MSP430 DriverLib for MSP430FR2xx_4xx Devices

User's Guide

DOCNUM-2.91.11.01

Copyright

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

Revision Information

This is version 2.91.11.01 of this document, last updated on Wed Feb 20 2019 12:05:21.

Table of Contents

1 Introduction

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430 FR2xx/FR4xx family of microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

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

- They are written entirely in C except where absolutely not possible.
- They demonstrate how to use the peripheral in its common mode of operation.
- They are easy to understand.
- They are reasonably efficient in terms of memory and processor usage.
- They are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They can be built with more than one tool chain.

Some consequences of these design goals are:

- The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.
- The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which cannot be utilized by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.
- The APIs have a means of removing all error checking code. Because the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

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

Each MSP430ware driverlib API takes in the base address of the corresponding peripheral as the first parameter. This base address is obtained from the msp430 device specific header files (or from the device datasheet). The example code for the various peripherals show how base address is used. When using CCS, the eclipse shortcut "Ctrl + Space" helps. Type __MSP430 and "Ctrl + Space", and the list of base addresses from the included device specific header files is listed.

The following tool chains are supported:

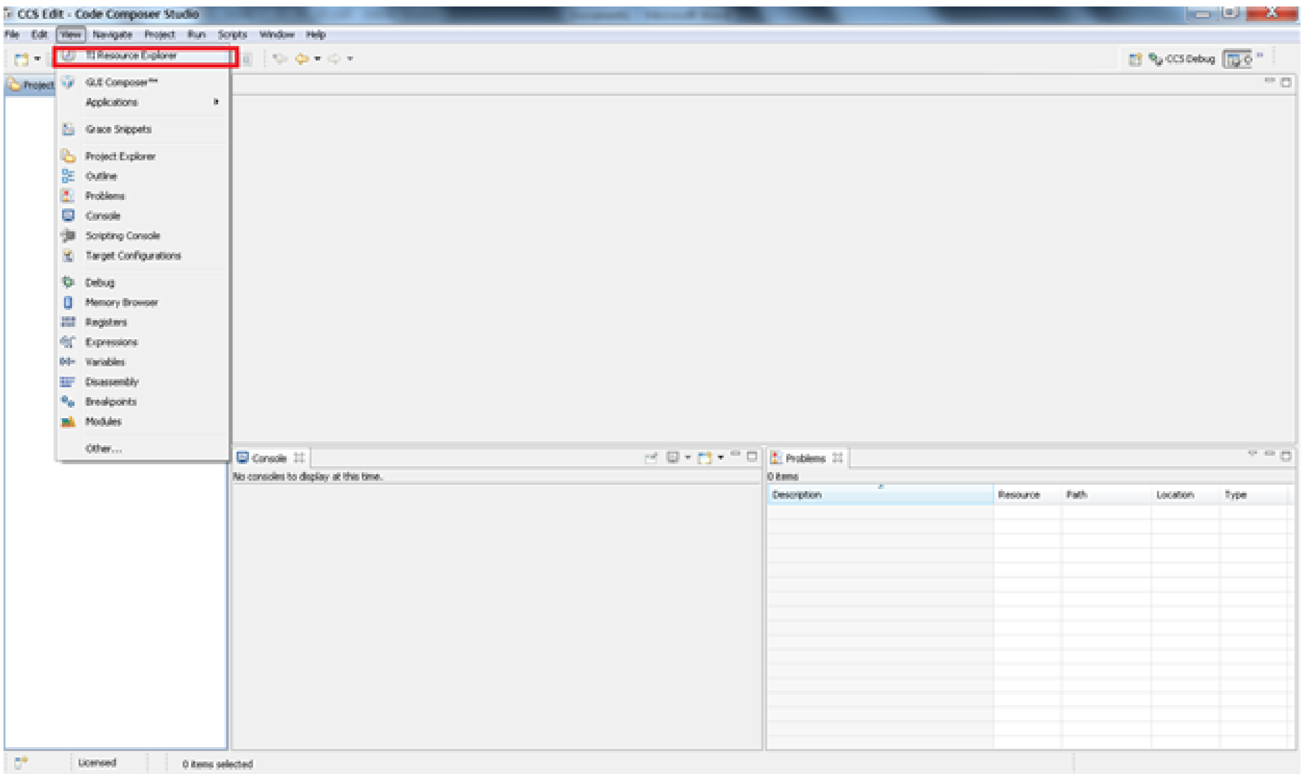
- IAR Embedded Workbench®
- Texas Instruments Code Composer Studio™

Using assert statements to debug

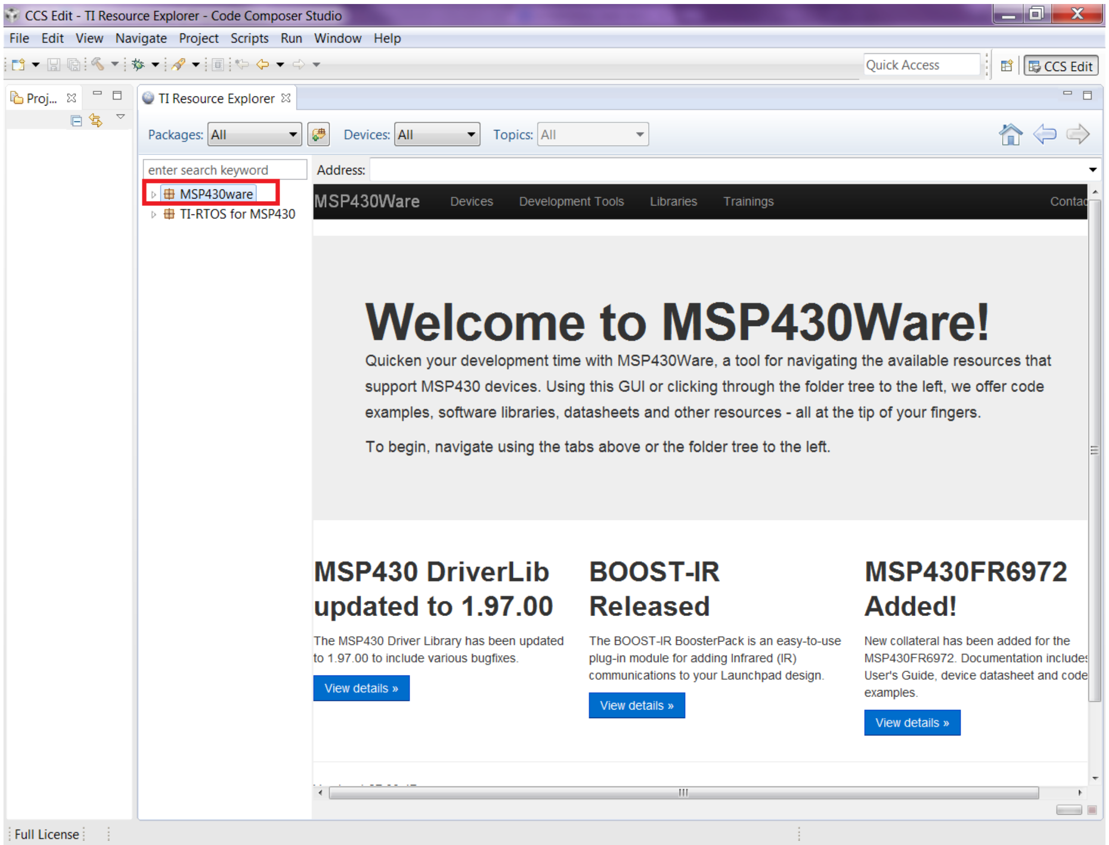
Assert statements are disabled by default. To enable the assert statement edit the hw_regaccess.h file in the inc folder. Comment out the statement #define NDEBUG -> //#define NDEBUG Asserts in CCS work only if the project is optimized for size.

2 Navigating to driverlib through CCS Resource Explorer

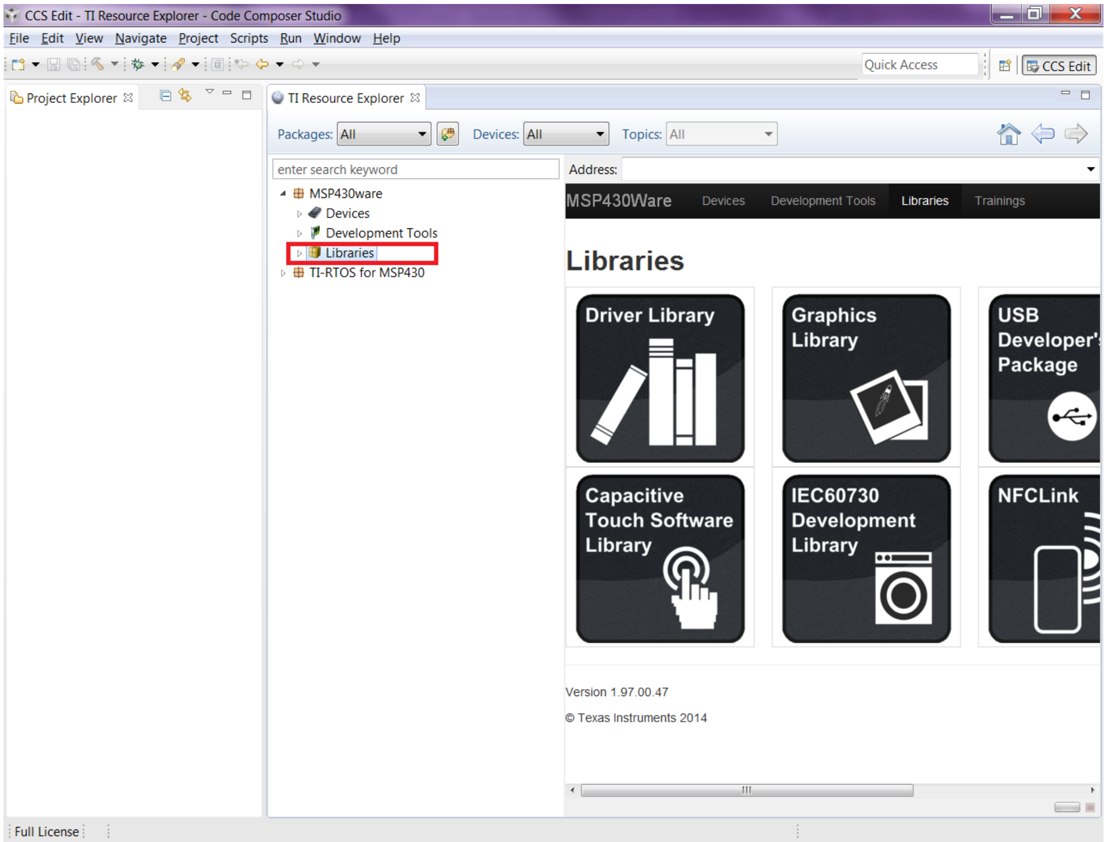
In CCS, click View->TI Resource Explorer

