTH_8** ■ **ADC_PULSE_WIDTH_16** ■ **ADC_PULSE_WIDTH_32** ■ **ADC_PULSE_WIDTH_64** ■ **ADC_PULSE_WIDTH_96** ■ **ADC_PULSE_WIDTH_128** ■ **ADC_PULSE_WIDTH_192**

Returns

false if setting fails due to an in progress conversion

2.6.2.31 bool ADC14_setSampleHoldTrigger (uint32_t *source,* bool *invertSignal*)

Sets the source for the trigger of the ADC module. By default, this value is configured to a software source (the ADCSC bit), however depending on the specific device the trigger can be set to different sources (for example, a timer output). These sources vary from part to part and the user should refer to the device specific datasheet.

Parameters

source	Trigger source for sampling. Possible values include: ■ **ADC_TRIGGER_ADCSC** [DEFAULT] ■ **ADC_TRIGGER_SOURCE1** ■ **ADC_TRIGGER_SOURCE2** ■ **ADC_TRIGGER_SOURCE3** ■ **ADC_TRIGGER_SOURCE4** ■ **ADC_TRIGGER_SOURCE5** ■ **ADC_TRIGGER_SOURCE6** ■ **ADC_TRIGGER_SOURCE7**
invertSignal	When set to true, will invert the trigger signal to a falling edge. When false, will use a rising edge.

Returns
false if setting fails due to an in progress conversion

2.6.2.32 bool ADC14_toggleConversionTrigger (void)

Toggles the trigger for conversion of the ADC module by toggling the trigger software bit. Note that this will cause the ADC to start conversion regardless if the software bit was set as the trigger using ADC14_setSampleHoldTrigger.

Returns
false if setting fails due to an in progress conversion

2.6.2.33 void ADC14_unregisterInterrupt (void)

Unregisters the interrupt handler for the ADCC module.

This function unregisters the handler to be called when an ADCC interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

3 Advanced Encryption Standard 256 Module (AES256)

3.1 Module Operation

The AES256 accelerator module performs encryption and decryption of 128-bit data with 128-bit keys according to the advanced encryption standard (AES256) (FIPS PUB 197) in hardware.

3.2 Key Features

The key features of the AES256 module include:

- Encryption and decryption according to AES256 FIPS PUB 197 with 128-bit key
- On-the-fly key expansion for encryption and decryption
- Off-line key generation for decryption
- Byte and word access to key, input, and output data
- AES256 ready interrupt flag

The AES256256 accelerator module performs encryption and decryption of 128-bit data with 128-/192-/256-bit keys according to the advanced encryption standard (AES256) (FIPS PUB 197) in hardware.

3.3 Encryption/Decryption Cycle Times

The the AES256 accelerator decryption/encryption cycle counts are as follows:

AES256 encryption

- 128 bit - 168 cycles
- 192 bit - 204 cycles
- 256 bit - 234 cycles

AES256 decryption:

- 128 bit - 168 cycles
- 192 bit - 206 cycles
- 256 bit - 234 cycles

3.4 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the AES256 module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. Below is a simple code example of how to encrypt/decrypt data using a cipher key with the AES256 module

```
/* Load a cipher key to module */
MAP_AES256_setCipherKey(AES256_BASE,  CipherKey,  AES256_KEYLENGTH_256BIT);

/* Encrypt data with preloaded cipher key */
MAP_AES256_encryptData(AES256_BASE,  Data,  DataAESencrypted);

/* Load a decipher key to module */
MAP_AES256_setDecipherKey(AES256_BASE,  CipherKey, AES256_KEYLENGTH_256BIT);

/* Decrypt data with keys that were generated during encryption - takes
 214 MCLK cyles. This function will generate all round keys needed for
 decryption first and then the encryption process starts */
MAP_AES256_decryptData(AES256_BASE,  DataAESencrypted,  DataAESdecrypted);
```

3.5 Definitions

Functions

- void AES256_clearErrorFlag (uint32_t moduleInstance)
- void AES256_clearInterruptFlag (uint32_t moduleInstance)
- void AES256_decryptData (uint32_t moduleInstance, const uint8_t *data, uint8_t *decryptedData)
- void AES256_disableInterrupt (uint32_t moduleInstance)
- void AES256_enableInterrupt (uint32_t moduleInstance)
- void AES256_encryptData (uint32_t moduleInstance, const uint8_t *data, uint8_t *encryptedData)
- bool AES256_getDataOut (uint32_t moduleInstance, uint8_t *outputData)
- uint32_t AES256_getErrorFlagStatus (uint32_t moduleInstance)
- uint32_t AES256_getInterruptFlagStatus (uint32_t moduleInstance)
- uint32_t AES256_getInterruptStatus (uint32_t moduleInstance)
- bool AES256_isBusy (uint32_t moduleInstance)
- void AES256_registerInterrupt (uint32_t moduleInstance, void(*intHandler)(void))
- void AES256_reset (uint32_t moduleInstance)
- bool AES256_setCipherKey (uint32_t moduleInstance, const uint8_t *cipherKey, uint_fast16_t keyLength)
- bool AES256_setDecipherKey (uint32_t moduleInstance, const uint8_t *cipherKey, uint_fast16_t keyLength)
- void AES256_startDecryptData (uint32_t moduleInstance, const uint8_t *data)
- void AES256_startEncryptData (uint32_t moduleInstance, const uint8_t *data)
- bool AES256_startSetDecipherKey (uint32_t moduleInstance, const uint8_t *cipherKey, uint_fast16_t keyLength)
- void AES256_unregisterInterrupt (uint32_t moduleInstance)

3.5.1 Detailed Description

The code for this module is contained in `driverlib/aes256.c` and `driverlib/legacy/MSP432xx/legacy_aes256.c`, with `driverlib/aes256.h` and `driverlib/legacy/MSP432xx/legacy_aes256.h` containing the API declarations for use by applications.

3.5.2 Function Documentation

3.5.2.1 void AES256_clearErrorFlag (uint32_t *moduleInstance*)

Clears the AES256 error flag.

Parameters

moduleInstance	is the base address of the AES256 module.

Modified bits are **AESERRFG** of **AESACTL0** register.

Returns
None

3.5.2.2 void AES256_clearInterruptFlag (uint32_t *moduleInstance*)

Clears the AES256 ready interrupt flag.

Parameters

moduleInstance	is the base address of the AES256 module.

Modified bits are **AESRDYIFG** of **AESACTL0** register.

Returns
None

3.5.2.3 void AES256_decryptData (uint32_t *moduleInstance,* const uint8_t * *data,* uint8_t * *decryptedData*)

Decrypts a block of data using the AES256 module.

This function requires a pregenerated decryption key. A key can be loaded and pregenerated by using function AES256_setDecipherKey() or AES256_startSetDecipherKey(). The decryption takes 167 MCLK.

Parameters

moduleInstance	is the base address of the AES256 module.
data	is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.
decryptedData	is a pointer to an uint8_t array with a length of 16 bytes in that the decrypted data will be written.

Returns
None

3.5.2.4 void AES256_disableInterrupt (uint32_t *moduleInstance*)

Disables AES256 ready interrupt.

Parameters

moduleInstance	is the base address of the AES256 module.

Modified bits are **AESRDYIE** of **AESACTL0** register.

Returns
None

3.5.2.5 void AES256_enableInterrupt (uint32_t *moduleInstance*)

Enables AES256 ready interrupt.

Parameters

moduleInstance	is the base address of the AES256 module.

Modified bits are **AESRDYIE** of **AESACTL0** register.

Returns
None

3.5.2.6 void AES256_encryptData (uint32_t *moduleInstance,* const uint8_t * *data,* uint8_t * *encryptedData*)

Encrypts a block of data using the AES256 module.

The cipher key that is used for encryption should be loaded in advance by using function AES256_setCipherKey()

Parameters

moduleInstance	is the base address of the AES256 module.
data	is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.
encryptedData	is a pointer to an uint8_t array with a length of 16 bytes in that the encrypted data will be written.

Returns
None

3.5.2.7 bool AES256_getDataOut (uint32_t *moduleInstance,* uint8_t * *outputData*)

Reads back the output data from AES256 module.

This function is meant to use after an encryption or decryption process that was started and finished by initiating an interrupt by use of AES256_startEncryptData or AES256_startDecryptData functions.

Parameters

moduleInstance	is the base address of the AES256 module.
outputData	is a pointer to an uint8_t array with a length of 16 bytes in that the data will be written.

Returns

true if data is valid, otherwise false

3.5.2.8 uint32_t AES256_getErrorFlagStatus (uint32_t *moduleInstance*)

Gets the AES256 error flag status.

Parameters

moduleInstance	is the base address of the AES256 module.

Returns

One of the following:

- **AES256_ERROR_OCCURRED**
- **AES256_NO_ERROR**

indicating the error flag status

3.5.2.9 uint32_t AES256_getInterruptFlagStatus (uint32_t *moduleInstance*)

Gets the AES256 ready interrupt flag status.

Parameters

moduleInstance	is the base address of the AES256 module.

Returns

One of the following:

- **AES256_READY_INTERRUPT**
- **AES256_NOTREADY_INTERRUPT**

indicating the status of the AES256 ready status

Referenced by AES256_getInterruptStatus().

3.5.2.10 uint32_t AES256_getInterruptStatus (uint32_t *moduleInstance*)

Returns the current interrupt flag for the peripheral.

Parameters

moduleInstance	Instance of the AES256 module

Returns

The currently triggered interrupt flag for the module.

References AES256_getInterruptFlagStatus().

3.5.2.11 bool AES256_isBusy (uint32_t *moduleInstance*)

Gets the AES256 module busy status.

Parameters

moduleInstance	is the base address of the AES256 module.

Returns
true if busy, false otherwise

3.5.2.12 void AES256_registerInterrupt (uint32_t *moduleInstance,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the AES interrupt.

Parameters

moduleInstance	Instance of the AES256 module
intHandler	is a pointer to the function to be called when the AES interrupt occurs.

This function registers the handler to be called when a AES interrupt occurs. This function enables the global interrupt in the interrupt controller; specific AES interrupts must be enabled via AES256_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via AES256_clearInterrupt().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

3.5.2.13 void AES256_reset (uint32_t *moduleInstance*)

Resets AES256 Module immediately.

Parameters

moduleInstance	is the base address of the AES256 module.

Modified bits are **AESSWRST** of **AESACTL0** register.

Returns
None

3.5.2.14 bool AES256_setCipherKey (uint32_t *moduleInstance,* const uint8_t ∗ *cipherKey,* uint_fast16_t *keyLength*)

Loads a 128, 192 or 256 bit cipher key to AES256 module.

Parameters

moduleInstance	is the base address of the AES256 module.
cipherKey	is a pointer to an uint8_t array with a length of 16 bytes that contains a 128 bit cipher key.
keyLength	is the length of the key. Valid values are: ■ **AES256_KEYLENGTH_128BIT** ■ **AES256_KEYLENGTH_192BIT** ■ **AES256_KEYLENGTH_256BIT**

Returns

true if set correctly, false otherwise

3.5.2.15 bool AES256_setDecipherKey (uint32_t *moduleInstance,* const uint8_t * *cipherKey,* uint_fast16_t *keyLength*)

Sets the decipher key.

The API AES256_startSetDecipherKey or AES256_setDecipherKey must be invoked before invoking AES256_startDecryptData.

Parameters

moduleInstance	is the base address of the AES256 module.
cipherKey	is a pointer to an uint8_t array with a length of 16 bytes that contains a 128 bit cipher key.
keyLength	is the length of the key. Valid values are: ■ **AES256_KEYLENGTH_128BIT** ■ **AES256_KEYLENGTH_192BIT** ■ **AES256_KEYLENGTH_256BIT**

Returns

true if set, false otherwise

3.5.2.16 void AES256_startDecryptData (uint32_t *moduleInstance,* const uint8_t * *data*)

Decypts a block of data using the AES256 module.

This is the non-blocking equivalant of AES256_decryptData(). This function requires a pregenerated decryption key. A key can be loaded and pregenerated by using function AES256_setDecipherKey() or AES256_startSetDecipherKey(). The decryption takes 167 MCLK. It is recommended to use interrupt to check for procedure completion then use the AES256_getDataOut() API to retrieve the decrypted data.

Parameters

moduleInstance	is the base address of the AES256 module.
data	is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

Returns
None

3.5.2.17 void AES256_startEncryptData (uint32_t *moduleInstance,* const uint8_t * *data*)

Starts an encryption process on the AES256 module.

The cipher key that is used for decryption should be loaded in advance by using function AES256_setCipherKey(). This is a non-blocking equivalent pf AES256_encryptData(). It is recommended to use the interrupt functionality to check for procedure completion then use the AES256_getDataOut() API to retrieve the encrypted data.

Parameters

moduleInstance	is the base address of the AES256 module.
data	is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.

Returns
None

3.5.2.18 bool AES256_startSetDecipherKey (uint32_t *moduleInstance,* const uint8_t * *cipherKey,* uint_fast16_t *keyLength*)

Sets the decipher key.

The API AES256_startSetDecipherKey() or AES256_setDecipherKey() must be invoked before invoking AES256_startDecryptData.

Parameters

moduleInstance	is the base address of the AES256 module.
cipherKey	is a pointer to an uint8_t array with a length of 16 bytes that contains a 128 bit cipher key.
keyLength	is the length of the key. Valid values are: ■ **AES256_KEYLENGTH_128BIT** ■ **AES256_KEYLENGTH_192BIT** ■ **AES256_KEYLENGTH_256BIT**

Returns
true if set correctly, false otherwise

3.5.2.19 void AES256_unregisterInterrupt (uint32_t *moduleInstance*)

Unregisters the interrupt handler for the AES interrupt

Parameters

moduleInstance	Instance of the AES256 module

This function unregisters the handler to be called when AES interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

4 Analog Comparator (COMP_E)

4.1 Module Operation

The Comparator (Comp) API provides a set of functions for using the SDK COMP_E modules. Functions are provided to initialize the COMP_E modules, setup reference voltages for input, and manage interrupts for the COMP_E modules.

The COMP_E module provides the ability to compare two analog signals and use the output in software and on an output pin. The output represents whether the signal on the positive terminal is higher than the signal on the negative terminal. The COMP_E moduke may be used to generate a hysteresis. There are 16 different inputs that can be used, as well as the ability to short 2 input together. The COMP_E module also has control over the REF_A module to generate a reference voltage as an input.

The COMP_E module can generate multiple interrupts. An interrupt may be asserted for the output, with separate interrupts on whether the output rises, or falls.

4.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the COMP_E module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a simple example of how to setup the COMP_E module to setup a comparator window with a Vcompare of 1.2v using the internal reference.

First, below is an example of setting up the COMP_E module configuration structure:

```
/* Comparator configuration structure */
const COMP_E_Config compConfig =
{
        COMP_E_VREF,                    // Positive Input Terminal
        COMP_E_INPUT6,                  // Negative Input Terminal
        COMP_E_FILTEROUTPUT_DLYLVL4,    // Delay Level 4 Filter
        COMP_E_NORMALOUTPUTPOLARITY    // Normal Output Polarity
};
```

Below are the actual DriverLib calls to configure/setup the Comp module:

```
    /* Initialize the Comparator module
     *  Comparator Instance 1
     *  Pin CE16 to Positive(+) Terminal
     *  Reference Voltage to Negative(-) Terminal
     *  Normal Power Mode
     *  Output Filter On with max delay
     *  Non-Inverted Output Polarity
     */
    MAP_COMP_E_initModule(COMP_E1_BASE,  &compConfig);

    /*
     * Base Address of Comparator E,
     * Reference Voltage of 1.2 V,
     * Lower Limit of 1.2*(32/32) = 1.2V,
     * Upper Limit of 1.2*(32/32) = 1.2V
     */
    MAP_COMP_E_setReferenceVoltage(COMP_E1_BASE,  COMP_E_VREFBASE1_2V,  32,  32);

    /* Enable COMP_E Interrupt on default rising edge for CEIFG */
    MAP_COMP_E_setInterruptEdgeDirection(COMP_E1_BASE,  COMP_E_RISINGEDGE);

    /* Enable Interrupts
     *  Comparator Instance 1,
     *  Enable COMPE Interrupt on default rising edge for CEIFG
     */
    MAP_COMP_E_clearInterruptFlag(COMP_E1_BASE,  COMP_E_OUTPUT_INTERRUPT);
    MAP_COMP_E_enableInterrupt(COMP_E1_BASE,  COMP_E_OUTPUT_INTERRUPT);
    MAP_Interrupt_enableInterrupt(INT_COMP_E1);
    MAP_Interrupt_enableMaster();
```

4.3 Definitions

Data Structures

- struct _COMP_E_Config

Functions

- void COMP_E_clearInterruptFlag (uint32_t comparator, uint_fast16_t mask)
- void COMP_E_disableInputBuffer (uint32_t comparator, uint_fast16_t inputPort)
- void COMP_E_disableInterrupt (uint32_t comparator, uint_fast16_t mask)
- void COMP_E_disableModule (uint32_t comparator)
- void COMP_E_enableInputBuffer (uint32_t comparator, uint_fast16_t inputPort)
- void COMP_E_enableInterrupt (uint32_t comparator, uint_fast16_t mask)
- void COMP_E_enableModule (uint32_t comparator)
- uint_fast16_t COMP_E_getEnabledInterruptStatus (uint32_t comparator)
- uint_fast16_t COMP_E_getInterruptStatus (uint32_t comparator)
- bool COMP_E_initModule (uint32_t comparator, const COMP_E_Config *config)
- uint8_t COMP_E_outputValue (uint32_t comparator)
- void COMP_E_registerInterrupt (uint32_t comparator, void(*intHandler)(void))
- void COMP_E_setInterruptEdgeDirection (uint32_t comparator, uint_fast8_t edgeDirection)
- void COMP_E_setPowerMode (uint32_t comparator, uint_fast16_t powerMode)
- void COMP_E_setReferenceAccuracy (uint32_t comparator, uint_fast16_t referenceAccuracy)
- void COMP_E_setReferenceVoltage (uint32_t comparator, uint_fast16_t supplyVoltageReferenceBase, uint_fast16_t lowerLimitSupplyVoltageFractionOf32, uint_fast16_t upperLimitSupplyVoltageFractionOf32)
- void COMP_E_shortInputs (uint32_t comparator)
- void COMP_E_swapIO (uint32_t comparator)
- void COMP_E_toggleInterruptEdgeDirection (uint32_t comparator)
- void COMP_E_unregisterInterrupt (uint32_t comparator)
- void COMP_E_unshortInputs (uint32_t comparator)

4.3.1 Detailed Description

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

4.3.2 Data Structure Documentation

4.3.2.1 struct _COMP_E_Config

Type definition for _COMP_E_Config structure.

ypedef COMP_E_Config

Configuration structure for Comparator module. See COMP_E_initModule for parameter documentation.

4.3.3 Function Documentation

4.3.3.1 void COMP_E_clearInterruptFlag (uint32_t *comparator,* uint_fast16_t *mask*)

Clears Comparator interrupt flags.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
mask	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following ■ **COMP_E_INTERRUPT_FLAG** - Output interrupt flag ■ **COMP_E_INTERRUPT_FLAG_INVERTED_POLARITY** - Output interrupt flag inverted polarity ■ **COMP_E_INTERRUPT_FLAG_READY** - Ready interrupt flag

The Comparator interrupt source is cleared, so that it no longer asserts. The highest interrupt flag is automatically cleared when an interrupt vector generator is used.

Returns
NONE

4.3.3.2 void COMP_E_disableInputBuffer (uint32_t *comparator,* uint_fast16_t *inputPort*)

Disables the input buffer of the selected input port to effectively allow for analog signals.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
inputPort	is the port in which the input buffer will be disabled. Valid values are a logical OR of the following: ■ **COMP_E_INPUT0** [Default] ■ **COMP_E_INPUT1** ■ **COMP_E_INPUT2** ■ **COMP_E_INPUT3** ■ **COMP_E_INPUT4** ■ **COMP_E_INPUT5** ■ **COMP_E_INPUT6** ■ **COMP_E_INPUT7** ■ **COMP_E_INPUT8** ■ **COMP_E_INPUT9** ■ **COMP_E_INPUT10** ■ **COMP_E_INPUT11** ■ **COMP_E_INPUT12** ■ **COMP_E_INPUT13** ■ **COMP_E_INPUT14** ■ **COMP_E_INPUT15** Modified bits are **CEPDx** of **CECTL3** register.

This function sets the bit to disable the buffer for the specified input port to allow for analog signals from any of the comparator input pins. This bit is automatically set when the input is initialized to be used with the comparator module. This function should be used whenever an analog input is connected to one of these pins to prevent parasitic voltage from causing unexpected results.

Returns
NONE

4.3.3.3 void COMP_E_disableInterrupt (uint32_t *comparator,* uint_fast16_t *mask*)

Disables selected Comparator interrupt sources.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
mask	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following ■ **COMP_E_OUTPUT_INTERRUPT** - Output interrupt ■ **COMP_E_INVERTED_POLARITY_INTERRUPT** - Output interrupt inverted polarity ■ **COMP_E_READY_INTERRUPT** - Ready interrupt

Disables the indicated Comparator 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

4.3.3.4 void COMP_E_disableModule (uint32_t *comparator*)

Turns off the Comparator module.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function clears the CEON bit disabling the operation of the Comparator module, saving from excess power consumption.

Modified bits are **CEON** of **CECTL1** register.

Returns
NONE

4.3.3.5 void COMP_E_enableInputBuffer (uint32_t *comparator,* uint_fast16_t *inputPort*)

Enables the input buffer of the selected input port to allow for digital signals.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
inputPort	is the port in which the input buffer will be enabled. Valid values are a logical OR of the following: ■ **COMP_E_INPUT0** [Default] ■ **COMP_E_INPUT1** ■ **COMP_E_INPUT2** ■ **COMP_E_INPUT3** ■ **COMP_E_INPUT4** ■ **COMP_E_INPUT5** ■ **COMP_E_INPUT6** ■ **COMP_E_INPUT7** ■ **COMP_E_INPUT8** ■ **COMP_E_INPUT9** ■ **COMP_E_INPUT10** ■ **COMP_E_INPUT11** ■ **COMP_E_INPUT12** ■ **COMP_E_INPUT13** ■ **COMP_E_INPUT14** ■ **COMP_E_INPUT15** Modified bits are **CEPDx** of **CECTL3** register.

This function clears the bit to enable the buffer for the specified input port to allow for digital signals from any of the comparator input pins. This should not be reset if there is an analog signal connected to the specified input pin to prevent from unexpected results.

Returns
NONE

4.3.3.6 void COMP_E_enableInterrupt (uint32_t *comparator,* uint_fast16_t *mask*)

Enables selected Comparator interrupt sources.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
mask	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following ■ **COMP_E_OUTPUT_INTERRUPT** - Output interrupt ■ **COMP_E_INVERTED_POLARITY_INTERRUPT** - Output interrupt inverted polarity ■ **COMP_E_READY_INTERRUPT** - Ready interrupt

Enables the indicated Comparator interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The default trigger for the non-inverted interrupt is a rising edge of the output, this can be changed with the interruptSetEdgeDirection() function.

Returns
NONE

4.3.3.7 void COMP_E_enableModule (uint32_t *comparator*)

Turns on the Comparator module.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function sets the bit that enables the operation of the Comparator module.

Returns
NONE

4.3.3.8 uint_fast16_t COMP_E_getEnabledInterruptStatus (uint32_t *comparator*)

Enables selected Comparator interrupt sources masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

Enables the indicated Comparator interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The default trigger for the non-inverted interrupt is a rising edge of the output, this can be changed with the COMP_E_setInterruptEdgeDirection() function.

Returns
NONE

References COMP_E_getInterruptStatus().

4.3.3.9 uint_fast16_t COMP_E_getInterruptStatus (uint32_t *comparator*)

Gets the current Comparator interrupt status.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This returns the interrupt status for the Comparator module based on which flag is passed.

Returns
The current interrupt flag status for the corresponding mask.

Referenced by COMP_E_getEnabledInterruptStatus().

4.3.3.10 bool COMP_E_initModule (uint32_t *comparator,* const **COMP_E_Config** * *config*)

Initializes the Comparator Module.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
config	Configuration structure for the Comparator module

Configuration options for COMP_E_Config structure.

Parameters

positiveTerminalInput	selects the input to the positive terminal. Valid values are ■ **COMP_E_INPUT0** [Default] ■ **COMP_E_INPUT1** ■ **COMP_E_INPUT2** ■ **COMP_E_INPUT3** ■ **COMP_E_INPUT4** ■ **COMP_E_INPUT5** ■ **COMP_E_INPUT6** ■ **COMP_E_INPUT7** ■ **COMP_E_INPUT8** ■ **COMP_E_INPUT9** ■ **COMP_E_INPUT10** ■ **COMP_E_INPUT11** ■ **COMP_E_INPUT12** ■ **COMP_E_INPUT13** ■ **COMP_E_INPUT14** ■ **COMP_E_INPUT15** ■ **COMP_E_VREF** Modified bits are **CEIPSEL** and **CEIPEN** of **CECTL0** register, **CERSEL** of **CECTL2** register, and CEPDx of **CECTL3** register.

negativeTerminalInput	selects the input to the negative terminal. Valid values are: ■ **COMP_E_INPUT0** [Default] ■ **COMP_E_INPUT1** ■ **COMP_E_INPUT2** ■ **COMP_E_INPUT3** ■ **COMP_E_INPUT4** ■ **COMP_E_INPUT5** ■ **COMP_E_INPUT6** ■ **COMP_E_INPUT7** ■ **COMP_E_INPUT8** ■ **COMP_E_INPUT9** ■ **COMP_E_INPUT10** ■ **COMP_E_INPUT11** ■ **COMP_E_INPUT12** ■ **COMP_E_INPUT13** ■ **COMP_E_INPUT14** ■ **COMP_E_INPUT15** ■ **COMP_E_VREF** Modified bits are **CEIMSEL** and **CEIMEN** of **CECTL0** register, **CERSEL** of **CECTL2** register, and CEPDx of **CECTL3** register.

outputFilterEnableAndDelayLevel	controls the output filter delay state, which is either off or enabled with a specified delay level. Valid values are ■ **COMP_E_FILTEROUTPUT_OFF** [Default] ■ **COMP_E_FILTEROUTPUT_DLYLVL1** ■ **COMP_E_FILTEROUTPUT_DLYLVL2** ■ **COMP_E_FILTEROUTPUT_DLYLVL3** ■ **COMP_E_FILTEROUTPUT_DLYLVL4** This parameter is device specific and delay levels should be found in the device's datasheet. Modified bits are **CEF** and **CEFDLY** of **CECTL1** register.
invertedOutputPolarity	controls if the output will be inverted or not. Valid values are ■ **COMP_E_NORMALOUTPUTPOLARITY** - indicates the output should be normal. [Default] ■ **COMP_E_INVERTEDOUTPUTPOLARITY** - the output should be inverted. Modified bits are **CEOUTPOL** of **CECTL1** register.
powerMode	controls the power mode of the module ■ **COMP_E_HIGH_SPEED_MODE** [default] ■ **COMP_E_NORMAL_MODE** ■ **COMP_E_ULTRA_LOW_POWER_MODE** Upon successful initialization of the Comparator module, this function will have reset all necessary register bits and set the given options in the registers. To actually use the comparator module, the COMP_E_enableModule() function must be explicitly called before use. If a Reference Voltage is set to a terminal, the Voltage should be set using the COMP_E_setReferenceVoltage() function.

Returns

true or false of the initialization process.

4.3.3.11 uint8_t COMP_E_outputValue (uint32_t *comparator*)

Returns the output value of the Comparator module.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

Returns the output value of the Comparator module.

Returns
COMP_E_HIGH or COMP_E_LOW as the output value of the Comparator module.

4.3.3.12 void COMP_E_registerInterrupt (uint32_t *comparator,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the Comparator E interrupt.

Parameters

intHandler	is a pointer to the function to be called when the Comparator interrupt occurs.
comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function registers the handler to be called when a Comparator interrupt occurs. This function enables the global interrupt in the interrupt controller; specific Comparator interrupts must be enabled via COMP_E_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via COMP_E_clearInterruptFlag().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

4.3.3.13 void COMP_E_setInterruptEdgeDirection (uint32_t *comparator,* uint_fast8_t *edgeDirection*)

Explicitly sets the edge direction that would trigger an interrupt.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
edgeDirection	determines which direction the edge would have to go to generate an interrupt based on the non-inverted interrupt flag. Valid values are ■ **COMP_E_FALLINGEDGE** - sets the bit to generate an interrupt when the output of the comparator falls from HIGH to LOW if the normal interrupt bit is set(and LOW to HIGH if the inverted interrupt enable bit is set). [Default] ■ **COMP_E_RISINGEDGE** - sets the bit to generate an interrupt when the output of the comparator rises from LOW to HIGH if the normal interrupt bit is set(and HIGH to LOW if the inverted interrupt enable bit is set). Modified bits are **CEIES** of **CECTL1** register.

This function will set which direction the output will have to go, whether rising or falling, to generate an interrupt based on a non-inverted interrupt.

Returns
NONE

4.3.3.14 void COMP_E_setPowerMode (uint32_t *comparator,* uint_fast16_t *powerMode*)

Sets the power mode

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
powerMode	decides the power mode Valid values are ■ **COMP_E_HIGH_SPEED_MODE** ■ **COMP_E_NORMAL_MODE** ■ **COMP_E_ULTRA_LOW_POWER_MODE** Modified bits are **CEPWRMD** of **CECTL1** register.

Returns
NONE

4.3.3.15 void COMP_E_setReferenceAccuracy (uint32_t *comparator,* uint_fast16_t *referenceAccuracy*)

Sets the reference accuracy

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
referenceAccuracy	is the reference accuracy setting of the comparator. Clocked is for low power/low accuracy. Valid values are ■ **COMP_E_ACCURACY_STATIC** ■ **COMP_E_ACCURACY_CLOCKED** Modified bits are **CEREFACC** of **CECTL2** register.

The reference accuracy is set to the desired setting. Clocked is better for low power operations but has a lower accuracy.

Returns
NONE

4.3.3.16 void COMP_E_setReferenceVoltage (uint32_t *comparator,* uint_fast16_t *supplyVoltageReferenceBase,* uint_fast16_t *lowerLimitSupplyVoltageFractionOf32,* uint_fast16_t *upperLimitSupplyVoltageFractionOf32*)

Generates a Reference Voltage to the terminal selected during initialization.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**
supplyVoltageReferenceBase	decides the source and max amount of Voltage that can be used as a reference. Valid values are ■ **COMP_E_REFERENCE_AMPLIFIER_DISABLED** ■ **COMP_E_VREFBASE1_2V** ■ **COMP_E_VREFBASE2_0V** ■ **COMP_E_VREFBASE2_5V**
upperLimitSupplyVoltageFractionOf32	is the numerator of the equation to generate the reference voltage for the upper limit reference voltage. Valid values are between 1 and 32.
lowerLimitSupplyVoltageFractionOf32	is the numerator of the equation to generate the reference voltage for the lower limit reference voltage. Valid values are between 1 and 32. Modified bits are **CEREF0** of **CECTL2** register.

Use this function to generate a voltage to serve as a reference to the terminal selected at initialization. The voltage is determined by the equation: Vbase $*$ (Numerator / 32). If the upper and lower limit voltage numerators are equal, then a static reference is defined, whereas they are different then a hysteresis effect is generated.

Returns
NONE

4.3.3.17 void COMP_E_shortInputs (uint32_t *comparator*)

Shorts the two input pins chosen during initialization.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function sets the bit that shorts the devices attached to the input pins chosen from the initialization of the comparator.

Modified bits are **CESHORT** of **CECTL1** register.

Returns
NONE

4.3.3.18 void COMP_E_swapIO (uint32_t *comparator*)

Toggles the bit that swaps which terminals the inputs go to, while also inverting the output of the comparator.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ \ bCOMP_E0 ■ \ bCOMP_E1

This function toggles the bit that controls which input goes to which terminal. After initialization, this bit is set to 0, after toggling it once the inputs are routed to the opposite terminal and the output is inverted.

Modified bits are **CEEX** of **CECTL1** register.

Returns
NONE

4.3.3.19 void COMP_E_toggleInterruptEdgeDirection (uint32_t *comparator*)

Toggles the edge direction that would trigger an interrupt.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function will toggle which direction the output will have to go, whether rising or falling, to generate an interrupt based on a non-inverted interrupt. If the direction was rising, it is now falling, if it was falling, it is now rising.

Modified bits are **CEIES** of **CECTL1** register.

Returns
NONE

4.3.3.20 void COMP_E_unregisterInterrupt (uint32_t *comparator*)

Unregisters the interrupt handler for the Comparator E interrupt

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function unregisters the handler to be called when Comparator E interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

4.3.3.21 void COMP_E_unshortInputs (uint32_t *comparator*)

Disables the short of the two input pins chosen during initialization.

Parameters

comparator	is the instance of the Comparator module. Valid parameters vary from part to part, but can include: ■ **COMP_E0_BASE** ■ **COMP_E1_BASE**

This function clears the bit that shorts the devices attached to the input pins chosen from the initialization of the comparator.

Modified bits are **CESHORT** of **CECTL1** register.

Returns
NONE

5 Cyclic Redundancy Check 32 (CRC32)

5.1 Module Operation

The Cyclic Redundancy Check 32 (CRC32) API provides a set of functions for using the SDK CRC32 module. Functions are provided to initialize the CRC and create a CRC signature to check the validity of data. This is mostly useful in the communication of data, or as a startup procedure to as a more complex and accurate check of data.

The CRC32 module offers no interrupts and is used only to generate CRC signatures to verify against pre-made CRC signatures (Checksums).

The CRC32 module provides the capability for both 32-bit and 16-bit calculations. As such, the DriverLib API provides functionality for the user to provide variable bit-length data for either 16-bit or 32-bit calculations.

5.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the CRC32 module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

In the following very simple code example, an array of data is fed into the CRC32 module and the 32-bit calculation is retrieved:

```
int main(void)
{
    volatile uint32_t hwCalculatedCRC,  swCalculatedCRC;
    uint32_t ii;

    /* Stop WDT */
    MAP_WDT_A_holdTimer();

    MAP_CRC32_setSeed(CRC32_INIT,  CRC32_MODE);

    for (ii = 0; ii < 9; ii++)
        MAP_CRC32_set8BitData(myData[ii],  CRC32_MODE);

    /* Getting the result from the hardware module */
    hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

    /* Calculating the CRC32 checksum through software */
    swCalculatedCRC = calculateCRC32((uint8_t*) myData,  9);

    /* Pause for the debugger */
    __no_operation();
}
```

5.3 Definitions

Functions

- uint32_t CRC32_getResult (uint_fast8_t crcType)
- uint32_t CRC32_getResultReversed (uint_fast8_t crcType)
- void CRC32_set16BitData (uint16_t dataIn, uint_fast8_t crcType)
- void CRC32_set16BitDataReversed (uint16_t dataIn, uint_fast8_t crcType)
- void CRC32_set32BitData (uint32_t dataIn)
- void CRC32_set32BitDataReversed (uint32_t dataIn)
- void CRC32_set8BitData (uint8_t dataIn, uint_fast8_t crcType)
- void CRC32_set8BitDataReversed (uint8_t dataIn, uint_fast8_t crcType)
- void CRC32_setSeed (uint32_t seed, uint_fast8_t crcType)

5.3.1 Detailed Description

The code for this module is contained in `driverlib/crc32.c` and `driverlib/legacy/MSP432xx/legacy_crc32.c`, with `driverlib/crc32.h` and `driverlib/legacy/MSP432xx/legacy_crc32.h` containing the API declarations for use by applications.

5.3.2 Function Documentation

5.3.2.1 uint32_t CRC32_getResult (uint_fast8_t *crcType*)

Returns the value of CRC Signature Result.

Parameters

crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function returns the value of the signature result generated by the CRC. Bit 0 is treated as LSB.

Returns
uint32_t Result

5.3.2.2 uint32_t CRC32_getResultReversed (uint_fast8_t *crcType*)

Returns the bit-wise reversed format of the 32 bit Signature Result.

Parameters

crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function returns the bit-wise reversed format of the Signature Result. Bit 0 is treated as MSB.

Returns
uint32_t Result

5.3.2.3 void CRC32_set16BitData (uint16_t *dataIn,* uint_fast8_t *crcType*)

Sets the 16 Bit data to add into the CRC module to generate a new signature.

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC16DIW0** of **CRC16DIW0** register. **CRC32DIW0** of **CRC32DIW0** register.
crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function sets the given data into the CRC module to generate the new signature from the current signature and new data. Bit 0 is treated as LSB

Returns
NONE

5.3.2.4 void CRC32_set16BitDataReversed (uint16_t *dataIn,* uint_fast8_t *crcType*)

Translates the data by reversing the bits in each 16 bit data and then sets this data to add into the CRC module to generate a new signature.

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC16DIRBW0** of **CRC16DIRBW0** register. **CRC32DIRBW0** of **CRC32DIRBW0** register.
crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function first reverses the bits in each byte of the data and then generates the new signature from the current signature and new translated data. Bit 0 is treated as MSB.

Returns
NONE

5.3.2.5 void CRC32_set32BitData (uint32_t *dataIn*)

Sets the 32 Bit data to add into the CRC module to generate a new signature. Available only for CRC32_MODE and not for CRC16_MODE

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC32DIL0** of **CRC32DIL0** register.

This function sets the given data into the CRC module to generate the new signature from the current signature and new data. Bit 0 is treated as LSB

Returns
NONE

5.3.2.6 void CRC32_set32BitDataReversed (uint32_t *dataIn*)

Translates the data by reversing the bits in each 32 Bit Data and then sets this data to add into the CRC module to generate a new signature. Available only for CRC32 mode and not for CRC16 mode

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC32DIRBL0** of **CRC32DIRBL0** register.

This function first reverses the bits in each byte of the data and then generates the new signature from the current signature and new translated data. Bit 0 is treated as MSB.

Returns
NONE

5.3.2.7 void CRC32_set8BitData (uint8_t *dataIn,* uint_fast8_t *crcType*)

Sets the 8 Bit data to add into the CRC module to generate a new signature.

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC16DIB0** of **CRC16DIB0** register. **CRC32DIB0** of **CRC32DIB0** register.
crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function sets the given data into the CRC module to generate the new signature from the current signature and new data. Bit 0 is treated as LSB.

Returns
NONE

5.3.2.8 void CRC32_set8BitDataReversed (uint8_t *dataIn,* uint_fast8_t *crcType*)

Translates the data by reversing the bits in each 8 bit data and then sets this data to add into the CRC module to generate a new signature.

Parameters

dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRC16DIRBB0** of **CRC16DIRBB0** register. **CRC32DIRBB0** of **CRC32DIRBB0** register.
crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function first reverses the bits in each byte of the data and then generates the new signature from the current signature and new translated data. Bit 0 is treated as MSB.

Returns
NONE

5.3.2.9 void CRC32_setSeed (uint32_t *seed,* uint_fast8_t *crcType*)

Sets the seed for the CRC.

Parameters

seed	is the seed for the CRC to start generating a signature from. Modified bits are **CRC16INIRESL0** of **CRC16INIRESL0** register. **CRC32INIRESL0** of **CRC32INIRESL0** register
crcType	selects between CRC32 and CRC16 Valid values are **CRC16_MODE** and **CRC32_MODE**

This function sets the seed for the CRC to begin generating a signature with the given seed and all passed data. Using this function resets the CRC32 signature.

Returns
NONE

6 Clock System (CS)

6.1 Module Operation

The clock system module for DriverLib gives users the ability to fully configure and control all aspects of the MSP432 clock system. This includes initializing and maintaining the MCLK, ACLK, HSMCLK, SMCLK, and BCLK clock systems. Additionally, APIs exist for configuring connected crystal oscillators as well as configuring/manipulating the DCO and reference oscillator.

6.2 Timeout Parameters

For crystal configuration APIs (starting the LFXT and HFXT), a "timeout" API exists that will return control of execution back to the user application if a specified duration passes. The variable that is passed into these functions is a unit of time specified by how many "loop iterations" elapse before unsuccessful stabilization of the respected crystal. The API will attempt to check to see if there was a crystal fault, clear the crystal fault flag, and repeat the check until no fault exists. If the user calls the API with a specified timeout, the loop will only check the given number of loop iterations for a successfully stabilized crystal.