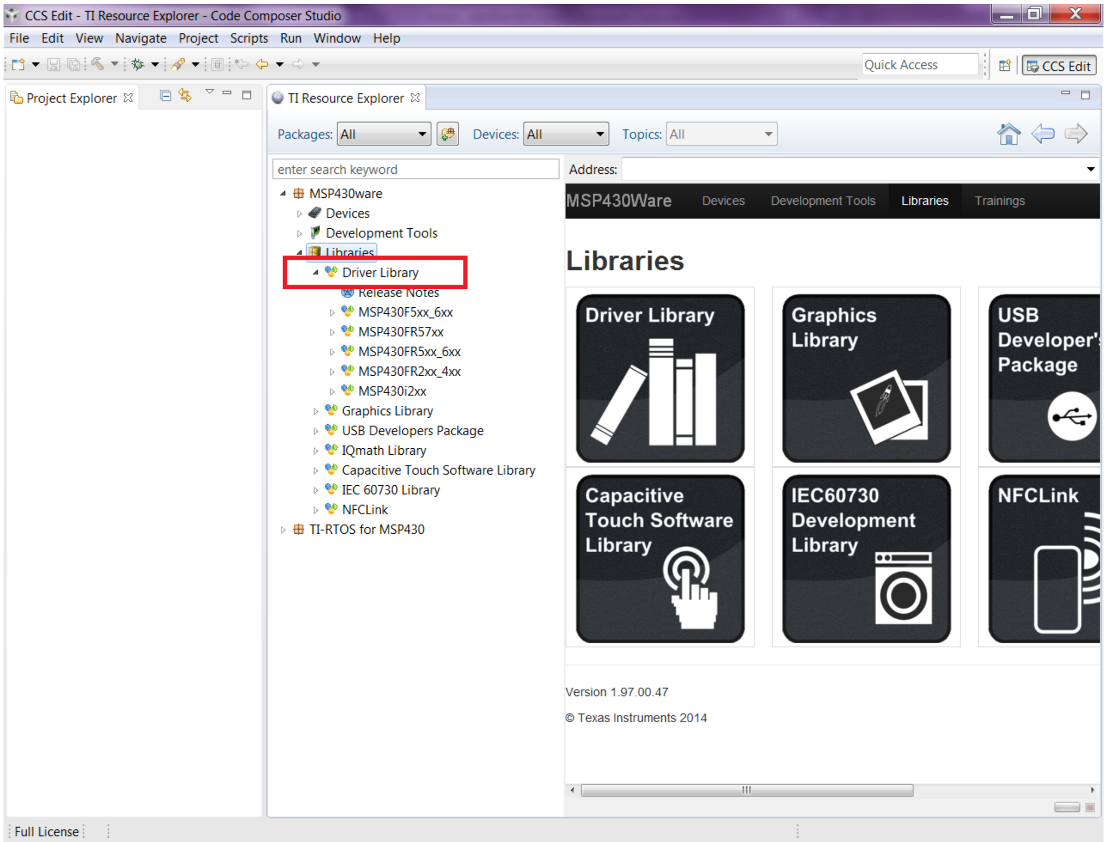


In Resource Explorer View, click on MSP430ware

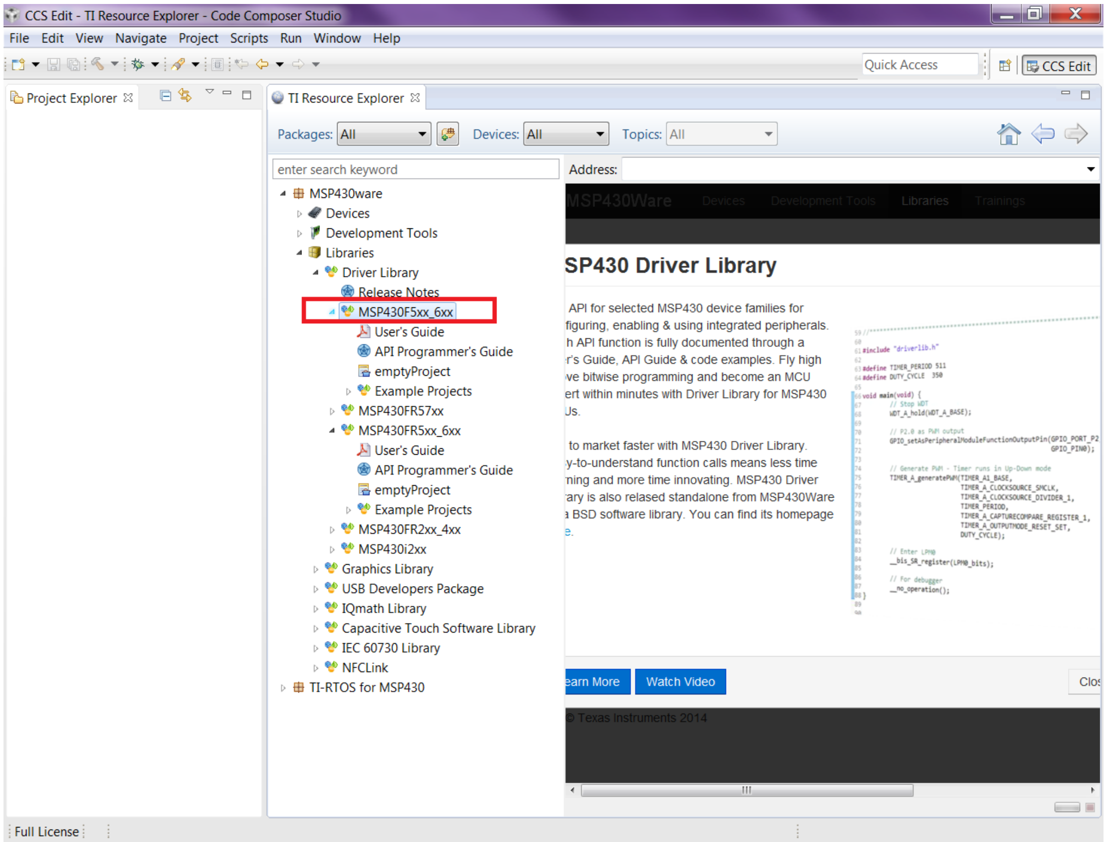


Clicking MSP430ware takes you to the introductory page. The version of the latest MSP430ware installed is available in this page. In this screenshot the version is 1.30.00.15 The various software, collateral, code examples, datasheets and user guides can be navigated by clicking the different topics under MSP430ware. To proceed to driverlib, click on Libraries->Driverlib as shown in the next two screenshots.

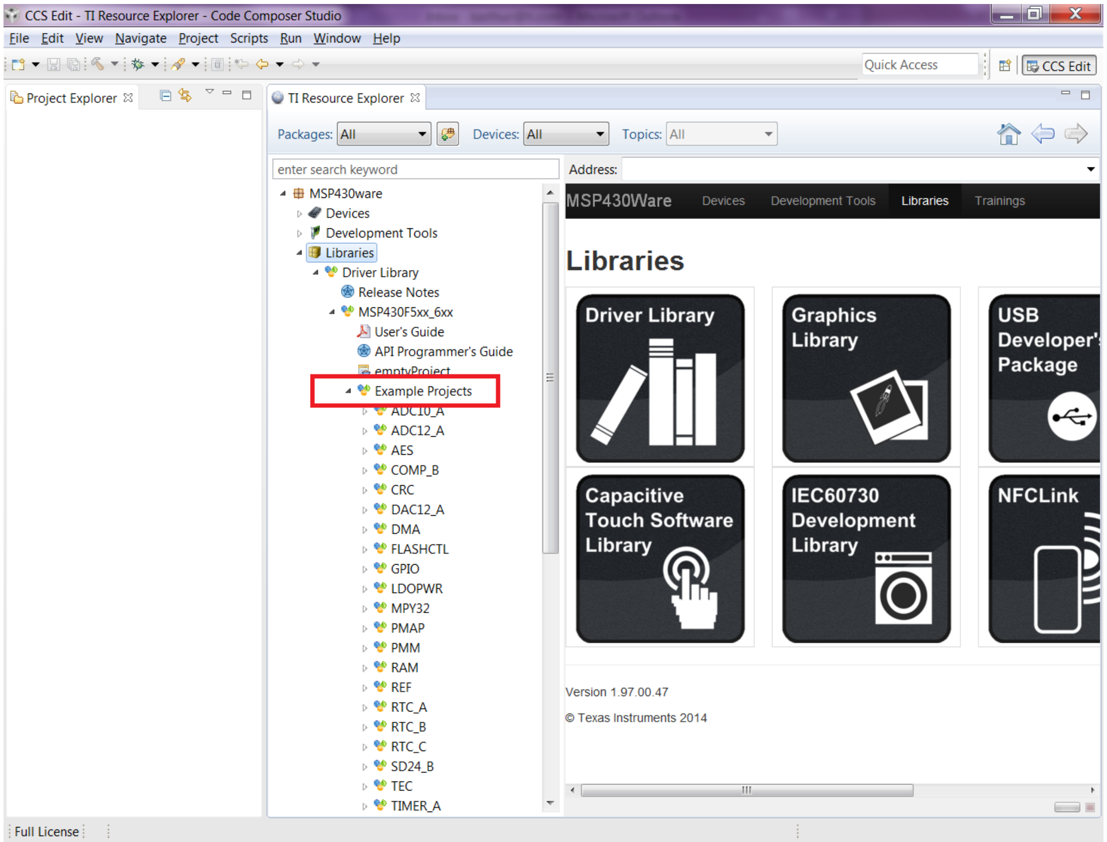

CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Scripts Run Window Help
Quick Access
CCS Edit
Project Explorer
TI Resource Explorer
Packages: All
Devices: All
Topics: All
enter search keyword
Address:
MSP430ware
Devices
Development Tools
Libraries
TI-RTOS for MSP430
MSP430Ware
Devices
Development Tools
Libraries
Trainings
Libraries
Driver Library
Graphics Library
USB Developer's Package
Capacitive Touch Software Library
IEC60730 Development Library
NFCLink
Version 1.97.00.47
© Texas Instruments 2014
Full License



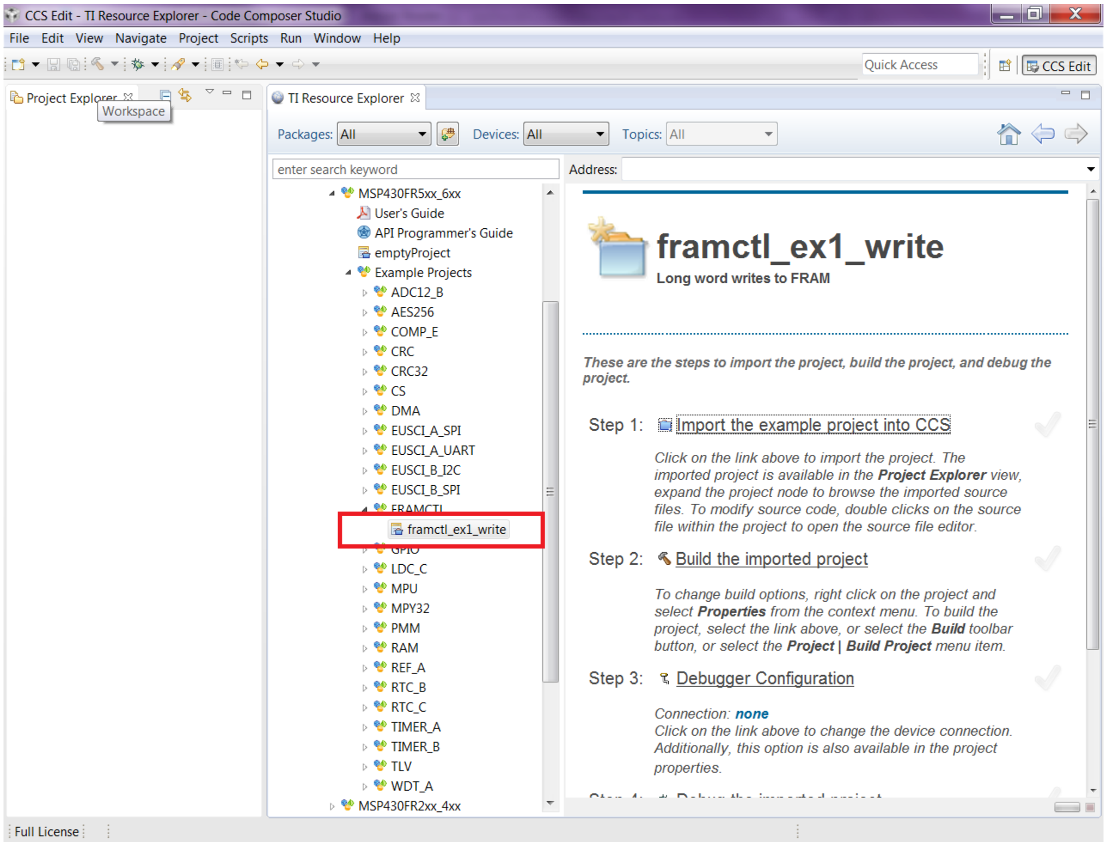
Driverlib is designed per Family. If a common device family user's guide exists for a group of devices, these devices belong to the same 'family'. Currently driverlib is available for the following family of devices. MSP430F5xx_6xx MSP430FR57xx MSP430FR2xx_4xx MSP430FR5xx_6xx MSP430i2xx



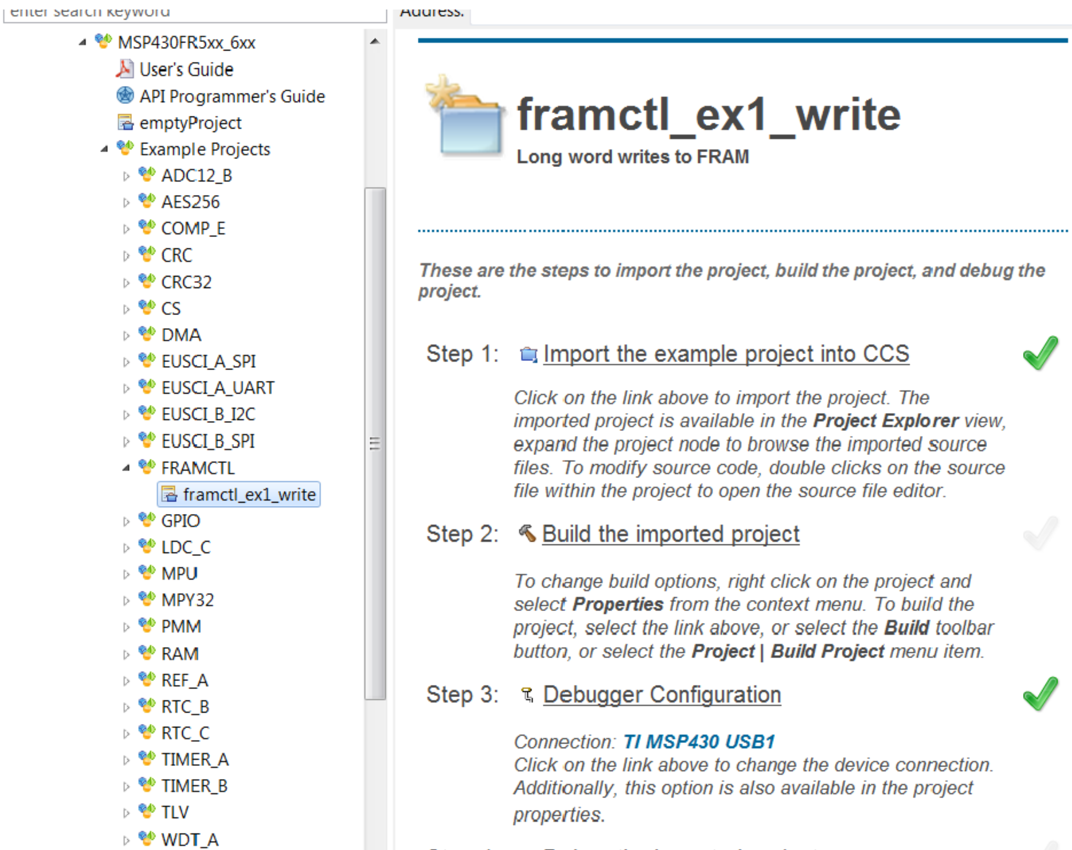
Click on the MSP430F5xx_6xx to navigate to the driverlib based example code for that family.



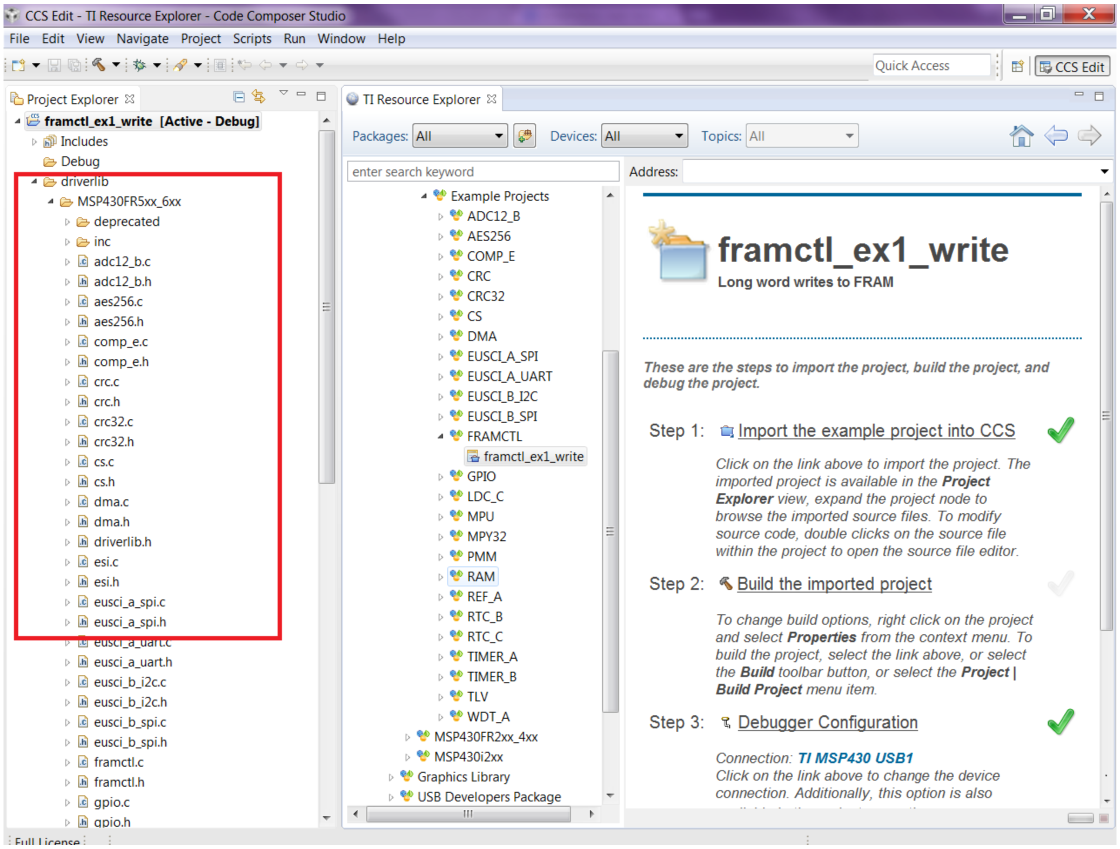
The various peripherals are listed in alphabetical order. The names of peripherals are as in device family user's guide. Clicking on a peripheral name lists the driverlib example code for that peripheral. The screenshot below shows an example when the user clicks on GPIO peripheral.