6.3 Custom DCO Frequency

For tuning the DCO frequency to a specific frequency, a convenient CS_setDCOFrequency function is provided to users. This function accepts any uint32_t frequency and automatically calculates the appropriate tuning parameters to get the DCO frequency as close as possible to the provided frequency. Note that with this function, floating point math is involved so if efficiency is a concern the user should enable the FPU using the FPU_enableModule function.

6.4 Specifying External Crystal Frequencies

MSP432 DriverLib has a variety of convenience functions for obtaining the specific frequency of a clock source (such as CS_getMCLK). If a clock source is sourced from an external crystal, the crystal frequency must be specified explicitly using the CS_setExternalClockSourceFrequency function. This function must be used prior to the getters for the clock source if an external crystal is used.

6.5 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the CS module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to start the external high frequency crystal and source MCLK from this crystal. An LED is configured as an output in this example as well. For a set of more detailed code examples, please refer to the code examples in the examples/ directory of the SDK release:

```
/* Configuring pins for peripheral/crystal usage and LED for output */
MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_PJ,
        GPIO_PIN3 | GPIO_PIN2,  GPIO_PRIMARY_MODULE_FUNCTION);
MAP_GPIO_setAsOutputPin(GPIO_PORT_P1,  GPIO_PIN0);

/* Just in case the user wants to use the getACLK,  getMCLK,  etc. functions,
 * let's set the clock frequency in the code.
 */
CS_setExternalClockSourceFrequency(32000, 48000000);

/* Starting HFXT in non-bypass mode without a timeout. Before we start
 * we have to change VCORE to 1 to support the 48MHz frequency */
MAP_PCM_setCoreVoltageLevel(PCM_VCORE1);
MAP_FlashCtl_A_setWaitState(FLASH_A_BANK0,  2);
MAP_FlashCtl_A_setWaitState(FLASH_A_BANK1,  2);
CS_startHFXT(false);

/* Initializing MCLK to HFXT (effectively 48MHz) */
MAP_CS_initClockSignal(CS_MCLK,  CS_HFXTCLK_SELECT,  CS_CLOCK_DIVIDER_1);
```

6.6 Definitions

Functions

- void CS_clearInterruptFlag (uint32_t flags)
- void CS_disableClockRequest (uint32_t selectClock)
- void CS_disableDCOExternalResistor (void)
- void CS_disableFaultCounter (uint_fast8_t counterSelect)
- void CS_disableInterrupt (uint32_t flags)
- void CS_enableClockRequest (uint32_t selectClock)
- void CS_enableDCOExternalResistor (void)
- void CS_enableFaultCounter (uint_fast8_t counterSelect)
- void CS_enableInterrupt (uint32_t flags)
- uint32_t CS_getACLK (void)
- uint32_t CS_getBCLK (void)
- uint32_t CS_getDCOFrequency (void)
- uint32_t CS_getEnabledInterruptStatus (void)
- uint32_t CS_getHSMCLK (void)
- uint32_t CS_getInterruptStatus (void)
- uint32_t CS_getMCLK (void)
- uint32_t CS_getSMCLK (void)
- void CS_initClockSignal (uint32_t selectedClockSignal, uint32_t clockSource, uint32_t clockSourceDivider)
- void CS_registerInterrupt (void(*intHandler)(void))
- void CS_resetFaultCounter (uint_fast8_t counterSelect)
- void CS_setDCOCenteredFrequency (uint32_t dcoFreq)
- void CS_setDCOExternalResistorCalibration (uint_fast8_t uiCalData, uint_fast8_t freqRange)
- void CS_setDCOFrequency (uint32_t dcoFrequency)
- void CS_setExternalClockSourceFrequency (uint32_t lfxt_XT_CLK_frequency, uint32_t hfxt_XT_CLK_frequency)
- void CS_setReferenceOscillatorFrequency (uint8_t referenceFrequency)
- void CS_startFaultCounter (uint_fast8_t counterSelect, uint_fast8_t countValue)
- bool CS_startHFXT (bool bypassMode)
- bool CS_startHFXTWithTimeout (bool bypassMode, uint32_t timeout)
- bool CS_startLFXT (uint32_t xtDrive)
- bool CS_startLFXTWithTimeout (uint32_t xtDrive, uint32_t timeout)
- void CS_tuneDCOFrequency (int16_t tuneParameter)
- void CS_unregisterInterrupt (void)

6.6.1 Detailed Description

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

6.6.2 Function Documentation

6.6.2.1 void CS_clearInterruptFlag (uint32_t *flags*)

Clears clock system interrupt sources.

Parameters

flags	is a bit mask of the interrupt sources to be cleared. Must be a logical OR of: ■ **CS_LFXT_FAULT**, ■ **CS_HFXT_FAULT**, ■ **CS_DCO_OPEN_FAULT**, ■ **CS_STARTCOUNT_LFXT_FAULT**, ■ **CS_STARTCOUNT_HFXT_FAULT**,

The specified clock system interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep it from being called again immediately upon exit.

Note

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns

None.

6.6.2.2 void CS_disableClockRequest (uint32_t *selectClock*)

Disables conditional module requests

Parameters

selectClock	selects specific request disables. Valid values are are a logical OR of the following values: ■ **CS_ACLK**, ■ **CS_HSMCLK**, ■ **CS_SMCLK**, ■ **CS_MCLK**

Returns

NONE

6.6.2.3 void CS_disableDCOExternalResistor (void)

Disables the external resistor for DCO operation

Returns
NONE

6.6.2.4 void CS_disableFaultCounter (uint_fast8_t *counterSelect*)

Disables the fault counter for the CS module. This function can disable either the HFXT fault counter or the LFXT fault counter.

Parameters

counterSelect	selects the fault counter to disable ■ **CS_HFXT_FAULT_COUNTER** ■ **CS_LFXT_FAULT_COUNTER**

Returns
NONE

6.6.2.5 void CS_disableInterrupt (uint32_t *flags*)

Disables individual clock system interrupt sources.

Parameters

flags	is a bit mask of the interrupt sources to be disabled. Must be a logical OR of: ■ **CS_LFXT_FAULT**, ■ **CS_HFXT_FAULT**, ■ **CS_DCOMIN_FAULT**, ■ **CS_DCOMAX_FAULT**, ■ **CS_DCO_OPEN_FAULT**, ■ **CS_STARTCOUNT_LFXT_FAULT**, ■ **CS_STARTCOUNT_HFXT_FAULT**,

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns
None.

6.6.2.6 void CS_enableClockRequest (uint32_t *selectClock*)

Enables conditional module requests

Parameters

selectClock	selects specific request enables. Valid values are are a logical OR of the following values: ■ **CS_ACLK**, ■ **CS_HSMCLK**, ■ **CS_SMCLK**, ■ **CS_MCLK**

Returns
NONE

6.6.2.7 void CS_enableDCOExternalResistor (void)

Enables the external resistor for DCO operation

Returns
NONE

6.6.2.8 void CS_enableFaultCounter (uint_fast8_t *counterSelect*)

Enables the fault counter for the CS module. This function can enable either the HFXT fault counter or the LFXT fault counter.

Parameters

counterSelect	selects the fault counter to enable ■ **CS_HFXT_FAULT_COUNTER** ■ **CS_LFXT_FAULT_COUNTER**

Returns
NONE

6.6.2.9 void CS_enableInterrupt (uint32_t *flags*)

Enables individual clock control interrupt sources.

Parameters

flags	is a bit mask of the interrupt sources to be enabled. Must be a logical OR of: ■ **CS_LFXT_FAULT**, ■ **CS_HFXT_FAULT**, ■ **CS_DCOMIN_FAULT**, ■ **CS_DCOMAX_FAULT**, ■ **CS_DCO_OPEN_FAULT**, ■ **CS_STARTCOUNT_LFXT_FAULT**, ■ **CS_STARTCOUNT_HFXT_FAULT**,

This function enables the indicated clock system interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns
None.

6.6.2.10 uint32_t CS_getACLK (void)

Get the current ACLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_setExternalClockSourceFrequency() API was invoked before in case LFXT is being used.

Returns
Current ACLK frequency in Hz

6.6.2.11 uint32_t CS_getBCLK (void)

Get the current BCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_setExternalClockSourceFrequency API was invoked before in case LFXT or HFXT is being used.

Returns
Current BCLK frequency in Hz

6.6.2.12 uint32_t CS_getDCOFrequency (void)

Gets the current tuned DCO frequency. If no tuning has been done, this returns the nominal DCO frequency of the current DCO range. Note that this function will grab any constant/calibration data from the DDDS table without any user interaction needed.

Note
This function uses floating point math to calculate the DCO tuning parameter. If efficiency is a concern, the user should use the FPU_enableModule function (if available) to enable the floating point co-processor.

Returns
Current DCO frequency in Hz

6.6.2.13 uint32_t CS_getEnabledInterruptStatus (void)

Gets the current interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Returns
The current interrupt status, enumerated as a bit field of
- **CS_LFXT_FAULT**,
- **CS_HFXT_FAULT**,
- **CS_DCO_OPEN_FAULT**,
- **CS_DCO_SHORT_FAULT**,
- **CS_STARTCOUNT_LFXT_FAULT**,
- **CS_STARTCOUNT_HFXT_FAULT**,

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

References CS_getInterruptStatus().

6.6.2.14 uint32_t CS_getHSMCLK (void)

Get the current HSMCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_setExternalClockSourceFrequency API was invoked before in case LFXT or HFXT is being used.

Returns
Current HSMCLK frequency in Hz

6.6.2.15 uint32_t CS_getInterruptStatus (void)

Gets the current interrupt status.

Returns
The current interrupt status, enumerated as a bit field of:
- **CS_LFXT_FAULT**,
- **CS_HFXT_FAULT**,
- **CS_DCO_OPEN_FAULT**,
- **CS_DCO_SHORT_FAULT**,
- **CS_STARTCOUNT_LFXT_FAULT**,
- **CS_STARTCOUNT_HFXT_FAULT**,

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Referenced by CS_getEnabledInterruptStatus().

6.6.2.16 uint32_t CS_getMCLK (void)

Get the current MCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_setExternalClockSourceFrequency API was invoked before in case LFXT or HFXT is being used.

Returns
Current MCLK frequency in Hz

6.6.2.17 uint32_t CS_getSMCLK (void)

Get the current SMCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_setExternalClockSourceFrequency API was invoked before in case LFXT or HFXT is being used.

Returns
Current SMCLK frequency in Hz

6.6.2.18 void CS_initClockSignal (uint32_t *selectedClockSignal,* uint32_t *clockSource,* uint32_t *clockSourceDivider*)

This function initializes each of the clock signals. The user must ensure that this function is called for each clock signal. If not, the default state is assumed for the particular clock signal. Refer to DriverLib documentation for CS module or Device Family User's Guide for details of default clock signal states.

Note that this function is blocking and will wait on the appropriate bit to be set in the CSSTAT READY register to be set before setting the clock source.

Also note that when HSMCLK and SMCLK share the same clock signal. If you change the clock signal for HSMCLK, the clock signal for SMCLK will change also (and vice-versa).

HFXTCLK is not available for BCLK or ACLK.

Parameters

selectedClockSignal	Clock signal to initialize. ■ **CS_ACLK**, ■ **CS_MCLK**, ■ **CS_HSMCLK** ■ **CS_SMCLK** ■ **CS_BCLK** [clockSourceDivider is ignored for this parameter]
clockSource	Clock source for the selectedClockSignal signal. ■ **CS_LFXTCLK_SELECT**, ■ **CS_HFXTCLK_SELECT**, ■ **CS_VLOCLK_SELECT**, [Not available for BCLK] ■ **CS_DCOCLK_SELECT**, [Not available for ACLK, BCLK] ■ **CS_REFOCLK_SELECT**, ■ **CS_MODOSC_SELECT** [Not available for ACLK, BCLK]
clockSourceDivider	- selected the clock divider to calculate clock signal from clock source. This parameter is ignored when setting BLCK. Valid values are: ■ **CS_CLOCK_DIVIDER_1**, ■ **CS_CLOCK_DIVIDER_2**, ■ **CS_CLOCK_DIVIDER_4**, ■ **CS_CLOCK_DIVIDER_8**, ■ **CS_CLOCK_DIVIDER_16**, ■ **CS_CLOCK_DIVIDER_32**, ■ **CS_CLOCK_DIVIDER_64**, ■ **CS_CLOCK_DIVIDER_128**

Returns
NONE

6.6.2.19 void CS_registerInterrupt (void(∗)(void) *intHandler*)

Registers an interrupt handler for the clock system interrupt.

Parameters

intHandler	is a pointer to the function to be called when the clock system interrupt occurs.

This function registers the handler to be called when a clock system interrupt occurs. This function enables the global interrupt in the interrupt controller; specific clock system interrupts must be enabled via CS_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via CS_clearInterruptFlag().

Clock System can generate interrupts when

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

6.6.2.20 void CS_resetFaultCounter (uint_fast8_t *counterSelect*)

Resets the fault counter for the CS module. This function can reset either the HFXT fault counter or the LFXT fault counter.

Parameters

counterSelect	selects the fault counter to reset ■ **CS_HFXT_FAULT_COUNTER** ■ **CS_LFXT_FAULT_COUNTER**

Returns
NONE

6.6.2.21 void CS_setDCOCenteredFrequency (uint32_t *dcoFreq*)

Sets the centered frequency of DCO operation. Each frequency represents the centred frequency of a particular frequency range. Further tuning can be achieved by using the CS_tuneDCOFrequency function. Note that setting the nominal frequency will reset the tuning parameters.

Parameters

dcoFreq	selects between the valid frequencies: ■ **CS_DCO_FREQUENCY_1_5**, [1MHz to 2MHz] ■ **CS_DCO_FREQUENCY_3**, [2MHz to 4MHz] ■ **CS_DCO_FREQUENCY_6**, [4MHz to 8MHz] ■ **CS_DCO_FREQUENCY_12**, [8MHz to 16MHz] ■ **CS_DCO_FREQUENCY_24**, [16MHz to 32MHz] ■ **CS_DCO_FREQUENCY_48** [32MHz to 64MHz]

Returns
NONE

Referenced by CS_setDCOFrequency().

6.6.2.22 void CS_setDCOExternalResistorCalibration (uint_fast8_t *uiCalData,* uint_fast8_t *freqRange*)

Sets the calibration value for the DCO when using the external resistor mode. This value is used for tuning the DCO to custom frequencies. By default, the value in the CS module is populated by the calibration data of the suggested external resistor (see device datasheet).

Parameters

calData	is the calibration data constant for the external resistor.
freqRange	is the range of the DCO to set the external calibration for. Frequencies above 32MHZ have a different calibration value than frequencies below 32MHZ.

Returns
None

6.6.2.23 void CS_setDCOFrequency (uint32_t *dcoFrequency*)

Automatically sets/tunes the DCO to the given frequency. Any valid value up to max frequency in the spec can be given to this function and the API will do its best to determine the correct tuning parameter.

Note
The frequency ranges that can be custom tuned on early release MSP432 devices is limited. For further details on supported tunable frequencies, please refer to the device errata sheet or data sheet.

Parameters

dcoFrequency	Frequency in Hz that the user wants to set the DCO to.

Note
This function uses floating point math to calculate the DCO tuning parameter. If efficiency is a concern, the user should use the FPU_enableModule function (if available) to enable the floating point co-processor.

Returns
None

Automatically sets/tunes the DCO to the given frequency. Any valid value up to (and including) 64Mhz can be given to this function and the API will do its best to determine the correct tuning parameter.

Note

This function is not currently available on pre-release MSP432 devices. On early release versions of MSP432, the DCO calibration information has not been populated making the DCO only able to operate at the pre-calibrated centered frequencies accessible by the CS_setDCOCenteredFrequency function. While this function will be added on the final devices being released, for early silicon please default to the pre-calibrated DCO center frequencies.

Parameters

dcoFrequency	Frequency in Hz (1500000 - 64000000) that the user wants to set the DCO to.

Note

This function uses floating point math to calculate the DCO tuning parameter. If efficiency is a concern, the user should use the FPU_enableModule function (if available) to enable the floating point co-processor.

Returns

None

References CS_setDCOCenteredFrequency(), and CS_tuneDCOFrequency().

6.6.2.24 void CS_setExternalClockSourceFrequency (uint32_t *lfxt_XT_CLK_frequency,* uint32_t *hfxt_XT_CLK_frequency*)

This function sets the external clock sources LFXT and HFXT crystal oscillator frequency values. This function must be called if an external crystal LFXT or HFXT is used and the user intends to call CS_getSMCLK, CS_getMCLK, CS_getBCLK, CS_getHSMCLK, CS_getACLK and any of the HFXT oscillator control functions

Parameters

lfxt_XT_CLK_frequency	is the LFXT crystal frequencies in Hz
hfxt_XT_CLK_frequency	is the HFXT crystal frequencies in Hz

Returns

None

6.6.2.25 void CS_setReferenceOscillatorFrequency (uint8_t *referenceFrequency*)

Selects between the frequency of the internal REFO clock source

Parameters

referenceFrequency	selects between the valid frequencies: ■ **CS_REFO_32KHZ**, ■ **CS_REFO_128KHZ**,

Returns
NONE

6.6.2.26 void CS_startFaultCounter (uint_fast8_t *counterSelect,* uint_fast8_t *countValue*)

Sets the count for the start value of the fault counter. This function can be used to set either the HFXT count or the LFXT count.

Parameters

counterSelect	selects the fault counter to reset ■ **CS_HFXT_FAULT_COUNTER** ■ **CS_LFXT_FAULT_COUNTER**
countValue	selects the cycles to set the fault counter to ■ **CS_FAULT_COUNTER_4096_CYCLES** ■ **CS_FAULT_COUNTER_8192_CYCLES** ■ **CS_FAULT_COUNTER_16384_CYCLES** ■ **CS_FAULT_COUNTER_32768_CYCLES**

Returns
NONE

6.6.2.27 bool CS_startHFXT (bool *bypassMode*)

Initializes the HFXT crystal oscillator, which supports crystal frequencies between 0 MHz and 48 MHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared, with no timeout. See the device-specific data sheet for appropriate drive settings. NOTE: User must call CS_setExternalClockSourceFrequency to set frequency of external clocks before calling this function.

Parameters

bypassMode	When this variable is set, the oscillator will start in bypass mode and the signal can be generated by a digital square wave.

Returns
true if started correctly, false otherwise

References CS_startHFXTWithTimeout().

6.6.2.28 bool CS_startHFXTWithTimeout (bool *bypassMode,* uint32_t *timeout*)

Initializes the HFXT crystal oscillator, which supports crystal frequencies between 0 MHz and 48 MHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared,

with no timeout. See the device-specific data sheet for appropriate drive settings. NOTE: User must call CS_setExternalClockSourceFrequency to set frequency of external clocks before calling this function. This function has a timeout associated with stabilizing the oscillator.

Parameters

bypassMode	When this variable is set, the oscillator will start in bypass mode and the signal can be generated by a digital square wave.
timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Returns

true if started correctly, false otherwise

Referenced by CS_startHFXT().

6.6.2.29 bool CS_startLFXT (uint32_t *xtDrive*)

Initializes the LFXT crystal oscillator, which supports crystal frequencies up to 50kHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared, with no timeout. See the device-specific data sheet for appropriate drive settings. NOTE: User must call CS_setExternalClockSourceFrequency to set frequency of external clocks before calling this function.

Parameters

xtDrive	is the target drive strength for the LFXT crystal oscillator. Valid values are: ■ **CS_LFXT_DRIVE0**, ■ **CS_LFXT_DRIVE1**, ■ **CS_LFXT_DRIVE2**, ■ **CS_LFXT_DRIVE3**, [Default Value] ■ **CS_LFXT_BYPASS**

Note

When CS_LFXT_BYPASS is passed as a parameter the oscillator will start in bypass mode and the signal can be generated by a digital square wave.

Returns

true if started correctly, false otherwise

References CS_startLFXTWithTimeout().

6.6.2.30 bool CS_startLFXTWithTimeout (uint32_t *xtDrive,* uint32_t *timeout*)

Initializes the LFXT crystal oscillator, which supports crystal frequencies up to 50kHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared. See the device-specific data sheet for appropriate drive settings. NOTE: User must call CS_setExternalClockSourceFrequency to set frequency of external clocks before calling this function. This function has a timeout associated with stabilizing the oscillator.

Parameters

xtDrive	is the target drive strength for the LFXT crystal oscillator. Valid values are: ■ **CS_LFXT_DRIVE0**, ■ **CS_LFXT_DRIVE1**, ■ **CS_LFXT_DRIVE2**, ■ **CS_LFXT_DRIVE3**, [Default Value] ■ **CS_LFXT_BYPASS**

Note

When CS_LFXT_BYPASS is passed as a parameter the oscillator will start in bypass mode and the signal can be generated by a digital square wave.

Parameters

timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Returns

true if started correctly, false otherwise

Referenced by CS_startLFXT().

6.6.2.31 void CS_tuneDCOFrequency (int16_t *tuneParameter*)

Tunes the DCO to a specific frequency. Tuning of the DCO is based off of the following equation in the user's guide:

See the user's guide for more detailed information about DCO tuning.

Note

This function is not currently available on pre-release MSP432 devices. On early release versions of MSP432, the DCO calibration information has not been populated making the DCO only able to operate at the pre-calibrated centered frequencies accessible by the CS_setDCOCenteredFrequency function. While this function will be added on the final devices being released, for early silicon please default to the pre-calibrated DCO center frequencies.

Parameters

tuneParameter	Tuning parameter in 2's Compliment representation. Can be negative or positive.

Returns

NONE

Referenced by CS_setDCOFrequency().

6.6.2.32 void CS_unregisterInterrupt (void)

Unregisters the interrupt handler for the clock system.

This function unregisters the handler to be called when a clock system interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

7 Direct Memory Access Controller (DMA)

7.1 Module Operation

The Micro Direct Memory Access (DMA) API provides functions to configure the MSP432 uDMA controller. The DMA controller is designed to work with the ARM Cortex-M processor and provides an efficient and low-overhead means of transferring blocks of data in the system.

The DMA controller has the following features:

- dedicated channels for supported peripherals
- one channel each for receive and transmit for devices with receive and transmit paths
- dedicated channel for software initiated data transfers
- channels can be independently configured and operated
- an arbitration scheme that is configurable per channel
- two levels of priority
- subordinate to Cortex-M processor bus usage
- data sizes of 8, 16, or 32 bits
- address increment of byte, half-word, word, or none
- maskable device requests
- optional software initiated transfers on any channel
- interrupt on transfer completion

The uDMA controller supports several different transfer modes, allowing for complex transfer schemes. The following transfer modes are provided:

- **Basic** mode performs a simple transfer when a request is asserted by a device. This mode is appropriate to use with peripherals where the peripheral asserts the request signal whenever data should be transferred. The transfer pauses if the request is de-asserted, even if the transfer is not complete.
- **Auto-request** mode performs a simple transfer that is started by a request, but always completes the entire transfer, even if the request is de-asserted. This mode is appropriate to use with software-initiated transfers.
- **Ping-Pong** mode is used to transfer data to or from two buffers, switching from one buffer to the other as each buffer fills. This mode is appropriate to use with peripherals as a way to ensure a continuous flow of data to or from the peripheral. However, it is more complex to set up and requires code to manage the ping-pong buffers in the interrupt handler.
- **Memory scatter-gather** mode is a complex mode that provides a way to set up a list of transfer “tasks” for the uDMA controller. Blocks of data can be transferred to and from arbitrary locations in memory.
- **Peripheral scatter-gather** mode is similar to memory scatter-gather mode except that it is controlled by a peripheral request.

Detailed explanation of the various transfer modes is beyond the scope of this document. Please refer to the device data sheet for more information on the operation of the uDMA controller.

7.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the DMA module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief example of how to configure the DMA controller to transfer from a data array (data_array) to the EUSCI I2C module to be sent over the I2C line. This is useful in the sense that the EUSCI module does not constantly have to wake up the CPU in order to load the next byte into the buffer.

```
/* Configuring DMA module */
MAP_DMA_enableModule();
MAP_DMA_setControlBase(controlTable);

/* Assigning Channel 2 to EUSCIB1TX0,  and Channel 5 to EUSCIB2RX0  and
 * enabling channels 2 and 5*/
MAP_DMA_assignChannel(DMA_CH2_EUSCIB1TX0);
MAP_DMA_assignChannel(DMA_CH5_EUSCIB2RX0);

 /* Disabling channel attributes */
MAP_DMA_disableChannelAttribute(DMA_CH2_EUSCIB1TX0,
                                UDMA_ATTR_ALTSELECT | UDMA_ATTR_USEBURST |
                                UDMA_ATTR_HIGH_PRIORITY |
                                UDMA_ATTR_REQMASK);
MAP_DMA_disableChannelAttribute(DMA_CH5_EUSCIB2RX0,
                                UDMA_ATTR_ALTSELECT | UDMA_ATTR_USEBURST |
                                UDMA_ATTR_HIGH_PRIORITY |
                                UDMA_ATTR_REQMASK);


/* Setting Control Indexes */
MAP_DMA_setChannelControl(UDMA_PRI_SELECT | DMA_CH2_EUSCIB1TX0,
        UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE | UDMA_ARB_1);
MAP_DMA_setChannelControl(UDMA_PRI_SELECT | DMA_CH5_EUSCIB2RX0,
       UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 | UDMA_ARB_1);
MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT | DMA_CH2_EUSCIB1TX0,
        UDMA_MODE_BASIC,  data_array,
        (void*) MAP_I2C_getTransmitBufferAddressForDMA(EUSCI_B1_BASE),  1024);
MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT | DMA_CH5_EUSCIB2RX0,
        UDMA_MODE_BASIC,
        (void*)MAP_I2C_getReceiveBufferAddressForDMA(EUSCI_B2_BASE),  recBuffer,
        1024);

/* Assigning/Enabling Interrupts */
MAP_DMA_assignInterrupt(DMA_INT1,  2);
MAP_Interrupt_enableInterrupt(INT_DMA_INT1);

/* Now that the DMA is primed and setup,  enabling the channels. The EUSCI
 * hardware should take over and transfer/receive all bytes */
MAP_DMA_enableChannel(2);
MAP_DMA_enableChannel(5);

/* Sending the start condition */
MAP_I2C_masterSendStart(EUSCI_B1_BASE);
while(!MAP_I2C_masterIsStartSent(EUSCI_B1_BASE));
```

7.3 Definitions

Macros

- #define DMA_TaskStructEntry(transferCount, itemSize, srcIncrement, srcAddr, dstIncrement, dstAddr, arbSize, mode)

Functions

- void DMA_assignChannel (uint32_t mapping)
- void DMA_assignInterrupt (uint32_t interruptNumber, uint32_t channel)
- void DMA_clearErrorStatus (void)
- void DMA_clearInterruptFlag (uint32_t channel)
- void DMA_disableChannel (uint32_t channelNum)
- void DMA_disableChannelAttribute (uint32_t channelNum, uint32_t attr)
- void DMA_disableInterrupt (uint32_t interruptNumber)
- void DMA_disableModule (void)
- void DMA_enableChannel (uint32_t channelNum)
- void DMA_enableChannelAttribute (uint32_t channelNum, uint32_t attr)
- void DMA_enableInterrupt (uint32_t interruptNumber)
- void DMA_enableModule (void)
- uint32_t DMA_getChannelAttribute (uint32_t channelNum)
- uint32_t DMA_getChannelMode (uint32_t channelStructIndex)
- uint32_t DMA_getChannelSize (uint32_t channelStructIndex)
- void * DMA_getControlAlternateBase (void)
- void * DMA_getControlBase (void)
- uint32_t DMA_getErrorStatus (void)
- uint32_t DMA_getInterruptStatus (void)
- bool DMA_isChannelEnabled (uint32_t channelNum)
- void DMA_registerInterrupt (uint32_t interruptNumber, void(*intHandler)(void))
- void DMA_requestChannel (uint32_t channelNum)
- void DMA_requestSoftwareTransfer (uint32_t channel)
- void DMA_setChannelControl (uint32_t channelStructIndex, uint32_t control)
- void DMA_setChannelScatterGather (uint32_t channelNum, uint32_t taskCount, void *taskList, uint32_t isPeriphSG)
- void DMA_setChannelTransfer (uint32_t channelStructIndex, uint32_t mode, void *srcAddr, void *dstAddr, uint32_t transferSize)
- void DMA_setControlBase (void *controlTable)
- void DMA_unregisterInterrupt (uint32_t interruptNumber)

7.3.1 Detailed Description

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

7.3.2 Macro Definition Documentation

7.3.2.1 #define DMA_TaskStructEntry(*transferCount, itemSize, srcIncrement, srcAddr, dstIncrement, dstAddr, arbSize, mode*)

A helper macro for building scatter-gather task table entries.

This macro is intended to be used to help populate a table of DMA tasks for a scatter-gather transfer. This macro will calculate the values for the fields of a task structure entry based on the input parameters.

There are specific requirements for the values of each parameter. No checking is done so it is up to the caller to ensure that correct values are used for the parameters.

The **transferCount** parameter is the number of items that will be transferred by this task. It must be in the range 1-1024.

The **itemSize** parameter is the bit size of the transfer data. It must be one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32**.

The *srcIncrement* parameter is the increment size for the source data. It must be one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE**.

The **srcAddr** parameter is a void pointer to the beginning of the source data.

The **dstIncrement** parameter is the increment size for the destination data. It must be one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_DST_INC_NONE**.

The **dstAddr** parameter is a void pointer to the beginning of the location where the data will be transferred.

The **arbSize** parameter is the arbitration size for the transfer, and must be one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, and so on up to **UDMA_ARB_1024**. This is used to select the arbitration size in powers of 2, from 1 to 1024.

The *mode* parameter is the mode to use for this transfer task. It must be one of **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**. Note that normally all tasks will be one of the scatter-gather modes while the last task is a task list will be AUTO or BASIC.

This macro is intended to be used to initialize individual entries of a structure of DMA_ControlTable type, like this:

```
DMA_ControlTable MyTaskList[] =
{
    DMA_TaskStructEntry(Task1Count,  UDMA_SIZE_8,
                        UDMA_SRC_INC_8,  MySourceBuf,
                        UDMA_DST_INC_8,  MyDestBuf,
                        UDMA_ARB_8,  UDMA_MODE_MEM_SCATTER_GATHER),
    DMA_TaskStructEntry(Task2Count,  ... ),
}
```

Parameters

transferCount	is the count of items to transfer for this task.

itemSize	is the bit size of the items to transfer for this task.
srcIncrement	is the bit size increment for source data.
srcAddr	is the starting address of the data to transfer.
dstIncrement	is the bit size increment for destination data.
dstAddr	is the starting address of the destination data.
arbSize	is the arbitration size to use for the transfer task.
mode	is the transfer mode for this task.

Returns
Nothing; this is not a function.

7.3.3 Function Documentation

7.3.3.1 void DMA_assignChannel (uint32_t *mapping*)

Assigns a peripheral mapping for a DMA channel.

Parameters

mapping	is a macro specifying the peripheral assignment for a channel.

This function assigns a peripheral mapping to a DMA channel. It is used to select which peripheral is used for a DMA channel. The parameter *mapping* should be one of the macros named **UDMA_CHn_tttt** from the header file *dma.h*. For example, to assign DMA channel 0 to the eUSCI AO RX channel, the parameter should be the macro **UDMA_CH1_EUSCIA0RX**.

Please consult the data sheet for a table showing all the possible peripheral assignments for the DMA channels for a particular device.

Returns
None.

7.3.3.2 void DMA_assignInterrupt (uint32_t *interruptNumber,* uint32_t *channel*)

Assigns a specific DMA channel to the corresponding interrupt handler. For MSP432 devices, there are three configurable interrupts, and one master interrupt. This function will assign a specific DMA channel to the provided configurable DMA interrupt.

Note that once a channel is assigned to a configurable interrupt, it will be masked in hardware from the master DMA interrupt (interruptNumber zero). This function can also be used in conjunction with the DMAIntTrigger function to provide the feature to software trigger specific channel interrupts.

Parameters

interruptNumber	is the configurable interrupt to assign the given channel. Valid values are: ■ **DMA_INT1** the first configurable DMA interrupt handler ■ **DMA_INT2** the second configurable DMA interrupt handler ■ **DMA_INT3** the third configurable DMA interrupt handler
channel	is the channel to assign the interrupt

Returns
None.

References DMA_enableInterrupt().

7.3.3.3 void DMA_clearErrorStatus (void)

Clears the DMA error interrupt.

This function clears a pending DMA error interrupt. This function should be called from within the DMA error interrupt handler to clear the interrupt.

Returns
None.

7.3.3.4 void DMA_clearInterruptFlag (uint32_t *channel*)

Clears the DMA controller channel interrupt mask for interrupt zero.

Parameters

channel	is the channel interrupt to clear.

This function is used to clear the interrupt status of the DMA controller. Note that only interrupts that weren't assigned to DMA interrupts one through three using the DMA_assignInterrupt function will be affected by this function. For other DMA interrupts, only one channel can be associated and therefore clearing is unnecessary.

Returns
None

7.3.3.5 void DMA_disableChannel (uint32_t *channelNum*)

Disables a DMA channel for operation.

Parameters

channelNum	is the channel number to disable.

This function disables a specific DMA channel. Once disabled, a channel cannot respond to DMA transfer requests until re-enabled via DMA_enableChannel().

Returns
None.

7.3.3.6 void DMA_disableChannelAttribute (uint32_t *channelNum,* uint32_t *attr*)

Disables attributes of a DMA channel.

Parameters

channelNum	is the channel to configure.
attr	is a combination of attributes for the channel.

This function is used to disable attributes of a DMA channel.

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

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns
None.

7.3.3.7 void DMA_disableInterrupt (uint32_t *interruptNumber*)

Disables the specified interrupt for the DMA controller.

Parameters

interruptNumber	identifies which DMA interrupt is to be disabled. This interrupt should be one of the following:

- **DMA_INT1** the first configurable DMA interrupt handler
- **DMA_INT2** the second configurable DMA interrupt handler
- **DMA_INT3** the third configurable DMA interrupt handler
  Note for interrupts that are associated with a specific DMA channel (DMA_INT1 - DMA_INT3), this function will also enable that specific channel for interrupts.

Returns
None.

7.3.3.8 void DMA_disableModule (void)

Disables the DMA controller for use.

This function disables the DMA controller. Once disabled, the DMA controller cannot operate until re-enabled with DMA_enableModule().

Returns
None.

7.3.3.9 void DMA_enableChannel (uint32_t *channelNum*)

Enables a DMA channel for operation.

Parameters

channelNum	is the channel number to enable.

When a DMA transfer is completed, the channel is automatically disabled by the DMA controller. Therefore, this function should be called prior to starting up any new transfer.

Returns
None.

7.3.3.10 void DMA_enableChannelAttribute (uint32_t *channelNum,* uint32_t *attr*)

Enables attributes of a DMA channel.

Parameters

channelNum	is the channel to configure.
attr	is a combination of attributes for the channel.

This function is used to enable attributes of a DMA channel.

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

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel (it is very unlikely that this flag should be used).
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns
None.

7.3.3.11 void DMA_enableInterrupt (uint32_t *interruptNumber*)

Enables the specified interrupt for the DMA controller. Note for interrupts one through three, specific channels have to be mapped to the interrupt using the DMA_assignInterrupt function.

Parameters

interruptNumber	identifies which DMA interrupt is to be enabled. This interrupt should be one of the following:

- **DMA_INT1** the first configurable DMA interrupt handler
- **DMA_INT2** the second configurable DMA interrupt handler
- **DMA_INT3** the third configurable DMA interrupt handler

Returns
None.

Referenced by DMA_assignInterrupt().

7.3.3.12 void DMA_enableModule (void)

Enables the DMA controller for use.

This function enables the DMA controller. The DMA controller must be enabled before it can be configured and used.

Returns
None.

7.3.3.13 uint32_t DMA_getChannelAttribute (uint32_t *channelNum*)

Gets the enabled attributes of a DMA channel.

Parameters

channelNum	is the channel to configure.

This function returns a combination of flags representing the attributes of the DMA channel.

Returns
Returns the logical OR of the attributes of the DMA channel, which can be any of the following:
- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

7.3.3.14 uint32_t DMA_getChannelMode (uint32_t *channelStructIndex*)

Gets the transfer mode for a DMA channel control structure.

Parameters

channelStructIndex	is the logical OR of the DMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

This function is used to get the transfer mode for the DMA channel and to query the status of a transfer on a channel. When the transfer is complete the mode is **UDMA_MODE_STOP**.

Returns
Returns the transfer mode of the specified channel and control structure, which is one of the following values: **UDMA_MODE_STOP**, **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_PINGPONG**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**.

7.3.3.15 uint32_t DMA_getChannelSize (uint32_t *channelStructIndex*)

Gets the current transfer size for a DMA channel control structure.

Parameters

channelStructIndex	is the logical OR of the DMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

This function is used to get the DMA transfer size for a channel. The transfer size is the number of items to transfer, where the size of an item might be 8, 16, or 32 bits. If a partial transfer has already occurred, then the number of remaining items is returned. If the transfer is complete, then 0 is returned.

Returns
Returns the number of items remaining to transfer.

7.3.3.16 void* DMA_getControlAlternateBase (void)

Gets the base address for the channel control table alternate structures.

This function gets the base address of the second half of the channel control table that holds the alternate control structures for each channel.

Returns
Returns a pointer to the base address of the second half of the channel control table.

7.3.3.17 void* DMA_getControlBase (void)

Gets the base address for the channel control table.

This function gets the base address of the channel control table. This table resides in system memory and holds control information for each DMA channel.

Returns
Returns a pointer to the base address of the channel control table.

7.3.3.18 uint32_t DMA_getErrorStatus (void)

Gets the DMA error status.

This function returns the DMA error status. It should be called from within the DMA error interrupt handler to determine if a DMA error occurred.

Returns
Returns non-zero if a DMA error is pending.

7.3.3.19 uint32_t DMA_getInterruptStatus (void)

Gets the DMA controller channel interrupt status for interrupt zero.

This function is used to get the interrupt status of the DMA controller. The returned value is a 32-bit bit mask that indicates which channels are requesting an interrupt. This function can be used from within an interrupt handler to determine or confirm which DMA channel has requested an interrupt.

Note that this will only apply to interrupt zero for the DMA controller as only one interrupt can be associated with interrupts one through three. If an interrupt is assigned to an interrupt other than interrupt zero, it will be masked by this function.

Returns

Returns a 32-bit mask which indicates requesting DMA channels. There is a bit for each channel and a 1 indicates that the channel is requesting an interrupt. Multiple bits can be set.

7.3.3.20 bool DMA_isChannelEnabled (uint32_t *channelNum*)

Checks if a DMA channel is enabled for operation.

Parameters

channelNum	is the channel number to check.

This function checks to see if a specific DMA channel is enabled. This function can be used to check the status of a transfer, as the channel is automatically disabled at the end of a transfer.

Returns

Returns **true** if the channel is enabled, **false** if disabled.

7.3.3.21 void DMA_registerInterrupt (uint32_t *interruptNumber,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the DMA controller.

Parameters

interruptNumber	identifies which DMA interrupt is to be registered.
intHandler	is a pointer to the function to be called when the interrupt is called.

This function registers and enables the handler to be called when the DMA controller generates an interrupt. The *interrupt* parameter should be one of the following:

- **DMA_INT0** the master DMA interrupt handler
- **DMA_INT1** the first configurable DMA interrupt handler
- **DMA_INT2** the second configurable DMA interrupt handler
- **DMA_INT3** the third configurable DMA interrupt handler
- **DMA_INTERR** the DMA error interrupt handler

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

7.3.3.22 void DMA_requestChannel (uint32_t *channelNum*)

Requests a DMA channel to start a transfer.

Parameters

channelNum	is the channel number on which to request a DMA transfer.

This function allows software to request a DMA channel to begin a transfer. This function could be used for performing a memory-to-memory transfer, or if for some reason a transfer needs to be initiated by software instead of the peripheral associated with that channel.

Returns
None.

7.3.3.23 void DMA_requestSoftwareTransfer (uint32_t *channel*)

Initializes a software transfer of the corresponding DMA channel. This is done if the user wants to force a DMA on the specified channel without the hardware precondition. Specific channels can be configured using the DMA_assignChannel function.

Parameters

channel	is the channel to trigger the interrupt

Returns
None

7.3.3.24 void DMA_setChannelControl (uint32_t *channelStructIndex,* uint32_t *control*)

Sets the control parameters for a DMA channel control structure.

Parameters

channelStructIndex	is the logical OR of the DMA channel number with **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.
control	is logical OR of several control values to set the control parameters for the channel.