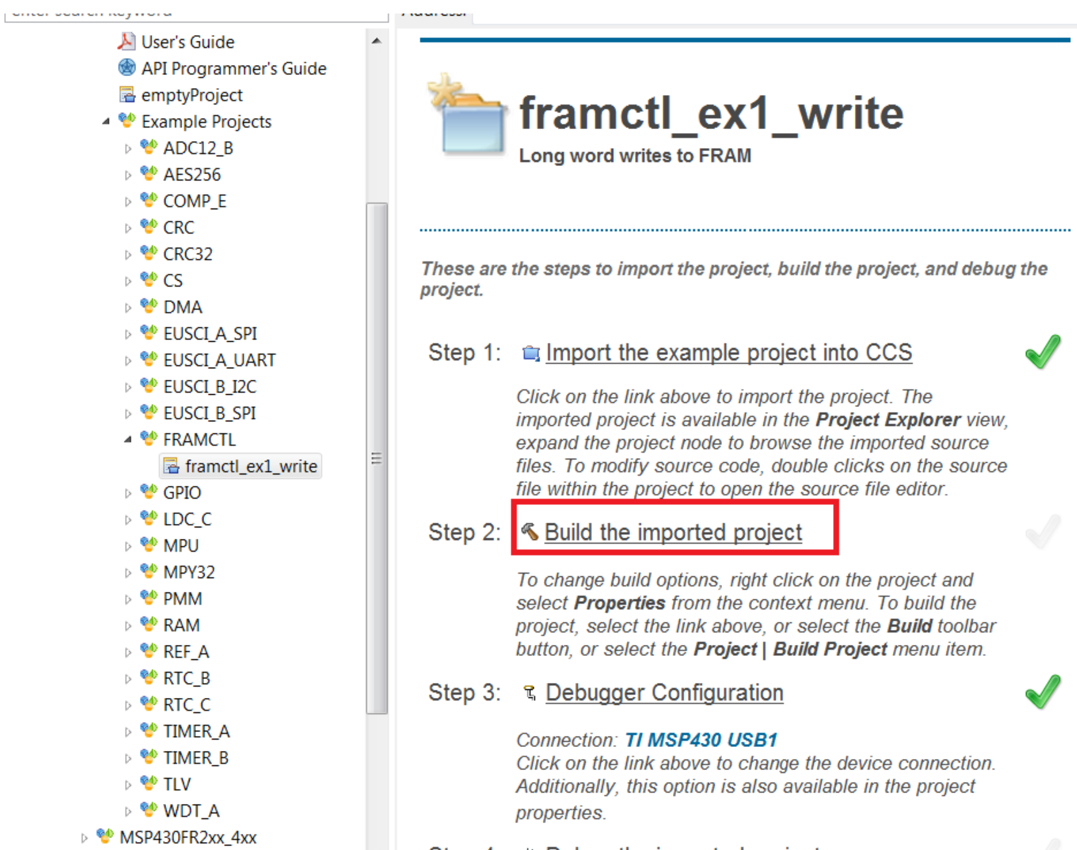
Now click on the specific example you are interested in. On the right side there are options to Import/Build/Download and Debug. Import the project by clicking on the "Import the example project into CCS"



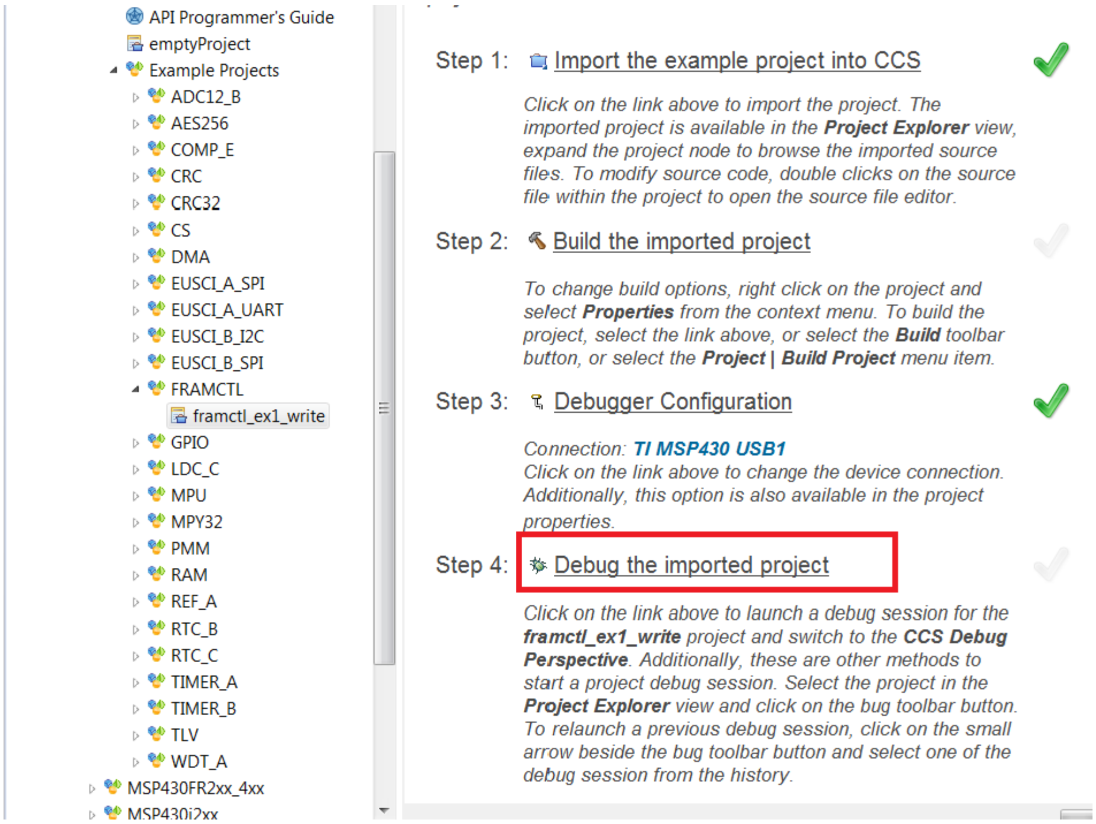
The imported project can be viewed on the left in the Project Explorer. All required driverlib source and header files are included inside the driverlib folder. All driverlib source and header files are linked to the example projects. So if the user modifies any of these source or header files, the original copy of the installed MSP430ware driverlib source and header files get modified.



Now click on Build the imported project on the right to build the example project.

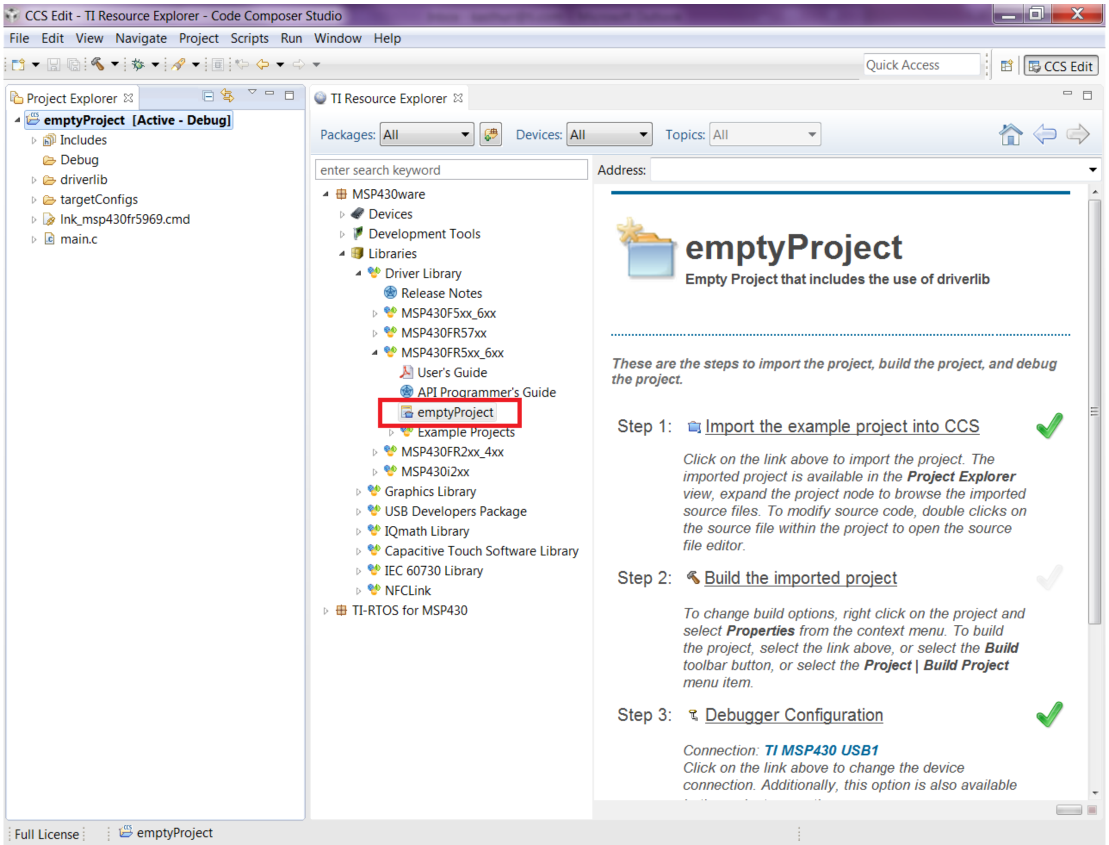


Now click on Build the imported project on the right to build the example project.



The COM port to download to can be changed using the Debugger Configuration option on the right if required.

To get started on a new project we recommend getting started on an empty project we provide. This project has all the driverlib source files, header files, project paths are set by default.

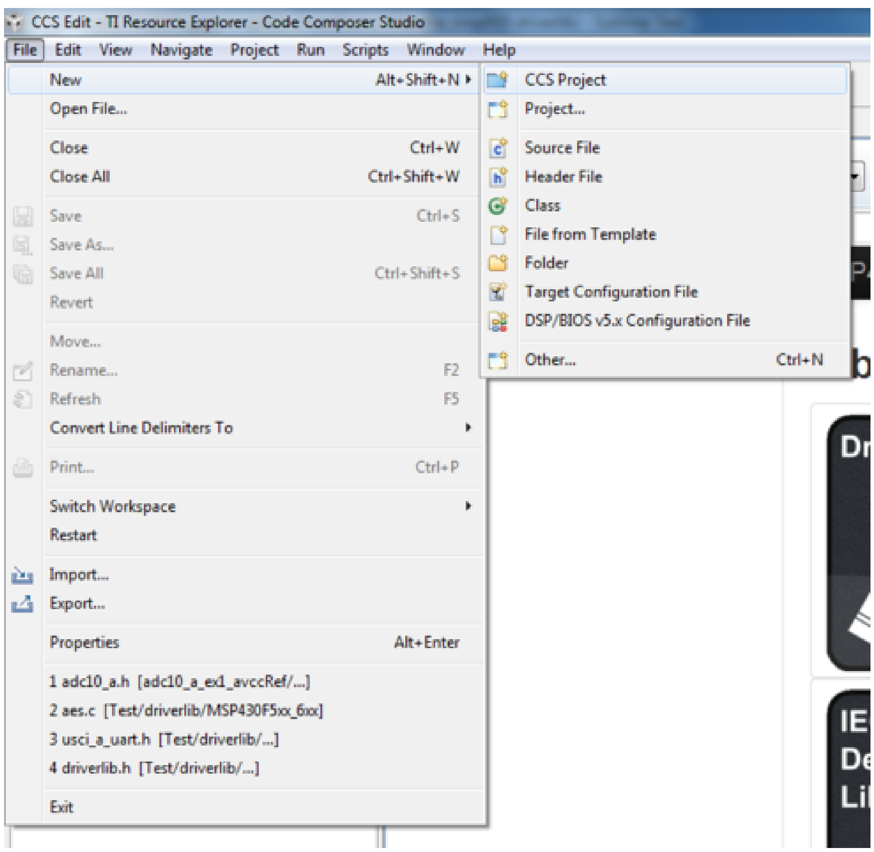


The main.c included with the empty project can be modified to include user code.