This function is used to set control parameters for a DMA transfer. These parameters are typically not changed often.

The *channelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *control* parameter is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. The choices available for each of these values is described below.

Choose the data size from one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32** to select a data size of 8, 16, or 32 bits.

Choose the source address increment from one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE** to select an address increment of 8-bit bytes, 16-bit half-words, 32-bit words, or to select non-incrementing.

Choose the destination address increment from one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_SRC_INC_8** to select an address increment of 8-bit bytes, 16-bit half-words, 32-bit words, or to select non-incrementing.

The arbitration size determines how many items are transferred before the DMA controller re-arbitrates for the bus. Choose the arbitration size from one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, **UDMA_ARB_8**, through **UDMA_ARB_1024** to select the arbitration size from 1 to 1024 items, in powers of 2.

The value **UDMA_NEXT_USEBURST** is used to force the channel to only respond to burst requests at the tail end of a scatter-gather transfer.

Note
The address increment cannot be smaller than the data size.

Returns
None.

7.3.3.25 void DMA_setChannelScatterGather (uint32_t *channelNum,* uint32_t *taskCount,* void * *taskList,* uint32_t *isPeriphSG*)

Configures a DMA channel for scatter-gather mode.

Parameters

channelNum	is the DMA channel number.
taskCount	is the number of scatter-gather tasks to execute.
taskList	is a pointer to the beginning of the scatter-gather task list.
isPeriphSG	is a flag to indicate it is a peripheral scatter-gather transfer (else it is memory scatter-gather transfer)

This function is used to configure a channel for scatter-gather mode. The caller must have already set up a task list and must pass a pointer to the start of the task list as the *taskList* parameter. The *taskCount* parameter is the count of tasks in the task list, not the size of the task list. The flag *bIsPeriphSG* should be used to indicate if scatter-gather should be configured for peripheral or memory operation.

See Also
DMA_TaskStructEntry

Returns
None.

7.3.3.26 void DMA_setChannelTransfer (uint32_t *channelStructIndex,* uint32_t *mode,* void * *srcAddr,* void * *dstAddr,* uint32_t *transferSize*)

Sets the transfer parameters for a DMA channel control structure.

Parameters

channelStructIndex	is the logical OR of the DMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

mode	is the type of DMA transfer.
srcAddr	is the source address for the transfer.
dstAddr	is the destination address for the transfer.
transferSize	is the number of data items to transfer.

This function is used to configure the parameters for a DMA transfer. These parameters are typically changed often. The function DMA_setChannelControl() MUST be called at least once for this channel prior to calling this function.

The *channelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *mode* parameter should be one of the following values:

- **UDMA_MODE_STOP** stops the DMA transfer. The controller sets the mode to this value at the end of a transfer.
- **UDMA_MODE_BASIC** to perform a basic transfer based on request.
- **UDMA_MODE_AUTO** to perform a transfer that always completes once started even if the request is removed.
- **UDMA_MODE_PINGPONG** to set up a transfer that switches between the primary and alternate control structures for the channel. This mode allows use of ping-pong buffering for DMA transfers.
- **UDMA_MODE_MEM_SCATTER_GATHER** to set up a memory scatter-gather transfer.
- **UDMA_MODE_PER_SCATTER_GATHER** to set up a peripheral scatter-gather transfer.

The *srcAddr* and *dstAddr* parameters are pointers to the first location of the data to be transferred. These addresses should be aligned according to the item size. The compiler takes care of this alignment if the pointers are pointing to storage of the appropriate data type.

The *transferSize* parameter is the number of data items, not the number of bytes.

The two scatter-gather modes, memory and peripheral, are actually different depending on whether the primary or alternate control structure is selected. This function looks for the **UDMA_PRI_SELECT** and **UDMA_ALT_SELECT** flag along with the channel number and sets the scatter-gather mode as appropriate for the primary or alternate control structure.

The channel must also be enabled using DMA_enableChannel() after calling this function. The transfer does not begin until the channel has been configured and enabled. Note that the channel is automatically disabled after the transfer is completed, meaning that DMA_enableChannel() must be called again after setting up the next transfer.

Note

Great care must be taken to not modify a channel control structure that is in use or else the results are unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the DMA_getChannelMode() returns **UDMA_MODE_STOP**. For PINGPONG or one of the SCATTER_GATHER modes, it is safe to modify the primary or alternate control structure only when the other is being used. The DMA_getChannelMode() function returns **UDMA_MODE_STOP** when a channel control structure is inactive and safe to modify.

Returns

None.

7.3.3.27 void DMA_setControlBase (void * *controlTable*)

Sets the base address for the channel control table.

Parameters

controlTable	is a pointer to the 1024-byte-aligned base address of the DMA channel control table.

This function configures the base address of the channel control table. This table resides in system memory and holds control information for each DMA channel. The table must be aligned on a 1024-byte boundary. The base address must be configured before any of the channel functions can be used.

The size of the channel control table depends on the number of DMA channels and the transfer modes that are used. Refer to the introductory text and the microcontroller datasheet for more information about the channel control table.

Returns

None.

7.3.3.28 void DMA_unregisterInterrupt (uint32_t *interruptNumber*)

Unregisters an interrupt handler for the DMA controller.

Parameters

interruptNumber	identifies which DMA interrupt to unregister.

This function disables and unregisters the handler to be called for the specified DMA interrupt. The *interrupt* parameter should be one of **the** parameters as documented for the function DMA_registerInterrupt().

Note for interrupts that are associated with a specific DMA channel (DMA_INT1 - DMA_INT3), this function will also disable that specific channel for interrupts.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

8 Flash Memory Controller (FlashCtl)

8.1 Module Operation

The MSP432 DriverLib Flash Controller peripheral is designed to simplify the process or writing, erasing, and configuring the flash memory on the MSP432 part. Many of the stringent verification requirements/preconditions are handled entirely inside the FlashCtl APIs.

8.2 Flash Controller Limitations

When utilizing the flash controller for MSP432, the user program has to take special consideration on a few critical limitations. The biggest obstacle that the user has to be mindful of is the stringent verification requirements imposed by the flash controller. Many operations (such as program and verify) will take multiple cycles to complete successfully and the usage is somewhat complicated for a normal user program. For this reason, it is strongly recommended that the user uses the DriverLib APIs for programming and erasing flash. Using the flash controller directly is strongly discouraged as the level of overhead and attention to verification requirements make for a very intricate user experience.

Furthermore, when using the FlashCtl APIs, the user must take special consideration of where the API is being executed. For the critical APIs (such as erase and program), the DriverLib APIs are required to be executed from either SRAM or ROM (using the ROM_ prefix). Due to the verification requirements of the flash controller, running these APIs out of Flash is not currently supported.

8.3 Wait State Considerations

When changing read modes on the MSP432 microcontroller, some read modes (such as erase verify) require an additional number of wait states. The wait states of the flash controller can be configured using the FlashCtl_setWaitState command. When using the DriverLib APIs, the wait states are automatically changed within the API.

8.4 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the FlashCtl module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to unprotect a sector and issue a mass erase with the FlashCtl module:

```
/* Unprotecting User Bank 1,  Sectors 30 and 31  */
MAP_FlashCtl_unprotectSector(FLASH_MAIN_MEMORY_SPACE_BANK1,
        FLASH_SECTOR30 | FLASH_SECTOR31);

/* Trying a mass erase. Since we unprotected User Bank 1,
 * sectors 31 and 32,  this should erase these sectors. If it fails,  we
 * trap inside an infinite loop.
 */
if(!MAP_FlashCtl_performMassErase())
    while(1);
```

8.5 Definitions

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

9 Flash Memory Controller (FlashCtl$_A$)

9.1 Module Operation

Note that this module is for use exclusively on the MSP432P4111. If using the MSP432P401, please refer to the non-a variant.

The MSP432 DriverLib Flash Controller A peripheral is designed to simplify the process or writing, erasing, and configuring the flash memory on the MSP432 part. Many of the stringent verification requirements/preconditions are handled entirely inside the FlashCtl APIs.

9.2 Flash Controller Limitations

When utilizing the flash controller for MSP432, the user program has to take special consideration on a few critical limitations. The biggest obstacle that the user has to be mindful of is the stringent verification requirements imposed by the flash controller. Many operations (such as program and verify) will take multiple cycles to complete successfully and the usage is somewhat complicated for a normal user program. For this reason, it is strongly recommended that the user uses the DriverLib APIs for programming and erasing flash. Using the flash controller directly is strongly discouraged as the level of overhead and attention to verification requirements make for a very intricate user experience.

Furthermore, when using the FlashCtl APIs, the user must take special consideration of where the API is being executed. For the critical APIs (such as erase and program), the DriverLib APIs are required to be executed from either SRAM or ROM (using the ROM_ prefix). Due to the verification requirements of the flash controller, running these APIs out of Flash is not currently supported.

9.3 Wait State Considerations

When changing read modes on the MSP432 microcontroller, some read modes (such as erase verify) require an additional number of wait states. The wait states of the flash controller can be configured using the FlashCtl_setWaitState command. When using the DriverLib APIs, the wait states are automatically changed within the API.

9.4 Programming Example

9.5 Definitions

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

10 Floating Point Unit (FPU)

10.1 Module Operation

The floating-point unit (FPU) driver provides methods for manipulating the behavior of the floating-point unit in the Cortex-M processor. By default, the floating-point is disabled and must be enabled prior to the execution of any floating-point instructions. If a floating-point instruction is executed when the floating-point unit is disabled, a NOCP usage fault is generated. This feature can be used by an RTOS, for example, to keep track of which tasks actually use the floating-point unit, and therefore only perform floating-point context save/restore on task switches that involve those tasks.

There are three methods of handling the floating-point context when the processor executes an interrupt handler: it can do nothing with the floating-point context, it can always save the floating-point context, or it can perform a lazy save/restore of the floating-point context. If nothing is done with the floating-point context, the interrupt stack frame is identical to a Cortex-M processor that does not have a floating-point unit, containing only the volatile registers of the integer unit. This method is useful for applications where the floating-point unit is used by the main thread of execution, but not in any of the interrupt handlers. By not saving the floating-point context, stack usage is reduced and interrupt latency is kept to a minimum.

Alternatively, the floating-point context can always be saved onto the stack. This method allows floating-point operations to be performed inside interrupt handlers without any special precautions, at the expense of increased stack usage (for the floating-point context) and increased interrupt latency (due to the additional writes to the stack). The advantage to this method is that the stack frame always contains the floating-point context when inside an interrupt handler.

The default handling of the floating-point context is to perform a lazy save/restore. When an interrupt is taken, space is reserved on the stack for the floating-point context but the context is not written. This method keeps the interrupt latency to a minimum because only the integer state is written to the stack. Then, if a floating-point instruction is executed from within the interrupt handler, the floating-point context is written to the stack prior to the execution of the floating-point instruction. Finally, upon return from the interrupt, the floating-point context is restored from the stack only if it was written. Using lazy save/restore provides a blend between fast interrupt response and the ability to use floating-point instructions in the interrupt handler.

The floating-point unit can generate an interrupt when one of several exceptions occur. The exceptions are underflow, overflow, divide by zero, invalid operation, input denormal, and inexact exception. The application can optionally choose to enable one or more of these interrupts and use the interrupt handler to decide upon a course of action to be taken in each case.

The behavior of the floating-point unit can also be adjusted, specifying the format of half-precision floating-point values, the handle of NaN values, the flush-to-zero mode (which sacrifices full IEEE compliance for execution speed), and the rounding mode for results.

10.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the FPU module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief example of floating point operation. While the compiler will usually enable the floating point unit by default, when executing floating point operations it is important to make sure that the coprocessor is enabled (otherwise a system fault will occur).

```
/* Enabling FPU for DCO Frequency calculation */
MAP_FPU_enableModule();

/* Setting the DCO Frequency to a non-standard 8.33MHz */
MAP_CS_setDCOFrequency(8330000);
```

10.3 Definitions

Functions

- void FPU_disableModule (void)
- void FPU_disableStacking (void)
- void FPU_enableLazyStacking (void)
- void FPU_enableModule (void)
- void FPU_enableStacking (void)
- void FPU_setFlushToZeroMode (uint32_t mode)
- void FPU_setHalfPrecisionMode (uint32_t mode)
- void FPU_setNaNMode (uint32_t mode)
- void FPU_setRoundingMode (uint32_t mode)

10.3.1 Detailed Description

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

10.3.2 Function Documentation

10.3.2.1 void FPU_disableModule (void)

Disables the floating-point unit.

This function disables the floating-point unit, preventing floating-point instructions from executing (generating a NOCP usage fault instead).

Returns
None.

10.3.2.2 void FPU_disableStacking (void)

Disables the stacking of floating-point registers.

This function disables the stacking of floating-point registers s0-s15 when an interrupt is handled. When floating-point context stacking is disabled, floating-point operations performed in an interrupt handler destroy the floating-point context of the main thread of execution.

Returns
None.

10.3.2.3 void FPU_enableLazyStacking (void)

Enables the lazy stacking of floating-point registers.

This function enables the lazy stacking of floating-point registers s0-s15 when an interrupt is handled. When lazy stacking is enabled, space is reserved on the stack for the floating-point context, but the floating-point state is not saved. If a floating-point instruction is executed from within the interrupt context, the floating-point context is first saved into the space reserved on the stack. On completion of the interrupt handler, the floating-point context is only restored if it was saved (as the result of executing a floating-point instruction).

This method provides a compromise between fast interrupt response (because the floating-point state is not saved on interrupt entry) and the ability to use floating-point in interrupt handlers (because the floating-point state is saved if floating-point instructions are used).

Returns
None.

10.3.2.4 void FPU_enableModule (void)

Enables the floating-point unit.

This function enables the floating-point unit, allowing the floating-point instructions to be executed. This function must be called prior to performing any hardware floating-point operations; failure to do so results in a NOCP usage fault.

Returns
None.

10.3.2.5 void FPU_enableStacking (void)

Enables the stacking of floating-point registers.

This function enables the stacking of floating-point registers s0-s15 when an interrupt is handled. When enabled, space is reserved on the stack for the floating-point context and the floating-point state is saved into this stack space. Upon return from the interrupt, the floating-point context is restored.

If the floating-point registers are not stacked, floating-point instructions cannot be safely executed in an interrupt handler because the values of s0-s15 are not likely to be preserved for the interrupted code. On the other hand, stacking the floating-point registers increases the stacking operation from 8 words to 26 words, also increasing the interrupt response latency.

Returns
None.

10.3.2.6 void FPU_setFlushToZeroMode (uint32_t *mode*)

Selects the flush-to-zero mode.

Parameters

mode	is the flush-to-zero mode; which is either **FPU_FLUSH_TO_ZERO_DIS** or **FPU_FLUSH_TO_ZERO_EN**.

This function enables or disables the flush-to-zero mode of the floating-point unit. When disabled (the default), the floating-point unit is fully IEEE compliant. When enabled, values close to zero are treated as zero, greatly improving the execution speed at the expense of some accuracy (as well as IEEE compliance).

Note
Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns
None.

10.3.2.7 void FPU_setHalfPrecisionMode (uint32_t *mode*)

Selects the format of half-precision floating-point values.

Parameters

mode	is the format for half-precision floating-point value, which is either **FPU_HALF_IEEE** or **FPU_HALF_ALTERNATE**.

This function selects between the IEEE half-precision floating-point representation and the Cortex-M processor alternative representation. The alternative representation has a larger range but does not have a way to encode infinity (positive or negative) or NaN (quiet or signalling). The default setting is the IEEE format.

Note
Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns
None.

10.3.2.8 void FPU_setNaNMode (uint32_t *mode*)

Selects the NaN mode.

Parameters

mode	is the mode for NaN results; which is either **FPU_NAN_PROPAGATE** or **FPU_NAN_DEFAULT**.

This function selects the handling of NaN results during floating-point computations. NaNs can either propagate (the default), or they can return the default NaN.

Note
Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns
None.

10.3.2.9 void FPU_setRoundingMode (uint32_t *mode*)

Selects the rounding mode for floating-point results.

Parameters

mode	is the rounding mode.

This function selects the rounding mode for floating-point results. After a floating-point operation, the result is rounded toward the specified value. The default mode is **FPU_ROUND_NEAREST**.

The following rounding modes are available (as specified by *mode*):

- **FPU_ROUND_NEAREST** - round toward the nearest value
- **FPU_ROUND_POS_INF** - round toward positive infinity
- **FPU_ROUND_NEG_INF** - round toward negative infinity
- **FPU_ROUND_ZERO** - round toward zero

Note

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns

None.

11 General Purpose Input/Output (GPIO)

11.1 Module Operation

The Digital I/O (GPIO) API provides a set of functions for using the SDK L GPIO modules. Functions are provided to setup and enable use of input/output pins, setting them up with or without interrupts and those that access the pin value.

11.2 Key Features

The digital I/O features include:

- Independently programmable individual I/Os
- Any combination of input or output
- Individually configurable P1 and P2 interrupts. Some devices may include additional port interrupts.
- Independent input and output data registers
- Individually configurable pullup or pulldown resistors

Devices within the family may have up to twelve digital I/O ports implemented (P1 to P11 and PJ). Most ports contain eight I/O lines; however, some ports may contain less (see the device-specific data sheet for ports available). Each I/O line is individually configurable for input or output direction, and each can be individually read or written. Each I/O line is individually configurable for pullup or pulldown resistors. PJ contains only four I/O lines.

Individual ports can be accessed as byte-wide ports or can be combined into word-wide ports and accessed via word formats. Port pairs P1/P2, P3/P4, P5/P6, P7/P8, etc., are associated with the names PA, PB, PC, PD, etc., respectively. All port registers are handled in this manner with this naming convention.

When writing to port PA with word operations, all 16 bits are written to the port. When writing to the lower byte of the PA port using byte operations, the upper byte remains unchanged. Similarly, writing to the upper byte of the PA port using byte instructions leaves the lower byte unchanged. When writing to a port that contains less than the maximum number of bits possible, the unused bits are a "don't care". Ports PB, PC, PD, PE, and PF behave similarly.

Reading of the PA port using word operations causes all 16 bits to be transferred to the destination. Reading the lower or upper byte of the PA port (P1 or P2) and storing to memory using byte operations causes only the lower or upper byte to be transferred to the destination, respectively. Reading of the PA port and storing to a general-purpose register using byte operations causes the byte transferred to be written to the least significant byte of the register. The upper significant byte of the destination register is cleared automatically. Ports PB, PC, PD, PE,

and PF behave similarly. When reading from ports that contain less than the maximum bits possible, unused bits are read as zeros (similarly for port PJ).

The GPIO pin may be configured as an I/O pin with GPIO_setAsOutputPin, GPIO_setAsInputPin, GPIO_setAsInputPinWithPullDownResistor or GPIO_setAsInputPinWithPullUpResistor . The GPIO pin may instead be configured to operate in the Peripheral Module assigned function by configuring the GPIO using GPIO_setAsPeripheralModuleFunctionOutputPin or GPIO_setAsPeripheralModuleFunctionInputPin.

11.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the GPIO module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a simple example of how to set up a GPIO in output mode and toggle an LED using a simple delay:

```
int main(void)
{
    volatile uint32_t ii;

    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();

    /* Configuring P1.0 as output */
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P1,  GPIO_PIN0);

    while (1)
    {
        /* Delay Loop */
        for(ii=0;ii<5000;ii++)
        {
        }

        MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1,  GPIO_PIN0);
    }
}
```

11.4 Definitions

Functions

- void GPIO_clearInterruptFlag (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_disableInterrupt (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_enableInterrupt (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- uint_fast16_t GPIO_getEnabledInterruptStatus (uint_fast8_t selectedPort)
- uint8_t GPIO_getInputPinValue (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- uint_fast16_t GPIO_getInterruptStatus (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_interruptEdgeSelect (uint_fast8_t selectedPort, uint_fast16_t selectedPins, uint_fast8_t edgeSelect)
- void GPIO_registerInterrupt (uint_fast8_t selectedPort, void(*intHandler)(void))
- void GPIO_setAsInputPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_setAsInputPinWithPullDownResistor (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_setAsInputPinWithPullUpResistor (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_setAsOutputPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_setAsPeripheralModuleFunctionInputPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins, uint_fast8_t mode)
- void GPIO_setAsPeripheralModuleFunctionOutputPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins, uint_fast8_t mode)
- void GPIO_setDriveStrengthHigh (uint_fast8_t selectedPort, uint_fast8_t selectedPins)
- void GPIO_setDriveStrengthLow (uint_fast8_t selectedPort, uint_fast8_t selectedPins)
- void GPIO_setOutputHighOnPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_setOutputLowOnPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_toggleOutputOnPin (uint_fast8_t selectedPort, uint_fast16_t selectedPins)
- void GPIO_unregisterInterrupt (uint_fast8_t selectedPort)

11.4.1 Detailed Description

The code for this module is contained in `driverlib/gpio.c` and `driverlib/legacy/MSP432xx/legacy_gpio.c`, with `driverlib/gpio.h` and `driverlib/legacy/MSP432xx/legacy_gpio.h` containing the API declarations for use by applications.

11.4.2 Function Documentation

11.4.2.1 void GPIO_clearInterruptFlag (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function clears the interrupt flag on the selected pin.

This function clears the interrupt flag on the selected pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_PA**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxIFG** register.

Returns
None

11.4.2.2 void GPIO_disableInterrupt (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function disables the port interrupt on the selected pin.

This function disables the port interrupt on the selected pin. Note that only Port 1, 2, A have this

capability.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_PA**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxIE** register.

Returns
None

11.4.2.3 void GPIO_enableInterrupt (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function enables the port interrupt on the selected pin.

This function enables the port interrupt on the selected pin. Note that only Port 1, 2, A have this capability.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_PA**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxIE** register.

Returns
None

11.4.2.4 uint_fast16_t GPIO_getEnabledInterruptStatus (uint_fast8_t *selectedPort*)

This function gets the interrupt status of the provided PIN and masks it with the interrupts that are actually enabled. This is useful for inside ISRs where the status of only the enabled interrupts needs to be checked.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

Returns

Logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**
- **GPIO_PIN6**
- **GPIO_PIN7**
- **GPIO_PIN8**
- **GPIO_PIN9**
- **GPIO_PIN10**
- **GPIO_PIN11**
- **GPIO_PIN12**
- **GPIO_PIN13**
- **GPIO_PIN14**
- **GPIO_PIN15**,
- **PIN_ALL8**,
- **PIN_ALL16**

indicating the interrupt status of the selected pins [Default: 0]

References GPIO_getInterruptStatus().

11.4.2.5 uint8_t GPIO_getInputPinValue (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function gets the input value on the selected pin.

This function gets the input value on the selected pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**
selectedPins	is the specified pin in the selected port. Valid values are: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Returns

One of the following:

- **GPIO_INPUT_PIN_HIGH**
- **GPIO_INPUT_PIN_LOW**

indicating the status of the pin

11.4.2.6 uint_fast16_t GPIO_getInterruptStatus (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function gets the interrupt status of the selected pin.

This function gets the interrupt status of the selected pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_PA**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Returns

Logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**

- **GPIO_PIN4**
- **GPIO_PIN5**
- **GPIO_PIN6**
- **GPIO_PIN7**
- **GPIO_PIN8**
- **GPIO_PIN9**
- **GPIO_PIN10**
- **GPIO_PIN11**
- **GPIO_PIN12**
- **GPIO_PIN13**
- **GPIO_PIN14**
- **GPIO_PIN15**

  indicating the interrupt status of the selected pins [Default: 0]

Referenced by GPIO_getEnabledInterruptStatus().

11.4.2.7 void GPIO_interruptEdgeSelect (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins,* uint_fast8_t *edgeSelect*)

This function selects on what edge the port interrupt flag should be set for a transition.

This function selects on what edge the port interrupt flag should be set for a transition. Values for edgeSelect should be GPIO_LOW_TO_HIGH_TRANSITION or GPIO_HIGH_TO_LOW_TRANSITION.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**
edgeSelect	specifies what transition sets the interrupt flag Valid values are: ■ **GPIO_HIGH_TO_LOW_TRANSITION** ■ **GPIO_LOW_TO_HIGH_TRANSITION**

Modified bits of **PxIES** register.

Returns
None

11.4.2.8 void GPIO_registerInterrupt (uint_fast8_t *selectedPort,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the port interrupt.

Parameters

selectedPort	is the port to register the interrupt handler
intHandler	is a pointer to the function to be called when the port interrupt occurs.

This function registers the handler to be called when a port interrupt occurs. This function enables the global interrupt in the interrupt controller; specific GPIO interrupts must be enabled via GPIO_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via GPIO_clearInterruptFlag().

Clock System can generate interrupts when

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

11.4.2.9 void GPIO_setAsInputPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function configures the selected Pin as input pin.

This function selected pins on a selected port as input pins.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxDIR** register, bits of **PxREN** register and bits of **PxSEL** register.

Returns
None

11.4.2.10 void GPIO_setAsInputPinWithPullDownResistor (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function sets the selected Pin in input Mode with Pull Down resistor.

This function sets the selected Pin in input Mode with Pull Down resistor.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxDIR** register, bits of **PxOUT** register and bits of **PxREN** register.

Returns
None

11.4.2.11 void GPIO_setAsInputPinWithPullUpResistor (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function sets the selected Pin in input Mode with Pull Up resistor.

This function sets the selected Pin in input Mode with Pull Up resistor.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxDIR** register, bits of **PxOUT** register and bits of **PxREN** register.

Returns
None

11.4.2.12 void GPIO_setAsOutputPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function configures the selected Pin as output pin.

This function selected pins on a selected port as output pins.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxDIR** register and bits of **PxSEL** register.

Returns
None

11.4.2.13 void GPIO_setAsPeripheralModuleFunctionInputPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins,* uint_fast8_t *mode*)

This function configures the peripheral module function in the input direction for the selected pin for either primary, secondary or ternary module function modes.

This function configures the peripheral module function in the input direction for the selected pin for either primary, secondary or ternary module function modes. Accepted values for mode are GPIO_PRIMARY_MODULE_FUNCTION, GPIO_SECONDARY_MODULE_FUNCTION, and GPIO_TERTIARY_MODULE_FUNCTION

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

mode	is the specified mode that the pin should be configured for the module function. Valid values are: ■ **GPIO_PRIMARY_MODULE_FUNCTION** ■ **GPIO_SECONDARY_MODULE_FUNCTION** ■ **GPIO_TERTIARY_MODULE_FUNCTION**

Modified bits of **PxDIR** register and bits of **PxSEL** register.

Returns
None

11.4.2.14 void GPIO_setAsPeripheralModuleFunctionOutputPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins,* uint_fast8_t *mode*)

This function configures the peripheral module function in the output direction for the selected pin for either primary, secondary or ternary module function modes.

This function configures the peripheral module function in the output direction for the selected pin for either primary, secondary or ternary module function modes. Accepted values for mode are GPIO_PRIMARY_MODULE_FUNCTION, GPIO_SECONDARY_MODULE_FUNCTION, and GPIO_TERTIARY_MODULE_FUNCTION

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

mode	is the specified mode that the pin should be configured for the module function. Valid values are: ■ **GPIO_PRIMARY_MODULE_FUNCTION** ■ **GPIO_SECONDARY_MODULE_FUNCTION** ■ **GPIO_TERTIARY_MODULE_FUNCTION**

Modified bits of **PxDIR** register and bits of **PxSEL** register.

Returns
None

11.4.2.15 void GPIO_setDriveStrengthHigh (uint_fast8_t *selectedPort,* uint_fast8_t *selectedPins*)

This function sets the drive strength to high for the selected port

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1**, ■ **GPIO_PORT_P2**, ■ **GPIO_PORT_P3**, ■ **GPIO_PORT_P4**, ■ **GPIO_PORT_P5**, ■ **GPIO_PORT_P6**, ■ **GPIO_PORT_P7**, ■ **GPIO_PORT_P8**, ■ **GPIO_PORT_P9**, ■ **GPIO_PORT_P10**, ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Valid values are: ■ **GPIO_PIN0**, ■ **GPIO_PIN1**, ■ **GPIO_PIN2**, ■ **GPIO_PIN3**, ■ **GPIO_PIN4**, ■ **GPIO_PIN5**, ■ **GPIO_PIN6**, ■ **GPIO_PIN7**, ■ **GPIO_PIN8**, ■ **PIN_ALL8**,

Returns
None

11.4.2.16 void GPIO_setDriveStrengthLow (uint_fast8_t *selectedPort,* uint_fast8_t *selectedPins*)

This function sets the drive strength to low for the selected port

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1**, ■ **GPIO_PORT_P2**, ■ **GPIO_PORT_P3**, ■ **GPIO_PORT_P4**, ■ **GPIO_PORT_P5**, ■ **GPIO_PORT_P6**, ■ **GPIO_PORT_P7**, ■ **GPIO_PORT_P8**, ■ **GPIO_PORT_P9**, ■ **GPIO_PORT_P10**, ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Valid values are: ■ **GPIO_PIN0**, ■ **GPIO_PIN1**, ■ **GPIO_PIN2**, ■ **GPIO_PIN3**, ■ **GPIO_PIN4**, ■ **GPIO_PIN5**, ■ **GPIO_PIN6**, ■ **GPIO_PIN7**, ■ **GPIO_PIN8**, ■ **PIN_ALL8**,

Returns
None

11.4.2.17 void GPIO_setOutputHighOnPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function sets output HIGH on the selected Pin.

This function sets output HIGH on the selected port's pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxOUT** register.

Returns
None

11.4.2.18 void GPIO_setOutputLowOnPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function sets output LOW on the selected Pin.

This function sets output LOW on the selected port's pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**
selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Returns
None

11.4.2.19 void GPIO_toggleOutputOnPin (uint_fast8_t *selectedPort,* uint_fast16_t *selectedPins*)

This function toggles the output on the selected Pin.

This function toggles the output on the selected port's pin.

Parameters

selectedPort	is the selected port. Valid values are: ■ **GPIO_PORT_P1** ■ **GPIO_PORT_P2** ■ **GPIO_PORT_P3** ■ **GPIO_PORT_P4** ■ **GPIO_PORT_P5** ■ **GPIO_PORT_P6** ■ **GPIO_PORT_P7** ■ **GPIO_PORT_P8** ■ **GPIO_PORT_P9** ■ **GPIO_PORT_P10** ■ **GPIO_PORT_P11** ■ **GPIO_PORT_PJ**

selectedPins	is the specified pin in the selected port. Mask value is the logical OR of any of the following: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Modified bits of **PxOUT** register.

Returns
None

11.4.2.20 void GPIO_unregisterInterrupt (uint_fast8_t *selectedPort*)

Unregisters the interrupt handler for the port.

Parameters

selectedPort	is the port to unregister the interrupt handler

This function unregisters the handler to be called when a port interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

12 Inter-Integrated Circuit (I2C)

12.1 I2C Module Operation

In I2C mode, the eUSCI_B module provides an interface between the device and I2C-compatible devices connected by the two-wire I2C serial bus. External components attached to the I2C bus serially transmit and/or receive serial data to/from the eUSCI_B module through the 2-wire I2C interface. The Inter-Integrated Circuit (I2C) API provides a set of functions for using the SDK I2C modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules. For the sake of simplicity and code readability, the EUSCI_B module name has been omitted from the API name space.

The I2C module provide the ability to communicate to other IC devices over an I2C bus. The I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the I2C bus can be designated as either a master or a slave. The SDK L I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave.

I2C module can generate interrupts. The I2C module configured as a master will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The I2C module configured as a slave will generate interrupts when data has been sent or requested by a master.

12.2 Master Operation

To drive the master module, the APIs need to be invoked in the following order

- **I2C_initMaster**
- **I2C_setSlaveAddress**
- **I2C_setMode**
- **I2C_enableModule**
- **I2C_enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first initialize the I2C module and configure it as a master with a call to I2C_initMaster . That function will set the clock and data rates. This is followed by a call to set the slave address with which the master intends to communicate with using I2C_setSlaveAddress . Then the mode of operation (transmit or receive) is chosen using I2C_setMode . The I2C module may now be enabled using I2C_enableModule . It is recommended to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Master Single Byte Transmission

- I2C_masterSendSingleByte

Master Multiple Byte Transmission

- I2C_masterSendMultiByteStart
- I2C_masterSendMultiByteNext
- I2C_masterSendMultiByteStop

Master Single Byte Reception

- I2C_masterReceiveSingleByte

Master Multiple Byte Reception

- I2C_masterReceiveStart
- I2C_masterReceiveMultiByteNext
- I2C_masterReceiveMultiByteFinish
- I2C_masterReceiveMultiByteStop

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

12.3 Slave Operation

To drive the slave module, the APIs need to be invoked in the following order

- **I2C_initSlave**
- **I2C_setMode**
- **I2C_enableModule**
- **I2C_enableInterrupt** (if interrupts are being used)

The user must first call the I2C_initSlave to initialize the slave module in I2C mode and set the slave address. This is followed by a call to set the mode of operation (transmit or receive). The I2C module may now be enabled using I2C_enableModule . It is recommended to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

The transaction can then be initiated on the bus by calling the transmit or receive related APIs as listed below.

Slave Transmission API

- I2C_slavePutData

Slave Reception API

- I2C_slaveGetData

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C interrupt.

12.4 Timeout Parameters

For serial transmission APIs (sending/receiving), a "timeout" API exists that will return control of execution back to the user application if a specified duration passes. The variable that is passed into these functions is a unit of time specified by how many "loop iterations" elapse before unsuccessful transmission of data.

12.5 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the I2C module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a simple example of how to setup the I2C module for master operation with a 400KHz clock.

First, below is an example of setting up the I2C module configuration structure:

```
/* I2C Master Configuration Parameter */
const eUSCI_I2C_MasterConfig i2cConfig =
{
        EUSCI_B_I2C_CLOCKSOURCE_SMCLK,           // SMCLK Clock Source
        3000000,                                 // SMCLK = 3MHz
        EUSCI_B_I2C_SET_DATA_RATE_400KBPS,       // Desired I2C Clock of 400khz
        0,                                       // No byte counter threshold
        EUSCI_B_I2C_NO_AUTO_STOP                // No Autostop
};
```

Below are the actual DriverLib calls to configure/setup the I2C module:

```
    /* Initializing I2C Master to SMCLK at 400kbs with no autostop */
    MAP_I2C_initMaster(EUSCI_B0_BASE,  &i2cConfig);

    /* Specify slave address */
    MAP_I2C_setSlaveAddress(EUSCI_B0_BASE,  SLAVE_ADDRESS);

    /* Set Master in transmit mode */
    MAP_I2C_setMode(EUSCI_B0_BASE,  EUSCI_B_I2C_TRANSMIT_MODE);

    /* Enable I2C Module to start operations */
    MAP_I2C_enableModule(EUSCI_B0_BASE);

    /* Enable and clear the interrupt flag */
    MAP_I2C_clearInterruptFlag(EUSCI_B0_BASE,
            EUSCI_B_I2C_TRANSMIT_INTERRUPT0 + EUSCI_B_I2C_NAK_INTERRUPT);

    /* Enable master transmit interrupt */
    MAP_I2C_enableInterrupt(EUSCI_B0_BASE,
            EUSCI_B_I2C_TRANSMIT_INTERRUPT0 + EUSCI_B_I2C_NAK_INTERRUPT);
    MAP_Interrupt_enableInterrupt(INT_EUSCIB0);
```

12.6 Definitions

Data Structures

- struct _eUSCI_I2C_MasterConfig

Functions

- void I2C_clearInterruptFlag (uint32_t moduleInstance, uint_fast16_t mask)
- void I2C_disableInterrupt (uint32_t moduleInstance, uint_fast16_t mask)
- void I2C_disableModule (uint32_t moduleInstance)
- void I2C_disableMultiMasterMode (uint32_t moduleInstance)
- void I2C_enableInterrupt (uint32_t moduleInstance, uint_fast16_t mask)
- void I2C_enableModule (uint32_t moduleInstance)
- void I2C_enableMultiMasterMode (uint32_t moduleInstance)
- uint_fast16_t I2C_getEnabledInterruptStatus (uint32_t moduleInstance)
- uint_fast16_t I2C_getInterruptStatus (uint32_t moduleInstance, uint16_t mask)
- uint_fast8_t I2C_getMode (uint32_t moduleInstance)
- uint32_t I2C_getReceiveBufferAddressForDMA (uint32_t moduleInstance)
- uint32_t I2C_getTransmitBufferAddressForDMA (uint32_t moduleInstance)
- void I2C_initMaster (uint32_t moduleInstance, const eUSCI_I2C_MasterConfig *config)
- void I2C_initSlave (uint32_t moduleInstance, uint_fast16_t slaveAddress, uint_fast8_t slaveAddressOffset, uint32_t slaveOwnAddressEnable)
- uint8_t I2C_isBusBusy (uint32_t moduleInstance)
- bool I2C_masterIsStartSent (uint32_t moduleInstance)
- uint8_t I2C_masterIsStopSent (uint32_t moduleInstance)
- uint8_t I2C_masterReceiveMultiByteFinish (uint32_t moduleInstance)
- bool I2C_masterReceiveMultiByteFinishWithTimeout (uint32_t moduleInstance, uint8_t *txData, uint32_t timeout)
- uint8_t I2C_masterReceiveMultiByteNext (uint32_t moduleInstance)
- void I2C_masterReceiveMultiByteStop (uint32_t moduleInstance)
- uint8_t I2C_masterReceiveSingle (uint32_t moduleInstance)
- uint8_t I2C_masterReceiveSingleByte (uint32_t moduleInstance)
- void I2C_masterReceiveStart (uint32_t moduleInstance)
- bool I2C_masterSendMultiByteFinish (uint32_t moduleInstance, uint8_t txData)
- bool I2C_masterSendMultiByteFinishWithTimeout (uint32_t moduleInstance, uint8_t txData, uint32_t timeout)
- void I2C_masterSendMultiByteNext (uint32_t moduleInstance, uint8_t txData)
- bool I2C_masterSendMultiByteNextWithTimeout (uint32_t moduleInstance, uint8_t txData, uint32_t timeout)
- void I2C_masterSendMultiByteStart (uint32_t moduleInstance, uint8_t txData)
- bool I2C_masterSendMultiByteStartWithTimeout (uint32_t moduleInstance, uint8_t txData, uint32_t timeout)
- void I2C_masterSendMultiByteStop (uint32_t moduleInstance)
- bool I2C_masterSendMultiByteStopWithTimeout (uint32_t moduleInstance, uint32_t timeout)
- void I2C_masterSendSingleByte (uint32_t moduleInstance, uint8_t txData)
- bool I2C_masterSendSingleByteWithTimeout (uint32_t moduleInstance, uint8_t txData, uint32_t timeout)
- void I2C_masterSendStart (uint32_t moduleInstance)
- void I2C_registerInterrupt (uint32_t moduleInstance, void(*intHandler)(void))
- void I2C_setMode (uint32_t moduleInstance, uint_fast8_t mode)
- void I2C_setSlaveAddress (uint32_t moduleInstance, uint_fast16_t slaveAddress)
- uint8_t I2C_slaveGetData (uint32_t moduleInstance)
- void I2C_slavePutData (uint32_t moduleInstance, uint8_t transmitData)
- void I2C_slaveSendNAK (uint32_t moduleInstance)
- void I2C_unregisterInterrupt (uint32_t moduleInstance)

12.6.1 Detailed Description

The code for this module is contained in `driverlib/i2c.c`, with `driverlib/i2c.h` containing the API declarations for use by applications.

12.6.2 Data Structure Documentation

12.6.2.1 struct _eUSCI_I2C_MasterConfig

Type definition for _eUSCI_I2C_MasterConfig structure.

ypedef eUSCI_I2C_MasterConfig

Configuration structure for master mode in the **I2C** module. See I2C_initMaster for parameter documentation.

12.6.3 Function Documentation

12.6.3.1 void I2C_clearInterruptFlag (uint32_t *moduleInstance,* uint_fast16_t *mask*)

Clears I2C interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
mask	is a bit mask of the interrupt sources to be cleared.

The I2C interrupt source is cleared, so that it no longer asserts. The highest interrupt flag is automatically cleared when an interrupt vector generator is used.

The mask parameter has the same definition as the mask parameter to I2C_enableInterrupt().

Modified register is **UCBxIFG**.

Returns
None.

12.6.3.2 void I2C_disableInterrupt (uint32_t *moduleInstance,* uint_fast16_t *mask*)

Disables individual I2C interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
mask	is the bit mask of the interrupt sources to be disabled.

Disables the indicated I2C interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The mask parameter is the logical OR of any of the following:

- **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3
- **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0
- **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1
- **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2
- **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3
- **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt
- **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt enable
- **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable
- **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Modified register is **UCBxIE**.

Returns
None.

12.6.3.3 void I2C_disableModule (uint32_t *moduleInstance*)

Disables the I2C block.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This will disable operation of the I2C block. Modified bits are **UCSWRST** of **UCBxCTL1** register.

Returns
None.

12.6.3.4 void I2C_disableMultiMasterMode (uint32_t *moduleInstance*)

Disables Multi Master Mode

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

At the end of this function, the I2C module is still disabled till I2C_enableModule is invoked

Modified bits are **UCSWRST** of **OFS_UCBxCTLW0**, **UCMM** bit of **UCBxCTLW0**

Returns
None.

12.6.3.5 void I2C_enableInterrupt (uint32_t *moduleInstance,* uint_fast16_t *mask*)

Enables individual I2C interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
mask	is the bit mask of the interrupt sources to be enabled.

Enables the indicated I2C interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The mask parameter is the logical OR of any of the following:

- **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3
- **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0
- **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1
- **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2
- **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3
- **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt
- **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt enable
- **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable
- **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Modified registers are UCBxIFG and OFS_UCBxIE.

Returns
None.

12.6.3.6 void I2C_enableModule (uint32_t *moduleInstance*)

Enables the I2C block.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This will enable operation of the I2C block. Modified bits are **UCSWRST** of **UCBxCTL1** register.

Returns
None.

12.6.3.7 void I2C_enableMultiMasterMode (uint32_t *moduleInstance*)

Enables Multi Master Mode

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

At the end of this function, the I2C module is still disabled till I2C_enableModule is invoked

Modified bits are **UCSWRST** of **OFS_UCBxCTLW0**, **UCMM** bit of **UCBxCTLW0**

Returns
None.

12.6.3.8 uint_fast16_t I2C_getEnabledInterruptStatus (uint32_t *moduleInstance*)

Gets the current I2C interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

Returns

the masked status of the interrupt flag

- **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3
- **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0
- **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1
- **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2
- **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3
- **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt
- **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt enable
- **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable
- **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

References I2C_getInterruptStatus().

12.6.3.9 uint_fast16_t I2C_getInterruptStatus (uint32_t *moduleInstance,* uint16_t *mask*)

Gets the current I2C interrupt status.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt ■ **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt ■ **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt ■ **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3 ■ **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt ■ **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable ■ **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Returns

the masked status of the interrupt flag

- **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3
- **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0
- **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1
- **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2
- **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3
- **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt

- **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt enable
- **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable
- **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Referenced by I2C_getEnabledInterruptStatus().

12.6.3.10 uint_fast8_t I2C_getMode (uint32_t *moduleInstance*)

Gets the mode of the I2C device.

Current I2C transmit/receive mode.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: - **EUSCI_B0_BASE** - **EUSCI_B1_BASE** - **EUSCI_B2_BASE** - **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

Modified bits are **UCTR** of **UCBxCTL1** register.

Returns

None Return one of the following:
- **EUSCI_B_I2C_TRANSMIT_MODE**
- **EUSCI_B_I2C_RECEIVE_MODE**

indicating the current mode

12.6.3.11 uint32_t I2C_getReceiveBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the RX Buffer of the I2C for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: - **EUSCI_B0_BASE** - **EUSCI_B1_BASE** - **EUSCI_B2_BASE** - **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

Returns the address of the I2C RX Buffer. This can be used in conjunction with the DMA to store the received data directly to memory.

Returns
NONE

12.6.3.12 uint32_t I2C_getTransmitBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the TX Buffer of the I2C for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

Returns the address of the I2C TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Returns
NONE

12.6.3.13 void I2C_initMaster (uint32_t *moduleInstance,* const eUSCI_I2C_MasterConfig * *config*)

Initializes the I2C Master block.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
config	Configuration structure for I2C master mode

Configuration options for eUSCI_I2C_MasterConfig structure.

Parameters

selectClock-Source	is the clock source. Valid values are ■ **EUSCI_B_I2C_CLOCKSOURCE_ACLK** ■ **EUSCI_B_I2C_CLOCKSOURCE_SMCLK**
i2cClk	is the rate of the clock supplied to the I2C module (the frequency in Hz of the clock source specified in selectClockSource).
dataRate	set up for selecting data transfer rate. Valid values are ■ **EUSCI_B_I2C_SET_DATA_RATE_1MBPS** ■ **EUSCI_B_I2C_SET_DATA_RATE_400KBPS** ■ **EUSCI_B_I2C_SET_DATA_RATE_100KBPS**
byteCoun-terThreshold	sets threshold for automatic STOP or UCSTPIFG
autoSTOPGen-eration	sets up the STOP condition generation. Valid values are ■ **EUSCI_B_I2C_NO_AUTO_STOP** ■ **EUSCI_B_I2C_SET_BYTECOUNT_THRESHOLD_FLAG** ■ **EUSCI_B_I2C_SEND_STOP_AUTOMATICALLY_ON_BYTECOUNT_THRESHOLD**

This function initializes operation of the I2C Master block. Upon successful initialization of the I2C block, this function will have set the bus speed for the master; however I2C module is still disabled till I2C_enableModule is invoked

Modified bits are **UCMST**, UCMODE_3, **UCSYNC** of **UCBxCTL0** register **UCSSELx**, **UCSWRST**, of **UCBxCTL1** register **UCBxBR0** and **UCBxBR1** registers

Returns
None.

12.6.3.14 void I2C_initSlave (uint32_t *moduleInstance,* uint_fast16_t *slaveAddress,* uint_fast8_t *slaveAddressOffset,* uint32_t *slaveOwnAddressEnable*)

Initializes the I2C Slave block.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
slaveAddress	7-bit or 10-bit slave address
slaveAddressOffset	Own address Offset referred to- 'x' value of UCBxI2COAx. Valid values are: ■ **EUSCI_B_I2C_OWN_ADDRESS_OFFSET0**, ■ **EUSCI_B_I2C_OWN_ADDRESS_OFFSET1**, ■ **EUSCI_B_I2C_OWN_ADDRESS_OFFSET2**, ■ **EUSCI_B_I2C_OWN_ADDRESS_OFFSET3**
slaveOwnAddressEnable	selects if the specified address is enabled or disabled. Valid values are: ■ **EUSCI_B_I2C_OWN_ADDRESS_DISABLE**, ■ **EUSCI_B_I2C_OWN_ADDRESS_ENABLE**

This function initializes operation of the I2C as a Slave mode. Upon successful initialization of the I2C blocks, this function will have set the slave address but the I2C module is still disabled till I2C_enableModule is invoked.

The parameter slaveAddress is the value that will be compared against the slave address sent by an I2C master.

Modified bits are **UCMODE_3**, **UCSYNC** of **UCBxCTL0** register **UCSWRST** of **UCBxCTL1** register **UCBxI2COA** register

Returns
None.

12.6.3.15 uint8_t I2C_isBusBusy (uint32_t *moduleInstance*)

Indicates whether or not the I2C bus is busy.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function returns an indication of whether or not the I2C bus is busy.This function checks the status of the bus via UCBBUSY bit in UCBxSTAT register.

Returns

Returns EUSCI_B_I2C_BUS_BUSY if the I2C Master is busy; otherwise, returns EUSCI_B_I2C_BUS_NOT_BUSY.

12.6.3.16 bool I2C_masterIsStartSent (uint32_t *moduleInstance*)

Indicates whether Start got sent.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function returns an indication of whether or not Start got sent This function checks the status of the bus via UCTXSTT bit in UCBxCTL1 register.

Returns

Returns true if the START has been sent, false if it is sending

12.6.3.17 uint8_t I2C_masterIsStopSent (uint32_t *moduleInstance*)

Indicates whether STOP got sent.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function returns an indication of whether or not STOP got sent This function checks the status of the bus via UCTXSTP bit in UCBxCTL1 register.

Returns
Returns EUSCI_B_I2C_STOP_SEND_COMPLETE if the I2C Master finished sending STOP; otherwise, returns EUSCI_B_I2C_SENDING_STOP.

12.6.3.18 uint8_t I2C_masterReceiveMultiByteFinish (uint32_t *moduleInstance*)

Finishes multi-byte reception at the Master end

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module to initiate completion of a multi-byte reception This function

- Receives the current byte and initiates the STOP from Master to Slave

Modified bits are **UCTXSTP** bit of **UCBxCTL1**.

Returns
Received byte at Master end.

12.6.3.19 bool I2C_masterReceiveMultiByteFinishWithTimeout (uint32_t *moduleInstance,* uint8_t * *txData,* uint32_t *timeout*)

Finishes multi-byte reception at the Master end with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is a pointer to the location to store the received byte at master end
timeout	is the amount of time to wait until giving up

This function is used by the Master module to initiate completion of a multi-byte reception This function

- Receives the current byte and initiates the STOP from Master to Slave

Modified bits are **UCTXSTP** bit of **UCBxCTL1**.

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.20 uint8_t I2C_masterReceiveMultiByteNext (uint32_t *moduleInstance*)

Starts multi-byte reception at the Master end one byte at a time

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module to receive each byte of a multi-byte reception This function reads currently received byte

Modified register is **UCBxRXBUF**.

Returns
Received byte at Master end.

12.6.3.21 void I2C_masterReceiveMultiByteStop (uint32_t *moduleInstance*)

Sends the STOP at the end of a multi-byte reception at the Master end

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module to initiate STOP

Modified bits are UCTXSTP bit of UCBxCTL1.

Returns
None.

12.6.3.22 uint8_t I2C_masterReceiveSingle (uint32_t *moduleInstance*)

Receives a byte that has been sent to the I2C Master Module.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function reads a byte of data from the I2C receive data Register.

Returns
Returns the byte received from by the I2C module, cast as an uint8_t.

12.6.3.23 uint8_t I2C_masterReceiveSingleByte (uint32_t *moduleInstance*)

Does single byte reception from the slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module to receive a single byte. This function:

- Sends START and STOP
- Waits for data reception
- Receives one byte from the Slave

Modified registers are **UCBxIE**, **UCBxCTL1**, **UCBxIFG**, **UCBxTXBUF**, **UCBxIE**

Returns
The byte that has been received from the slave

12.6.3.24 void I2C_masterReceiveStart (uint32_t *moduleInstance*)

Starts reception at the Master end

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module initiate reception of a single byte. This function

- Sends START

Modified bits are **UCTXSTT** bit of **UCBxCTL1**.

Returns
None.

12.6.3.25 bool I2C_masterSendMultiByteFinish (uint32_t *moduleInstance,* uint8_t *txData*)

Finishes multi-byte transmission from Master to Slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the last data byte to be transmitted in a multi-byte transmsission

This function is used by the Master module to send the last byte and STOP. This function

- Transmits the last data byte of a multi-byte transmission to the Slave
- Sends STOP

Modified registers are **UCBxTXBUF** and **UCBxCTL1**.

Returns
false if NAK occurred, false otherwise

12.6.3.26 bool I2C_masterSendMultiByteFinishWithTimeout (uint32_t *moduleInstance,* uint8_t *txData,* uint32_t *timeout*)

Finishes multi-byte transmission from Master to Slave with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the last data byte to be transmitted in a multi-byte transmission
timeout	is the amount of time to wait until giving up

This function is used by the Master module to send the last byte and STOP. This function

- Transmits the last data byte of a multi-byte transmission to the Slave
- Sends STOP

Modified registers are **UCBxTXBUF** and **UCBxCTL1**.

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.27 void I2C_masterSendMultiByteNext (uint32_t *moduleInstance,* uint8_t *txData*)

Continues multi-byte transmission from Master to Slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the next data byte to be transmitted

This function is used by the Master module continue each byte of a multi-byte trasmission. This function

- Transmits each data byte of a multi-byte transmission to the Slave

Modified registers are **UCBxTXBUF**

Returns
None.

12.6.3.28 bool I2C_masterSendMultiByteNextWithTimeout (uint32_t *moduleInstance,* uint8_t *txData,* uint32_t *timeout*)

Continues multi-byte transmission from Master to Slave with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: - **EUSCI_B0_BASE** - **EUSCI_B1_BASE** - **EUSCI_B2_BASE** - **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the next data byte to be transmitted
timeout	is the amount of time to wait until giving up

This function is used by the Master module continue each byte of a multi-byte transmission. This function

- Transmits each data byte of a multi-byte transmission to the Slave

Modified registers are **UCBxTXBUF**

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.29 void I2C_masterSendMultiByteStart (uint32_t *moduleInstance,* uint8_t *txData*)

Starts multi-byte transmission from Master to Slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the first data byte to be transmitted

This function is used by the Master module to send a single byte. This function

- Sends START
- Transmits the first data byte of a multi-byte transmission to the Slave

Modified registers are **UCBxIE**, **UCBxCTL1**, **UCBxIFG**, **UCBxTXBUF**, **UCBxIE**

Returns
None.

12.6.3.30 bool I2C_masterSendMultiByteStartWithTimeout (uint32_t *moduleInstance,* uint8_t *txData,* uint32_t *timeout*)

Starts multi-byte transmission from Master to Slave with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the first data byte to be transmitted
timeout	is the amount of time to wait until giving up

This function is used by the Master module to send a single byte. This function

- Sends START
- Transmits the first data byte of a multi-byte transmission to the Slave

Modified registers are **UCBxIE**, **UCBxCTL1**, **UCBxIFG**, **UCBxTXBUF**, **UCBxIE**

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.31 void I2C_masterSendMultiByteStop (uint32_t *moduleInstance*)

Send STOP byte at the end of a multi-byte transmission from Master to Slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: - **EUSCI_B0_BASE** - **EUSCI_B1_BASE** - **EUSCI_B2_BASE** - **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module send STOP at the end of a multi-byte transmission

This function

- Send a STOP after current transmission is complete

Modified bits are **UCTXSTP** bit of **UCBxCTL1**.

Returns
None.

12.6.3.32 bool I2C_masterSendMultiByteStopWithTimeout (uint32_t *moduleInstance,* uint32_t *timeout*)

Send STOP byte at the end of a multi-byte transmission from Master to Slave with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
timeout	is the amount of time to wait until giving up

This function is used by the Master module send STOP at the end of a multi-byte transmission

This function

- Send a STOP after current transmission is complete

Modified bits are **UCTXSTP** bit of **UCBxCTL1**.

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.33 void I2C_masterSendSingleByte (uint32_t *moduleInstance,* uint8_t *txData*)

Does single byte transmission from Master to Slave

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the data byte to be transmitted

This function is used by the Master module to send a single byte. This function

- Sends START
- Transmits the byte to the Slave
- Sends STOP

Modified registers are **UCBxIE**, **UCBxCTL1**, **UCBxIFG**, **UCBxTXBUF**, **UCBxIE**

Returns
none

12.6.3.34 bool I2C_masterSendSingleByteWithTimeout (uint32_t *moduleInstance,* uint8_t *txData,* uint32_t *timeout*)

Does single byte transmission from Master to Slave with timeout

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: - **EUSCI_B0_BASE** - **EUSCI_B1_BASE** - **EUSCI_B2_BASE** - **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
txData	is the data byte to be transmitted
timeout	is the amount of time to wait until giving up

This function is used by the Master module to send a single byte. This function

- Sends START
- Transmits the byte to the Slave
- Sends STOP

Modified registers are **UCBxIE**, **UCBxCTL1**, **UCBxIFG**, **UCBxTXBUF**, **UCBxIE**

Returns
0x01 or 0x00URE of the transmission process.

12.6.3.35 void I2C_masterSendStart (uint32_t *moduleInstance*)

This function is used by the Master module to initiate START

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function is used by the Master module to initiate STOP

Modified bits are UCTXSTT bit of UCBxCTLW0.

Returns
None.

12.6.3.36 void I2C_registerInterrupt (uint32_t *moduleInstance,* void(∗)(void) *intHandler*)

Registers an interrupt handler for I2C interrupts.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
intHandler	is a pointer to the function to be called when the timer capture compare interrupt occurs.

This function registers the handler to be called when an I2C interrupt occurs. This function enables the global interrupt in the interrupt controller; specific I2C interrupts must be enabled via I2C_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via I2C_clearInterruptFlag().

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

12.6.3.37 void I2C_setMode (uint32_t *moduleInstance,* uint_fast8_t *mode*)

Sets the mode of the I2C device

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
mode	indicates whether module is in transmit/receive mode ■ **EUSCI_B_I2C_TRANSMIT_MODE** ■ **EUSCI_B_I2C_RECEIVE_MODE** [Default value]

Modified bits are **UCTR** of **UCBxCTL1** register

Returns
None.

12.6.3.38 void I2C_setSlaveAddress (uint32_t *moduleInstance,* uint_fast16_t *slaveAddress*)

Sets the address that the I2C Master will place on the bus.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
slaveAddress	7-bit or 10-bit slave address

This function will set the address that the I2C Master will place on the bus when initiating a transaction. Modified register is **UCBxI2CSA** register

Returns
None.

12.6.3.39 uint8_t I2C_slaveGetData (uint32_t *moduleInstance*)

Receives a byte that has been sent to the I2C Module.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function reads a byte of data from the I2C receive data Register.

Returns
Returns the byte received from by the I2C module, cast as an uint8_t. Modified bit is **UCBxRXBUF** register

12.6.3.40 void I2C_slavePutData (uint32_t *moduleInstance,* uint8_t *transmitData*)

Transmits a byte from the I2C Module.

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
transmitData	data to be transmitted from the I2C module

This function will place the supplied data into I2C transmit data register to start transmission
Modified register is **UCBxTXBUF** register

Returns
None.

12.6.3.41 void I2C_slaveSendNAK (uint32_t *moduleInstance*)

This function is used by the slave to send a NAK out over the I2C line

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

Returns
None.

12.6.3.42 void I2C_unregisterInterrupt (uint32_t *moduleInstance*)

Unregisters the interrupt handler for the timer

Parameters

moduleInstance	is the instance of the eUSCI B (I2C) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.

This function unregisters the handler to be called when timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

13 Nested Vector Interrupt Controller (NVIC)

13.1 Module Operation

The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, register interrupt handlers, and set the priority of interrupts.

The NVIC provides global interrupt masking, prioritization, and handler dispatching. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The NVIC is tightly coupled with the Cortex-M microprocessor. When the processor responds to an interrupt, the NVIC supplies the address of the function to handle the interrupt directly to the processor. This action eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, reducing interrupt response time.

The interrupt prioritization in the NVIC allows higher priority interrupts to be handled before lower priority interrupts, as well as allowing preemption of lower priority interrupt handlers by higher priority interrupts. Again, this helps reduce interrupt response time (for example, a 1 ms system control interrupt is not held off by the execution of a lower priority 1 second housekeeping interrupt handler).

Sub-prioritization is also possible; instead of having N bits of preemptable prioritization, the NVIC can be configured (via software) for N - M bits of preemptable prioritization and M bits of sub-priority. In this scheme, two interrupts with the same preemptable prioritization but different sub-priorities do not cause a preemption; tail chaining is used instead to process the two interrupts back-to-back.

If two interrupts with the same priority (and sub-priority if so configured) are asserted at the same time, the one with the lower interrupt number is processed first. The NVIC keeps track of the nesting of interrupt handlers, allowing the processor to return from interrupt context only once all nested and pending interrupts have been handled.

Interrupt handlers can be configured in one of two ways; statically at compile time or dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the application's startup code. When statically configured, the interrupts must be explicitly enabled in the NVIC via Interrupt_enableInterrupt() before the processor can respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself). Statically configuring the interrupt table provides the fastest interrupt response time because the stacking operation (a write to SRAM) can be performed in parallel with the interrupt handler table fetch (a read from Flash), as well as the prefetch of the interrupt handler itself (assuming it is also in Flash).

Alternatively, interrupts can be configured at run-time using Interrupt_registerInterrupt(). When using Interrupt_registerInterrupt(), the interrupt must also be enabled as before; when using the analogue in each individual driver, Interrupt_enableInterrupt() is called by the driver and does not need to be called by the application. Run-time configuration of interrupts adds a small latency to

the interrupt response time because the stacking operation (a write to SRAM) and the interrupt handler table fetch (a read from SRAM) must be performed sequentially.

Run-time configuration of interrupt handlers requires that the interrupt handler table be placed on a 1-kB boundary in SRAM (typically this is at the beginning of SRAM). Failure to do so results in an incorrect vector address being fetched in response to an interrupt. The vector table is in a section called “vtable” and should be placed appropriately with a linker script.

13.2 Basic Operation Modes

The primary function of the interrupt controller API is to manage the interrupt vector table used by the NVIC to dispatch interrupt requests. Registering an interrupt handler is a simple matter of inserting the handler address into the table. By default, the table is filled with pointers to an internal handler that loops forever; it is an error for an interrupt to occur when there is no interrupt handler registered to process it. Therefore, interrupt sources should not be enabled before a handler has been registered, and interrupt sources should be disabled before a handler is unregistered. Interrupt handlers are managed with Interrupt_registerInterrupt() and Interrupt_unregisterInterrupt().

Each interrupt source can be individually enabled and disabled via Interrupt_enableInterrupt() and Interrupt_disableInterrupt(). 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 used as a simple critical section (only an NMI can interrupt the processor while the processor interrupt is disabled), although masking the processor interrupt can have adverse effects on the interrupt response time.

The priority of each interrupt source can be set and examined via Interrupt_setPriority() and Interrupt_getPriority(). The priority assignments are defined by the hardware; the upper N bits of the 8-bit priority are examined to determine the priority of an interrupt (for the MSP432 family, N is 3). This protocol allows priorities to be defined without knowledge of the exact number of supported priorities; moving to a device with more or fewer priority bits is made easier as the interrupt source continues to have a similar level of priority. Smaller priority numbers correspond to higher interrupt priority, so 0 is the highest priority.

13.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the Interrupt module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure interrupt priorities. For a set of more detailed code examples, please refer to the code examples in the examples/ directory of the SDK release:

```
/* Configuring interrupt priorities  */
MAP_Interrupt_setPriority(INT_EUSCIB0,  0x20);
MAP_Interrupt_setPriority(INT_EUSCIA0,  0x40);
```

13.4 Definitions

Functions

- void Interrupt_disableInterrupt (uint32_t interruptNumber)
- bool Interrupt_disableMaster (void)
- void Interrupt_disableSleepOnIsrExit (void)
- void Interrupt_enableInterrupt (uint32_t interruptNumber)
- bool Interrupt_enableMaster (void)
- void Interrupt_enableSleepOnIsrExit (void)
- uint8_t Interrupt_getPriority (uint32_t interruptNumber)
- uint32_t Interrupt_getPriorityGrouping (void)
- uint8_t Interrupt_getPriorityMask (void)
- uint32_t Interrupt_getVectorTableAddress (void)
- bool Interrupt_isEnabled (uint32_t interruptNumber)
- void Interrupt_pendInterrupt (uint32_t interruptNumber)
- void Interrupt_registerInterrupt (uint32_t interruptNumber, void(*intHandler)(void))
- void Interrupt_setPriority (uint32_t interruptNumber, uint8_t priority)
- void Interrupt_setPriorityGrouping (uint32_t bits)
- void Interrupt_setPriorityMask (uint8_t priorityMask)
- void Interrupt_setVectorTableAddress (uint32_t addr)
- void Interrupt_unpendInterrupt (uint32_t interruptNumber)
- void Interrupt_unregisterInterrupt (uint32_t interruptNumber)

13.4.1 Detailed Description

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

13.4.2 Function Documentation

13.4.2.1 void Interrupt_disableInterrupt (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.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
None.

Referenced by ADC14_unregisterInterrupt(), AES256_unregisterInterrupt(), COMP_E_unregisterInterrupt(), CS_unregisterInterrupt(), DMA_unregisterInterrupt(), GPIO_unregisterInterrupt(), I2C_unregisterInterrupt(), MPU_disableInterrupt(), PCM_unregisterInterrupt(), PSS_unregisterInterrupt(), RTC_C_unregisterInterrupt(), SPI_unregisterInterrupt(), Timer32_unregisterInterrupt(), Timer_A_unregisterInterrupt(), UART_unregisterInterrupt(), and WDT_A_unregisterInterrupt().

13.4.2.2 bool Interrupt_disableMaster (void)

Disables the processor interrupt.

This function prevents the processor from receiving interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Returns
Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

Referenced by PCM_gotoLPM0InterruptSafe(), PCM_gotoLPM3InterruptSafe(), and PCM_gotoLPM4InterruptSafe().

13.4.2.3 void Interrupt_disableSleepOnIsrExit (void)

Disables the processor to sleep when exiting an ISR.

Returns
None

13.4.2.4 void Interrupt_enableInterrupt (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.

Valid values will vary from part to part, so it is important to check the device specific datasheet, however for MSP432 101 the following values can be provided:

- **FAULT_NMI**
- **FAULT_HARD**
- **FAULT_MPU**
- **FAULT_BUS**
- **FAULT_USAGE**
- **FAULT_SVCALL**
- **FAULT_DEBUG**
- **FAULT_PENDSV**
- **FAULT_SYSTICK**
- **INT_PSS**
- **INT_CS**
- **INT_PCM**
- **INT_WDT_A**
- **INT_FPU**
- **INT_FLCTL**
- **INT_COMP0**
- **INT_COMP1**
- **INT_TA0_0**
- **INT_TA0_N**
- **INT_TA1_0**
- **INT_TA1_N**
- **INT_TA2_0**
- **INT_TA2_N**
- **INT_TA3_0**
- **INT_TA3_N**
- **INT_EUSCIA0**
- **INT_EUSCIA1**
- **INT_EUSCIA2**
- **INT_EUSCIA3**
- **INT_EUSCIB0**
- **INT_EUSCIB1**
- **INT_EUSCIB2**
- **INT_EUSCIB3**
- **INT_ADC14**

- **INT_T32_INT1**
- **INT_T32_INT2**
- **INT_T32_INTC**
- **INT_AES**
- **INT_RTCC**
- **INT_DMA_ERR**
- **INT_DMA_INT3**
- **INT_DMA_INT2**
- **INT_DMA_INT1**
- **INT_DMA_INT0**
- **INT_PORT1**
- **INT_PORT2**
- **INT_PORT3**
- **INT_PORT4**
- **INT_PORT5**
- **INT_PORT6**

Returns
None.

Referenced by ADC14_registerInterrupt(), AES256_registerInterrupt(), COMP_E_registerInterrupt(), CS_registerInterrupt(), DMA_registerInterrupt(), GPIO_registerInterrupt(), I2C_registerInterrupt(), MPU_enableInterrupt(), PCM_registerInterrupt(), PSS_registerInterrupt(), RTC_C_registerInterrupt(), SPI_registerInterrupt(), Timer32_registerInterrupt(), Timer_A_registerInterrupt(), UART_registerInterrupt(), and WDT_A_registerInterrupt().

13.4.2.5 bool Interrupt_enableMaster (void)

Enables the processor interrupt.

This function allows the processor to respond to interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Returns
Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

Referenced by PCM_gotoLPM0InterruptSafe(), PCM_gotoLPM3InterruptSafe(), and PCM_gotoLPM4InterruptSafe().

13.4.2.6 void Interrupt_enableSleepOnIsrExit (void)

Enables the processor to sleep when exiting an ISR. For low power operation, this is ideal as power cycles are not wasted with the processing required for waking up from an ISR and going back to sleep.

Returns
None

13.4.2.7 uint8_t Interrupt_getPriority (uint32_t *interruptNumber*)

Gets the priority of an interrupt.

Parameters

interruptNumber	specifies the interrupt in question.

This function gets the priority of an interrupt. See Interrupt_setPriority() for a definition of the priority value.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
Returns the interrupt priority, or -1 if an invalid interrupt was specified.

13.4.2.8 uint32_t Interrupt_getPriorityGrouping (void)

Gets the priority grouping of the interrupt controller.

This function returns the split between preemptable priority levels and sub-priority levels in the interrupt priority specification.

Returns
The number of bits of preemptable priority.

13.4.2.9 uint8_t Interrupt_getPriorityMask (void)

Gets the priority masking level

This function gets the current setting of the interrupt priority masking level. The value returned is the priority level such that all interrupts of that and lesser priority are masked. A value of 0 means that priority masking is disabled.

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater are blocked.

The hardware priority mechanism only looks at the upper N bits of the priority level (where N is 3 for the MSP432 family), so any prioritization must be performed in those bits.

Returns
Returns the value of the interrupt priority level mask.

13.4.2.10 uint32_t Interrupt_getVectorTableAddress (void)

Returns the address of the interrupt vector table.

Returns
Address of the vector table.

13.4.2.11 bool Interrupt_isEnabled (uint32_t *interruptNumber*)

Returns if a peripheral interrupt is enabled.

Parameters

interruptNumber	specifies the interrupt to check.

This function checks if the specified interrupt is enabled in the interrupt controller.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
A non-zero value if the interrupt is enabled.

13.4.2.12 void Interrupt_pendInterrupt (uint32_t *interruptNumber*)

Pends an interrupt.

Parameters

interruptNumber	specifies the interrupt to be pended.

The specified interrupt is pended in the interrupt controller. Pending an interrupt causes the interrupt controller to execute the corresponding interrupt handler at the next available time, based on the current interrupt state priorities. For example, if called by a higher priority interrupt handler, the specified interrupt handler is not called until after the current interrupt handler has completed execution. The interrupt must have been enabled for it to be called.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
None.

13.4.2.13 void Interrupt_registerInterrupt (uint32_t *interruptNumber,* void(∗)(void) *intHandler*)

Registers a function to be called when an interrupt occurs.

Parameters

interruptNumber	specifies the interrupt in question.
intHandler	is a pointer to the function to be called.

Note

The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise the NVIC does not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter.
This function is only used if the customer wants to specify the interrupt handler at run time. In most cases, this is done through means of the user setting the ISR function pointer in the startup file. Refer Refer to the Module Operation section for more details.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns

None.

Referenced by ADC14_registerInterrupt(), AES256_registerInterrupt(), COMP_E_registerInterrupt(), CS_registerInterrupt(), DMA_registerInterrupt(), GPIO_registerInterrupt(), I2C_registerInterrupt(), MPU_registerInterrupt(), PCM_registerInterrupt(), PSS_registerInterrupt(), RTC_C_registerInterrupt(), SPI_registerInterrupt(), SysTick_registerInterrupt(), Timer32_registerInterrupt(), Timer_A_registerInterrupt(), UART_registerInterrupt(), and WDT_A_registerInterrupt().

13.4.2.14 void Interrupt_setPriority (uint32_t *interruptNumber,* uint8_t *priority*)

Sets the priority of an interrupt.

Parameters

interruptNumber	specifies the interrupt in question.
priority	specifies the priority of the interrupt.

This function is used to set the priority of an interrupt. When multiple interrupts are asserted simultaneously, the ones with the highest priority are processed before the lower priority interrupts. Smaller numbers correspond to higher interrupt priorities; priority 0 is the highest interrupt priority.

The hardware priority mechanism only looks at the upper N bits of the priority level (where N is 3 for the MSP432 family), so any prioritization must be performed in those bits. The remaining bits can be used to sub-prioritize the interrupt sources, and may be used by the hardware priority mechanism on a future part. This arrangement allows priorities to migrate to different NVIC implementations without changing the gross prioritization of the interrupts.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns

None.

13.4.2.15 void Interrupt_setPriorityGrouping (uint32_t *bits*)

Sets the priority grouping of the interrupt controller.

Parameters

bits	specifies the number of bits of preemptable priority.

This function specifies the split between preemptable priority levels and sub-priority levels in the interrupt priority specification. The range of the grouping values are dependent upon the hardware implementation; on the MSP432 family, three bits are available for hardware interrupt prioritization and therefore priority grouping values of three through seven have the same effect.

Returns
None.

13.4.2.16 void Interrupt_setPriorityMask (uint8_t *priorityMask*)

Sets the priority masking level

Parameters

priorityMask	is the priority level that is masked.

This function sets the interrupt priority masking level so that all interrupts at the specified or lesser priority level are masked. Masking interrupts can be used to globally disable a set of interrupts with priority below a predetermined threshold. A value of 0 disables priority masking.

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater are blocked.

The hardware priority mechanism only looks at the upper N bits of the priority level (where N is 3 for the MSP432 family), so any prioritization must be performed in those bits.

Returns
None.

13.4.2.17 void Interrupt_setVectorTableAddress (uint32_t *addr*)

Sets the address of the vector table. This function is for advanced users who might want to switch between multiple instances of vector tables (perhaps between flash/ram).

Parameters

addr	is the new address of the vector table.

Returns
None.

13.4.2.18 void Interrupt_unpendInterrupt (uint32_t *interruptNumber*)

Un-pends an interrupt.

Parameters

interruptNumber	specifies the interrupt to be un-pended.

The specified interrupt is un-pended in the interrupt controller. This will cause any previously generated interrupts that have not been handled yet (due to higher priority interrupts or the interrupt no having been enabled yet) to be discarded.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
None.

13.4.2.19 void Interrupt_unregisterInterrupt (uint32_t *interruptNumber*)

Unregisters the function to be called when an interrupt occurs.

Parameters

interruptNumber	specifies the interrupt in question.

This function is used to indicate that no handler should be called when the given interrupt is asserted to the processor. The interrupt source is automatically disabled (via Interrupt_disableInterrupt()) if necessary.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

See Interrupt_enableInterrupt for details about the interrupt parameter

Returns
None.

Referenced by ADC14_unregisterInterrupt(), AES256_unregisterInterrupt(), COMP_E_unregisterInterrupt(), CS_unregisterInterrupt(), DMA_unregisterInterrupt(), GPIO_unregisterInterrupt(), I2C_unregisterInterrupt(), MPU_unregisterInterrupt(), PCM_unregisterInterrupt(), PSS_unregisterInterrupt(), RTC_C_unregisterInterrupt(), SPI_unregisterInterrupt(), SysTick_unregisterInterrupt(), Timer32_unregisterInterrupt(), Timer_A_unregisterInterrupt(), UART_unregisterInterrupt(), and WDT_A_unregisterInterrupt().

14 LCD Module (LCD_F)

14.1 Module Operation

14.2 Definitions

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

15 Memory Protection Unit (MPU)

15.1 Module Operation

The Memory Protection Unit (MPU) API provides functions to configure the MPU. The MPU is tightly coupled to the Cortex-M processor core and provides a means to establish access permissions on regions of memory.

Up to eight memory regions can be defined. Each region has a base address and a size. The size is specified as a power of 2 between 32 bytes and 4 GB, inclusive. The region's base address must be aligned to the size of the region. Each region also has access permissions. Code execution can be allowed or disallowed for a region. A region can be configured for read-only access, read/write access, or no access for both privileged and user modes. Access permissions can be used to create an environment where only kernel or system code can access certain hardware registers or sections of code.

The MPU creates 8 sub-regions within each region. Any sub-region or combination of sub-regions can be disabled, allowing creation of "holes" or complex overlaying regions with different permissions. The sub-regions can also be used to create an unaligned beginning or ending of a region by disabling one or more of the leading or trailing sub-regions.

Once the regions are defined and the MPU is enabled, any access violation of a region causes a memory management fault, and the fault handler is acted.

15.2 Module Operation

The MPU APIs provide a means to enable and configure the MPU and memory protection regions.

Generally, the memory protection regions should be defined before enabling the MPU. The regions can be configured by calling MPU_setRegion() once for each region to be configured.

A region that is defined by MPU_setRegion() can be initially enabled or disabled. If the region is not initially enabled, it can be enabled later by calling MPU_enableRegion(). An enabled region can be disabled by calling MPU_disableRegion(). When a region is disabled, its configuration is preserved as long as it is not overwritten. In this case, it can be enabled again with MPU_enableRegion() without the need to reconfigure the region.

Care must be taken when setting up a protection region using MPU_setRegion(). The function writes to multiple registers and is not protected from interrupts. Therefore, it is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to protect against this is to make sure that a region is always disabled before making any changes. Otherwise, it is up to the caller to ensure that MPU_setRegion() is always called from within code that cannot be interrupted, or from code that is not be affected if an interrupt occurs while the region attributes are being changed.

The attributes of a region that have already been programmed can be retrieved and saved using

the MPU_getRegionCount() function. This function is intended to save the attributes in a format that can be used later to reload the region using the MPU_setRegion() function. Note that the enable state of the region is saved with the attributes and takes effect when the region is reloaded.

When one or more regions are defined, the MPU can be enabled by calling MPU_enableModule(). This function turns on the MPU and also defines the behavior in privileged mode and in the Hard Fault and NMI fault handlers. The MPU can be configured so that when in privileged mode and no regions are enabled, a default memory map is applied. If this feature is not enabled, then a memory management fault is generated if the MPU is enabled and no regions are configured and enabled. The MPU can also be set to use a default memory map when in the Hard Fault or NMI handlers, instead of using the configured regions. All of these features are selected when calling MPU_enableModule(). When the MPU is enabled, it can be disabled by calling MPU_disableModule().

Finally, if the application is using run-time interrupt registration (see Interrupt_registerInterrupt()), then the function MPU_registerInterrupt() can be used to install the fault handler which is called whenever a memory protection violation occurs. This function also enables the fault handler. If compile-time interrupt registration is used, then the Interrupt_enableInterrupt() function with the parameter **FAULT_MPU** must be used to enable the memory management fault handler. When the memory management fault handler has been installed with MPU_disableModule(), it can be removed by calling MPU_unregisterInterrupt().

15.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the MPU module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the MPU module to define a new memory region and set it as read only:

```
/* Memory region that we will protect  */
uint8_t memoryRegion[32];

/* MPU Configuration flag set - 32B region with read only for both privileged
 * and user accesses
 */
const uint32_t flagSet = MPU_RGN_SIZE_32B | MPU_RGN_PERM_EXEC
        | MPU_RGN_PERM_PRV_RO_USR_RO | MPU_SUB_RGN_DISABLE_7 | MPU_RGN_ENABLE;


 int main(void)
{
    /* Holding the Watchdog  and enabling master interrupts */
    WDT_A_holdTimer();
    Interrupt_enableMaster();

    /* Configuring P1.0 as output and P1.1 (switch) as input */
    GPIO_setAsOutputPin(GPIO_PORT_P1,  GPIO_PIN0);

    /* Configuring P1.1 as an input and enabling interrupts */
    MAP_GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1,  GPIO_PIN1);
    GPIO_clearInterruptFlag(GPIO_PORT_P1,  GPIO_PIN1);
    GPIO_enableInterrupt(GPIO_PORT_P1,  GPIO_PIN1);
    Interrupt_enableInterrupt(INT_PORT1);

    /* Setting and enabling the region - This will cause 0x3E000 - 0x3F000 to
     * read only
     */
```

```
    MPU_setRegion(0,   (uint32_t)memoryRegion,  flagSet);

    Interrupt_enableInterrupt(FAULT_MPU);

    MPU_enableModule(MPU_CONFIG_PRIV_DEFAULT);

    while(1)
    {
        /* Going to LPM3 and waiting for a GPIO interrupt */
        PCM_gotoLPM3();

        /* Trying to program the sector that we protected - This will cause
         * the MPU to fault and interrupt
         */
        memoryRegion[0] = 0xA5;
    }
}

/* GPIO ISR to wake up the CPU */
void PORT1_IRQHandler(void)
{
    GPIO_clearInterruptFlag(GPIO_PORT_P1,  GPIO_PIN1);
}

/* MPU Fault ISR */
void MemManage_Handler(void)
{
    GPIO_setOutputHighOnPin(GPIO_PORT_P1,  GPIO_PIN0);
    while(1);
}
```

15.4 Definitions

Functions

- void MPU_disableInterrupt (void)
- void MPU_disableModule (void)
- void MPU_disableRegion (uint32_t region)
- void MPU_enableInterrupt (void)
- void MPU_enableModule (uint32_t mpuConfig)
- void MPU_enableRegion (uint32_t region)
- void MPU_getRegion (uint32_t region, uint32_t *addr, uint32_t *pflags)
- uint32_t MPU_getRegionCount (void)
- void MPU_registerInterrupt (void(*intHandler)(void))
- void MPU_setRegion (uint32_t region, uint32_t addr, uint32_t flags)
- void MPU_unregisterInterrupt (void)

15.4.1 Detailed Description

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

15.4.2 Function Documentation

15.4.2.1 void MPU_disableInterrupt (void)

Disables the interrupt for the memory management fault.

Returns
None.

References Interrupt_disableInterrupt().

15.4.2.2 void MPU_disableModule (void)

Disables the MPU for use.