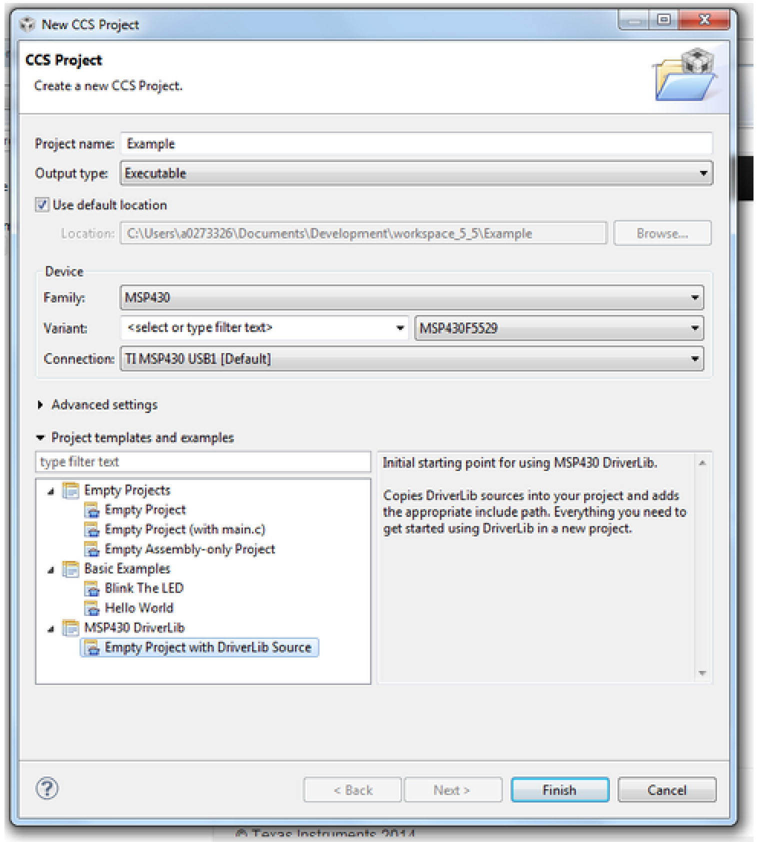
3 How to create a new CCS project that uses Driverlib

3.1 Introduction

To get started on a new project we recommend using the new project wizard. For driver library to work with the new project wizard CCS must have discovered the driver library RTSC product. For more information refer to the installation steps of the release notes. The new project wizard adds the needed driver library source files and adds the driver library include path.
To open the new project wizard go to File -> New -> CCS Project as seen in the screenshot below.



Once the new project wizard has been opened name your project and choose the device you would like to create a Driver Library project for. The device must be supported by driver library. Then under "Project templates and examples" choose "Empty Project with DriverLib Source" as seen below.



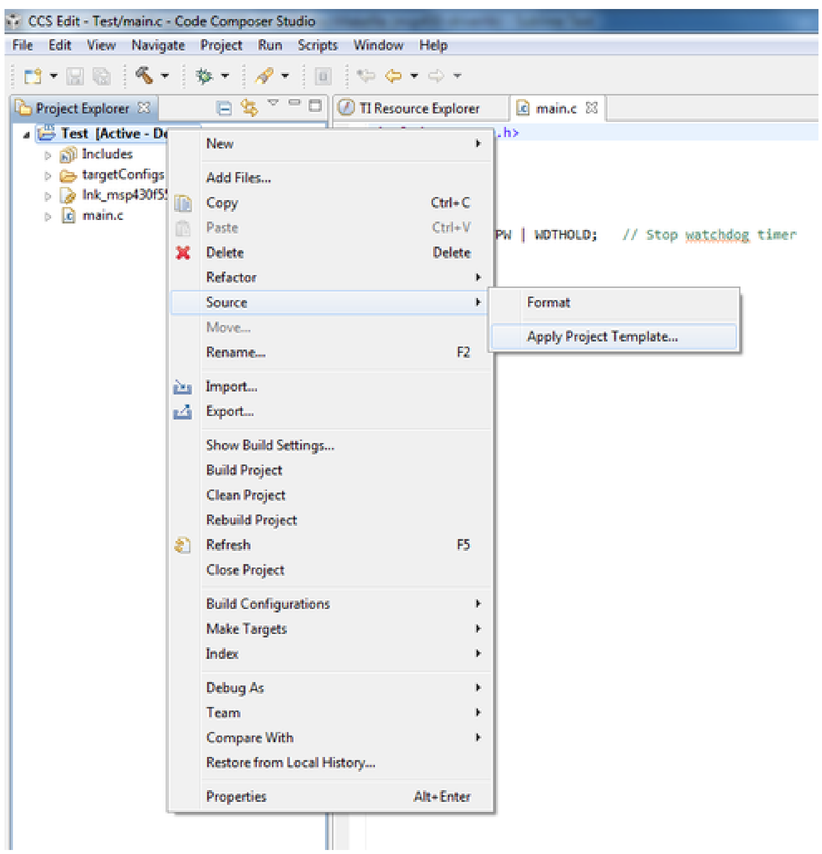
Finally click "Finish" and begin developing with your Driver Library enabled project.

We recommend -O4 compiler settings for more efficient optimizations for projects using driverlib

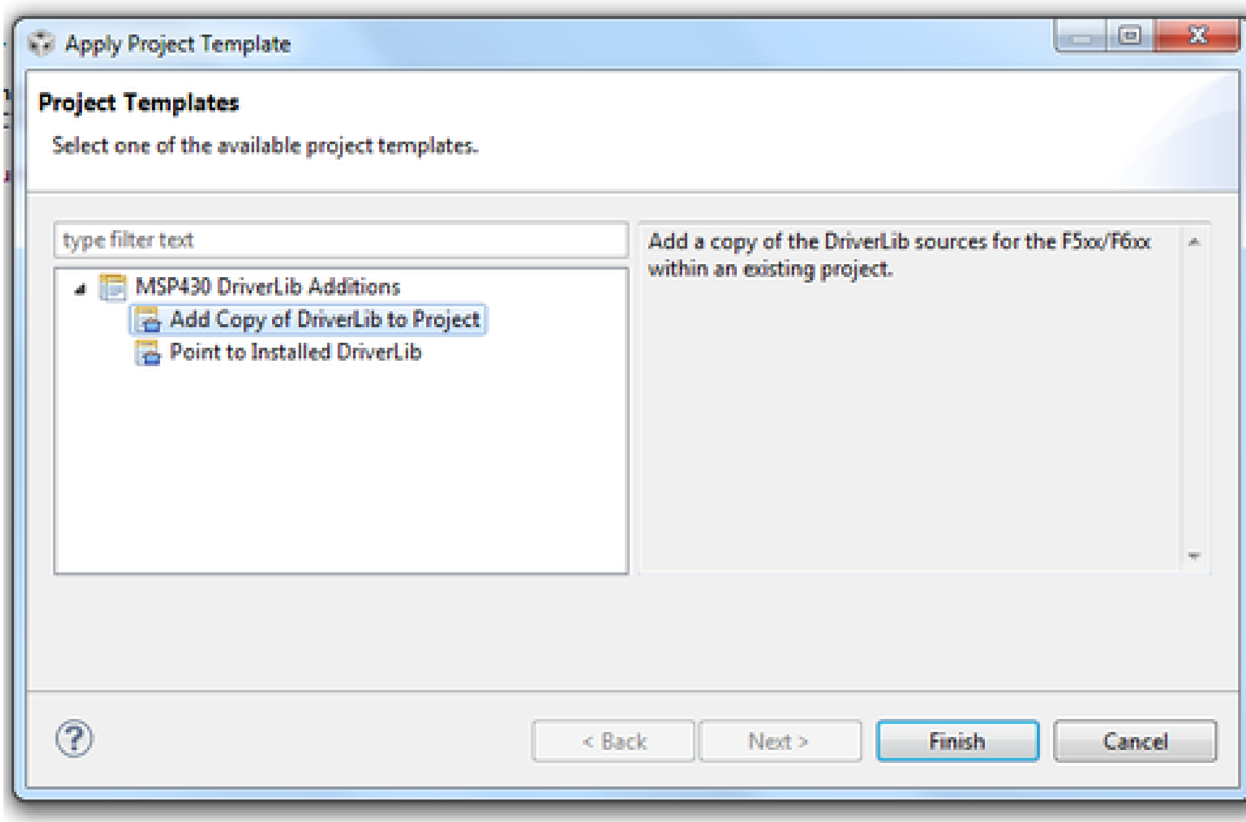
4 How to include driverlib into your existing CCS project

4.1 Introduction

To add driver library to an existing project we recommend using CCS project templates. For driver library to work with project templates CCS must have discovered the driver library RTSC product. For more information refer to the installation steps of the release notes. CCS project templates adds the needed driver library source files and adds the driver library include path.
To apply a project template right click on an existing project then go to Source -> Apply Project Template as seen in the screenshot below.



In the "Apply Project Template" dialog box under "MSP430 DriverLib Additions" choose either "Add Local Copy" or "Point to Installed DriverLib" as seen in the screenshot below. Most users will want to add a local copy which copies the DriverLib source into the project and sets the compiler settings needed.
Pointing to an installed DriverLib is for advandced users who are including a static library in their project and want to add the DriverLib header files to their include path.