This function disables the Cortex-M memory protection unit. When the MPU is disabled, the default memory map is used and memory management faults are not generated.

Returns
None.

15.4.2.3 void MPU_disableRegion (uint32_t *region*)

Disables a specific region.

Parameters

region	is the region number to disable. Valid values are between 0 and 7 inclusively.

This function is used to disable a previously enabled memory protection region. The region remains configured if it is not overwritten with another call to MPU_setRegion(), and can be enabled again by calling MPU_enableRegion().

Returns
None.

15.4.2.4 void MPU_enableInterrupt (void)

Enables the interrupt for the memory management fault.

Returns
None.

References Interrupt_enableInterrupt().

15.4.2.5 void MPU_enableModule (uint32_t *mpuConfig*)

Enables and configures the MPU for use.

Parameters

mpuConfig	is the logical OR of the possible configurations.

This function enables the Cortex-M memory protection unit. It also configures the default behavior when in privileged mode and while handling a hard fault or NMI. Prior to enabling the MPU, at least one region must be set by calling MPU_setRegion() or else by enabling the default region for privileged mode by passing the **MPU_CONFIG_PRIV_DEFAULT** flag to MPU_enableModule(). Once the MPU is enabled, a memory management fault is generated for memory access violations.

The *mpuConfig* parameter should be the logical OR of any of the following:

- **MPU_CONFIG_PRIV_DEFAULT** enables the default memory map when in privileged mode and when no other regions are defined. If this option is not enabled, then there must be at least one valid region already defined when the MPU is enabled.
- **MPU_CONFIG_HARDFLT_NMI** enables the MPU while in a hard fault or NMI exception handler. If this option is not enabled, then the MPU is disabled while in one of these exception handlers and the default memory map is applied.
- **MPU_CONFIG_NONE** chooses none of the above options. In this case, no default memory map is provided in privileged mode, and the MPU is not enabled in the fault handlers.

Returns
None.

15.4.2.6 void MPU_enableRegion (uint32_t *region*)

Enables a specific region.

Parameters

region	is the region number to enable. Valid values are between 0 and 7 inclusively.

This function is used to enable a memory protection region. The region should already be configured with the MPU_setRegion() function. Once enabled, the memory protection rules of the region are applied and access violations cause a memory management fault.

Returns
None.

15.4.2.7 void MPU_getRegion (uint32_t *region,* uint32_t * *addr,* uint32_t * *pflags*)

Gets the current settings for a specific region.

Parameters

region	is the region number to get. Valid values are between 0 and 7 inclusively.
addr	points to storage for the base address of the region.

pflags	points to the attribute flags for the region.

This function retrieves the configuration of a specific region. The meanings and format of the parameters is the same as that of the MPU_setRegion() function.

This function can be used to save the configuration of a region for later use with the MPU_setRegion() function. The region's enable state is preserved in the attributes that are saved.

Returns
None.

15.4.2.8 uint32_t MPU_getRegionCount (void)

Gets the count of regions supported by the MPU.

This function is used to get the total number of regions that are supported by the MPU, including regions that are already programmed.

Returns
The number of memory protection regions that are available for programming using MPU_setRegion().

15.4.2.9 void MPU_registerInterrupt (void(*)(void) *intHandler*)

Registers an interrupt handler for the memory management fault.

Parameters

intHandler	is a pointer to the function to be called when the memory management fault occurs.

This function sets and enables the handler to be called when the MPU generates a memory management fault due to a protection region access violation.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_registerInterrupt().

15.4.2.10 void MPU_setRegion (uint32_t *region*, uint32_t *addr*, uint32_t *flags*)

Sets up the access rules for a specific region.

Parameters

region	is the region number to set up.

addr	is the base address of the region. It must be aligned according to the size of the region specified in flags.
flags	is a set of flags to define the attributes of the region.

This function sets up the protection rules for a region. The region has a base address and a set of attributes including the size. The base address parameter, *addr*, must be aligned according to the size, and the size must be a power of 2.

Parameters

region	is the region number to set. Valid values are between 0 and 7 inclusively.

The *flags* parameter is the logical OR of all of the attributes of the region. It is a combination of choices for region size, execute permission, read/write permissions, disabled sub-regions, and a flag to determine if the region is enabled.

The size flag determines the size of a region and must be one of the following:

- **MPU_RGN_SIZE_32B**
- **MPU_RGN_SIZE_64B**
- **MPU_RGN_SIZE_128B**
- **MPU_RGN_SIZE_256B**
- **MPU_RGN_SIZE_512B**
- **MPU_RGN_SIZE_1K**
- **MPU_RGN_SIZE_2K**
- **MPU_RGN_SIZE_4K**
- **MPU_RGN_SIZE_8K**
- **MPU_RGN_SIZE_16K**
- **MPU_RGN_SIZE_32K**
- **MPU_RGN_SIZE_64K**
- **MPU_RGN_SIZE_128K**
- **MPU_RGN_SIZE_256K**
- **MPU_RGN_SIZE_512K**
- **MPU_RGN_SIZE_1M**
- **MPU_RGN_SIZE_2M**
- **MPU_RGN_SIZE_4M**
- **MPU_RGN_SIZE_8M**
- **MPU_RGN_SIZE_16M**
- **MPU_RGN_SIZE_32M**
- **MPU_RGN_SIZE_64M**
- **MPU_RGN_SIZE_128M**
- **MPU_RGN_SIZE_256M**
- **MPU_RGN_SIZE_512M**
- **MPU_RGN_SIZE_1G**
- **MPU_RGN_SIZE_2G**
- **MPU_RGN_SIZE_4G**

The execute permission flag must be one of the following:

- **MPU_RGN_PERM_EXEC** enables the region for execution of code
- **MPU_RGN_PERM_NOEXEC** disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

- **MPU_RGN_PERM_PRV_NO_USR_NO** - no access in privileged or user mode
- **MPU_RGN_PERM_PRV_RW_USR_NO** - privileged read/write, user no access
- **MPU_RGN_PERM_PRV_RW_USR_RO** - privileged read/write, user read-only
- **MPU_RGN_PERM_PRV_RW_USR_RW** - privileged read/write, user read/write
- **MPU_RGN_PERM_PRV_RO_USR_NO** - privileged read-only, user no access
- **MPU_RGN_PERM_PRV_RO_USR_RO** - privileged read-only, user read-only

The region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled, allowing for creation of “holes” in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

- **MPU_SUB_RGN_DISABLE_0**
- **MPU_SUB_RGN_DISABLE_1**
- **MPU_SUB_RGN_DISABLE_2**
- **MPU_SUB_RGN_DISABLE_3**
- **MPU_SUB_RGN_DISABLE_4**
- **MPU_SUB_RGN_DISABLE_5**
- **MPU_SUB_RGN_DISABLE_6**
- **MPU_SUB_RGN_DISABLE_7**

Finally, the region can be initially enabled or disabled with one of the following flags:

- **MPU_RGN_ENABLE**
- **MPU_RGN_DISABLE**

As an example, to set a region with the following attributes: size of 32 KB, execution enabled, read-only for both privileged and user, one sub-region disabled, and initially enabled; the *flags* parameter would have the following value:

```
(MPU_RGN_SIZE_32K | MPU_RGN_PERM_EXEC | MPU_RGN_PERM_PRV_RO_USR_RO |
MPU_SUB_RGN_DISABLE_2 | MPU_RGN_ENABLE)
```

Note

This function writes to multiple registers and is not protected from interrupts. It is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to handle this is to disable a region before changing it. Refer to the discussion of this in the API Detailed Description section.

Returns

None.

15.4.2.11 void MPU_unregisterInterrupt (void)

Unregisters an interrupt handler for the memory management fault.

This function disables and clears the handler to be called when a memory management fault occurs.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_unregisterInterrupt().

16 Power Control Module (PCM)

16.1 Module Operation

The Power Control Manager (PCM) module for DriverLib is meant to simplify the management of power states and provide a level of intelligence to users for switching between power states.

16.2 Switching States

One of the most useful features of the PCM module is the ability for the user to switch between power states without having to worry about the logic requirements of the state transitions. By using the PCM_setPowerState function, DriverLib will take in a parameter for the power state and automatically handle all of the state transitions. Say that the user wants to switch to use the DCDC converter with a voltage level of VCORE1 (PCM_AM_DCDC_VCORE1). Say that that same user is currently in the default mode of using the LDO with a voltage level of VCORE0 (PCM_AM_LDO_VCORE0). Normally, the user would have to take into account that there is a state transition that must happen to PCM_AM_LDO_VCORE1, however with the PCM_setPowerState API the user does not need to worry about this. The call to change the power state in this example would be:

```
PCM_setPowerState(PCM_AM_DCDC_VCORE1);
```

16.3 Switching Modes/Levels

In addition to being able to switch between individual power states, the PCM DriverLib API module also gives the user the ability to switch between different power modes and levels. This gives the user a more granular approach to power management and allows for a more refined customization of the power driver.

For changing between power levels, the user will be able to switch back and forth between **PCM_VCORE0** and **PCM_VCORE1** using the PCM_setCoreVoltageLevel function. While using this function it is important to note that the underlying power mode will be preserved. For example, if PCM_setCoreVoltageLevel is called with the **PCM_VCORE1** parameter while the devices is in **PCM_AM_LDO_VCORE0** mode, the power state will be changed to **PCM_AM_LDO_VCORE1**. If the same API is called with the same parameter in **PCM_AM_DCDC_VCORE0** mode, the power state will be changed to **PCM_AM_DCDC_VCORE1** mode.

The same preservation logic also applies while switching between power modes. If the PCM_setPowerMode function is called with the **PCM_DCDC_MODE** parameter while the device is in **PCM_AM_LDO_VCORE0** mode, the device will change to **PCM_AM_DCDC_VCORE0** mode (leaving the voltage level unchanged).

16.4 Low Power Mode and State Retention

In addition to being able to manipulate individual states/modes/levels, APIs are also provided to simplify entry into the low power modes of MSP432.

Low Power Entry Functions:

- PCM_gotoLPM0
- PCM_gotoLPM3
- PCM_shutdownDevice

When using these low power modes entry functions, it is important to note that the original state of the device before low power mode entry is retained. After the devices wakes up from low power mode, the original power mode is restored. For example, say that the device is in **PCM_AM_DCDC_VCORE0** mode and then the user calls the PCM_gotoLPM3 API. Since MSP432 devices are not allowed to go into LMP3 while in a DCDC power mode, the API will have the intelligence to first change into **PCM_AM_LDO_VCORE0** mode, and then go to LPM3. When the device wakes up, the API will automatically switch back to **PCM_AM_DCDC_VCORE0** mode. If the user wants to go into DSL in the previous example without the state preservation, the PCM_setPowerState function should be used with the **PCM_LPM3** parameter.

16.5 Enabling/Disabling Rude Mode

If the user calls a low power entry function that disables a clock source while an active peripheral is accessing the clock source, by default MSP432 will not allow the transition. This can be enabled/disabled by using the PCM_enableRudeMode and PCM_disableRudeMode functions respectively. By using these functions, the user can set the device to "force" its way into the low power mode by forcibly halting any dependent clock resource.

16.6 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the PCM module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to change power levels with the PCM module. This is done in order to facilitate a higher frequency of 48Mhz. For a set of more detailed code examples, please refer to the code examples in the examples/ directory of the SDK release:

```
/* Re-enabling port pin interrupt */
MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1,  GPIO_PIN1);
MAP_Interrupt_enableInterrupt(INT_PORT1);
MAP_Interrupt_enableMaster();

/* Change to new power state */
MAP_PCM_setPowerState(powerStates[curPowerState]);
```

16.7 Definitions

Functions

- void PCM_clearInterruptFlag (uint32_t flags)
- void PCM_disableInterrupt (uint32_t flags)
- void PCM_disableRudeMode (void)
- void PCM_enableInterrupt (uint32_t flags)
- void PCM_enableRudeMode (void)
- uint8_t PCM_getCoreVoltageLevel (void)
- uint32_t PCM_getEnabledInterruptStatus (void)
- uint32_t PCM_getInterruptStatus (void)
- uint8_t PCM_getPowerMode (void)
- uint8_t PCM_getPowerState (void)
- bool PCM_gotoLPM0 (void)
- bool PCM_gotoLPM0InterruptSafe (void)
- bool PCM_gotoLPM3 (void)
- bool PCM_gotoLPM3InterruptSafe (void)
- bool PCM_gotoLPM4 (void)
- bool PCM_gotoLPM4InterruptSafe (void)
- void PCM_registerInterrupt (void(*intHandler)(void))
- bool PCM_setCoreVoltageLevel (uint_fast8_t voltageLevel)
- bool PCM_setCoreVoltageLevelNonBlocking (uint_fast8_t voltageLevel)
- bool PCM_setCoreVoltageLevelWithTimeout (uint_fast8_t voltageLevel, uint32_t timeOut)
- bool PCM_setPowerMode (uint_fast8_t powerMode)
- bool PCM_setPowerModeNonBlocking (uint_fast8_t powerMode)
- bool PCM_setPowerModeWithTimeout (uint_fast8_t powerMode, uint32_t timeOut)
- bool PCM_setPowerState (uint_fast8_t powerState)
- bool PCM_setPowerStateNonBlocking (uint_fast8_t powerState)
- bool PCM_setPowerStateWithTimeout (uint_fast8_t powerState, uint32_t timeout)
- bool PCM_shutdownDevice (uint32_t shutdownMode)
- void PCM_unregisterInterrupt (void)

16.7.1 Detailed Description

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

16.7.2 Function Documentation

16.7.2.1 void PCM_clearInterruptFlag (uint32_t *flags*)

Clears power system interrupt sources.

The specified power system interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep it from being called again immediately upon exit.

Note

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Parameters

flags	is a bit mask of the interrupt sources to be cleared. Must be a logical OR of ■ **PCM_DCDCERROR**, ■ **PCM_AM_INVALIDTRANSITION**, ■ **PCM_SM_INVALIDCLOCK**, ■ **PCM_SM_INVALIDTRANSITION**

Note

The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns

None.

16.7.2.2 void PCM_disableInterrupt (uint32_t *flags*)

Disables individual power control interrupt sources.

Parameters

flags	is a bit mask of the interrupt sources to be enabled. Must be a logical OR of: ■ **PCM_DCDCERROR**, ■ **PCM_AM_INVALIDTRANSITION**, ■ **PCM_SM_INVALIDCLOCK**, ■ **PCM_SM_INVALIDTRANSITION**

This function disables the indicated power control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns
None.

16.7.2.3 void PCM_disableRudeMode (void)

Disables "rude mode" entry into LPM3 and shutdown modes. With this mode disabled, an entry into shutdown or LPM3 will wait for any active clock requests to free up before going into LPM3 or shutdown.

Returns
None

16.7.2.4 void PCM_enableInterrupt (uint32_t *flags*)

Enables individual power control interrupt sources.

Parameters

flags	is a bit mask of the interrupt sources to be enabled. Must be a logical OR of: ■ **PCM_DCDCERROR**, ■ **PCM_AM_INVALIDTRANSITION**, ■ **PCM_SM_INVALIDCLOCK**, ■ **PCM_SM_INVALIDTRANSITION**

This function enables the indicated power control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Returns
None.

16.7.2.5 void PCM_enableRudeMode (void)

Enables "rude mode" entry into LPM3 and shutdown modes. With this mode enabled, an entry into shutdown or LPM3 will occur even if there are clock systems active. The system will forcibly turn off all clock/systems when going into these modes.

Returns
None

16.7.2.6 uint8_t PCM_getCoreVoltageLevel (void)

Returns the current powers state of the system see the PCM_setCoreVoltageLevel function for specific information about the modes.

Returns
The current voltage of the system

Possible return values include:

- **PCM_VCORE0**
- **PCM_VCORE1**
- **PCM_VCORELPM3**

References PCM_getPowerState().

16.7.2.7 uint32_t PCM_getEnabledInterruptStatus (void)

Gets the current interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Returns
The current interrupt status, enumerated as a bit field of:
- **PCM_DCDCERROR**,
- **PCM_AM_INVALIDTRANSITION**,
- **PCM_SM_INVALIDCLOCK**,
- **PCM_SM_INVALIDTRANSITION**

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

References PCM_getInterruptStatus().

16.7.2.8 uint32_t PCM_getInterruptStatus (void)

Gets the current interrupt status.

Returns
The current interrupt status, enumerated as a bit field of:
- **PCM_DCDCERROR**,
- **PCM_AM_INVALIDTRANSITION**,
- **PCM_SM_INVALIDCLOCK**,
- **PCM_SM_INVALIDTRANSITION**

Note
The interrupt sources vary based on the part in use. Please consult the data sheet for the part you are using to determine which interrupt sources are available.

Referenced by PCM_getEnabledInterruptStatus().

16.7.2.9 uint8_t PCM_getPowerMode (void)

Returns the current powers state of the system see the **PCM_setPowerState** function for specific information about the modes.

Returns
The current power mode of the system

References PCM_getPowerState().

Referenced by PCM_gotoLPM3().

16.7.2.10 uint8_t PCM_getPowerState (void)

Returns the current powers state of the system see the PCMChangePowerState function for specific information about the states.

Refer to PCM_setPowerState for possible return values.

Returns
The current power state of the system

Referenced by PCM_getCoreVoltageLevel(), PCM_getPowerMode(), and PCM_gotoLPM3().

16.7.2.11 bool PCM_gotoLPM0 (void)

Transitions the device into LPM0.

Refer to the device specific data sheet for specifics about low power modes.

Returns
false if LPM0 state cannot be entered, true otherwise.

Referenced by PCM_gotoLPM0InterruptSafe().

16.7.2.12 bool PCM_gotoLPM0InterruptSafe (void)

Transitions the device into LPM0 while maintaining a safe interrupt handling mentality. This function is meant to be used in situations where the user wants to go to LPM0, however does not want to go to "miss" any interrupts due to the fact that going to LPM0 is not an atomic operation. This function will modify the PRIMASK and on exit of the program the master interrupts will be disabled.

Refer to the device specific data sheet for specifics about low power modes.

Returns
false if LPM0 state cannot be entered, true otherwise.

References Interrupt_disableMaster(), Interrupt_enableMaster(), and PCM_gotoLPM0().

16.7.2.13 bool PCM_gotoLPM3 (void)

Transitions the device into LPM3

Refer to the device specific data sheet for specifics about low power modes. Note that since LPM3 cannot be entered from a DCDC power modes, the power mode is first switched to LDO operation (if in DCDC mode), LPM3 is entered, and the DCDC mode is restored on wake up.

Returns
false if LPM3 state cannot be entered, true otherwise.

References PCM_getPowerMode(), PCM_getPowerState(), PCM_setPowerMode(), and PCM_setPowerState().

Referenced by PCM_gotoLPM3InterruptSafe(), and PCM_gotoLPM4().

16.7.2.14 bool PCM_gotoLPM3InterruptSafe (void)

Transitions the device into LPM3 while maintaining a safe interrupt handling mentality. This function is meant to be used in situations where the user wants to go to LPM3, however does not want to go to "miss" any interrupts due to the fact that going to LPM3 is not an atomic operation. This function will modify the PRIMASK and on exit of the program the master interrupts will be disabled.

Refer to the device specific data sheet for specifics about low power modes. Note that since LPM3 cannot be entered from a DCDC power modes, the power mode is first switched to LDO operation (if in DCDC mode), the LPM3 is entered, and the DCDC mode is restored on wake up.

Returns
false if LPM3 cannot be entered, true otherwise.

References Interrupt_disableMaster(), Interrupt_enableMaster(), and PCM_gotoLPM3().

16.7.2.15 bool PCM_gotoLPM4 (void)

Transitions the device into LPM4. LPM4 is the exact same with LPM3, just with RTC_C and WDT_A disabled. When waking up, RTC_C and WDT_A will remain disabled until reconfigured by the user.

Returns
false if LPM4 state cannot be entered, true otherwise.

References PCM_gotoLPM3(), RTC_C_holdClock(), and WDT_A_holdTimer().

Referenced by PCM_gotoLPM4InterruptSafe().

16.7.2.16 bool PCM_gotoLPM4InterruptSafe (void)

Transitions the device into LPM4 while maintaining a safe interrupt handling mentality. This function is meant to be used in situations where the user wants to go to LPM4, however does not want to go to "miss" any interrupts due to the fact that going to LPM4 is not an atomic operation.

This function will modify the PRIMASK and on exit of the program the master interrupts will be disabled.

Refer to the device specific data sheet for specifics about low power modes. Note that since LPM3 cannot be entered from a DCDC power modes, the power mode is first switched to LDO operation (if in DCDC mode), LPM4 is entered, and the DCDC mode is restored on wake up.

Returns
false if LPM4 state cannot be entered, true otherwise.

References Interrupt_disableMaster(), Interrupt_enableMaster(), and PCM_gotoLPM4().

16.7.2.17 void PCM_registerInterrupt (void(∗)(void) *intHandler*)

Registers an interrupt handler for the power system interrupt.

Parameters

intHandler	is a pointer to the function to be called when the power system interrupt occurs.

This function registers the handler to be called when a clock system interrupt occurs. This function enables the global interrupt in the interrupt controller; specific PCM interrupts must be enabled via PCM_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via PCM_clearInterruptFlag .

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

16.7.2.18 bool PCM_setCoreVoltageLevel (uint_fast8_t *voltageLevel*)

Sets the core voltage level (Vcore). The function will take care of all power state transitions needed to shift between core voltage levels. Because transitions between voltage levels may require changes power modes, the power mode might temporarily be change. The power mode will be returned to the original state (with the new voltage level) at the end of a successful execution of this function.

Refer to the device specific data sheet for specifics about core voltage levels.

Parameters

voltageLevel	The voltage level to be shifted to. ■ **PCM_VCORE0**, ■ **PCM_VCORE1**

Returns
true if voltage level set, false otherwise.

16.7.2.19 bool PCM_setCoreVoltageLevelNonBlocking (uint_fast8_t *voltageLevel*)

Sets the core voltage level (Vcore). This function is similar to PCM_setCoreVoltageLevel, however there are no polling flags to ensure a state has changed. Execution is returned back to the calling program correctly. For MSP432, changing into different power modes/states require very specific logic. This function will initiate only one state transition and then return. It is up to the user to keep calling this function until the correct power state has been achieved.

Refer to the device specific data sheet for specifics about core voltage levels.

Parameters

voltageLevel	The voltage level to be shifted to. ■ **PCM_VCORE0**, ■ **PCM_VCORE1**

Returns
true if voltage level set, false otherwise.

16.7.2.20 bool PCM_setCoreVoltageLevelWithTimeout (uint_fast8_t *voltageLevel,* uint32_t *timeOut*)

Sets the core voltage level (Vcore). This function will take care of all power state transitions needed to shift between core voltage levels. Because transitions between voltage levels may require changes power modes, the power mode might temporarily be change. The power mode will be returned to the original state (with the new voltage level) at the end of a successful execution of this function.

This function is similar to PCMSetCoreVoltageLevel, however a timeout mechanism is used.

Refer to the device specific data sheet for specifics about core voltage levels.

Parameters

voltageLevel	The voltage level to be shifted to. ■ **PCM_VCORE0**, ■ **PCM_VCORE1**
timeOut	Number of loop iterations to timeout when checking for power state transitions. This should be used for debugging initial power/hardware configurations. After a stable hardware base is established, the PCMSetCoreVoltageLevel function should be used

Returns
true if voltage level set, false otherwise.

16.7.2.21 bool PCM_setPowerMode (uint_fast8_t *powerMode*)

Switches between power modes. This function will take care of all power state transitions needed to shift between power modes. Note for changing to DCDC mode, specific hardware considerations are required.

Refer to the device specific data sheet for specifics about power modes.

Parameters

powerMode	The voltage modes to be shifted to. Valid values are: ■ **PCM_LDO_MODE**, ■ **PCM_DCDC_MODE**, ■ **PCM_LF_MODE**

Returns
true if power mode is set, false otherwise.

Referenced by PCM_gotoLPM3().

16.7.2.22 bool PCM_setPowerModeNonBlocking (uint_fast8_t *powerMode*)

Sets the core voltage level (Vcore). This function is similar to PCM_setPowerMode, however there are no polling flags to ensure a state has changed. Execution is returned back to the calling program correctly. For MSP432, changing into different power modes/states require very specific logic. This function will initiate only one state transition and then return. It is up to the user to keep calling this function until the correct power state has been achieved.

Refer to the device specific data sheet for specifics about core voltage levels.

Parameters

powerMode	The voltage modes to be shifted to. Valid values are: ■ **PCM_LDO_MODE**, ■ **PCM_DCDC_MODE**, ■ **PCM_LF_MODE**

Returns
true if power mode change was initiated, false otherwise

16.7.2.23 bool PCM_setPowerModeWithTimeout (uint_fast8_t *powerMode,* uint32_t *timeOut*)

Switches between power modes. This function will take care of all power state transitions needed to shift between power modes. Note for changing to DCDC mode, specific hardware considerations are required.

This function is similar to PCMSetPowerMode, however a timeout mechanism is used.

Refer to the device specific data sheet for specifics about power modes.

Parameters

powerMode	The voltage modes to be shifted to. Valid values are: ■ **PCM_LDO_MODE**, ■ **PCM_DCDC_MODE**, ■ **PCM_LF_MODE**
timeOut	Number of loop iterations to timeout when checking for power state transitions. This should be used for debugging initial power/hardware configurations. After a stable hardware base is established, the PCMSetPowerMode function should be used

Returns

true if power mode is set, false otherwise.

16.7.2.24 bool PCM_setPowerState (uint_fast8_t *powerState*)

Switches between power states. This is a convenience function that combines the functionality of PCM_setPowerMode and PCM_setCoreVoltageLevel as well as the LPM0/LPM3 functions.

Refer to the device specific data sheet for specifics about power states.

Parameters

powerState	The voltage modes to be shifted to. Valid values are: ■ **PCM_AM_LDO_VCORE0**, [Active Mode, LDO, VCORE0] ■ **PCM_AM_LDO_VCORE1**, [Active Mode, LDO, VCORE1] ■ **PCM_AM_DCDC_VCORE0**, [Active Mode, DCDC, VCORE0] ■ **PCM_AM_DCDC_VCORE1**, [Active Mode, DCDC, VCORE1] ■ **PCM_AM_LF_VCORE0**, [Active Mode, Low Frequency, VCORE0] ■ **PCM_AM_LF_VCORE1**, [Active Mode, Low Frequency, VCORE1] ■ **PCM_LPM0_LDO_VCORE0**, [LMP0, LDO, VCORE0] ■ **PCM_LPM0_LDO_VCORE1**, [LMP0, LDO, VCORE1] ■ **PCM_LPM0_DCDC_VCORE0**, [LMP0, DCDC, VCORE0] ■ **PCM_LPM0_DCDC_VCORE1**, [LMP0, DCDC, VCORE1] ■ **PCM_LPM0_LF_VCORE0**, [LMP0, Low Frequency, VCORE0] ■ **PCM_LPM0_LF_VCORE1**, [LMP0, Low Frequency, VCORE1] ■ **PCM_LPM3**, [LPM3] ■ **PCM_LPM35_VCORE0**, [LPM3.5 VCORE 0] ■ **PCM_LPM4**, [LPM4] ■ **PCM_LPM45**, [LPM4.5]

Returns

true if power state is set, false otherwise.

Referenced by PCM_gotoLPM3().

16.7.2.25 bool PCM_setPowerStateNonBlocking (uint_fast8_t *powerState*)

Sets the power state of the part. This function is similar to PCM_getPowerState, however there are no polling flags to ensure a state has changed. Execution is returned back to the calling program correctly. For MSP432, changing into different power modes/states require very specific logic. This function will initiate only one state transition and then return. It is up to the user to keep calling this function until the correct power state has been achieved.

Refer to the device specific data sheet for specifics about core voltage levels.

Parameters

powerState	The voltage modes to be shifted to. Valid values are: ■ **PCM_AM_LDO_VCORE0**, [Active Mode, LDO, VCORE0] ■ **PCM_AM_LDO_VCORE1**, [Active Mode, LDO, VCORE1] ■ **PCM_AM_DCDC_VCORE0**, [Active Mode, DCDC, VCORE0] ■ **PCM_AM_DCDC_VCORE1**, [Active Mode, DCDC, VCORE1] ■ **PCM_AM_LF_VCORE0**, [Active Mode, Low Frequency, VCORE0] ■ **PCM_AM_LF_VCORE1**, [Active Mode, Low Frequency, VCORE1] ■ **PCM_LPM0_LDO_VCORE0**, [LMP0, LDO, VCORE0] ■ **PCM_LPM0_LDO_VCORE1**, [LMP0, LDO, VCORE1] ■ **PCM_LPM0_DCDC_VCORE0**, [LMP0, DCDC, VCORE0] ■ **PCM_LPM0_DCDC_VCORE1**, [LMP0, DCDC, VCORE1] ■ **PCM_LPM0_LF_VCORE0**, [LMP0, Low Frequency, VCORE0] ■ **PCM_LPM0_LF_VCORE1**, [LMP0, Low Frequency, VCORE1] ■ **PCM_LPM3**, [LPM3] ■ **PCM_LPM35_VCORE0**, [LPM3.5 VCORE 0] ■ **PCM_LPM45**, [LPM4.5]

Returns

true if power state change was initiated, false otherwise

16.7.2.26 bool PCM_setPowerStateWithTimeout (uint_fast8_t *powerState,* uint32_t *timeout*)

Switches between power states. This is a convenience function that combines the functionality of PCM_setPowerMode and PCM_setCoreVoltageLevel as well as the LPM modes.

This function is similar to PCM_setPowerState, however a timeout mechanism is used.

Refer to the device specific data sheet for specifics about power states.

Parameters

powerState	The voltage modes to be shifted to. Valid values are: ■ **PCM_AM_LDO_VCORE0**, [Active Mode, LDO, VCORE0] ■ **PCM_AM_LDO_VCORE1**, [Active Mode, LDO, VCORE1] ■ **PCM_AM_DCDC_VCORE0**, [Active Mode, DCDC, VCORE0] ■ **PCM_AM_DCDC_VCORE1**, [Active Mode, DCDC, VCORE1] ■ **PCM_AM_LF_VCORE0**, [Active Mode, Low Frequency, VCORE0] ■ **PCM_AM_LF_VCORE1**, [Active Mode, Low Frequency, VCORE1] ■ **PCM_LPM0_LDO_VCORE0**, [LMP0, LDO, VCORE0] ■ **PCM_LPM0_LDO_VCORE1**, [LMP0, LDO, VCORE1] ■ **PCM_LPM0_DCDC_VCORE0**, [LMP0, DCDC, VCORE0] ■ **PCM_LPM0_DCDC_VCORE1**, [LMP0, DCDC, VCORE1] ■ **PCM_LPM0_LF_VCORE0**, [LMP0, Low Frequency, VCORE0] ■ **PCM_LPM0_LF_VCORE1**, [LMP0, Low Frequency, VCORE1] ■ **PCM_LPM3**, [LPM3] ■ **PCM_LPM35_VCORE0**, [LPM3.5 VCORE 0] ■ **PCM_LPM4**, [LPM4] ■ **PCM_LPM45**, [LPM4.5]
timeout	Number of loop iterations to timeout when checking for power state transitions. This should be used for debugging initial power/hardware configurations. After a stable hardware base is established, the PCMSetPowerMode function should be used

Returns

true if power state is set, false otherwise. It is important to note that if a timeout occurs, false will be returned, however the power state at this point is not guaranteed to be the same as the state prior to the function call

16.7.2.27 bool PCM_shutdownDevice (uint32_t *shutdownMode*)

Transitions the device into LPM3.5/LPM4.5 mode.

Refer to the device specific data sheet for specifics about shutdown modes.

The following events will cause a wake up from LPM3.5 mode:

- Device reset
- External reset RST
- Enabled RTC, WDT, and wake-up I/O only interrupt events

The following events will cause a wake up from the LPM4.5 mode:

- Device reset
- External reset RST
- Wake-up I/O only interrupt events

Parameters

shutdownMode	Specific mode to go to. Valid values are: ■ **PCM_LPM35_VCORE0** ■ **PCM_LPM45**

Returns
false if LPM state cannot be entered, true otherwise.

16.7.2.28 void PCM_unregisterInterrupt (void)

Unregisters the interrupt handler for the power system.

This function unregisters the handler to be called when a power system interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

17 Port Mapper (PMAP)

17.1 Module Operation

The port mapping controller allows the flexible and reconfigurable mapping of digital functions to port pins.

The port mapping controller features are:

- Configuration protected by write access key.
- Default mapping provided for each port pin (device-dependent, the device pinout in the device-specific data sheet).
- Mapping can be reconfigured during runtime.
- Each output signal can be mapped to several output pins.

17.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the PMAP module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to use the PMAP module to redirect the output of a TimerA CCR register.

First is the array configuration to remap the port:

```
/* Port mapper configuration register */
const uint8_t port_mapping[] =
{
        //Port P2:
        PMAP_NONE,  PMAP_NONE,  PMAP_NONE,  PMAP_NONE,  PMAP_TA1CCR1A,  PMAP_NONE,
        PMAP_NONE,  PMAP_NONE
};
```

Next is the call to the actual PMAP API that persists the configuration:

```
    /* Remapping  TACCR0 to P2.4 */
    MAP_PMAP_configurePorts((const uint8_t *) port_mapping,  PMAP_P2MAP,  1,
            PMAP_DISABLE_RECONFIGURATION);
```

17.3 Definitions

Functions

- void PMAP_configurePorts (const uint8_t *portMapping, uint8_t pxMAPy, uint8_t numberOfPorts, uint8_t portMapReconfigure)

17.3.1 Detailed Description

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

17.3.2 Function Documentation

17.3.2.1 void PMAP_configurePorts (const uint8_t * *portMapping,* uint8_t *pxMAPy,* uint8_t *numberOfPorts,* uint8_t *portMapReconfigure*)

This function configures the MSP432 Port Mapper

Parameters

portMapping	is the pointer to init Data
pxMAPy	is the Port Mapper to initialize
numberOfPorts	is the number of Ports to initialize
portMapReconfigure	is used to enable/disable reconfiguration Valid values are **PMAP_ENABLE_RECONFIGURATION** **PMAP_DISABLE_RECONFIGURATION** [Default value] Modified registers are **PMAPKEYID**, **PMAPCTL**

Returns
None

18 Power Supply System (PSS)

18.1 Module Operation

The PSS module for the DriverLib allows the user to fully configure/setup the various analog power sources on the MSP432 device. This mainly involves enabling and disabling the high side supervisor/monitor. Performance modes of both the high side power supply can be configured and manipulated in order to optimize power efficiency. Additionally, the PSS interrupt can be configured to fire an interrupt on a power supply violation.

18.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the PSS module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to disable the high side power supervisor:

```
MAP_PSS_enableHighSide();
```

18.3 Definitions

Functions

- void PSS_clearInterruptFlag (void)
- void PSS_disableForcedDCDCOperation (void)
- void PSS_disableHighSide (void)
- void PSS_disableHighSideMonitor (void)
- void PSS_disableHighSidePinToggle (void)
- void PSS_disableInterrupt (void)
- void PSS_enableForcedDCDCOperation (void)
- void PSS_enableHighSide (void)
- void PSS_enableHighSideMonitor (void)
- void PSS_enableHighSidePinToggle (bool activeLow)
- void PSS_enableInterrupt (void)
- uint_fast8_t PSS_getHighSidePerformanceMode (void)
- uint_fast8_t PSS_getHighSideVoltageTrigger (void)
- uint32_t PSS_getInterruptStatus (void)
- void PSS_registerInterrupt (void(*intHandler)(void))
- void PSS_setHighSidePerformanceMode (uint_fast8_t powerMode)
- void PSS_setHighSideVoltageTrigger (uint_fast8_t triggerVoltage)
- void PSS_unregisterInterrupt (void)

18.3.1 Detailed Description

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

18.3.2 Function Documentation

18.3.2.1 void PSS_clearInterruptFlag (void)

Clears power supply system interrupt source.

Returns
None.

18.3.2.2 void PSS_disableForcedDCDCOperation (void)

Disables the "forced" mode of the DCDC regulator. In this mode, the fail safe mechanism that disables the regulator to LDO mode when the supply voltage falls below the minimum supply voltage required for DCDC operation is turned on.

Returns
None.

18.3.2.3 void PSS_disableHighSide (void)

Disables high side voltage supervisor/monitor.

Returns
None.

18.3.2.4 void PSS_disableHighSideMonitor (void)

Switches the high side of the power supply system to be a supervisor instead of a monitor

Returns
None.

18.3.2.5 void PSS_disableHighSidePinToggle (void)

Disables output of the High Side interrupt flag on the device **SVMHOUT** pin

Returns
None.

18.3.2.6 void PSS_disableInterrupt (void)

Disables the power supply system interrupt source.

Returns
None.

18.3.2.7 void PSS_enableForcedDCDCOperation (void)

Enables the "forced" mode of the DCDC regulator. In this mode, the fail safe mechanism that disables the regulator to LDO mode when the supply voltage falls below the minimum supply voltage required for DCDC operation is turned off.

Returns
None.

18.3.2.8 void PSS_enableHighSide (void)

Enables high side voltage supervisor/monitor.

Returns
None.

18.3.2.9 void PSS_enableHighSideMonitor (void)

Sets the high side voltage supervisor to monitor mode

Returns
None.

18.3.2.10 void PSS_enableHighSidePinToggle (bool *activeLow*)

Enables output of the High Side interrupt flag on the device **SVMHOUT** pin

Parameters

activeLow	True if the signal should be logic low when SVSMHIFG is set. False if signal should be high when **SVSMHIFG** is set.

Returns
None.

18.3.2.11 void PSS_enableInterrupt (void)

Enables the power supply system interrupt source.

Returns
None.

18.3.2.12 uint_fast8_t PSS_getHighSidePerformanceMode (void)

Gets the performance mode of the high side voltage regulator. Refer to the user's guide for specific information about information about the different performance modes.

Returns
Performance mode of the voltage regulator

18.3.2.13 uint_fast8_t PSS_getHighSideVoltageTrigger (void)

Returns the voltage level at which the high side of the device voltage regulator triggers a reset.

Returns
The voltage level that the high side voltage supervisor/monitor triggers a reset. This value is represented as an unsigned eight bit integer where only the lowest three bits are most significant. See PSS_setHighSideVoltageTrigger for information regarding the return value

18.3.2.14 uint32_t PSS_getInterruptStatus (void)

Gets the current interrupt status.

Returns
The current interrupt status (**PSS_SVSMH**)

18.3.2.15 void PSS_registerInterrupt (void(*)(void) *intHandler*)

Registers an interrupt handler for the power supply system interrupt.

Parameters

intHandler	is a pointer to the function to be called when the power supply system interrupt occurs.

This function registers the handler to be called when a power supply system interrupt occurs. This function enables the global interrupt in the interrupt controller; specific PSS interrupts must be enabled via PSS_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via PSS_clearInterruptFlag().

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

18.3.2.16 void PSS_setHighSidePerformanceMode (uint_fast8_t *powerMode*)

Sets the performance mode of the high side regulator. Full performance mode allows for the best response times while normal performance mode is optimized for the lowest possible current consumption.

Parameters

powerMode	is the performance mode to set. Valid values are one of the following: ■ **PSS_FULL_PERFORMANCE_MODE**, ■ **PSS_NORMAL_PERFORMANCE_MODE**

Returns
None.

18.3.2.17 void PSS_setHighSideVoltageTrigger (uint_fast8_t *triggerVoltage*)

Sets the voltage level at which the high side of the device voltage regulator triggers a reset. This value is represented as an unsigned eight bit integer where only the lowest three bits are most significant.

Parameters

triggerVoltage	Voltage level in which high side supervisor/monitor triggers a reset. See the device specific data sheet for details on these voltage levels.

Typical values will vary from part to part (so it is very important to check the SVSH section of the data sheet. For reference only, the typical MSP432 101 values are listed below:

- 0 –> 1.57V
- 1 –> 1.62V
- 2 –> 1.83V
- 3 –> 2V
- 4 –> 2.25V
- 5 –> 2.4V
- 6 –> 2.6V
- 7 –> 2.8V

Returns
None.

18.3.2.18 void PSS_unregisterInterrupt (void)

Unregisters the interrupt handler for the power supply system

This function unregisters the handler to be called when a power supply system interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

19 Reference Module (REF_A)

19.1 Module Operation

The Internal Reference (REF_A) API provides a set of functions for using the SDK REF_A modules. Functions are provided to setup and enable use of the Reference voltage, enable or disable the internal temperature sensor, and view the status of the inner workings of the REF module.

The reference module (REF_A) is responsible for generation of all critical reference voltages that can be used by various analog peripherals in a given device. The heart of the reference system is the bandgap from which all other references are derived by unity or non-inverting gain stages. The REFGEN sub-system consists of the bandgap, the bandgap bias, and the non-inverting buffer stage which generates the three primary voltage reference available in the system, namely 1.2 V, 1.45, 2.0 V, and 2.5 V. In addition, when enabled, a buffered bandgap voltage is available.

19.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the REF_A module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to enable the REF_A module for a 2.5v reference:

```
/* Setting reference voltage to 2.5  and enabling reference */
MAP_REF_A_setReferenceVoltage(REF_A_VREF2_5V);
MAP_REF_A_enableReferenceVoltage();
```

19.3 Definitions

Functions

- void REF_A_disableReferenceVoltage (void)
- void REF_A_disableReferenceVoltageOutput (void)
- void REF_A_disableTempSensor (void)
- void REF_A_enableReferenceVoltage (void)
- void REF_A_enableReferenceVoltageOutput (void)
- void REF_A_enableTempSensor (void)
- uint_fast8_t REF_A_getBandgapMode (void)
- bool REF_A_getBufferedBandgapVoltageStatus (void)
- bool REF_A_getVariableReferenceVoltageStatus (void)
- bool REF_A_isBandgapActive (void)
- bool REF_A_isRefGenActive (void)
- bool REF_A_isRefGenBusy (void)
- void REF_A_setBufferedBandgapVoltageOneTimeTrigger (void)
- void REF_A_setReferenceVoltage (uint_fast8_t referenceVoltageSelect)
- void REF_A_setReferenceVoltageOneTimeTrigger (void)

19.3.1 Detailed Description

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

19.3.2 Function Documentation

19.3.2.1 void REF_A_disableReferenceVoltage (void)

Disables the reference voltage.

This function is used to disable the generated reference voltage. Please note, if the REF_A_isRefGenBusy() returns **REF_A_BUSY**, this function will have no effect.

Modified bits are **REFON** of **REFCTL0** register.

Returns
none

19.3.2.2 void REF_A_disableReferenceVoltageOutput (void)

Disables the reference voltage as an output to a pin.

This function is used to disables the reference voltage being generated to be given to an output pin. Please note, if the REF_A_isRefGenBusy() returns **REF_A_BUSY**, this function will have no effect.

Modified bits are **REFOUT** of **REFCTL0** register.

Returns
none

19.3.2.3 void REF_A_disableTempSensor (void)

Disables the internal temperature sensor to save power consumption.

This function is used to turn off the internal temperature sensor to save on power consumption. The temperature sensor is enabled by default. Please note, that giving ADC12 module control over the REF module, the state of the temperature sensor is dependent on the controls of the ADC12 module. Please note, if the REF_A_isRefGenBusy() returns **REF_A_BUSY**, this function will have no effect.

Modified bits are **REFTCOFF** of **REFCTL0** register.

Returns
none

19.3.2.4 void REF_A_enableReferenceVoltage (void)

Enables the reference voltage to be used by peripherals.

This function is used to enable the generated reference voltage to be used other peripherals or by an output pin, if enabled. Please note, that giving ADC12 module control over the REF module, the state of the reference voltage is dependent on the controls of the ADC12 module. Please note, if the REF_A_isRefGenBusy() returns REF_A_BUSY, this function will have no effect.

Modified bits are **REFON** of **REFCTL0** register.

Returns
none

19.3.2.5 void REF_A_enableReferenceVoltageOutput (void)

Outputs the reference voltage to an output pin.

This function is used to output the reference voltage being generated to an output pin. Please note, the output pin is device specific. Please note, that giving ADC12 module control over the REF module, the state of the reference voltage as an output to a pin is dependent on the controls of the ADC12 module. Please note, if the REF_A_isRefGenBusy() returns **REF_A_BUSY**, this function will have no effect.

Modified bits are **REFOUT** of **REFCTL0** register.

Returns
none

19.3.2.6 void REF_A_enableTempSensor (void)

Enables the internal temperature sensor.

This function is used to turn on the internal temperature sensor to use by other peripherals. The temperature sensor is enabled by default. Please note, if the REF_A_isRefGenBusy() returns **REF_A_BUSY**, this function will have no effect.

Modified bits are **REFTCOFF** of **REFCTL0** register.

Returns
none

19.3.2.7 uint_fast8_t REF_A_getBandgapMode (void)

Returns the bandgap mode of the REF module.

This function is used to return the bandgap mode of the REF module, requested by the peripherals using the bandgap. If a peripheral requests static mode, then the bandgap mode will be static for all modules, whereas if all of the peripherals using the bandgap request sample mode, then that will be the mode returned. Sample mode allows the bandgap to be active only when necessary to save on power consumption, static mode requires the bandgap to be active until no peripherals are using it anymore.

Returns
The bandgap mode of the REF module:
- **REF_A_STATICMODE** if the bandgap is operating in static mode
- **REF_A_SAMPLEMODE** if the bandgap is operating in sample mode

19.3.2.8 bool REF_A_getBufferedBandgapVoltageStatus (void)

Returns the busy status of the reference generator in the REF module.

This function is used to return the buys status of the buffered bandgap voltage in the REF module. If the ref. generator is on and ready to use, then the status will be seen as active.

Returns
true if the buffered bandgap voltage is ready to be used, false otherwise

19.3.2.9 bool REF_A_getVariableReferenceVoltageStatus (void)

Returns the busy status of the variable reference voltage in the REF module.

This function is used to return the buys status of the variable reference voltage in the REF module. If the ref. generator is on and ready to use, then the status will be seen as active.

Returns
true if the variable bandgap voltage is ready to be used, false otherwise

19.3.2.10 bool REF_A_isBandgapActive (void)

Returns the active status of the bandgap in the REF module.

This function is used to return the active status of the bandgap in the REF module. If the bandgap is in use by a peripheral, then the status will be seen as active.

Returns
true if the bandgap is being used, false otherwise

19.3.2.11 bool REF_A_isRefGenActive (void)

Returns the active status of the reference generator in the REF module.

This function is used to return the active status of the reference generator in the REF module. If the ref. generator is on and ready to use, then the status will be seen as active.

Returns
true if the reference generator is active, false otherwise.

19.3.2.12 bool REF_A_isRefGenBusy (void)

Returns the busy status of the reference generator in the REF module.

This function is used to return the busy status of the reference generator in the REF module. If the ref. generator is in use by a peripheral, then the status will be seen as busy.

Returns
true if the reference generator is being used, false otherwise.

19.3.2.13 void REF_A_setBufferedBandgapVoltageOneTimeTrigger (void)

Enables the one-time trigger of the buffered bandgap voltage.

Triggers the one-time generation of the buffered bandgap voltage. Once the buffered bandgap voltage request is set, this bit is cleared by hardware

Modified bits are **RefGOT** of **REFCTL0** register.

Returns
none

19.3.2.14 void REF_A_setReferenceVoltage (uint_fast8_t *referenceVoltageSelect*)

Sets the reference voltage for the voltage generator.

Parameters

referenceVoltageSelect	is the desired voltage to generate for a reference voltage. Valid values are: ■ **REF_A_VREF1_2V** [Default] ■ **REF_A_VREF1_45V** ■ **REF_A_VREF2_5V** Modified bits are **REFVSEL** of **REFCTL0** register.

This function sets the reference voltage generated by the voltage generator to be used by other peripherals. This reference voltage will only be valid while the REF module is in control. Please note, if the REF_A_isRefGenBusy() returns **REF_BUSY**, this function will have no effect.

Returns
none

19.3.2.15 void REF_A_setReferenceVoltageOneTimeTrigger (void)

Enables the one-time trigger of the reference voltage.

Triggers the one-time generation of the variable reference voltage. Once the reference voltage request is set, this bit is cleared by hardware

Modified bits are **REFGENOT** of **REFCTL0** register.

Returns
none

20 Reset Controller (ResetCtl)

20.1 Module Operation

The DriverLib APIs for the MSP432 Reset Control are a set of power functions that enables programmers to manipulate all aspects of a system reset. The user is able to initiate both hard and soft resets as well as determine the cause of a prior system reset.

20.2 Reset Sources

Reset sources will vary from device to device (see the device specific datasheet for the reset source mappings relevant to your device). The ResetCtl for DriverLib defines a set of generic reset sources (such as RESET_SRC_0). In practice, it is a good idea to use a define statement to match these to a specific reset source. For example, MSP432's mapping could look something similar to the following:

```
#define RESET_SYSTEM_SRC    RESET_SRC_0
#define RESET_WDTTIME_SRC   RESET_SRC_1
#define RESET_WDTPW_SRC     RESET_SRC_2
#define RESET_CS_SRC        RESET_SRC_3
#define RESET_PCM_SRC       RESET_SRC_14
#define RESET_SYS_SRC       RESET_SRC_15
```

By defining these extra set of macros, the user code that accesses the DriverLib ResetCtl APIs are more legible. For example, when checking to see if a device was reset because of a CS violation (such as a XTAL fault), the user could write code similar to the following:

```
if(ResetCtl_getSoftResetSource() == RESET_CS_SRC)
{
    // Do reset handling here
}
```

20.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the ResetCtl module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing an ISR that initiates a software reset of the device. The idea here is that a push button could cause a software initiated reset.

```
/* GPIO ISR */
void PORT1_IRQHandler(void)
{
    uint32_t status;

    status = MAP_GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1,  status);

    /* Initiated a hard reset */
    if(status & GPIO_PIN1)
    {
        MAP_ResetCtl_initiateHardResetWithSource(RESET_SRC_8);
    }

}
```

20.4 Definitions

Functions

- void ResetCtl_clearHardResetSource (uint32_t mask)
- void ResetCtl_clearPCMFlags (void)
- void ResetCtl_clearPSSFlags (void)
- void ResetCtl_clearSoftResetSource (uint32_t mask)
- uint32_t ResetCtl_getHardResetSource (void)
- uint32_t ResetCtl_getPCMSource (void)
- uint32_t ResetCtl_getPSSSource (void)
- uint32_t ResetCtl_getSoftResetSource (void)
- void ResetCtl_initiateHardReset (void)
- void ResetCtl_initiateHardResetWithSource (uint32_t source)
- void ResetCtl_initiateSoftReset (void)
- void ResetCtl_initiateSoftResetWithSource (uint32_t source)

20.4.1 Detailed Description

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

20.4.2 Function Documentation

20.4.2.1 void ResetCtl_clearHardResetSource (uint32_t *mask*)

Clears the reset sources associated with at hard reset

Parameters

mask	- Bitwise OR of any of the following values: ■ **RESET_SRC_0**, ■ **RESET_SRC_1**, ■ **RESET_SRC_2**, ■ **RESET_SRC_3**, ■ **RESET_SRC_4**, ■ **RESET_SRC_5**, ■ **RESET_SRC_6**, ■ **RESET_SRC_7**, ■ **RESET_SRC_8**, ■ **RESET_SRC_9**, ■ **RESET_SRC_10**, ■ **RESET_SRC_11**, ■ **RESET_SRC_12**, ■ **RESET_SRC_13**, ■ **RESET_SRC_14**, ■ **RESET_SRC_15**

Returns
none

20.4.2.2 void ResetCtl_clearPCMFlags (void)

Clears the corresponding PCM reset source flags

Returns
none

20.4.2.3 void ResetCtl_clearPSSFlags (void)

Clears the PSS reset source flags

Returns
none

20.4.2.4 void ResetCtl_clearSoftResetSource (uint32_t *mask*)

Clears the reset sources associated with at soft reset

Parameters

mask	- Bitwise OR of any of the following values: ■ **RESET_SRC_0**, ■ **RESET_SRC_1**, ■ **RESET_SRC_2**, ■ **RESET_SRC_3**, ■ **RESET_SRC_4**, ■ **RESET_SRC_5**, ■ **RESET_SRC_6**, ■ **RESET_SRC_7**, ■ **RESET_SRC_8**, ■ **RESET_SRC_9**, ■ **RESET_SRC_10**, ■ **RESET_SRC_11**, ■ **RESET_SRC_12**, ■ **RESET_SRC_13**, ■ **RESET_SRC_14**, ■ **RESET_SRC_15**

Returns
none

20.4.2.5 uint32_t ResetCtl_getHardResetSource (void)

Retrieves previous hard reset sources

Returns
the bitwise or of previous reset sources. These sources must be cleared using the ResetCtl_clearHardResetSource function to be cleared. Possible values include:
- **RESET_SRC_0**,
- **RESET_SRC_1**,
- **RESET_SRC_2**,
- **RESET_SRC_3**,
- **RESET_SRC_4**,
- **RESET_SRC_5**,
- **RESET_SRC_6**,
- **RESET_SRC_7**,
- **RESET_SRC_8**,
- **RESET_SRC_9**,
- **RESET_SRC_10**,
- **RESET_SRC_11**,
- **RESET_SRC_12**,

- **RESET_SRC_13**,
- **RESET_SRC_14**,
- **RESET_SRC_15**

20.4.2.6 uint32_t ResetCtl_getPCMSource (void)

Indicates the last cause of a power-on reset (POR) due to PCM operation.

Returns

Bitwise OR of any of the following values:

- RESET_LPM35,
- RESET_LPM45

20.4.2.7 uint32_t ResetCtl_getPSSSource (void)

Indicates the last cause of a power-on reset (POR) due to PSS operation. Note that the bits returned from this function may be set in different combinations. When a cold power up occurs, the value of all the values ORed together could be returned as a cold power up causes these conditions.

Returns

Bitwise OR of any of the following values:

- RESET_VCCDET,
- RESET_SVSH_TRIP,
- RESET_BGREF_BAD

20.4.2.8 uint32_t ResetCtl_getSoftResetSource (void)

Retrieves previous soft reset sources

Returns

the bitwise or of previous reset sources. These sources must be cleared using the ResetCtl_clearSoftResetSource function to be cleared. Possible values include:

- **RESET_SRC_0**,
- **RESET_SRC_1**,
- **RESET_SRC_2**,
- **RESET_SRC_3**,
- **RESET_SRC_4**,
- **RESET_SRC_5**,
- **RESET_SRC_6**,
- **RESET_SRC_7**,
- **RESET_SRC_8**,
- **RESET_SRC_9**,
- **RESET_SRC_10**,
- **RESET_SRC_11**,

- **RESET_SRC_12**,
- **RESET_SRC_13**,
- **RESET_SRC_14**,
- **RESET_SRC_15**

20.4.2.9 void ResetCtl_initiateHardReset (void)

Initiates a hard system reset.

Returns
none

20.4.2.10 void ResetCtl_initiateHardResetWithSource (uint32_t *source*)

Initiates a hard system reset with a particular source given. This source is generic and can be assigned by the user.

Parameters

source	- Valid values are one the following values: ■ **RESET_SRC_0**, ■ **RESET_SRC_1**, ■ **RESET_SRC_2**, ■ **RESET_SRC_3**, ■ **RESET_SRC_4**, ■ **RESET_SRC_5**, ■ **RESET_SRC_6**, ■ **RESET_SRC_7**, ■ **RESET_SRC_8**, ■ **RESET_SRC_9**, ■ **RESET_SRC_10**, ■ **RESET_SRC_11**, ■ **RESET_SRC_12**, ■ **RESET_SRC_13**, ■ **RESET_SRC_14**, ■ **RESET_SRC_15**

Returns
none

20.4.2.11 void ResetCtl_initiateSoftReset (void)

Initiates a soft system reset.

Returns
none

20.4.2.12 void ResetCtl_initiateSoftResetWithSource (uint32_t *source*)

Initiates a soft system reset with a particular source given. This source is generic and can be assigned by the user.

Parameters

source	Source of the reset. Valid values are: ■ **RESET_SRC_0**, ■ **RESET_SRC_1**, ■ **RESET_SRC_2**, ■ **RESET_SRC_3**, ■ **RESET_SRC_4**, ■ **RESET_SRC_5**, ■ **RESET_SRC_6**, ■ **RESET_SRC_7**, ■ **RESET_SRC_8**, ■ **RESET_SRC_9**, ■ **RESET_SRC_10**, ■ **RESET_SRC_11**, ■ **RESET_SRC_12**, ■ **RESET_SRC_13**, ■ **RESET_SRC_14**, ■ **RESET_SRC_15**

Returns
none

21 Real Time Clock (RTC_C)

21.1 Module Operation

The Real Time Clock (RTC_C) API provides a set of functions for using the SDK L RTC_C modules. Functions are provided to calibrate the clock, initialize the RTC_C modules in Calendar mode, and setup conditions for, and enable, interrupts for the RTC_C modules.

The RTC_C module provides the ability to keep track of the current time and date in calendar mode.

The RTC_C module generates multiple interrupts. There are 2 interrupts that can be defined in calendar mode, and 1 interrupt in counter mode for counter overflow, as well as an interrupt for each prescaler.

21.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the RTC_C module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the RTC_C module and create a calendar event.

The following is the configuration structure that sets the date:

```
/* Time is November 12th 1955 10:03:00 PM */
const RTC_C_Calendar currentTime =
{
        0x00,
        0x03,
        0x22,
        0x12,
        0x11,
        0x1955
};
```

Next are the actual calls to DriverLib that configure the module:

```
   /* Initializing RTC with current time as described in time in
    * definitions section */
   MAP_RTC_C_initCalendar(&currentTime,  RTC_C_FORMAT_BCD);

   /* Setup Calendar Alarm for 10:04pm (for the flux capacitor) */
   MAP_RTC_C_setCalendarAlarm(0x04,  0x22,  RTC_C_ALARMCONDITION_OFF,
           RTC_C_ALARMCONDITION_OFF);

   /* Specify an interrupt to assert every minute */
   MAP_RTC_C_setCalendarEvent(RTC_C_CALENDAREVENT_MINUTECHANGE);

   /* Enable interrupt for RTC Ready Status,  which asserts when the RTC
    * Calendar registers are ready to read.
    * Also,  enable interrupts for the Calendar alarm and Calendar event. */
   MAP_RTC_C_clearInterruptFlag(
           RTC_C_CLOCK_READ_READY_INTERRUPT | RTC_C_TIME_EVENT_INTERRUPT
                   | RTC_C_CLOCK_ALARM_INTERRUPT);
   MAP_RTC_C_enableInterrupt(
           RTC_C_CLOCK_READ_READY_INTERRUPT | RTC_C_TIME_EVENT_INTERRUPT
                   | RTC_C_CLOCK_ALARM_INTERRUPT);

   /* Start RTC Clock */
   MAP_RTC_C_startClock();
```

21.3 Definitions

Functions

- void RTC_C_clearInterruptFlag (uint_fast8_t interruptFlagMask)
- uint16_t RTC_C_convertBCDToBinary (uint16_t valueToConvert)
- uint16_t RTC_C_convertBinaryToBCD (uint16_t valueToConvert)
- void RTC_C_definePrescaleEvent (uint_fast8_t prescaleSelect, uint_fast8_t prescaleEventDivider)
- void RTC_C_disableInterrupt (uint8_t interruptMask)
- void RTC_C_enableInterrupt (uint8_t interruptMask)
- RTC_C_Calendar RTC_C_getCalendarTime (void)
- uint_fast8_t RTC_C_getEnabledInterruptStatus (void)
- uint_fast8_t RTC_C_getInterruptStatus (void)
- uint_fast8_t RTC_C_getPrescaleValue (uint_fast8_t prescaleSelect)
- void RTC_C_holdClock (void)
- void RTC_C_initCalendar (const RTC_C_Calendar *calendarTime, uint_fast16_t formatSelect)
- void RTC_C_registerInterrupt (void(*intHandler)(void))
- void RTC_C_setCalendarAlarm (uint_fast8_t minutesAlarm, uint_fast8_t hoursAlarm, uint_fast8_t dayOfWeekAlarm, uint_fast8_t dayOfmonthAlarm)
- void RTC_C_setCalendarEvent (uint_fast16_t eventSelect)
- void RTC_C_setCalibrationData (uint_fast8_t offsetDirection, uint_fast8_t offsetValue)
- void RTC_C_setCalibrationFrequency (uint_fast16_t frequencySelect)
- void RTC_C_setPrescaleValue (uint_fast8_t prescaleSelect, uint_fast8_t prescaleCounterValue)
- bool RTC_C_setTemperatureCompensation (uint_fast16_t offsetDirection, uint_fast8_t offsetValue)
- void RTC_C_startClock (void)
- void RTC_C_unregisterInterrupt (void)

21.3.1 Detailed Description

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

21.3.2 Function Documentation

21.3.2.1 void RTC_C_clearInterruptFlag (uint_fast8_t *interruptFlagMask*)

Clears selected RTC interrupt flags.

Parameters

interruptFlagMask	is a bit mask of the interrupt flags to be cleared. Mask Value is the logical OR of any of the following ■ **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by setCalendarEvent() is met. ■ **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

This function clears the RTC interrupt flag is cleared, so that it no longer asserts.

Returns
None

21.3.2.2 uint16_t RTC_C_convertBCDToBinary (uint16_t *valueToConvert*)

Returns the given BCD value in Binary Format

Parameters

valueToConvert	is the raw value in BCD format to convert to Binary.

This function converts BCD values to Binary format.

Returns
The Binary version of the valueToConvert parameter.

21.3.2.3 uint16_t RTC_C_convertBinaryToBCD (uint16_t *valueToConvert*)

Returns the given Binary value in BCD Format

Parameters

valueToConvert	is the raw value in Binary format to convert to BCD.

This function converts Binary values to BCD format.

Returns
The BCD version of the valueToConvert parameter.

21.3.2.4 void RTC_C_definePrescaleEvent (uint_fast8_t *prescaleSelect,* uint_fast8_t *prescaleEventDivider*)

Sets up an interrupt condition for the selected Prescaler.

Parameters

prescaleSelect	is the prescaler to define an interrupt for. Valid values are ■ **RTC_C_PRESCALE_0** ■ **RTC_C_PRESCALE_1**
prescaleEvent-Divider	is a divider to specify when an interrupt can occur based on the clock source of the selected prescaler. (Does not affect timer of the selected prescaler). Valid values are ■ **RTC_C_PSEVENTDIVIDER_2** [Default] ■ **RTC_C_PSEVENTDIVIDER_4** ■ **RTC_C_PSEVENTDIVIDER_8** ■ **RTC_C_PSEVENTDIVIDER_16** ■ **RTC_C_PSEVENTDIVIDER_32** ■ **RTC_C_PSEVENTDIVIDER_64** ■ **RTC_C_PSEVENTDIVIDER_128** ■ **RTC_C_PSEVENTDIVIDER_256**

This function sets the condition for an interrupt to assert based on the individual prescalers.

Returns
None

21.3.2.5 void RTC_C_disableInterrupt (uint8_t *interruptMask*)

Disables selected RTC interrupt sources.

Parameters

interruptMask	is a bit mask of the interrupts to disable. Mask Value is the logical OR of any of the following ■ **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by setCalendarEvent() is met. ■ **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

This function disables the selected RTC interrupt source. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns
None

21.3.2.6 void RTC_C_enableInterrupt (uint8_t *interruptMask*)

Enables selected RTC interrupt sources.

Parameters

interruptMask	is a bit mask of the interrupts to enable. Mask Value is the logical OR of any of the following ■ **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by setCalendarEvent() is met. ■ **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

This function enables the selected RTC interrupt source. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns
None

21.3.2.7 RTC_C_Calendar RTC_C_getCalendarTime (void)

Returns the Calendar Time stored in the Calendar registers of the RTC.

This function returns the current Calendar time in the form of a Calendar structure.

Returns
A Calendar structure containing the current time.

21.3.2.8 uint_fast8_t RTC_C_getEnabledInterruptStatus (void)

Returns the status of the interrupts flags masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Returns
A bit mask of the selected interrupt flag's status. Mask Value is the logical OR of any of the following
- **RTC_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by setCalendarEvent() is met.
- **RTC_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.

- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

References RTC_C_getInterruptStatus().

21.3.2.9 uint_fast8_t RTC_C_getInterruptStatus (void)

Returns the status of the interrupts flags.

Returns

A bit mask of the selected interrupt flag's status. Mask Value is the logical OR of any of the following

- **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by setCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Referenced by RTC_C_getEnabledInterruptStatus().

21.3.2.10 uint_fast8_t RTC_C_getPrescaleValue (uint_fast8_t *prescaleSelect*)

Returns the selected Prescaler value.

Parameters

prescaleSelect	is the prescaler to obtain the value of. Valid values are ■ **RTC_C_PRESCALE_0** ■ **RTC_C_PRESCALE_1**

This function returns the value of the selected prescale counter register. The counter should be held before reading. If in counter mode, the individual prescaler can be held, while in Calendar mode the whole RTC must be held.

Returns

The value of the specified Prescaler count register

21.3.2.11 void RTC_C_holdClock (void)

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Returns
None

Referenced by PCM_gotoLPM4().

21.3.2.12 void RTC_C_initCalendar (const RTC_C_Calendar * *calendarTime,* uint_fast16_t *formatSelect*)

Initializes the settings to operate the RTC in Calendar mode.

Parameters

calendarTime	is the structure containing the values for the Calendar to be initialized to. Valid values should be of type Calendar and should contain the following members and corresponding values: ■ **seconds** between 0-59 ■ **minutes** between 0-59 ■ **hours** between 0-23 ■ **dayOfWeek** between 0-6 ■ **dayOfmonth** between 1-31 ■ **month** between 1-12 ■ **year** between 0-4095

Note
Values beyond the ones specified may result in erratic behavior.

Parameters

formatSelect	is the format for the Calendar registers to use. Valid values are ■ **RTC_C_FORMAT_BINARY** [Default] ■ **RTC_C_FORMAT_BCD**

This function initializes the Calendar mode of the RTC module.

Returns
None

21.3.2.13 void RTC_C_registerInterrupt (void(*)(void) *intHandler*)

Registers an interrupt handler for the RTC interrupt.

Parameters

intHandler	is a pointer to the function to be called when the RTC interrupt occurs.

This function registers the handler to be called when a RTC interrupt occurs. This function enables the global interrupt in the interrupt controller; specific AES interrupts must be enabled via RTC_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via RTC_clearInterruptFlag().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

21.3.2.14 void RTC_C_setCalendarAlarm (uint_fast8_t *minutesAlarm,* uint_fast8_t *hoursAlarm,* uint_fast8_t *dayOfWeekAlarm,* uint_fast8_t *dayOfmonthAlarm*)

Sets and Enables the desired Calendar Alarm settings.

Parameters

minutesAlarm	is the alarm condition for the minutes. Valid values are ■ An integer between 0-59, OR ■ **RTC_C_ALARMCONDITION_OFF** [Default]
hoursAlarm	is the alarm condition for the hours. Valid values are ■ An integer between 0-24, OR ■ **RTC_C_ALARMCONDITION_OFF** [Default]
dayOfWeekAlarm	is the alarm condition for the day of week. Valid values are ■ An integer between 0-6, OR ■ **RTC_C_ALARMCONDITION_OFF** [Default]
dayOfmonthAlarm	is the alarm condition for the day of the month. Valid values are ■ An integer between 0-31, OR ■ **RTC_C_ALARMCONDITION_OFF** [Default]

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Returns
None

21.3.2.15 void RTC_C_setCalendarEvent (uint_fast16_t *eventSelect*)

Sets a single specified Calendar interrupt condition.

Parameters

eventSelect	is the condition selected. Valid values are ■ **RTC_C_CALENDAREVENT_MINUTECHANGE** - assert interrupt on every minute ■ **RTC_C_CALENDAREVENT_HOURCHANGE** - assert interrupt on every hour ■ **RTC_C_CALENDAREVENT_NOON** - assert interrupt when hour is 12 ■ **RTC_C_CALENDAREVENT_MIDNIGHT** - assert interrupt when hour is 0

This function sets a specified event to assert the RTCTEVIFG interrupt. This interrupt is independent from the Calendar alarm interrupt.

Returns
None

21.3.2.16 void RTC_C_setCalibrationData (uint_fast8_t *offsetDirection,* uint_fast8_t *offsetValue*)

Sets the specified calibration for the RTC.

Parameters

offsetDirection	is the direction that the calibration offset will go. Valid values are ■ **RTC_C_CALIBRATION_DOWN1PPM** - calibrate at steps of -1 ■ **RTC_C_CALIBRATION_UP1PPM** - calibrat at steps of +1
offsetValue	is the value that the offset will be a factor of; a valid value is any integer from 1-240.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +1-ppm or -1-ppm, and the offsetValue should be from 1-240 and is multiplied by the direction setting (i.e. +1-ppm $*$ 8 (offsetValue) = +8-ppm).

Returns
None

21.3.2.17 void RTC_C_setCalibrationFrequency (uint_fast16_t *frequencySelect*)

Allows and Sets the frequency output to RTCLK pin for calibration measurement.

Parameters

frequencySelect	is the frequency output to RTCLK. Valid values are ■ **RTC_C_CALIBRATIONFREQ_OFF** - turn off calibration output [Default] ■ **RTC_C_CALIBRATIONFREQ_512HZ** - output signal at 512Hz for calibration ■ **RTC_C_CALIBRATIONFREQ_256HZ** - output signal at 256Hz for calibration ■ **RTC_C_CALIBRATIONFREQ_1HZ** - output signal at 1Hz for calibration

This function sets a frequency to measure at the RTCLK output pin. After testing the set frequency, the calibration could be set accordingly.

Returns
None

21.3.2.18 void RTC_C_setPrescaleValue (uint_fast8_t *prescaleSelect,* uint_fast8_t *prescaleCounterValue*)

Sets the selected Prescaler value.

Parameters

prescaleSelect	is the prescaler to set the value for. Valid values are ■ **RTC_C_PRESCALE_0** ■ **RTC_C_PRESCALE_1**
prescaleCounterValue	is the specified value to set the prescaler to; a valid value is any integer from 0-255.

This function sets the prescale counter value. Before setting the prescale counter, it should be held.

Returns
None

21.3.2.19 bool RTC_C_setTemperatureCompensation (uint_fast16_t *offsetDirection,* uint_fast8_t *offsetValue*)

Sets the specified temperature compensation for the RTC.

Parameters

offsetDirection	is the direction that the calibration offset will go. Valid values are ■ **RTC_C_COMPENSATION_DOWN1PPM** - calibrate at steps of -1 ■ **RTC_C_COMPENSATION_UP1PPM** - calibrate at steps of +1
offsetValue	is the value that the offset will be a factor of; a value is any integer from 1-240.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +1-ppm or -1-ppm, and the offsetValue should be from 1-240 and is multiplied by the direction setting (i.e. +1-ppm $*$ 8 (offsetValue) = +8-ppm).

Returns
true if calibration was set, false if it could not be set

21.3.2.20 void RTC_C_startClock (void)

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Returns
None

21.3.2.21 void RTC_C_unregisterInterrupt (void)

Unregisters the interrupt handler for the RTC interrupt

This function unregisters the handler to be called when RTC interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

22 Serial Peripheral Interface (SPI)

22.1 Module Operation

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame. Note for simplicity, the module name EUSCI_A and EUSCI_B have been omitted from the API names.

This library provides the API for handling a 3-wire SPI communication

The SPI module can be configured as either a master or a slave device.

The SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock.

22.2 Basic Operation Modes

To use the module as a master, the user must call SPI_masterInit() to configure the SPI Master. This is followed by enabling the SPI module using SPI_enable(). The interrupts are then enabled (if needed). It is recommended to enable the SPI module before enabling the interrupts. A data transmit is then initiated using SPI_transmitData and then when the receive flag is set, the received data is read using SPI_receiveData and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using SPI_initSlave and this is followed by enabling the module using SPI_enableModule . Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using SPI_transmitData and this is followed by a data reception by SPI_receiveData .

22.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the SPI module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the SPI module in 3wire master mode.

In the code snippet below, the configuration settings for SPI in 3wire master mode can be seen:

```
/* SPI Master Configuration Parameter */
const eUSCI_SPI_MasterConfig spiMasterConfig =
{
        EUSCI_B_SPI_CLOCKSOURCE_SMCLK,              // SMCLK Clock Source
        3000000,                                    // SMCLK = DCO = 3MHZ
        500000,                                     // SPICLK = 500khz
        EUSCI_B_SPI_MSB_FIRST,                      // MSB First
        EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,     // Phase
        EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH,  // High polarity
        EUSCI_B_SPI_3PIN                           // 3Wire SPI Mode
};
```

In this code snippet, the SPI module is configured and enabled for 3wire SPI operation using the DriverLib APIs:

```
    /* Selecting P1.5 P1.6 and P1.7 in SPI mode */
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
            GPIO_PIN5 | GPIO_PIN6 | GPIO_PIN7,  GPIO_PRIMARY_MODULE_FUNCTION);

    /* Configuring SPI in 3wire master mode */
    SPI_initMaster(EUSCI_B0_BASE,  &spiMasterConfig);

    /* Enable SPI module */
    SPI_enableModule(EUSCI_B0_BASE);

    /* Enabling interrupts */
    SPI_enableInterrupt(EUSCI_B0_BASE,  EUSCI_B_SPI_RECEIVE_INTERRUPT);
    Interrupt_enableInterrupt(INT_EUSCIB0);
    Interrupt_enableSleepOnIsrExit();
```

22.4 Definitions

Data Structures

- struct _eUSCI_SPI_MasterConfig
- struct _eUSCI_SPI_SlaveConfig

Functions

- void EUSCI_A_SPI_changeClockPhasePolarity (uint32_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
- void EUSCI_A_SPI_clearInterruptFlag (uint32_t baseAddress, uint8_t mask)
- void EUSCI_A_SPI_disable (uint32_t baseAddress)
- void EUSCI_A_SPI_disableInterrupt (uint32_t baseAddress, uint8_t mask)
- void EUSCI_A_SPI_enable (uint32_t baseAddress)
- void EUSCI_A_SPI_enableInterrupt (uint32_t baseAddress, uint8_t mask)
- uint8_t EUSCI_A_SPI_getInterruptStatus (uint32_t baseAddress, uint8_t mask)
- uint32_t EUSCI_A_SPI_getReceiveBufferAddressForDMA (uint32_t baseAddress)
- uint32_t EUSCI_A_SPI_getTransmitBufferAddressForDMA (uint32_t baseAddress)
- bool EUSCI_A_SPI_isBusy (uint32_t baseAddress)
- void EUSCI_A_SPI_masterChangeClock (uint32_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- uint8_t EUSCI_A_SPI_receiveData (uint32_t baseAddress)
- void EUSCI_A_SPI_select4PinFunctionality (uint32_t baseAddress, uint8_t select4PinFunctionality)
- bool EUSCI_A_SPI_slaveInit (uint32_t baseAddress, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- void EUSCI_A_SPI_transmitData (uint32_t baseAddress, uint8_t transmitData)
- void EUSCI_B_SPI_changeClockPhasePolarity (uint32_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
- void EUSCI_B_SPI_clearInterruptFlag (uint32_t baseAddress, uint8_t mask)
- void EUSCI_B_SPI_disable (uint32_t baseAddress)
- void EUSCI_B_SPI_disableInterrupt (uint32_t baseAddress, uint8_t mask)
- void EUSCI_B_SPI_enable (uint32_t baseAddress)
- void EUSCI_B_SPI_enableInterrupt (uint32_t baseAddress, uint8_t mask)
- uint8_t EUSCI_B_SPI_getInterruptStatus (uint32_t baseAddress, uint8_t mask)
- uint32_t EUSCI_B_SPI_getReceiveBufferAddressForDMA (uint32_t baseAddress)
- uint32_t EUSCI_B_SPI_getTransmitBufferAddressForDMA (uint32_t baseAddress)
- bool EUSCI_B_SPI_isBusy (uint32_t baseAddress)
- void EUSCI_B_SPI_masterChangeClock (uint32_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- uint8_t EUSCI_B_SPI_receiveData (uint32_t baseAddress)
- void EUSCI_B_SPI_select4PinFunctionality (uint32_t baseAddress, uint8_t select4PinFunctionality)
- bool EUSCI_B_SPI_slaveInit (uint32_t baseAddress, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- void EUSCI_B_SPI_transmitData (uint32_t baseAddress, uint8_t transmitData)
- void SPI_changeClockPhasePolarity (uint32_t moduleInstance, uint_fast16_t clockPhase, uint_fast16_t clockPolarity)
- void SPI_changeMasterClock (uint32_t moduleInstance, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- void SPI_clearInterruptFlag (uint32_t moduleInstance, uint_fast8_t mask)
- void SPI_disableInterrupt (uint32_t moduleInstance, uint_fast8_t mask)
- void SPI_disableModule (uint32_t moduleInstance)

- void SPI_enableInterrupt (uint32_t moduleInstance, uint_fast8_t mask)
- void SPI_enableModule (uint32_t moduleInstance)
- uint_fast8_t SPI_getEnabledInterruptStatus (uint32_t moduleInstance)
- uint_fast8_t SPI_getInterruptStatus (uint32_t moduleInstance, uint16_t mask)
- uint32_t SPI_getReceiveBufferAddressForDMA (uint32_t moduleInstance)
- uint32_t SPI_getTransmitBufferAddressForDMA (uint32_t moduleInstance)
- bool SPI_initMaster (uint32_t moduleInstance, const eUSCI_SPI_MasterConfig *config)
- bool SPI_initSlave (uint32_t moduleInstance, const eUSCI_SPI_SlaveConfig *config)
- uint_fast8_t SPI_isBusy (uint32_t moduleInstance)
- uint8_t SPI_receiveData (uint32_t moduleInstance)
- void SPI_registerInterrupt (uint32_t moduleInstance, void(*intHandler)(void))
- void SPI_selectFourPinFunctionality (uint32_t moduleInstance, uint_fast8_t select4PinFunctionality)
- void SPI_transmitData (uint32_t moduleInstance, uint_fast8_t transmitData)
- void SPI_unregisterInterrupt (uint32_t moduleInstance)

22.4.1 Detailed Description

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

22.4.2 Data Structure Documentation

22.4.2.1 struct _eUSCI_SPI_MasterConfig

Type definition for _eUSCI_SPI_MasterConfig structure.

ypedef eUSCI_SPI_MasterConfig

Configuration structure for master mode in the **SPI** module. See SPI_initMaster for parameter documentation.

22.4.2.2 struct _eUSCI_SPI_SlaveConfig

Type definition for _eUSCI_SPI_SlaveConfig structure.

ypedef eUSCI_SPI_SlaveConfig

Configuration structure for slave mode in the **SPI** module. See SPI_initSlave for parameter documentation.

22.4.3 Function Documentation

22.4.3.1 void EUSCI_A_SPI_changeClockPhasePolarity (uint32_t *baseAddress,* uint16_t *clockPhase,* uint16_t *clockPolarity*)

Changes the SPI colock phase and polarity. At the end of this function call, SPI module is left enabled.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **EUSCI_A_CTLW0_CKPL**, **EUSCI_A_CTLW0_CKPH** and **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_changeClockPhasePolarity().

22.4.3.2 void EUSCI_A_SPI_clearInterruptFlag (uint32_t *baseAddress,* uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns
None

Referenced by SPI_clearInterruptFlag().

22.4.3.3 void EUSCI_A_SPI_disable (uint32_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_disableModule().

22.4.3.4 void EUSCI_A_SPI_disableInterrupt (uint32_t *baseAddress,* uint8_t *mask*)

Disables individual SPI interrupt sources.

Disables 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.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIE** register.

Returns
None

Referenced by SPI_disableInterrupt().

22.4.3.5 void EUSCI_A_SPI_enable (uint32_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_enableModule().

22.4.3.6 void EUSCI_A_SPI_enableInterrupt (uint32_t *baseAddress,* uint8_t *mask*)

Enables individual SPI interrupt sources.

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. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register and bits of **UCAxIE** register.

Returns
None

Referenced by SPI_enableInterrupt().

22.4.3.7 uint8_t EUSCI_A_SPI_getInterruptStatus (uint32_t *baseAddress,* uint8_t *mask*)

Gets the current SPI interrupt status.

This returns the interrupt status for the SPI module based on which flag is passed.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Returns

Logical OR of any of the following:
- **EUSCI_A_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_A_SPI_RECEIVE_INTERRUPT**

indicating the status of the masked interrupts

Referenced by SPI_getInterruptStatus().

22.4.3.8 uint32_t EUSCI_A_SPI_getReceiveBufferAddressForDMA (uint32_t *baseAddress*)

Returns the address of the RX Buffer of the SPI for the DMA module.