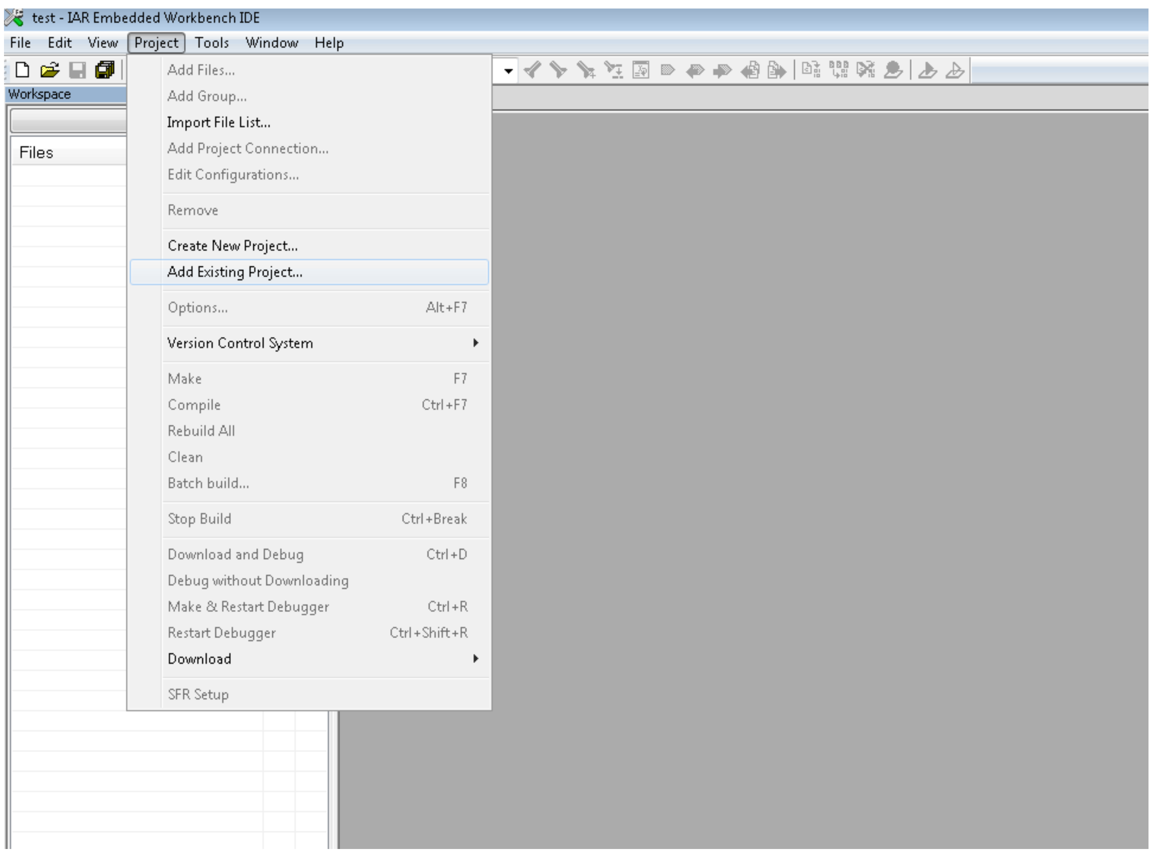


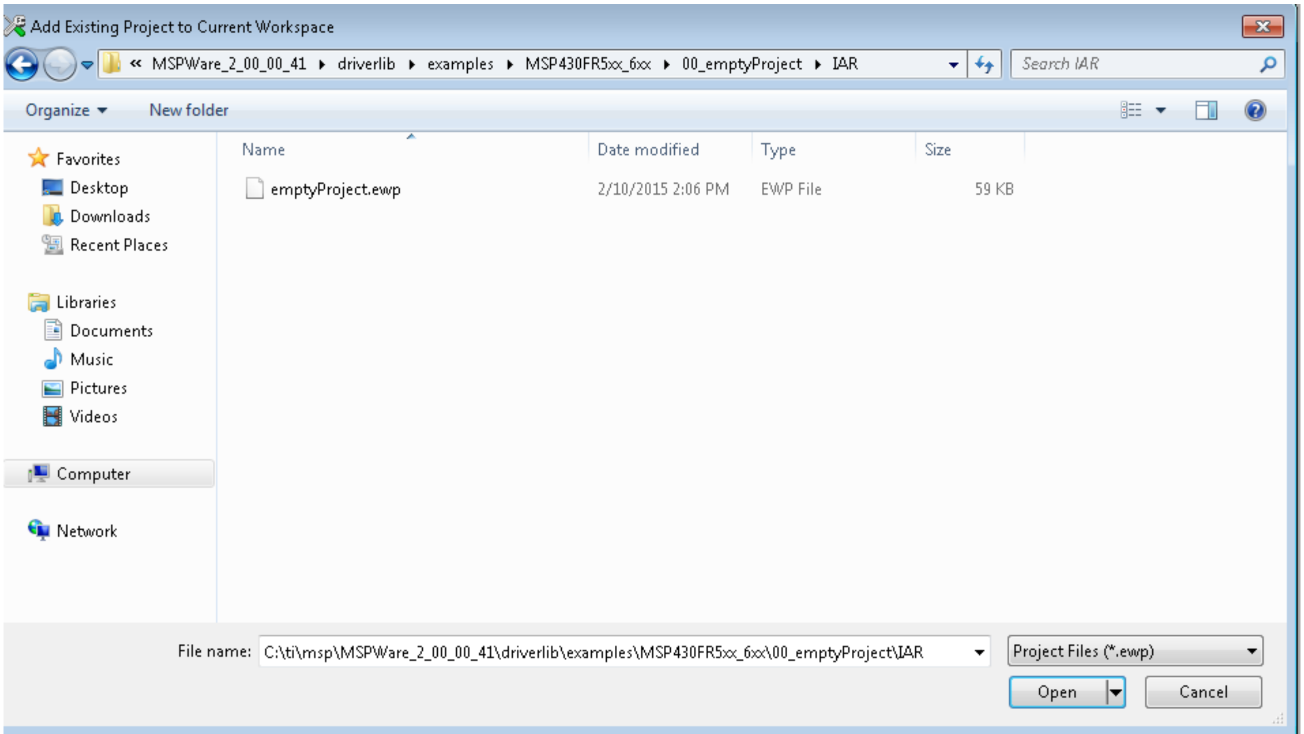
Click "Finish" and start developing with driver library in your project.

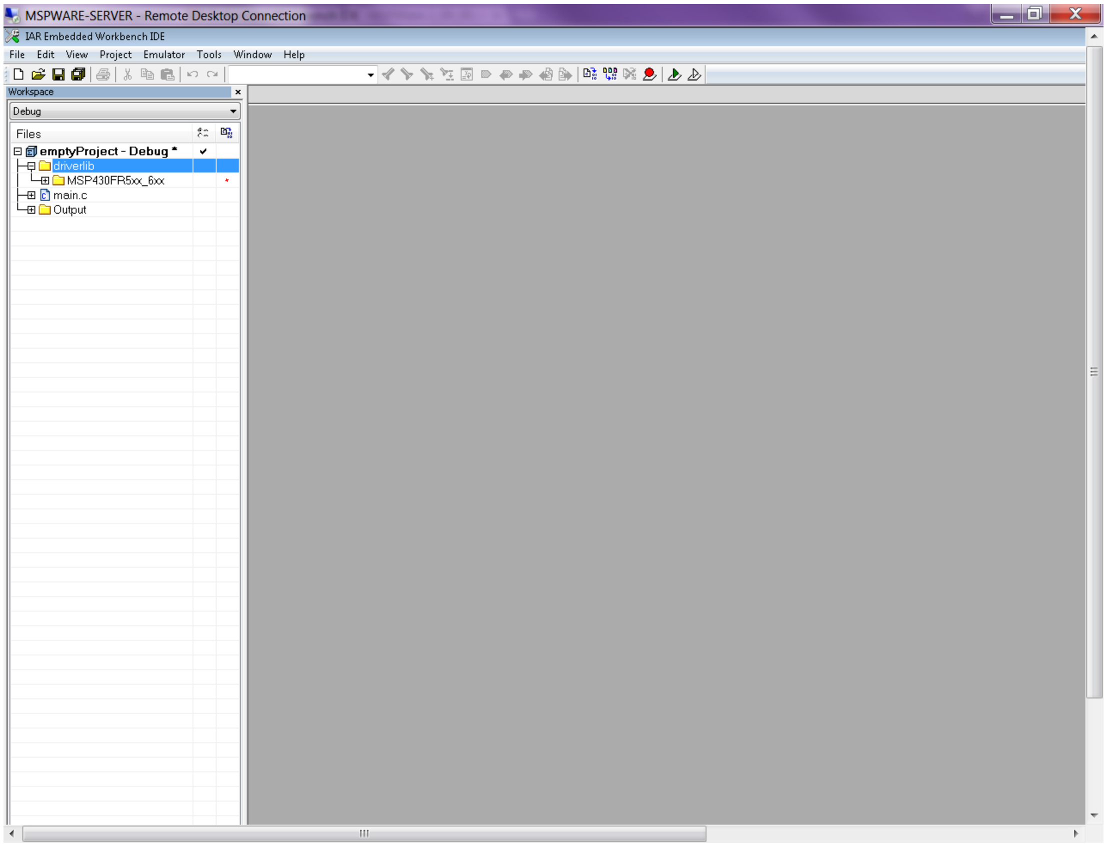
5 How to create a new IAR project that uses Driverlib

5.1 Introduction

It is recommended to get started with an Empty Driverlib Project. Browse to the empty project in your device's family. This is available in the driverlib instal folder\00_emptyProject



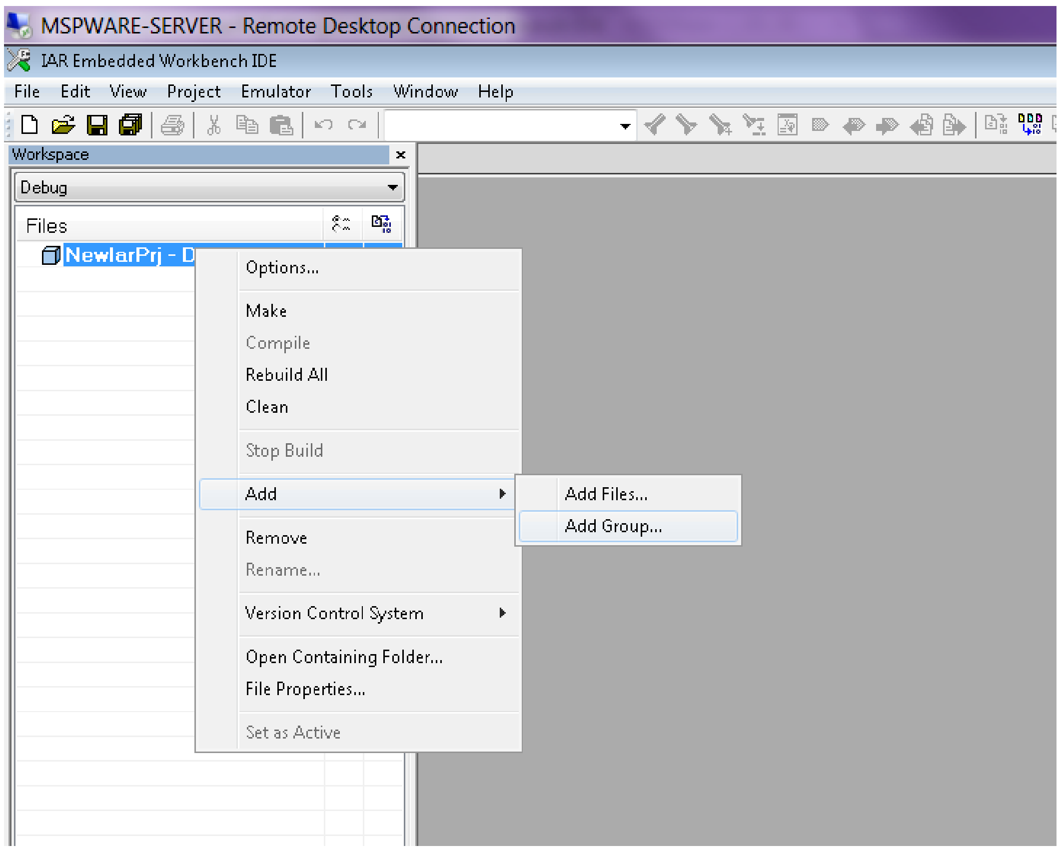

Add Existing Project to Current Workspace
« MSPWare_2_00_00_41 ▸ driverlib ▸ examples ▸ MSP430FR5xx_6xx ▸ 00_emptyProject ▸ IAR
Search IAR
Organize
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Network
Name
Date modified
Type
Size
emptyProject.ewp
2/10/2015 2:06 PM
EWP File
59 KB
File name:
C:\ti\msp\MSPWare_2_00_00_41\driverlib\examples\MSP430FR5xx_6xx\00_emptyProject\IAR
Project Files (*.ewp)
Open
Cancel


MSPWARE-SERVER - Remote Desktop Connection
IAR Embedded Workbench IDE
File Edit View Project Emulator Tools Window Help
Workspace
Debug
Files
emptyProject - Debug *
driverlib
MSP430FR5xx_6xx
main.c
Output

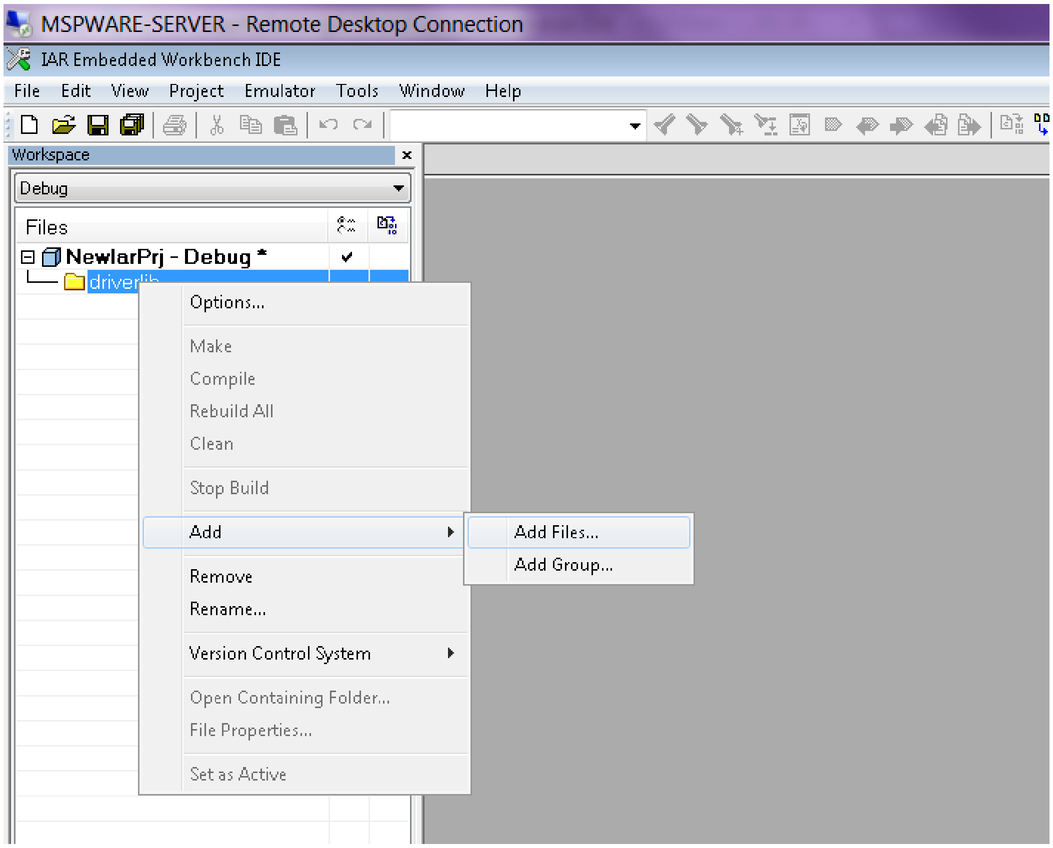
6 How to include driverlib into your existing IAR project

6.1 Introduction

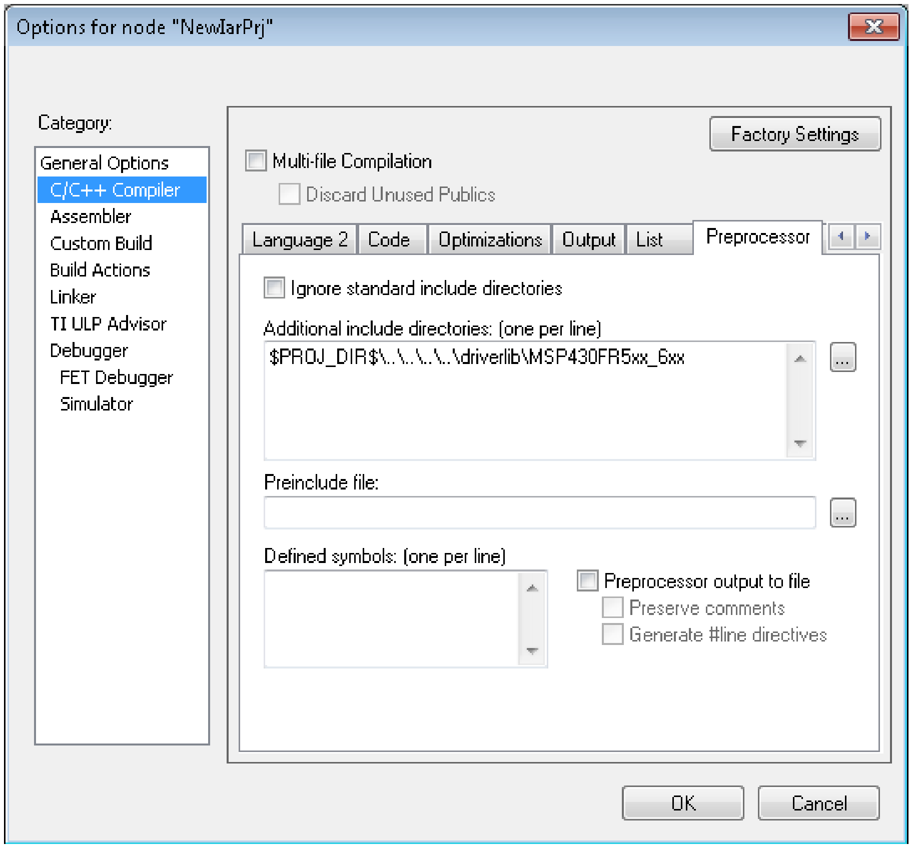
To add driver library to an existing project, right click project click on Add Group - "driverlib"



Now click Add files and browse through driverlib folder and add all source files of the family the device belongs to.



Add another group via "Add Group" and add inc folder. Add all files in the same driverlib family inc folder



Click "Finish" and start developing with driver library in your project.

7 Analog-to-Digital Converter (ADC)

7.1 Introduction

The Analog-to-Digital (ADC) API provides a set of functions for using the MSP430Ware ADC modules. Functions are provided to initialize the ADC modules, setup signal sources and reference voltages, and manage interrupts for the ADC modules.

The ADC module supports fast analog-to-digital conversions. The module implements a SAR core together, sample select control and a window comparator.

ADC features include:

- Greater than 200-ksps maximum conversion rate
- Monotonic up-to-12-bit converter with no missing codes
- Sample-and-hold with programmable sampling periods controlled by software or timers
- Conversion initiation by software or different timers
- Software-selectable on chip reference using the REF module or external reference
- Twelve individually configurable external input channels
- Conversion channel for temperature sensor of the REF module
- Selectable conversion clock source
- Single-channel, repeat-single-channel, sequence, and repeat-sequence conversion modes
- Window comparator for low-power monitoring of input signals
- Interrupt vector register for fast decoding of six ADC interrupts (ADCIFG0, ADCTOVIFG, ADCOVIFG, ADCLOIFG, ADCINIFG, ADCHIIFG)

This driver is contained in `adc.c`, with `adc.h` containing the API definitions for use by applications.

7.2 API Functions

Functions

- void ADC_init (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
  Initializes the ADC Module.
- void ADC_enable (uint16_t baseAddress)
  Enables the ADC block.
- void ADC_disable (uint16_t baseAddress)
  Disables the ADC block.

- void ADC_setupSamplingTimer (uint16_t baseAddress, uint16_t clockCycleHoldCount, uint16_t multipleSamplesEnabled)
  Sets up and enables the Sampling Timer Pulse Mode.
- void ADC_disableSamplingTimer (uint16_t baseAddress)
  Disables Sampling Timer Pulse Mode.
- void ADC_configureMemory (uint16_t baseAddress, uint8_t inputSourceSelect, uint8_t positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect)
  Configures the controls of the selected memory buffer.
- void ADC_enableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Enables selected ADC interrupt sources.
- void ADC_disableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Disables selected ADC interrupt sources.
- void ADC_clearInterrupt (uint16_t baseAddress, uint16_t interruptFlagMask)
  Clears ADC10B selected interrupt flags.
- uint8_t ADC_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
  Returns the status of the selected memory interrupt flags.
- void ADC_startConversion (uint16_t baseAddress, uint16_t conversionSequenceModeSelect)
  Enables/Starts an Analog-to-Digital Conversion.
- void ADC_disableConversions (uint16_t baseAddress, bool preempt)
  Disables the ADC from converting any more signals.
- int16_t ADC_getResults (uint16_t baseAddress)
  Returns the raw contents of the specified memory buffer.
- void ADC_setResolution (uint16_t baseAddress, uint16_t resolutionSelect)
  Use to change the resolution of the converted data.
- void ADC_setSampleHoldSignalInversion (uint16_t baseAddress, uint16_t invertedSignal)
  Use to invert or un-invert the sample/hold signal.
- void ADC_setDataReadBackFormat (uint16_t baseAddress, uint16_t readBackFormat)
  Use to set the read-back format of the converted data.
- void ADC_setReferenceBufferSamplingRate (uint16_t baseAddress, uint16_t samplingRateSelect)
  Use to set the reference buffer's sampling rate.
- void ADC_setWindowComp (uint16_t baseAddress, uint16_t highThreshold, uint16_t lowThreshold)
  Sets the high and low threshold for the window comparator feature.
- uint32_t ADC_getMemoryAddressForDMA (uint16_t baseAddress)
  Returns the address of the memory buffer for the DMA module.
- uint8_t ADC_isBusy (uint16_t baseAddress)
  Returns the busy status of the ADC core.