Returns the address of the SPI RX Buffer. This can be used in conjunction with the DMA to store the received data directly to memory.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

the address of the RX Buffer

Referenced by SPI_getReceiveBufferAddressForDMA().

22.4.3.9 uint32_t EUSCI_A_SPI_getTransmitBufferAddressForDMA (uint32_t *baseAddress*)

Returns the address of the TX Buffer of the SPI for the DMA module.

Returns the address of the SPI TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

the address of the TX Buffer

Referenced by SPI_getTransmitBufferAddressForDMA().

22.4.3.10 bool EUSCI_A_SPI_isBusy (uint32_t *baseAddress*)

Indicates whether or not the SPI bus is busy.

This function returns an indication of whether or not the SPI bus is busy.This function checks the status of the bus via UCBBUSY bit

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns
true if busy, false otherwise

Referenced by SPI_isBusy().

22.4.3.11 void EUSCI_A_SPI_masterChangeClock (uint32_t *baseAddress,* uint32_t *clockSourceFrequency,* uint32_t *desiredSpiClock*)

Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
clockSourceFrequency	is the frequency of the slected clock source
desiredSpiClock	is the desired clock rate for SPI communication

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_changeMasterClock().

22.4.3.12 uint8_t EUSCI_A_SPI_receiveData (uint32_t *baseAddress*)

Receives a byte that has been sent to the SPI Module.

This function reads a byte of data from the SPI receive data Register.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns
Returns the byte received from by the SPI module, cast as an uint8_t.

Referenced by SPI_receiveData().

22.4.3.13 void EUSCI_A_SPI_select4PinFunctionality (uint32_t *baseAddress,* uint8_t *select4PinFunctionality*)

Selects 4Pin Functionality.

This function should be invoked only in 4-wire mode. Invoking this function has no effect in 3-wire mode.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
select4PinFunctionality	selects 4 pin functionality Valid values are: ■ **EUSCI_A_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_A_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

Modified bits are **UCSTEM** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_selectFourPinFunctionality().

22.4.3.14 bool EUSCI_A_SPI_slaveInit (uint32_t *baseAddress,* uint16_t *msbFirst,* uint16_t *clockPhase,* uint16_t *clockPolarity,* uint16_t *spiMode*)

Initializes the SPI Slave block.

Upon successful initialization of the SPI slave block, this function will have initailized the slave block, but the SPI Slave block still remains disabled and must be enabled with EUSCI_A_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_A_SPI Slave module.
msbFirst	controls the direction of the receive and transmit shift register. Valid values are: ■ **EUSCI_A_SPI_MSB_FIRST** ■ **EUSCI_A_SPI_LSB_FIRST** [Default]

clockPhase	is clock phase select. Valid values are: ■ **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]
spiMode	is SPI mode select Valid values are: ■ **EUSCI_A_SPI_3PIN** ■ **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_HIGH** ■ **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Modified bits are **EUSCI_A_CTLW0_MSB**, **EUSCI_A_CTLW0_MST**, **EUSCI_A_CTLW0_SEVENBIT**, **EUSCI_A_CTLW0_CKPL**, **EUSCI_A_CTLW0_CKPH**, **UCMODE** and **UCSWRST** of **UCAxCTLW0** register.

Returns
true

22.4.3.15 void EUSCI_A_SPI_transmitData (uint32_t *baseAddress,* uint8_t *transmitData*)

Transmits a byte from the SPI Module.

This function will place the supplied data into SPI trasmit data register to start transmission.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
transmitData	data to be transmitted from the SPI module

Returns
None

Referenced by SPI_transmitData().

22.4.3.16 void EUSCI_B_SPI_changeClockPhasePolarity (uint32_t *baseAddress,* uint16_t *clockPhase,* uint16_t *clockPolarity*)

Changes the SPI colock phase and polarity. At the end of this function call, SPI module is left enabled.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **EUSCI_A_CTLW0_CKPL**, **EUSCI_A_CTLW0_CKPH** and **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_changeClockPhasePolarity().

22.4.3.17 void EUSCI_B_SPI_clearInterruptFlag (uint32_t *baseAddress,* uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
mask	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: ■ **EUSCI_B_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns
None

Referenced by SPI_clearInterruptFlag().

22.4.3.18 void EUSCI_B_SPI_disable (uint32_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Modified bits are **UCSWRST** of **UCBxCTLW0** register.

Returns
None

Referenced by SPI_disableModule().

22.4.3.19 void EUSCI_B_SPI_disableInterrupt (uint32_t *baseAddress,* uint8_t *mask*)

Disables individual SPI interrupt sources.

Disables 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.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
mask	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **EUSCI_B_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIE** register.

Returns
None

Referenced by SPI_disableInterrupt().

22.4.3.20 void EUSCI_B_SPI_enable (uint32_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Modified bits are **UCSWRST** of **UCBxCTLW0** register.

Returns
None

Referenced by SPI_enableModule().

22.4.3.21 void EUSCI_B_SPI_enableInterrupt (uint32_t *baseAddress,* uint8_t *mask*)

Enables individual SPI interrupt sources.

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. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
mask	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **EUSCI_B_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register and bits of **UCAxIE** register.

Returns
None

Referenced by SPI_enableInterrupt().

22.4.3.22 uint8_t EUSCI_B_SPI_getInterruptStatus (uint32_t *baseAddress,* uint8_t *mask*)

Gets the current SPI interrupt status.

This returns the interrupt status for the SPI module based on which flag is passed.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_B_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Returns
Logical OR of any of the following:
- **EUSCI_B_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_B_SPI_RECEIVE_INTERRUPT**

indicating the status of the masked interrupts

Referenced by SPI_getInterruptStatus().

22.4.3.23 uint32_t EUSCI_B_SPI_getReceiveBufferAddressForDMA (uint32_t *baseAddress*)

Returns the address of the RX Buffer of the SPI for the DMA module.

Returns the address of the SPI RX Buffer. This can be used in conjunction with the DMA to store the received data directly to memory.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns
the address of the RX Buffer

Referenced by SPI_getReceiveBufferAddressForDMA().

22.4.3.24 uint32_t EUSCI_B_SPI_getTransmitBufferAddressForDMA (uint32_t *baseAddress*)

Returns the address of the TX Buffer of the SPI for the DMA module.

Returns the address of the SPI TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns
the address of the TX Buffer

Referenced by SPI_getTransmitBufferAddressForDMA().

22.4.3.25 bool EUSCI_B_SPI_isBusy (uint32_t *baseAddress*)

Indicates whether or not the SPI bus is busy.

This function returns an indication of whether or not the SPI bus is busy.This function checks the status of the bus via UCBBUSY bit

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns
true if busy, false otherwise

Referenced by SPI_isBusy().

22.4.3.26 void EUSCI_B_SPI_masterChangeClock (uint32_t *baseAddress,* uint32_t *clockSourceFrequency,* uint32_t *desiredSpiClock*)

Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
clockSourceFrequency	is the frequency of the slected clock source
desiredSpiClock	is the desired clock rate for SPI communication

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_changeMasterClock().

22.4.3.27 uint8_t EUSCI_B_SPI_receiveData (uint32_t *baseAddress*)

Receives a byte that has been sent to the SPI Module.

This function reads a byte of data from the SPI receive data Register.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns
Returns the byte received from by the SPI module, cast as an uint8_t.

Referenced by SPI_receiveData().

22.4.3.28 void EUSCI_B_SPI_select4PinFunctionality (uint32_t *baseAddress,* uint8_t *select4PinFunctionality*)

Selects 4Pin Functionality.

This function should be invoked only in 4-wire mode. Invoking this function has no effect in 3-wire mode.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
select4PinFunctionality	selects 4 pin functionality Valid values are: ■ **EUSCI_B_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_B_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

Modified bits are **UCSTEM** of **UCAxCTLW0** register.

Returns
None

Referenced by SPI_selectFourPinFunctionality().

22.4.3.29 bool EUSCI_B_SPI_slaveInit (uint32_t *baseAddress,* uint16_t *msbFirst,* uint16_t *clockPhase,* uint16_t *clockPolarity,* uint16_t *spiMode*)

Initializes the SPI Slave block.

Upon successful initialization of the SPI slave block, this function will have initailized the slave block, but the SPI Slave block still remains disabled and must be enabled with EUSCI_B_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_B_SPI Slave module.
msbFirst	controls the direction of the receive and transmit shift register. Valid values are: ■ **EUSCI_B_SPI_MSB_FIRST** ■ **EUSCI_B_SPI_LSB_FIRST** [Default]
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]
spiMode	is SPI mode select Valid values are: ■ **EUSCI_B_SPI_3PIN** ■ **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_HIGH** ■ **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Modified bits are **EUSCI_A_CTLW0_MSB**, **EUSCI_A_CTLW0_MST**, **EUSCI_A_CTLW0_SEVENBIT**, **EUSCI_A_CTLW0_CKPL**, **EUSCI_A_CTLW0_CKPH**, **UCMODE** and **UCSWRST** of **UCAxCTLW0** register.

Returns
true

22.4.3.30 void EUSCI_B_SPI_transmitData (uint32_t *baseAddress,* uint8_t *transmitData*)

Transmits a byte from the SPI Module.

This function will place the supplied data into SPI trasmit data register to start transmission.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
transmitData	data to be transmitted from the SPI module

Returns
None

Referenced by SPI_transmitData().

22.4.3.31 void SPI_changeClockPhasePolarity (uint32_t *moduleInstance,* uint_fast16_t *clockPhase,* uint_fast16_t *clockPolarity*)

Changes the SPI clock phase and polarity.At the end of this function call, SPI module is left enabled.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default Value] ■ **EUSCI_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select. Valid values are: ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default Value]

Modified bits are **UCSWRST**, **UCCKPH**, **UCCKPL**, **UCSWRST** bits of **UCAxCTLW0**

Returns
None

References EUSCI_A_SPI_changeClockPhasePolarity(), and EUSCI_B_SPI_changeClockPhasePolarity().

22.4.3.32 void SPI_changeMasterClock (uint32_t *moduleInstance,* uint32_t *clockSourceFrequency,* uint32_t *desiredSpiClock*)

Initializes the SPI Master clock.At the end of this function call, SPI module is left enabled.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
clockSourceFrequency	is the frequency of the selected clock source
desiredSpiClock	is the desired clock rate for SPI communication.

Modified bits are **UCSWRST** bit of **UCAxCTLW0** register and **UCAxBRW** register

Returns
None

References EUSCI_A_SPI_masterChangeClock(), and EUSCI_B_SPI_masterChangeClock().

22.4.3.33 void SPI_clearInterruptFlag (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Clears the selected SPI interrupt status flag.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
mask	is the masked interrupt flag to be cleared.

The mask parameter is the logical OR of any of the following:

- **EUSCI_SPI_RECEIVE_INTERRUPT** -Receive interrupt
- **EUSCI_SPI_TRANSMIT_INTERRUPT** - Transmit interrupt Modified registers are **UCAxIFG**.

Returns
None

References EUSCI_A_SPI_clearInterruptFlag(), and EUSCI_B_SPI_clearInterruptFlag().

22.4.3.34 void SPI_disableInterrupt (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Disables individual SPI interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
mask	is the bit mask of the interrupt sources to be disabled.

Disables 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 mask parameter is the logical OR of any of the following:

- **EUSCI_SPI_RECEIVE_INTERRUPT** Receive interrupt
- **EUSCI_SPI_TRANSMIT_INTERRUPT** Transmit interrupt

Modified register is **UCAxIE**

Returns
None.

References EUSCI_A_SPI_disableInterrupt(), and EUSCI_B_SPI_disableInterrupt().

22.4.3.35 void SPI_disableModule (uint32_t *moduleInstance*)

Disables the SPI block.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

This will disable operation of the SPI block.

Modified bits are **UCSWRST** bit of **UCAxCTLW0** register.

Returns
None.

References EUSCI_A_SPI_disable(), and EUSCI_B_SPI_disable().

22.4.3.36 void SPI_enableInterrupt (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Enables individual SPI interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
mask	is the bit mask of the interrupt sources to be enabled.

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 mask parameter is the logical OR of any of the following:

- **EUSCI_SPI_RECEIVE_INTERRUPT** Receive interrupt
- **EUSCI_SPI_TRANSMIT_INTERRUPT** Transmit interrupt

Modified registers are **UCAxIFG** and **UCAxIE**

Returns
None.

References EUSCI_A_SPI_enableInterrupt(), and EUSCI_B_SPI_enableInterrupt().

22.4.3.37 void SPI_enableModule (uint32_t *moduleInstance*)

Enables the SPI block.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

This will enable operation of the SPI block. Modified bits are **UCSWRST** bit of **UCAxCTLW0** register.

Returns
None.

References EUSCI_A_SPI_enable(), and EUSCI_B_SPI_enable().

22.4.3.38 uint_fast8_t SPI_getEnabledInterruptStatus (uint32_t *moduleInstance*)

Gets the current SPI interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

Modified registers are **UCAxIFG**.

Returns
The current interrupt status as the mask of the set flags Mask parameter can be either any of the following selection:
- **EUSCI_SPI_RECEIVE_INTERRUPT** -Receive interrupt
- **EUSCI_SPI_TRANSMIT_INTERRUPT** - Transmit interrupt

References SPI_getInterruptStatus().

22.4.3.39 uint_fast8_t SPI_getInterruptStatus (uint32_t *moduleInstance,* uint16_t *mask*)

Gets the current SPI interrupt status.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
mask	Mask of interrupt to filter. This can include: ■ **EUSCI_SPI_RECEIVE_INTERRUPT** -Receive interrupt ■ **EUSCI_SPI_TRANSMIT_INTERRUPT** - Transmit interrupt

Modified registers are **UCAxIFG**.

Returns

The current interrupt status as the mask of the set flags Mask parameter can be either any of the following selection:

- **EUSCI_SPI_RECEIVE_INTERRUPT** -Receive interrupt
- **EUSCI_SPI_TRANSMIT_INTERRUPT** - Transmit interrupt

References EUSCI_A_SPI_getInterruptStatus(), and EUSCI_B_SPI_getInterruptStatus().

Referenced by SPI_getEnabledInterruptStatus().

22.4.3.40 uint32_t SPI_getReceiveBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the RX Buffer of the SPI for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

Returns the address of the SPI RX Buffer. This can be used in conjunction with the DMA to store the received data directly to memory.

Returns
NONE

References EUSCI_A_SPI_getReceiveBufferAddressForDMA(), and EUSCI_B_SPI_getReceiveBufferAddressForDMA().

22.4.3.41 uint32_t SPI_getTransmitBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the TX Buffer of the SPI for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

Returns the address of the SPI TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Returns
NONE

References EUSCI_A_SPI_getTransmitBufferAddressForDMA(), and EUSCI_B_SPI_getTransmitBufferAddressForDMA().

22.4.3.42 bool SPI_initMaster (uint32_t *moduleInstance,* const **eUSCI_SPI_MasterConfig** ∗ *config*)

Initializes the SPI Master block.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
config	Configuration structure for SPI master mode

Configuration options for eUSCI_SPI_MasterConfig structure.

Parameters

selectClock-Source	selects clock source. Valid values are ■ **EUSCI_SPI_CLOCKSOURCE_ACLK** ■ **EUSCI_SPI_CLOCKSOURCE_SMCLK**
clockSourceFre-quency	is the frequency of the selected clock source
desiredSpiClock	is the desired clock rate for SPI communication
msbFirst	controls the direction of the receive and transmit shift register. Valid values are ■ **EUSCI_SPI_MSB_FIRST** ■ **EUSCI_SPI_LSB_FIRST** [Default Value]

clockPhase	is clock phase select. Valid values are ■ **EUSCI_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default Value] ■ **EUSCI_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select. Valid values are ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default Value]
spiMode	is SPI mode select. Valid values are ■ **EUSCI_SPI_3PIN** [Default Value] ■ **EUSCI_SPI_4PIN_UCxSTE_ACTIVE_HIGH** ■ **EUSCI_SPI_4PIN_UCxSTE_ACTIVE_LOW** Upon successful initialization of the SPI master block, this function will have set the bus speed for the master, but the SPI Master block still remains disabled and must be enabled with SPI_enableModule()

Modified bits are **UCCKPH**, **UCCKPL**, **UC7BIT**, **UCMSB**, **UCSSELx**, **UCSWRST** bits of **UCAxCTLW0** register

Returns
true

22.4.3.43 bool SPI_initSlave (uint32_t *moduleInstance,* const **eUSCI_SPI_SlaveConfig** * *config*)

Initializes the SPI Slave block.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

config	Configuration structure for SPI slave mode

Configuration options for eUSCI_SPI_SlaveConfig structure.

Parameters

msbFirst	controls the direction of the receive and transmit shift register. Valid values are ■ **EUSCI_SPI_MSB_FIRST** ■ **EUSCI_SPI_LSB_FIRST** [Default Value]
clockPhase	is clock phase select. Valid values are ■ **EUSCI_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default Value] ■ **EUSCI_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select. Valid values are ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default Value]
spiMode	is SPI mode select. Valid values are ■ **EUSCI_SPI_3PIN** [Default Value] ■ **EUSCI_SPI_4PIN_UCxSTE_ACTIVE_HIGH** ■ **EUSCI_SPI_4PIN_UCxSTE_ACTIVE_LOW** Upon successful initialization of the SPI slave block, this function will have initialized the slave block, but the SPI Slave block still remains disabled and must be enabled with SPI_enableModule()

Modified bits are **UCMSB**, **UC7BIT**, **UCMST**, **UCCKPL**, **UCCKPH**, **UCMODE**, **UCSWRST** bits of **UCAxCTLW0**

Returns
true

22.4.3.44 uint_fast8_t SPI_isBusy (uint32_t *moduleInstance*)

Indicates whether or not the SPI bus is busy.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

This function returns an indication of whether or not the SPI bus is busy.This function checks the status of the bus via UCBBUSY bit

Returns
EUSCI_SPI_BUSY if the SPI module transmitting or receiving is busy; otherwise, returns EUSCI_SPI_NOT_BUSY.

References EUSCI_A_SPI_isBusy(), and EUSCI_B_SPI_isBusy().

22.4.3.45 uint8_t SPI_receiveData (uint32_t *moduleInstance*)

Receives a byte that has been sent to the SPI Module.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

This function reads a byte of data from the SPI receive data Register.

Returns
Returns the byte received from by the SPI module, cast as an uint8_t.

References EUSCI_A_SPI_receiveData(), and EUSCI_B_SPI_receiveData().

22.4.3.46 void SPI_registerInterrupt (uint32_t *moduleInstance,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the timer capture compare interrupt.

Parameters

moduleInstance	is the instance of the eUSCI (SPI) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE** It is important to note that for eUSCI modules, only "B" modules such as EUSCI_B0 can be used. "A" modules such as EUSCI_A0 do not support the I2C mode.
intHandler	is a pointer to the function to be called when the timer capture compare interrupt occurs.

This function registers the handler to be called when a timer interrupt occurs. This function enables the global interrupt in the interrupt controller; specific SPI interrupts must be enabled via SPI_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via SPI_clearInterruptFlag().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

22.4.3.47 void SPI_selectFourPinFunctionality (uint32_t *moduleInstance,* uint_fast8_t *select4PinFunctionality*)

Selects 4Pin Functionality

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**
select4PinFunctionality	selects Clock source. Valid values are ■ **EUSCI_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE** This function should be invoked only in 4-wire mode. Invoking this function has no effect in 3-wire mode.

Modified bits are **UCSTEM** bit of **UCAxCTLW0** register

Returns
true

References EUSCI_A_SPI_select4PinFunctionality(), and EUSCI_B_SPI_select4PinFunctionality().

22.4.3.48 void SPI_transmitData (uint32_t *moduleInstance,* uint_fast8_t *transmitData*)

Transmits a byte from the SPI Module.

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

transmitData	data to be transmitted from the SPI module

This function will place the supplied data into SPI transmit data register to start transmission

Modified register is **UCAxTXBUF**

Returns
None.

References EUSCI_A_SPI_transmitData(), and EUSCI_B_SPI_transmitData().

22.4.3.49 void SPI_unregisterInterrupt (uint32_t *moduleInstance*)

Unregisters the interrupt handler for the timer

Parameters

moduleInstance	is the instance of the eUSCI A/B module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** ■ **EUSCI_B0_BASE** ■ **EUSCI_B1_BASE** ■ **EUSCI_B2_BASE** ■ **EUSCI_B3_BASE**

This function unregisters the handler to be called when timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

23 System Control Module (SysCtl)

23.1 Module Operation

The SysCtl module is a conglomeration of miscellaneous system control modules that do not fit into any specific hardware peripheral.

Some of the functionalities of the SysCtl module include:

- Configure and enable/disable NMI sources
- Retrieve the SRAM/Flash size through software calls
- Disable/enable SRAM banks completely as well as disable retention during sleep
- Enable/disable GPIO glitch filters
- Change the type of reset that occurs on a WDT violation

23.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the SysCtl module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to retrieve the Flash and SRAM sizes using a software API. This is useful if the programmer is making a program that is meant to be run on multiple devices in the MSP432 family with different memory footprints.

```
int main(void)
{
    /* Variables we will store the sizes in. Declared volatile so the compiler
     * does not optimize out
     */
    volatile uint32_t sramSize,  flashSize;

    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();

    sramSize = MAP_SysCtl_getSRAMSize();
    flashSize = MAP_SysCtl_getFlashSize();

    /* No operation. Set Breakpoint here */
    __no_operation();
}
```

23.3 Definitions

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

24 System Control Module (SysCtl$_A$)

24.1 Module Operation

Note that this module is for use exclusively on the MSP432P4111. If using the MSP432P401, please refer to the non-a variant.

The SysCtl_A module is a conglomeration of miscellaneous system control modules that do not fit into any specific hardware peripheral.

Some of the functionalities of the SysCtl_A module include:

- Configure and enable/disable NMI sources
- Retrieve the SRAM/Flash size through software calls
- Disable/enable SRAM banks completely as well as disable retention during sleep
- Enable/disable GPIO glitch filters
- Change the type of reset that occurs on a WDT violation

24.2 Programming Example

24.3 Definitions

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

25 System Tick (SysTick)

25.1 Module Operation

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M microprocessor. Its intended purpose is to provide a periodic interrupt for an RTOS, but it can be used for other simple timing purposes.

The SysTick interrupt handler does not need to clear the SysTick interrupt source as it is cleared automatically by the NVIC when the SysTick interrupt handler is called.

25.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the SysTick module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the SysTick module to interrupt periodically and blink an LED attached to P1.0.

```
int main(void)
{
    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();

    /* Configuring GPIO as an output */
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P1,  GPIO_PIN0);

    /* Configuring SysTick to trigger at 1500000 (MCLK is 1.5MHz so this will
     * make it toggle every 1s) */
    MAP_SysTick_enableModule();
    MAP_SysTick_setPeriod(1500000);
    MAP_Interrupt_enableSleepOnIsrExit();
    MAP_SysTick_enableInterrupt();

    /* Enabling MASTER interrupts */
    MAP_Interrupt_enableMaster();

    while (1)
    {
        MAP_PCM_gotoLPM0();
    }
}

void SysTick_Handler(void)
{
    MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1,  GPIO_PIN0);
}
```

25.3 Definitions

Functions

- void SysTick_disableInterrupt (void)
- void SysTick_disableModule (void)
- void SysTick_enableInterrupt (void)
- void SysTick_enableModule (void)
- uint32_t SysTick_getPeriod (void)
- uint32_t SysTick_getValue (void)
- void SysTick_registerInterrupt (void(*intHandler)(void))
- void SysTick_setPeriod (uint32_t period)
- void SysTick_unregisterInterrupt (void)

25.3.1 Detailed Description

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

25.3.2 Function Documentation

25.3.2.1 void SysTick_disableInterrupt (void)

Disables the SysTick interrupt.

This function disables the SysTick interrupt, preventing it from being reflected to the processor.

Returns
None.

25.3.2.2 void SysTick_disableModule (void)

Disables the SysTick counter.

This function stops the SysTick counter. If an interrupt handler has been registered, it is not called until SysTick is restarted.

Returns
None.

25.3.2.3 void SysTick_enableInterrupt (void)

Enables the SysTick interrupt.

This function enables the SysTick interrupt, allowing it to be reflected to the processor.

Note
The SysTick interrupt handler is not required to clear the SysTick interrupt source because it is cleared automatically by the NVIC when the interrupt handler is called.

Returns
None.

25.3.2.4 void SysTick_enableModule (void)

Enables the SysTick counter.

This function starts the SysTick counter. If an interrupt handler has been registered, it is called when the SysTick counter rolls over.

Note
Calling this function causes the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to SysTick_setPeriod(). If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written to force the reload. Any write to this register clears the SysTick counter to 0 and causes a reload with the supplied period on the next clock.

Returns
None.

25.3.2.5 uint32_t SysTick_getPeriod (void)

Gets the period of the SysTick counter.

This function returns the rate at which the SysTick counter wraps, which equates to the number of processor clocks between interrupts.

Returns
Returns the period of the SysTick counter.

25.3.2.6 uint32_t SysTick_getValue (void)

Gets the current value of the SysTick counter.

This function returns the current value of the SysTick counter, which is a value between the period - 1 and zero, inclusive.

Returns
Returns the current value of the SysTick counter.

25.3.2.7 void SysTick_registerInterrupt (void(*)(void) *intHandler*)

Registers an interrupt handler for the SysTick interrupt.

Parameters

intHandler	is a pointer to the function to be called when the SysTick interrupt occurs.

This function registers the handler to be called when a SysTick interrupt occurs.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_registerInterrupt().

25.3.2.8 void SysTick_setPeriod (uint32_t *period*)

Sets the period of the SysTick counter.

Parameters

period	is the number of clock ticks in each period of the SysTick counter and must be between 1 and 16, 777, 216, inclusive.

This function sets the rate at which the SysTick counter wraps, which equates to the number of processor clocks between interrupts.

Note

Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and causes a reload with the *period* supplied here on the next clock after SysTick is enabled.

Returns

None.

25.3.2.9 void SysTick_unregisterInterrupt (void)

Unregisters the interrupt handler for the SysTick interrupt.

This function unregisters the handler to be called when a SysTick interrupt occurs.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_unregisterInterrupt().

26 32-bit ARM Timer (Timer32)

26.1 Module Operation

The Timer32 module in MSP432 is a simple 32-bit (or 16-bit depending on configuration) down counter which was implemented by ARM. While the user's guide for Timer32 treats the module as one unified timer, the DriverLib API separates the two timers into two separate modules. To choose between the module, the user either provides TIMER32_0 or TIMER32_1 to the timer in order to specify which timer is to be used.

26.2 Basic Operation Modes

Free Run Mode In free run mode, the timer will run from a value of UINT16_MAX or UINT32_MAX (depending on what resolution is selected).

Periodic Mode In periodic mode, the timer will run to a specified period by the user.

For both periodic and free run modes, the one shot boolean option in the Timer32_startTimer() function. If specified, when the count reaches zero from the specified period the timer will stop and not automatically resume with the next iteration of the count.

26.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the Timer32 module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the Timer32 as a simple down counter with interrupts enabled:

```
int main(void)
{
    volatile uint32_t curValue;

    /* Holding the Watchdog */
    MAP_WDT_A_holdTimer();

    /* Initializing Timer32 in module in 32-bit free-run mode (with max value
     * of 0xFFFFFFFF
     */
    MAP_Timer32_initModule(TIMER32_BASE,  TIMER32_PRESCALER_256,  TIMER32_32BIT,
            TIMER32_FREE_RUN_MODE);

    /* Starting the timer */
    MAP_Timer32_startTimer(TIMER32_BASE,  true);

    while(1)
    {
        /* Getting the current value of the Timer32 */
        curValue = MAP_Timer32_getValue(TIMER32_BASE);
    }
}
```

26.4 Definitions

Functions

- void Timer32_clearInterruptFlag (uint32_t timer)
- void Timer32_disableInterrupt (uint32_t timer)
- void Timer32_enableInterrupt (uint32_t timer)
- uint32_t Timer32_getInterruptStatus (uint32_t timer)
- uint32_t Timer32_getValue (uint32_t timer)
- void Timer32_haltTimer (uint32_t timer)
- void Timer32_initModule (uint32_t timer, uint32_t preScaler, uint32_t resolution, uint32_t mode)
- void Timer32_registerInterrupt (uint32_t timerInterrupt, void(*intHandler)(void))
- void Timer32_setCount (uint32_t timer, uint32_t count)
- void Timer32_setCountInBackground (uint32_t timer, uint32_t count)
- void Timer32_startTimer (uint32_t timer, bool oneShot)
- void Timer32_unregisterInterrupt (uint32_t timerInterrupt)

26.4.1 Detailed Description

The code for this module is contained in `driverlib/timer32.c`, with `driverlib/timer32.h` containing the API declarations for use by applications.

26.4.2 Function Documentation

26.4.2.1 void Timer32_clearInterruptFlag (uint32_t *timer*)

Clears Timer32 interrupt source.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

The Timer32 interrupt source is cleared, so that it no longer asserts.

Returns
None.

26.4.2.2 void Timer32_disableInterrupt (uint32_t *timer*)

Disables a Timer32 interrupt source.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

Disables the indicated Timer32 interrupt source.

Returns
None.

26.4.2.3 void Timer32_enableInterrupt (uint32_t *timer*)

Enables a Timer32 interrupt source.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

Enables the indicated Timer32 interrupt source.

Returns
None.

26.4.2.4 uint32_t Timer32_getInterruptStatus (uint32_t *timer*)

Gets the current Timer32 interrupt status.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

This returns the interrupt status for the Timer32 module. A positive value will indicate that an interrupt is pending while a zero value will indicate that no interrupt is pending.

Returns
The current interrupt status

26.4.2.5 uint32_t Timer32_getValue (uint32_t *timer*)

Returns the current value of the timer.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

Returns
The current count of the timer.

26.4.2.6 void Timer32_haltTimer (uint32_t *timer*)

Halts the timer. Current count and setting values are preserved.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**

Returns
None

26.4.2.7 void Timer32_initModule (uint32_t *timer,* uint32_t *preScaler,* uint32_t *resolution,* uint32_t *mode*)

Initializes the Timer32 module

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**
preScaler	is the prescaler (or divider) to apply to the clock source given to the Timer32 module. Valid values are ■ **TIMER32_PRESCALER_1** [DEFAULT] ■ **TIMER32_PRESCALER_16** ■ **TIMER32_PRESCALER_256**
resolution	is the bit resolution of the Timer32 module. Valid values are ■ **TIMER32_16BIT** [DEFAULT] ■ **TIMER32_32BIT**
mode	selects between free run and periodic mode. In free run mode, the value of the timer is reset to UINT16_MAX (for 16-bit mode) or UINT32_MAX (for 16-bit mode) when the timer reaches zero. In periodic mode, the timer is reset to the value set by the Timer32_setCount function. Valid values are ■ **TIMER32_FREE_RUN_MODE** [DEFAULT] ■ **TIMER32_PERIODIC_MODE**

Returns
None.

26.4.2.8 void Timer32_registerInterrupt (uint32_t *timerInterrupt,* void(∗)(void) *intHandler*)

Registers an interrupt handler for Timer32 interrupts.

Parameters

timerInterrupt	is the specific interrupt to register. For the Timer32 module, there are a total of three different interrupts: one interrupt for each two Timer32 modules, and a "combined" interrupt which is a logical OR of each individual Timer32 interrupt. ■ **TIMER32_0_INTERRUPT** ■ **TIMER32_1_INTERRUPT** ■ **TIMER32_COMBINED_INTERRUPT**
intHandler	is a pointer to the function to be called when the Timer32 interrupt occurs.

This function registers the handler to be called when an Timer32 interrupt occurs. This function enables the global interrupt in the interrupt controller; specific Timer32 interrupts must be enabled via Timer32_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via Timer32_clearInterruptFlag().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

26.4.2.9 void Timer32_setCount (uint32_t *timer,* uint32_t *count*)

Sets the count of the timer and resets the current value to the value passed. This value is set on the next rising edge of the clock provided to the timer module

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**
count	Value of the timer to set. Note that if the timer is in 16-bit mode and a value is passed in that exceeds UINT16_MAX, the value will be truncated to UINT16_MAX.

Also note that if the timer is operating in periodic mode, the value passed into this function will represent the new period of the timer (the value which is reloaded into the timer each time it reaches a zero value).

Returns
None

26.4.2.10 void Timer32_setCountInBackground (uint32_t *timer,* uint32_t *count*)

Sets the count of the timer without resetting the current value. When the current value of the timer reaches zero, the value passed into this function will be set as the new count value.

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**
count	Value of the timer to set in the background. Note that if the timer is in 16-bit mode and a value is passed in that exceeds UINT16_MAX, the value will be truncated to UINT16_MAX.

Also note that if the timer is operating in periodic mode, the value passed into this function will represent the new period of the timer (the value which is reloaded into the timer each time it reaches a zero value).

Returns
None

26.4.2.11 void Timer32_startTimer (uint32_t *timer,* bool *oneShot*)

Starts the timer. The Timer32_initModule function should be called (in conjunction with Timer32_setCount if periodic mode is desired) prior to

Parameters

timer	is the instance of the Timer32 module. Valid parameters must be one of the following values: ■ **TIMER32_0_BASE** ■ **TIMER32_1_BASE**
oneShot	sets whether the Timer32 module operates in one shot or continuous mode. In one shot mode, the timer will halt when a zero is reached and stay halted until either: ■ The user calls the Timer32PeriodSet function ■ The Timer32_initModule is called to reinitialize the timer with one-shot mode disabled.

A true value will cause the timer to operate in one shot mode while a false value will cause the timer to operate in continuous mode

Returns
None

26.4.2.12 void Timer32_unregisterInterrupt (uint32_t *timerInterrupt*)

Unregisters the interrupt handler for the Timer32 interrupt.

Parameters

timerInterrupt	is the specific interrupt to register. For the Timer32 module, there are a total of three different interrupts: one interrupt for each two Timer32 modules, and a "combined" interrupt which is a logical OR of each individual Timer32 interrupt. ■ **TIMER32_0_INTERRUPT** ■ **TIMER32_1_INTERRUPT** ■ **TIMER32_COMBINED_INTERRUPT**

This function unregisters the handler to be called when a Timer32 interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also

Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns

None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

27 16-Bit Timer with Precision PWM (Timer_A)

27.1 Module Operation

TimerA is a 16-bit timer/counter with multiple capture/compare registers. TimerA can support multiple capture/compares, PWM outputs, and interval timing. TimerA also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

This peripheral API handles Timer A hardware peripheral.

TimerA features include:

- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Up to seven configurable capture/compare registers
- Configurable outputs with pulse width modulation (PWM) capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer interrupts

27.2 Basic Operation Modes

TimerA can operate in 3 modes:

- Continuous Mode
- Up Mode
- Down Mode

TimerA Interrupts may be generated on counter overflow conditions and during capture compare events.

The TimerA may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TimerA_initCompare() and the necessary parameters. The PWM may be customized by selecting a desired timer mode (continuous/up/upDown), duty cycle, output mode, timer period etc. The library also provides a simpler way to generate PWM using TimerA_generatePWM() API. However the level of customization and the kinds of PWM generated are limited in this API. Depending on how complex the PWM is and what level of customization is required, the user can use TimerA_generatePWM() or a combination of TimerA_initCompare and timer start APIs.

The TimerA API provides a set of functions for dealing with the TimerA module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured.

27.3 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the TimerA module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to generate a PWM signal using the TimerA DriverLib module.

Below is the configuration parameter for the TimerA PWM config API:

```
/* Timer_A PWM Configuration Parameter */
Timer_A_PWMConfig pwmConfig =
{
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        32000,
        TIMER_A_CAPTURECOMPARE_REGISTER_1,
        TIMER_A_OUTPUTMODE_RESET_SET,
        3200
};
```

The next snippet of code is used to actually configure the PWM signal:

```
    /* Setting MCLK to REFO at 128Khz for LF mode
     * Setting SMCLK to 64Khz */
    MAP_CS_setReferenceOscillatorFrequency(CS_REFO_128KHZ);
    MAP_CS_initClockSignal(CS_MCLK,  CS_REFOCLK_SELECT,  CS_CLOCK_DIVIDER_1);
    MAP_CS_initClockSignal(CS_SMCLK,  CS_REFOCLK_SELECT,  CS_CLOCK_DIVIDER_2);
    MAP_PCM_setPowerState(PCM_AM_LF_VCORE0);

    /* Configuring GPIO2.4 as peripheral output for PWM  and P6.7 for button
     * interrupt */
    MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P2,  GPIO_PIN4,
            GPIO_PRIMARY_MODULE_FUNCTION);
    MAP_GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1,  GPIO_PIN1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1,  GPIO_PIN1);
    MAP_GPIO_enableInterrupt(GPIO_PORT_P1,  GPIO_PIN1);

    /* Configuring Timer_A to have a period of approximately 500ms and
     * an initial duty cycle of 10% of that (3200 ticks)  */
    MAP_Timer_A_generatePWM(TIMER_A0_BASE,  &pwmConfig);
```

27.4 Definitions

Data Structures

- struct _Timer_A_CaptureModeConfig
- struct _Timer_A_CompareModeConfig
- struct _Timer_A_ContinuousModeConfig
- struct _Timer_A_PWMConfig
- struct _Timer_A_UpDownModeConfig
- struct _Timer_A_UpModeConfig

Functions

- void Timer_A_clearCaptureCompareInterrupt (uint32_t timer, uint_fast16_t captureCompareRegister)
- void Timer_A_clearInterruptFlag (uint32_t timer)
- void Timer_A_clearTimer (uint32_t timer)
- void Timer_A_configureContinuousMode (uint32_t timer, const Timer_A_ContinuousModeConfig *config)
- void Timer_A_configureUpDownMode (uint32_t timer, const Timer_A_UpDownModeConfig *config)
- void Timer_A_configureUpMode (uint32_t timer, const Timer_A_UpModeConfig *config)
- void Timer_A_disableCaptureCompareInterrupt (uint32_t timer, uint_fast16_t captureCompareRegister)
- void Timer_A_disableInterrupt (uint32_t timer)
- void Timer_A_enableCaptureCompareInterrupt (uint32_t timer, uint_fast16_t captureCompareRegister)
- void Timer_A_enableInterrupt (uint32_t timer)
- void Timer_A_generatePWM (uint32_t timer, const Timer_A_PWMConfig *config)
- uint_fast16_t Timer_A_getCaptureCompareCount (uint32_t timer, uint_fast16_t captureCompareRegister)
- uint32_t Timer_A_getCaptureCompareEnabledInterruptStatus (uint32_t timer, uint_fast16_t captureCompareRegister)
- uint32_t Timer_A_getCaptureCompareInterruptStatus (uint32_t timer, uint_fast16_t captureCompareRegister, uint_fast16_t mask)
- uint16_t Timer_A_getCounterValue (uint32_t timer)
- uint32_t Timer_A_getEnabledInterruptStatus (uint32_t timer)
- uint32_t Timer_A_getInterruptStatus (uint32_t timer)
- uint_fast8_t Timer_A_getOutputForOutputModeOutBitValue (uint32_t timer, uint_fast16_t captureCompareRegister)
- uint_fast8_t Timer_A_getSynchronizedCaptureCompareInput (uint32_t timer, uint_fast16_t captureCompareRegister, uint_fast16_t synchronizedSetting)
- void Timer_A_initCapture (uint32_t timer, const Timer_A_CaptureModeConfig *config)
- void Timer_A_initCompare (uint32_t timer, const Timer_A_CompareModeConfig *config)
- void Timer_A_registerInterrupt (uint32_t timer, uint_fast8_t interruptSelect, void(*intHandler)(void))
- void Timer_A_setCompareValue (uint32_t timer, uint_fast16_t compareRegister, uint_fast16_t compareValue)
- void Timer_A_setOutputForOutputModeOutBitValue (uint32_t timer, uint_fast16_t captureCompareRegister, uint_fast8_t outputModeOutBitValue)
- void Timer_A_startCounter (uint32_t timer, uint_fast16_t timerMode)
- void Timer_A_stopTimer (uint32_t timer)
- void Timer_A_unregisterInterrupt (uint32_t timer, uint_fast8_t interruptSelect)

27.4.1 Detailed Description

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

27.4.2 Data Structure Documentation

27.4.2.1 struct _Timer_A_CaptureModeConfig

Type definition for _Timer_A_CaptureModeConfig structure.

ypedef Timer_A_CaptureModeConfig

Configuration structure for capture mode in the **Timer_A** module. See Timer_A_initCapture for parameter documentation.

27.4.2.2 struct _Timer_A_CompareModeConfig

Type definition for _Timer_A_CompareModeConfig structure.

ypedef Timer_A_CompareModeConfig

Configuration structure for compare mode in the **Timer_A** module. See Timer_A_initCompare for parameter documentation.

27.4.2.3 struct _Timer_A_ContinuousModeConfig

Type definition for _Timer_A_ContinuousModeConfig structure.

ypedef Timer_A_ContinuousModeConfig

Configuration structure for continuous mode in the **Timer_A** module. See Timer_A_configureContinuousMode for parameter documentation.

27.4.2.4 struct _Timer_A_PWMConfig

Type definition for _Timer_A_PWMConfig structure.

ypedef Timer_A_PWMConfig

Configuration structure for PWM mode in the **Timer_A** module. See Timer_A_generatePWM for parameter documentation.

27.4.2.5 struct _Timer_A_UpDownModeConfig

Type definition for _Timer_A_UpDownModeConfig structure.

ypedef Timer_A_UpDownModeConfig

Configuration structure for UpDown mode in the **Timer_A** module. See Timer_A_configureUpDownMode for parameter documentation.

27.4.2.6 struct _Timer_A_UpModeConfig

Type definition for _Timer_A_UpModeConfig structure.

ypedef Timer_A_UpModeConfig

Configuration structure for Up mode in the **Timer_A** module. See Timer_A_configureUpMode for parameter documentation.

27.4.3 Function Documentation

27.4.3.1 void Timer_A_clearCaptureCompareInterrupt (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
captureCompareRegister	selects the Capture-compare register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to the datasheet to ensure the device has the capture compare register being used

Returns
None

27.4.3.2 void Timer_A_clearInterruptFlag (uint32_t *timer*)

Clears the Timer TAIFG interrupt flag

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns
None

27.4.3.3 void Timer_A_clearTimer (uint32_t *timer*)

Reset/Clear the timer clock divider, count direction, count

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns
None

27.4.3.4 void Timer_A_configureContinuousMode (uint32_t *timer,* const **Timer_A_ContinuousModeConfig** ∗ *config*)

Configures Timer_A in continuous mode.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
config	Configuration structure for Timer_A continuous mode

Configuration options for Timer_A_ContinuousModeConfig structure.

Parameters

clockSource	selects Clock source. Valid values are ■ **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default value] ■ **TIMER_A_CLOCKSOURCE_ACLK** ■ **TIMER_A_CLOCKSOURCE_SMCLK** ■ **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**
timerInterruptEnable_TAIE	is the divider for Clock source. Valid values are: ■ **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default value] ■ **TIMER_A_CLOCKSOURCE_DIVIDER_2** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_4** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_8** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_3** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_5** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_6** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_7** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_10** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_12** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_14** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_16** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_20** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_24** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_28** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_32** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_40** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_48** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_56** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_64**
timerInterruptEnable_TAIE	is to enable or disable Timer_A interrupt. Valid values are ■ **TIMER_A_TAIE_INTERRUPT_ENABLE** ■ **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default value]

timerClear	decides if Timer_A clock divider, count direction, count need to be reset. Valid values are ■ **TIMER_A_DO_CLEAR** ■ **TIMER_A_SKIP_CLEAR** [Default value]

Note

This API does not start the timer. Timer needs to be started when required using the Timer_A_startCounter API.

Returns

None

27.4.3.5 void Timer_A_configureUpDownMode (uint32_t *timer,* const **Timer_A_UpDownModeConfig** ∗ *config*)

Configures Timer_A in up down mode.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
config	Configuration structure for Timer_A UpDown mode

Configuration options for Timer_A_UpDownModeConfig structure.

Parameters

clockSource	selects Clock source. Valid values are ■ **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default value] ■ **TIMER_A_CLOCKSOURCE_ACLK** ■ **TIMER_A_CLOCKSOURCE_SMCLK** ■ **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

clockSourceDivider	is the divider for Clock source. Valid values are: ■ **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default value] ■ **TIMER_A_CLOCKSOURCE_DIVIDER_2** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_4** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_8** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_3** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_5** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_6** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_7** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_10** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_12** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_14** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_16** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_20** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_24** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_28** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_32** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_40** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_48** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_56** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_64**

timerPeriod	is the specified Timer_A period
timerInterruptEnable_TAIE	is to enable or disable Timer_A interrupt. Valid values are ■ **TIMER_A_TAIE_INTERRUPT_ENABLE** ■ **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default value]
captureCompareInterruptEnable_CCR0_CCIE	is to enable or disable Timer_A CCR0 captureComapre interrupt. Valid values are ■ **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE** and ■ **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default value]
timerClear	decides if Timer_A clock divider, count direction, count need to be reset. Valid values are ■ **TIMER_A_DO_CLEAR** ■ **TIMER_A_SKIP_CLEAR** [Default value]

This API does not start the timer. Timer needs to be started when required using the Timer_A_startCounter API.

Returns
None

27.4.3.6 void Timer_A_configureUpMode (uint32_t *timer,* const **Timer_A_UpModeConfig** ∗ *config*)

Configures Timer_A in up mode.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
config	Configuration structure for Timer_A Up mode

Configuration options for Timer_A_UpModeConfig structure.

Parameters

clockSource	selects Clock source. Valid values are ■ **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default value] ■ **TIMER_A_CLOCKSOURCE_ACLK** ■ **TIMER_A_CLOCKSOURCE_SMCLK** ■ **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**
clockSourceDivider	is the divider for Clock source. Valid values are: ■ **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default value] ■ **TIMER_A_CLOCKSOURCE_DIVIDER_2** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_4** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_8** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_3** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_5** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_6** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_7** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_10** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_12** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_14** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_16** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_20** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_24** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_28** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_32** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_40** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_48** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_56** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_64**

timerPeriod	is the specified Timer_A period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]
timerInterruptEnable_TAIE	is to enable or disable Timer_A interrupt. Valid values are: ■ **TIMER_A_TAIE_INTERRUPT_ENABLE** and ■ **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default value]
captureCompareInterruptEnable_CCR0_CCIE	is to enable or disable Timer_A CCR0 captureComapre interrupt. Valid values are ■ **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE** and ■ **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default value]
timerClear	decides if Timer_A clock divider, count direction, count need to be reset. Valid values are ■ **TIMER_A_DO_CLEAR** ■ **TIMER_A_SKIP_CLEAR** [Default value]

Note

This API does not start the timer. Timer needs to be started when required using the Timer_A_startCounter API.

Returns

None

27.4.3.7 void Timer_A_disableCaptureCompareInterrupt (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Disable capture compare interrupt

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

captureCompar-eRegister	is the selected capture compare register

Returns
None

27.4.3.8 void Timer_A_disableInterrupt (uint32_t *timer*)

Disable timer interrupt

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns
None

27.4.3.9 void Timer_A_enableCaptureCompareInterrupt (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Enable capture compare interrupt

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

captureCompar-eRegister	is the selected capture compare register

Returns
None

27.4.3.10 void Timer_A_enableInterrupt (uint32_t *timer*)

Enable timer interrupt

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns
None

27.4.3.11 void Timer_A_generatePWM (uint32_t *timer,* const **Timer_A_PWMConfig** ∗ *config*)

Generate a PWM with timer running in up mode

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

config	Configuration structure for Timer_A PWM mode

Configuration options for Timer_A_PWMConfig structure.

Parameters

clockSource	selects Clock source. Valid values are ■ **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** ■ **TIMER_A_CLOCKSOURCE_ACLK** ■ **TIMER_A_CLOCKSOURCE_SMCLK** ■ **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**
clockSourceDivider	is the divider for Clock source. Valid values are ■ **TIMER_A_CLOCKSOURCE_DIVIDER_1** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_2** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_4** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_8** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_3** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_5** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_6** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_7** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_10** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_12** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_14** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_16** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_20** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_24** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_28** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_32** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_40** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_48** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_56** ■ **TIMER_A_CLOCKSOURCE_DIVIDER_64**

timerPeriod	selects the desired timer period
compareRegister	selects the compare register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used
compareOutputMode	specifies the ouput mode. Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE**, ■ **TIMER_A_OUTPUTMODE_SET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_RESET**, ■ **TIMER_A_OUTPUTMODE_SET_RESET** ■ **TIMER_A_OUTPUTMODE_TOGGLE**, ■ **TIMER_A_OUTPUTMODE_RESET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_SET**, ■ **TIMER_A_OUTPUTMODE_RESET_SET**
dutyCycle	specifies the dutycycle for the generated waveform

Returns
None

27.4.3.12 uint_fast16_t Timer_A_getCaptureCompareCount (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Get current capture compare count

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

captureCompareRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used

Returns

current count as uint16_t

27.4.3.13 uint32_t Timer_A_getCaptureCompareEnabledInterruptStatus (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Return capture compare interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
captureCompareRegister	is the selected capture compare register

Returns

uint32_t. The mask of the set flags. Valid values is an OR of

- **TIMER_A_CAPTURE_OVERFLOW**,
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG**

References Timer_A_getCaptureCompareInterruptStatus().

27.4.3.14 uint32_t Timer_A_getCaptureCompareInterruptStatus (uint32_t *timer,* uint_fast16_t *captureCompareRegister,* uint_fast16_t *mask*)

Return capture compare interrupt status

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
captureCompareRegister	is the selected capture compare register
mask	is the mask for the interrupt status Mask value is the logical OR of any of the following: ■ **TIMER_A_CAPTURE_OVERFLOW** ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG**

Returns

uint32_t. The mask of the set flags. Valid values is an OR of
- **TIMER_A_CAPTURE_OVERFLOW**,
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG**

Referenced by Timer_A_getCaptureCompareEnabledInterruptStatus().

27.4.3.15 uint16_t Timer_A_getCounterValue (uint32_t *timer*)

Returns the current value of the specified timer. Note that according to the Timer A user guide, reading the value of the counter is unreliable if the system clock is asynchronous from the timer clock. The API addresses this concern by reading the timer count register twice and then determining the integrity of the value. If the two values are within 10 timer counts of each other, the value is deemed safe and returned. If not, the process is repeated until a reliable timer value is determined.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns

The value of the specified timer

27.4.3.16 uint32_t Timer_A_getEnabledInterruptStatus (uint32_t *timer*)

Get timer interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns

uint32_t. Return interrupt status. Valid values are

- **TIMER_A_INTERRUPT_PENDING**
- **TIMER_A_INTERRUPT_NOT_PENDING**

References Timer_A_getInterruptStatus().

27.4.3.17 uint32_t Timer_A_getInterruptStatus (uint32_t *timer*)

Get timer interrupt status

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns

uint32_t. Return interrupt status. Valid values are

- **TIMER_A_INTERRUPT_PENDING**
- **TIMER_A_INTERRUPT_NOT_PENDING**

Referenced by Timer_A_getEnabledInterruptStatus().

27.4.3.18 uint_fast8_t Timer_A_getOutputForOutputModeOutBitValue (uint32_t *timer,* uint_fast16_t *captureCompareRegister*)

Get ouput bit for output mode

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
captureCompareRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used

Returns

TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH or
- **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

27.4.3.19 uint_fast8_t Timer_A_getSynchronizedCaptureCompareInput (uint32_t *timer,* uint_fast16_t *captureCompareRegister,* uint_fast16_t *synchronizedSetting*)

Get synchronized capture compare input

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

captureCompareRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used
synchronizedSetting	is to select type of capture compare input. Valid values are ■ **TIMER_A_READ_CAPTURE_COMPARE_INPUT** ■ **TIMER_A_READ_SYNCHRONIZED_CAPTURECOMPAREINPUT**

Returns

TIMER_A_CAPTURECOMPARE_INPUT_HIGH or
- **TIMER_A_CAPTURECOMPARE_INPUT_LOW**

27.4.3.20 void Timer_A_initCapture (uint32_t *timer,* const **Timer_A_CaptureModeConfig** ∗ *config*)

Initializes Capture Mode

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
config	Configuration structure for Timer_A capture mode

Configuration options for Timer_A_CaptureModeConfig structure.

Parameters

captureRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used
captureMode	is the capture mode selected. Valid values are ■ **TIMER_A_CAPTUREMODE_NO_CAPTURE** [Default value] ■ **TIMER_A_CAPTUREMODE_RISING_EDGE** ■ **TIMER_A_CAPTUREMODE_FALLING_EDGE** ■ **TIMER_A_CAPTUREMODE_RISING_AND_FALLING_EDGE**
captureInputSelect	decides the Input Select ■ **TIMER_A_CAPTURE_INPUTSELECT_CCIxA** [Default value] ■ **TIMER_A_CAPTURE_INPUTSELECT_CCIxB** ■ **TIMER_A_CAPTURE_INPUTSELECT_GND** ■ **TIMER_A_CAPTURE_INPUTSELECT_Vcc**
synchronizeCaptureSource	decides if capture source should be synchronized with timer clock Valid values are ■ **TIMER_A_CAPTURE_ASYNCHRONOUS** [Default value] ■ **TIMER_A_CAPTURE_SYNCHRONOUS**
captureInterruptEnable	is to enable or disable timer captureComapre interrupt. Valid values are ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default value] ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**
captureOutputMode	specifies the ouput mode. Valid values are ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default value], ■ **TIMER_A_OUTPUTMODE_SET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_RESET**, ■ **TIMER_A_OUTPUTMODE_SET_RESET** ■ **TIMER_A_OUTPUTMODE_TOGGLE**, ■ **TIMER_A_OUTPUTMODE_RESET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_SET**, ■ **TIMER_A_OUTPUTMODE_RESET_SET**

Returns
None

27.4.3.21 void Timer_A_initCompare (uint32_t *timer,* const **Timer_A_CompareModeConfig** ∗ *config*)

Initializes Compare Mode

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
config	Configuration structure for Timer_A compare mode

Configuration options for Timer_A_CompareModeConfig structure.

Parameters

compareRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used

compareInterruptEnable	is to enable or disable timer captureComapre interrupt. Valid values are ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE** and ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default value]
compareOutputMode	specifies the output mode. Valid values are ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default value], ■ **TIMER_A_OUTPUTMODE_SET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_RESET**, ■ **TIMER_A_OUTPUTMODE_SET_RESET** ■ **TIMER_A_OUTPUTMODE_TOGGLE**, ■ **TIMER_A_OUTPUTMODE_RESET**, ■ **TIMER_A_OUTPUTMODE_TOGGLE_SET**, ■ **TIMER_A_OUTPUTMODE_RESET_SET**
compareValue	is the count to be compared with in compare mode

Returns
None

27.4.3.22 void Timer_A_registerInterrupt (uint32_t *timer,* uint_fast8_t *interruptSelect,* void(∗)(void) *intHandler*)

Registers an interrupt handler for the timer capture compare interrupt.

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

interruptSelect	Selects which timer interrupt handler to register. For the timer module, there are two separate interrupt handlers that can be registered: ■ **TIMER_A_CCR0_INTERRUPT** Corresponds to the interrupt for CCR0 ■ **TIMER_A_CCRX_AND_OVERFLOW_INTERRUPT** Corresponds to the interrupt for CCR1-6, as well as the overflow interrupt.
intHandler	is a pointer to the function to be called when the timer capture compare interrupt occurs.

This function registers the handler to be called when a timer interrupt occurs. This function enables the global interrupt in the interrupt controller; specific Timer_Ainterrupts must be enabled via Timer_A_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via Timer_A_clearCaptureCompareInterrupt().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

27.4.3.23 void Timer_A_setCompareValue (uint32_t *timer,* uint_fast16_t *compareRegister,* uint_fast16_t *compareValue*)

Sets the value of the capture-compare register

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
compareRegister	selects the Capture register being used. Valid values are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used

compareValue	is the count to be compared with in compare mode

Returns
None

27.4.3.24 void Timer_A_setOutputForOutputModeOutBitValue (uint32_t *timer,* uint_fast16_t *captureCompareRegister,* uint_fast8_t *outputModeOutBitValue*)

Set ouput bit for output mode

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
captureCompareRegister	selects the Capture register being used. are ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6** Refer to datasheet to ensure the device has the capture compare register being used

outputModeOut-BitValue	the value to be set for out bit. Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

Returns
None

27.4.3.25 void Timer_A_startCounter (uint32_t *timer,* uint_fast16_t *timerMode*)

Starts Timer_A counter

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
timerMode	selects Clock source. Valid values are ■ **TIMER_A_CONTINUOUS_MODE** [Default value] ■ **TIMER_A_UPDOWN_MODE** ■ **TIMER_A_UP_MODE**

Note
This function assumes that the timer has been previously configured using Timer_A_configureContinuousMode, Timer_A_configureUpMode or Timer_A_configureUpDownMode.

Returns
None

27.4.3.26 void Timer_A_stopTimer (uint32_t *timer*)

Stops the timer

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**

Returns
None

27.4.3.27 void Timer_A_unregisterInterrupt (uint32_t *timer,* uint_fast8_t *interruptSelect*)

Unregisters the interrupt handler for the timer

Parameters

timer	is the instance of the Timer_A module. Valid parameters vary from part to part, but can include: ■ **TIMER_A0_BASE** ■ **TIMER_A1_BASE** ■ **TIMER_A2_BASE** ■ **TIMER_A3_BASE**
interruptSelect	Selects which timer interrupt handler to register. For the timer module, there are two separate interrupt handlers that can be registered: ■ **TIMER_A_CCR0_INTERRUPT** Corresponds to the interrupt for CCR0 ■ **TIMER_A_CCRX_AND_OVERFLOW_INTERRUPT** Corresponds to the interrupt for CCR1-6, as well as the overflow interrupt.

This function unregisters the handler to be called when timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

28 Universal Asynchronous Receiver/Transmitter (UART)

28.1 Module Operation

The SDK library for UART mode features include:

- Odd, even, or non-parity
- Independent transmit and receive shift registers
- Separate transmit and receive buffer registers
- LSB-first or MSB-first data transmit and receive
- Built-in idle-line and address-bit communication protocols for multiprocessor systems
- Status flags for error detection and suppression
- Status flags for address detection
- Independent interrupt capability for receive and transmit

The modes of operations supported by the UART and the library include

- UART mode
- Idle-line multiprocessor mode
- Address-bit multiprocessor mode
- UART mode with automatic baud-rate detection

In UART mode, the USCI transmits and receives characters at a bit rate asynchronous to another device. Timing for each character is based on the selected baud rate of the USCI. The transmit and receive functions use the same baud-rate frequency.

28.2 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the UART module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure and enable the UART module. In the case of this example, we assume the MCLK is operating off of the DCO and the DCO is tuned to 12MHz. This makes the configuration parameters so that the baud rate is 9600.

Below is an example of the UART configuration parameter:

```
/* UART Configuration Parameter. These are the configuration parameters to
 * make the eUSCI A UART module to operate with a 9600 baud rate. These
 * values were calculated using the online calculator that TI provides
 * at:
 *http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.html
 */
const eUSCI_UART_Config uartConfig =
{
        EUSCI_A_UART_CLOCKSOURCE_SMCLK,           // SMCLK Clock Source
        78,                                      // BRDIV = 78
        2,                                        // UCxBRF = 2
        0,                                        // UCxBRS = 0
        EUSCI_A_UART_NO_PARITY,                   // No Parity
        EUSCI_A_UART_LSB_FIRST,                   // LSB First
        EUSCI_A_UART_ONE_STOP_BIT,                // One stop bit
        EUSCI_A_UART_MODE,                        // UART mode
        EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION  // Oversampling
};
```

This code snippet is the actual configuration of the UART module using the DriverLib APIs:

```
    /* Configuring UART Module */
    MAP_UART_initModule(EUSCI_A0_BASE,  &uartConfig);

    /* Enable UART module */
    MAP_UART_enableModule(EUSCI_A0_BASE);

    /* Enabling interrupts */
    MAP_UART_enableInterrupt(EUSCI_A0_BASE,  EUSCI_A_UART_RECEIVE_INTERRUPT);
    MAP_Interrupt_enableInterrupt(INT_EUSCIA0);
    MAP_Interrupt_enableSleepOnIsrExit();
    MAP_Interrupt_enableMaster();
```

28.3 Definitions

Data Structures

- struct _eUSCI_eUSCI_UART_Config

Functions

- void UART_clearInterruptFlag (uint32_t moduleInstance, uint_fast8_t mask)
- void UART_disableInterrupt (uint32_t moduleInstance, uint_fast8_t mask)
- void UART_disableModule (uint32_t moduleInstance)
- void UART_enableInterrupt (uint32_t moduleInstance, uint_fast8_t mask)
- void UART_enableModule (uint32_t moduleInstance)
- uint_fast8_t UART_getEnabledInterruptStatus (uint32_t moduleInstance)
- uint_fast8_t UART_getInterruptStatus (uint32_t moduleInstance, uint8_t mask)
- uint32_t UART_getReceiveBufferAddressForDMA (uint32_t moduleInstance)
- uint32_t UART_getTransmitBufferAddressForDMA (uint32_t moduleInstance)
- bool UART_initModule (uint32_t moduleInstance, const eUSCI_UART_Config *config)
- uint_fast8_t UART_queryStatusFlags (uint32_t moduleInstance, uint_fast8_t mask)
- uint8_t UART_receiveData (uint32_t moduleInstance)
- void UART_registerInterrupt (uint32_t moduleInstance, void(*intHandler)(void))
- void UART_resetDormant (uint32_t moduleInstance)
- void UART_selectDeglitchTime (uint32_t moduleInstance, uint32_t deglitchTime)
- void UART_setDormant (uint32_t moduleInstance)
- void UART_transmitAddress (uint32_t moduleInstance, uint_fast8_t transmitAddress)
- void UART_transmitBreak (uint32_t moduleInstance)
- void UART_transmitData (uint32_t moduleInstance, uint_fast8_t transmitData)
- void UART_unregisterInterrupt (uint32_t moduleInstance)

28.3.1 Detailed Description

The code for this module is contained in `uart/adc14.c`, with `driverlib/uart.h` containing the API declarations for use by applications.

28.3.2 Data Structure Documentation

28.3.2.1 struct _eUSCI_eUSCI_UART_Config

Type definition for _eUSCI_UART_Config structure.

ypedef eUSCI_eUSCI_UART_Config

Configuration structure for compare mode in the **UART** module. See UART_initModule for parameter documentation.

28.3.3 Function Documentation

28.3.3.1 void UART_clearInterruptFlag (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Clears UART interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
mask	is a bit mask of the interrupt sources to be cleared.

The UART interrupt source is cleared, so that it no longer asserts. The highest interrupt flag is automatically cleared when an interrupt vector generator is used.

The mask parameter has the same definition as the mask parameter to EUSCI_A_UART_enableInterrupt().

Modified register is **UCAxIFG**

Returns
None.

28.3.3.2 void UART_disableInterrupt (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Disables individual UART interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
mask	is the bit mask of the interrupt sources to be disabled.

Disables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The mask parameter is the logical OR of any of the following:

- **EUSCI_A_UART_RECEIVE_INTERRUPT** -Receive interrupt
- **EUSCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt
- **EUSCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable
- **EUSCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable

Modified register is **UCAxIFG**, **UCAxIE** and **UCAxCTL1**

Returns
None.

28.3.3.3 void UART_disableModule (uint32_t *moduleInstance*)

Disables the UART block.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

This will disable operation of the UART block.

Modified register is **UCAxCTL1**

Returns
None.

28.3.3.4 void UART_enableInterrupt (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Enables individual UART interrupt sources.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
mask	is the bit mask of the interrupt sources to be enabled.

Enables the indicated UART interrupt sources. The interrupt flag is first and then the corresponding interrupt is enabled. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The mask parameter is the logical OR of any of the following:

- **EUSCI_A_UART_RECEIVE_INTERRUPT** -Receive interrupt
- **EUSCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt
- **EUSCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable
- **EUSCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable

Modified register is **UCAxIFG**, **UCAxIE** and **UCAxCTL1**

Returns
None.

28.3.3.5 void UART_enableModule (uint32_t *moduleInstance*)

Enables the UART block.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

This will enable operation of the UART block.

Modified register is **UCAxCTL1**

Returns
None.

28.3.3.6 uint_fast8_t UART_getEnabledInterruptStatus (uint32_t *moduleInstance*)

Gets the current UART interrupt status masked with the enabled interrupts. This function is useful to call in ISRs to get a list of pending interrupts that are actually enabled and could have caused the ISR.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Returns
The current interrupt status as an ORed bit mask:
- **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG** -Receive interrupt flag
- **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG** - Transmit interrupt flag

References UART_getInterruptStatus().

28.3.3.7 uint_fast8_t UART_getInterruptStatus (uint32_t *moduleInstance,* uint8_t *mask*)

Gets the current UART interrupt status.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_STARTBIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT_FLAG**

Returns

The current interrupt status as an ORed bit mask:

- **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG** -Receive interrupt flag
- **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG** - Transmit interrupt flag

Referenced by UART_getEnabledInterruptStatus().

28.3.3.8 uint32_t UART_getReceiveBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the RX Buffer of the UART for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Returns the address of the UART RX Buffer. This can be used in conjunction with the DMA to store the received data directly to memory.

Returns

None

28.3.3.9 uint32_t UART_getTransmitBufferAddressForDMA (uint32_t *moduleInstance*)

Returns the address of the TX Buffer of the UART for the DMA module.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Returns the address of the UART TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Returns
None

28.3.3.10 bool UART_initModule (uint32_t *moduleInstance,* const **eUSCI_UART_Config** ∗ *config*)

Initialization routine for the UART block. The values to be written into the UCAxBRW and UCAxMCTLW registers should be pre-computed and passed into the initialization function

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE**
config	Configuration structure for the UART module

Configuration options for eUSCI_UART_Config structure.

It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode.

Parameters

selectClock-Source	selects Clock source. Valid values are ■ **EUSCI_A_UART_CLOCKSOURCE_SMCLK** ■ **EUSCI_A_UART_CLOCKSOURCE_ACLK**

clockPrescalar	is the value to be written into UCBRx bits
firstModReg	is First modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device User Guide.This value is written into UCBRFx bits of UCAxMCTLW.
secondModReg	is Second modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device User Guide. This value is written into UCBRSx bits of UCAxMCTLW.
parity	is the desired parity. Valid values are ■ **EUSCI_A_UART_NO_PARITY** [Default Value], ■ **EUSCI_A_UART_ODD_PARITY**, ■ **EUSCI_A_UART_EVEN_PARITY**
msborLsbFirst	controls direction of receive and transmit shift register. Valid values are ■ **EUSCI_A_UART_MSB_FIRST** ■ **EUSCI_A_UART_LSB_FIRST** [Default Value]
numberofStopBits	indicates one/two STOP bits Valid values are ■ **EUSCI_A_UART_ONE_STOP_BIT** [Default Value] ■ **EUSCI_A_UART_TWO_STOP_BITS**
uartMode	selects the mode of operation Valid values are ■ **EUSCI_A_UART_MODE** [Default Value], ■ **EUSCI_A_UART_IDLE_LINE_MULTI_PROCESSOR_MODE**, ■ **EUSCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR_MODE**, ■ **EUSCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION_MODE**
overSampling	indicates low frequency or oversampling baud generation Valid values are ■ **EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION** ■ **EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

Upon successful initialization of the UART block, this function will have initialized the module, but the UART block still remains disabled and must be enabled with UART_enableModule()

Refer to `this calculator` for help on calculating values for the parameters.

Modified bits are **UCPEN**, **UCPAR**, **UCMSB**, **UC7BIT**, **UCSPB**, **UCMODEx**, **UCSYNC** bits of **UCAxCTL0** and **UCSSELx**, **UCSWRST** bits of **UCAxCTL1**

Returns
true or false of the initialization process

28.3.3.11 uint_fast8_t UART_queryStatusFlags (uint32_t *moduleInstance,* uint_fast8_t *mask*)

Gets the current UART status flags.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
mask	is the masked interrupt flag status to be returned.

This returns the status for the UART module based on which flag is passed. mask parameter can be either any of the following selection.

- **EUSCI_A_UART_LISTEN_ENABLE**
- **EUSCI_A_UART_FRAMING_ERROR**
- **EUSCI_A_UART_OVERRUN_ERROR**
- **EUSCI_A_UART_PARITY_ERROR**
- **eUARTBREAK_DETECT**
- **EUSCI_A_UART_RECEIVE_ERROR**
- **EUSCI_A_UART_ADDRESS_RECEIVED**
- **EUSCI_A_UART_IDLELINE**
- **EUSCI_A_UART_BUSY**

Modified register is **UCAxSTAT**

Returns
the masked status flag

28.3.3.12 uint8_t UART_receiveData (uint32_t *moduleInstance*)

Receives a byte that has been sent to the UART Module.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

This function reads a byte of data from the UART receive data Register.

Modified register is **UCAxRXBUF**

Returns
Returns the byte received from by the UART module, cast as an uint8_t.

28.3.3.13 void UART_registerInterrupt (uint32_t *moduleInstance,* void(∗)(void) *intHandler*)

Registers an interrupt handler for UART interrupts.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode.
intHandler	is a pointer to the function to be called when the timer capture compare interrupt occurs.

This function registers the handler to be called when an UART interrupt occurs. This function enables the global interrupt in the interrupt controller; specific UART interrupts must be enabled via UART_enableInterrupt(). It is the interrupt handler's responsibility to clear the interrupt source via UART_clearInterruptFlag().

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

28.3.3.14 void UART_resetDormant (uint32_t *moduleInstance*)

Re-enables UART module from dormant mode

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Not dormant. All received characters set UCRXIFG.

Modified bits are **UCDORM** of **UCAxCTL1** register.

Returns
None.

28.3.3.15 void UART_selectDeglitchTime (uint32_t *moduleInstance,* uint32_t *deglitchTime*)

Sets the deglitch time

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
deglitchTime	is the selected deglitch time Valid values are ■ **EUSCI_A_UART_DEGLITCH_TIME_2ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_50ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_100ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_200ns**

Returns the address of the UART TX Buffer. This can be used in conjunction with the DMA to obtain transmitted data directly from memory.

Returns
None

28.3.3.16 void UART_setDormant (uint32_t *moduleInstance*)

Sets the UART module in dormant mode

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Puts USCI in sleep mode Only characters that are preceded by an idle-line or with address bit set UCRXIFG. In UART mode with automatic baud-rate detection, only the combination of a break and synch field sets UCRXIFG.

Modified register is **UCAxCTL1**

Returns
None.

28.3.3.17 void UART_transmitAddress (uint32_t *moduleInstance,* uint_fast8_t *transmitAddress*)

Transmits the next byte to be transmitted marked as address depending on selected multiprocessor mode

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
transmitAddress	is the next byte to be transmitted

Modified register is **UCAxCTL1**, **UCAxTXBUF**

Returns
None.

28.3.3.18 void UART_transmitBreak (uint32_t *moduleInstance*)

Transmit break. Transmits a break with the next write to the transmit buffer. In UART mode with automatic baud-rate detection, EUSCI_A_UART_AUTOMATICBAUDRATE_SYNC(0x55) must be written into UCAxTXBUF to generate the required break/synch fields. Otherwise, DEFAULT_SYNC(0x00) must be written into the transmit buffer. Also ensures module is ready for transmitting the next data

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such asEUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode

Modified register is **UCAxCTL1**, **UCAxTXBUF**

Returns
None.

28.3.3.19 void UART_transmitData (uint32_t *moduleInstance,* uint_fast8_t *transmitData*)

Transmits a byte from the UART Module.

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode
transmitData	data to be transmitted from the UART module

This function will place the supplied data into UART transmit data register to start transmission

Modified register is **UCAxTXBUF**

Returns
None.

28.3.3.20 void UART_unregisterInterrupt (uint32_t *moduleInstance*)

Unregisters the interrupt handler for the UART module

Parameters

moduleInstance	is the instance of the eUSCI A (UART) module. Valid parameters vary from part to part, but can include: ■ **EUSCI_A0_BASE** ■ **EUSCI_A1_BASE** ■ **EUSCI_A2_BASE** ■ **EUSCI_A3_BASE** It is important to note that for eUSCI modules, only "A" modules such as EUSCI_A0 can be used. "B" modules such as EUSCI_B0 do not support the UART mode.

This function unregisters the handler to be called when timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

29 Watchdog Timer (WDT_A)

29.1 Module Operation

MSP432 includes a standard watchdog module that is identical to the WDT_A module of MSP430. By using DriverLib, the user can configure all aspects of the watchdog peripheral including using the watchdog in interval mode as well as watchdog mode.

29.2 Watchdog Mode

Once the module is initiated in watchdog mode, the timer will reset part if the count expires. The reset can be set as either a soft or hard reset. This use case is useful when the programmer wants to make sure that the code execution isn't perpetually stuck/locked in an unrecoverable state.

To configure the WDT module in watchdog mode, the WDT_initWatchdogTimer function is used such as follows:

```
/* Configuring WDT to timeout after 512k iterations of SMCLK,  at 128k,
 * this will roughly equal 4 seconds*/
MAP_SysCtl_A_setWDTTimeoutResetType(SYSCTL_A_SOFT_RESET);
MAP_WDT_A_initWatchdogTimer(WDT_A_CLOCKSOURCE_SMCLK,
                            WDT_A_CLOCKITERATIONS_512K);
```

This will set the watchdog timer to be sourced from SMCLK and have a duration of 512, 000 SMCLK cycles. This means that once started, if the watchdog timer goes 512, 000 iterations without being reset a reset will occur. To reset the counter (after using WDT_startTimer to start the timer), the user should use the WDT_resetTimer function.

29.3 Interval Mode

MSP432 Driverlib can also configure the WDT module to work in interval mode. This turns the WDT into an ordinary 16-bit down counter with interrupt support. This can be used if the user needs access to another low power counter, however has already used other resources. To configure the module in interval mode, use the WDT_initIntervalTimer function such as follows:

```
/* Configuring WDT in interval mode to trigger every 32K clock iterations.
 * This comes out to roughly every 3.5 seconds */
MAP_WDT_A_initIntervalTimer(WDT_A_CLOCKSOURCE_VLOCLK,
                            WDT_A_CLOCKITERATIONS_32K);
```

This will configure the WDT module to be sourced from SMCLK and have a period of 32, 000 cycles. In this example, we have previously configured SMCLK to be 64Khz making this timer's period be approximately half a second. After using the WDT_startTimer function to start the timer, the user can service interrupts from interval mode after enabling interrupts using the Interrupt_enableInterrupt function.

29.4 Setting Reset Type

The type of reset that occurs on watchdog timeout/password violation can be configured through the SysCtl module using the SysCtl_setWDTPasswordViolationResetType and SysCtl_setWDTTimeoutResetType APIs. These APIs will allow the user to change whether a soft or hard reset occurs on a watchdog timeout and password violation. For the user, the convenience functions WDT_setPasswordViolationReset and WDT_setTimeoutReset exist in the WDT APIs.

29.5 Programming Example

The DriverLib package contains a variety of different code examples that demonstrate the usage of the WDT module. These code examples are accessible under the examples/ folder of the SDK release as well as through TI Resource Explorer if using Code Composer Studio. These code examples provide a comprehensive list of use cases as well as practical applications involving each module.

Below is a very brief code example showing how to configure the WDT module in interval mode:

```
/* Configuring WDT in interval mode to trigger every 32K clock iterations.
 * This comes out to roughly every 3.5 seconds */
MAP_WDT_A_initIntervalTimer(WDT_A_CLOCKSOURCE_VLOCLK,
                                    WDT_A_CLOCKITERATIONS_32K);
```

29.6 Definitions

Functions

- void WDT_A_clearTimer (void)
- void WDT_A_holdTimer (void)
- void WDT_A_initIntervalTimer (uint_fast8_t clockSelect, uint_fast8_t clockDivider)
- void WDT_A_initWatchdogTimer (uint_fast8_t clockSelect, uint_fast8_t clockDivider)
- void WDT_A_registerInterrupt (void(*intHandler)(void))
- void WDT_A_setPasswordViolationReset (uint_fast8_t resetType)
- void WDT_A_setTimeoutReset (uint_fast8_t resetType)
- void WDT_A_startTimer (void)
- void WDT_A_unregisterInterrupt (void)

29.6.1 Detailed Description

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

29.6.2 Function Documentation

29.6.2.1 void WDT_A_clearTimer (void)

Clears the timer counter of the Watchdog Timer.

This function clears the watchdog timer count to 0x0000h. This function is used to "service the dog" when operating in watchdog mode.

Returns
None

29.6.2.2 void WDT_A_holdTimer (void)

Holds the Watchdog Timer.

This function stops the watchdog timer from running. This way no interrupt or PUC is asserted.

Returns
None

Referenced by PCM_gotoLPM4().

29.6.2.3 void WDT_A_initIntervalTimer (uint_fast8_t *clockSelect,* uint_fast8_t *clockDivider*)

Sets the clock source for the Watchdog Timer in timer interval mode.

Parameters

clockSelect	is the clock source that the watchdog timer will use. Valid values are ■ **WDT_A_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_A_CLOCKSOURCE_ACLK** ■ **WDT_A_CLOCKSOURCE_VLOCLK** ■ **WDT_A_CLOCKSOURCE_BCLK**
clockIterations	is the number of clock iterations for a watchdog interval. Valid values are ■ **WDT_A_CLOCKITERATIONS_2G** [Default] ■ **WDT_A_CLOCKITERATIONS_128M** ■ **WDT_A_CLOCKITERATIONS_8192K** ■ **WDT_A_CLOCKITERATIONS_512K** ■ **WDT_A_CLOCKITERATIONS_32K** ■ **WDT_A_CLOCKITERATIONS_8192** ■ **WDT_A_CLOCKITERATIONS_512** ■ **WDT_A_CLOCKITERATIONS_64**

This function sets the watchdog timer as timer interval mode, which will assert an interrupt without causing a PUC.

Returns
None

29.6.2.4 void WDT_A_initWatchdogTimer (uint_fast8_t *clockSelect,* uint_fast8_t *clockDivider*)

Sets the clock source for the Watchdog Timer in watchdog mode.

Parameters

clockSelect	is the clock source that the watchdog timer will use. Valid values are ■ **WDT_A_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_A_CLOCKSOURCE_ACLK** ■ **WDT_A_CLOCKSOURCE_VLOCLK** ■ **WDT_A_CLOCKSOURCE_BCLK**
clockIterations	is the number of clock iterations for a watchdog timeout. Valid values are ■ **WDT_A_CLOCKITERATIONS_2G** [Default] ■ **WDT_A_CLOCKITERATIONS_128M** ■ **WDT_A_CLOCKITERATIONS_8192K** ■ **WDT_A_CLOCKITERATIONS_512K** ■ **WDT_A_CLOCKITERATIONS_32K** ■ **WDT_A_CLOCKITERATIONS_8192** ■ **WDT_A_CLOCKITERATIONS_512** ■ **WDT_A_CLOCKITERATIONS_64**

This function sets the watchdog timer in watchdog mode, which will cause a PUC when the timer overflows. When in the mode, a PUC can be avoided with a call to WDT_A_resetTimer() before the timer runs out.

Returns
None

29.6.2.5 void WDT_A_registerInterrupt (void(∗)(void) *intHandler*)

Registers an interrupt handler for the watchdog interrupt.

Parameters

intHandler	is a pointer to the function to be called when the watchdog interrupt occurs.

Returns
None.

References Interrupt_enableInterrupt(), and Interrupt_registerInterrupt().

29.6.2.6 void WDT_A_setPasswordViolationReset (uint_fast8_t *resetType*)

Sets the type of RESET that happens when a watchdog password violation occurs.

Parameters

resetType	The type of reset to set

The *resetType* parameter must be only one of the following values:

- **WDT_A_HARD_RESET**
- **WDT_A_SOFT_RESET**

Returns
None.

29.6.2.7 void WDT_A_setTimeoutReset (uint_fast8_t *resetType*)

Sets the type of RESET that happens when a watchdog timeout occurs.

Parameters

resetType	The type of reset to set

The *resetType* parameter must be only one of the following values:

- **WDT_A_HARD_RESET**
- **WDT_A_SOFT_RESET**

Returns
None.

29.6.2.8 void WDT_A_startTimer (void)

Starts the Watchdog Timer.

This function starts the watchdog timer functionality to start counting.

Returns
None

29.6.2.9 void WDT_A_unregisterInterrupt (void)

Unregisters the interrupt handler for the watchdog.

This function unregisters the handler to be called when a watchdog interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See Also
Interrupt_registerInterrupt() for important information about registering interrupt handlers.

Returns
None.

References Interrupt_disableInterrupt(), and Interrupt_unregisterInterrupt().

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265