7.2.1 Detailed Description

The ADC API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxiliary features of the ADC.

The ADC initialization and conversion functions are

- ADC_init()
- ADC_configureMemory()
- ADC_setupSamplingTimer()
- ADC_disableSamplingTimer()

- ADC_setWindowComp()
- ADC_startConversion()
- ADC_disableConversions()
- ADC_getResults()
- ADC_isBusy()

The ADC interrupts are handled by

- ADC_enableInterrupt()
- ADC_disableInterrupt()
- ADC_clearInterrupt()
- ADC_getInterruptStatus()

Auxiliary features of the ADC are handled by

- ADC_setResolution()
- ADC_setSampleHoldSignalInversion()
- ADC_setDataReadBackFormat()
- ADC_enableReferenceBurst()
- ADC_disableReferenceBurst()
- ADC_setReferenceBufferSamplingRate()
- ADC_getMemoryAddressForDMA()
- ADC_enable()
- ADC_disable()

7.2.2 Function Documentation

ADC_clearInterrupt()

```
void ADC_clearInterrupt (
            uint16_t baseAddress,
            uint16_t interruptFlagMask )
```

Clears ADC10B selected interrupt flags.

The selected ADC interrupt flags are cleared, so that it no longer asserts. The memory buffer interrupt flags are only cleared when the memory buffer is accessed.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

interruptFlagMask	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: ■ **ADC_OVERFLOW_INTERRUPT_FLAG** - Interrupt flag for when a new conversion is about to overwrite the previous one ■ **ADC_TIMEOVERFLOW_INTERRUPT_FLAG** - Interrupt flag for when a new conversion is starting before the previous one has finished ■ **ADC_ABOVETHRESHOLD_INTERRUPT_FLAG** - Interrup flag for when the input signal has gone above the high threshold of the window comparator ■ **ADC_BELOWTHRESHOLD_INTERRUPT_FLAG** - Interrupt flag for when the input signal has gone below the low threshold of the window comparator ■ **ADC_INSIDEWINDOW_INTERRUPT_FLAG** - Interrupt flag for when the input signal is in between the high and low thresholds of the window comparator ■ **ADC_COMPLETED_INTERRUPT_FLAG** - Interrupt flag for new conversion data in the memory buffer

Modified bits of **ADCIFG** register.

Returns

None

ADC_configureMemory()

```
void ADC_configureMemory (
            uint16_t baseAddress,
            uint8_t inputSourceSelect,
            uint8_t positiveRefVoltageSourceSelect,
            uint8_t negativeRefVoltageSourceSelect )
```

Configures the controls of the selected memory buffer.

Maps an input signal conversion into the memory buffer, as well as the positive and negative reference voltages for each conversion being stored into the memory buffer. If the internal reference is used for the positive reference voltage, the internal REF module has to control the voltage level. Note that if a conversion has been started with the startConversion() function, then a call to disableConversions() is required before this function may be called. If conversion is not disabled, this function does nothing.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

inputSourceSelect	is the input that will store the converted data into the specified memory buffer. Valid values are: ■ **ADC_INPUT_A0** [Default] ■ **ADC_INPUT_A1** ■ **ADC_INPUT_A2** ■ **ADC_INPUT_A3** ■ **ADC_INPUT_A4** ■ **ADC_INPUT_A5** ■ **ADC_INPUT_A6** ■ **ADC_INPUT_A7** ■ **ADC_INPUT_A8** - [Valid for FR4xx devices] ■ **ADC_INPUT_A9** - [Valid for FR4xx devices] ■ **ADC_INPUT_TEMPSENSOR** ■ **ADC_INPUT_REFVOLTAGE** ■ **ADC_INPUT_DVSS** ■ **ADC_INPUT_DVCC** Modified bits are **ADCINCHx** of **ADCMCTL0** register.
positiveRefVoltageSourceSelect	is the reference voltage source to set as the upper limit for the conversion that is to be stored in the specified memory buffer. Valid values are: ■ **ADC_VREFPOS_AVCC** [Default] ■ **ADC_VREFPOS_INT** ■ **ADC_VREFPOS_EXT_BUF** ■ **ADC_VREFPOS_EXT_NOBUF** Modified bits are **ADCSREF** of **ADCMCTL0** register.
negativeRefVoltageSourceSelect	is the reference voltage source to set as the lower limit for the conversion that is to be stored in the specified memory buffer. Valid values are: ■ **ADC_VREFNEG_AVSS** [Default] ■ **ADC_VREFNEG_EXT** Modified bits are **ADCSREF** of **ADCMCTL0** register.

Returns

None

ADC_disable()

```
void ADC_disable (
```

```
uint16_t baseAddress )
```

Disables the ADC block.

This will disable operation of the ADC block.

Parameters

baseAddress	is the base address of the ADC module.

Modified bits are **ADCON** of **ADCCTL0** register.

Returns

None

ADC_disableConversions()

```
void ADC_disableConversions (
        uint16_t baseAddress,
        bool preempt )
```

Disables the ADC from converting any more signals.

Disables the ADC from converting any more signals. If there is a conversion in progress, this function can stop it immediatly if the preempt parameter is set as ADC_PREEMPTCONVERSION, by changing the conversion mode to single- channel, single-conversion and disabling conversions. If the conversion mode is set as single-channel, single-conversion and this function is called without preemption, then the ADC core conversion status is polled until the conversion is complete before disabling conversions to prevent unpredictable data. If the ADC_startConversion() has been called, then this function has to be called to re-initialize the ADC, reconfigure a memory buffer control, enable/disable the sampling pulse mode, or change the internal reference voltage.

Parameters

baseAddress	is the base address of the ADC module.
preempt	specifies if the current conversion should be preemptly stopped before the end of the conversion Valid values are: ■ **ADC_COMPLETECONVERSION** - Allows the ADC to end the current conversion before disabling conversions. ■ **ADC_PREEMPTCONVERSION** - Stops the ADC10B immediately, with unpredicatble results of the current conversion. Cannot be used with repeated conversion.

Modified bits of **ADCCTL0** register and bits of **ADCCTL1** register.

Returns

None

ADC_disableInterrupt()

```
void ADC_disableInterrupt (
            uint16_t baseAddress,
            uint16_t interruptMask )
```

Disables selected ADC interrupt sources.

Disables the indicated ADC 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 ADC module.
interruptMask	is the bit mask of the memory buffer interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **ADC_OVERFLOW_INTERRUPT** - Interrupts when a new conversion is about to overwrite the previous one ■ **ADC_TIMEOVERFLOW_INTERRUPT** - Interrupts when a new conversion is starting before the previous one has finished ■ **ADC_ABOVETHRESHOLD_INTERRUPT** - Interrups when the input signal has gone above the high threshold of the window comparator ■ **ADC_BELOWTHRESHOLD_INTERRUPT** - Interrupts when the input signal has gone below the low threshold of the low window comparator ■ **ADC_INSIDEWINDOW_INTERRUPT** - Interrupts when the input signal is in between the high and low thresholds of the window comparator ■ **ADC_COMPLETED_INTERRUPT** - Interrupt for new conversion data in the memory buffer

Modified bits of **ADCIE** register.

Returns

None

ADC_disableSamplingTimer()

```
void ADC_disableSamplingTimer (
            uint16_t baseAddress )
```

Disables Sampling Timer Pulse Mode.

Disables the Sampling Timer Pulse Mode. Note that if a conversion has been started with the startConversion() function, then a call to disableConversions() is required before this function may be called.

Parameters

baseAddress	is the base address of the ADC module.

Modified bits are **ADCSHP** of **ADCCTL1** register.

Returns

None

ADC_enable()

```
void ADC_enable (
            uint16_t baseAddress )
```

Enables the ADC block.

This will enable operation of the ADC block.

Parameters

baseAddress	is the base address of the ADC module.

Modified bits are **ADCON** of **ADCCTL0** register.

Returns

None

ADC_enableInterrupt()

```
void ADC_enableInterrupt (
            uint16_t baseAddress,
            uint16_t interruptMask )
```

Enables selected ADC interrupt sources.

Enables the indicated ADC 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 ADC module.**

Parameters

interruptMask	**is the bit mask of the memory buffer interrupt sources to be enabled. Mask value is the logical OR of any of the following:** ■ **ADC_OVERFLOW_INTERRUPT - Interrupts when a new conversion is about to overwrite the previous one** ■ **ADC_TIMEOVERFLOW_INTERRUPT - Interrupts when a new conversion is starting before the previous one has finished** ■ **ADC_ABOVETHRESHOLD_INTERRUPT - Interrups when the input signal has gone above the high threshold of the window comparator** ■ **ADC_BELOWTHRESHOLD_INTERRUPT - Interrupts when the input signal has gone below the low threshold of the low window comparator** ■ **ADC_INSIDEWINDOW_INTERRUPT - Interrupts when the input signal is in between the high and low thresholds of the window comparator** ■ **ADC_COMPLETED_INTERRUPT - Interrupt for new conversion data in the memory buffer**

Modified bits of ADCIE register.

Returns

None

ADC_getInterruptStatus()

```
uint8_t ADC_getInterruptStatus (
            uint16_t baseAddress,
            uint8_t interruptFlagMask )
```

Returns the status of the selected memory interrupt flags.

Returns the status of the selected interrupt flags.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

interruptFlagMask	is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following: ■ **ADC_OVERFLOW_INTERRUPT_FLAG** - Interrupt flag for when a new conversion is about to overwrite the previous one ■ **ADC_TIMEOVERFLOW_INTERRUPT_FLAG** - Interrupt flag for when a new conversion is starting before the previous one has finished ■ **ADC_ABOVETHRESHOLD_INTERRUPT_FLAG** - Interrup flag for when the input signal has gone above the high threshold of the window comparator ■ **ADC_BELOWTHRESHOLD_INTERRUPT_FLAG** - Interrupt flag for when the input signal has gone below the low threshold of the window comparator ■ **ADC_INSIDEWINDOW_INTERRUPT_FLAG** - Interrupt flag for when the input signal is in between the high and low thresholds of the window comparator ■ **ADC_COMPLETED_INTERRUPT_FLAG** - Interrupt flag for new conversion data in the memory buffer

Modified bits of **ADC10IFG** register.

Returns

The current interrupt flag status for the corresponding mask.

ADC_getMemoryAddressForDMA()

```
uint32_t ADC_getMemoryAddressForDMA (
            uint16_t baseAddress )
```

Returns the address of the memory buffer for the DMA module.

Parameters

baseAddress	is the base address of the ADC module.

Returns

the address of the memory buffer. This can be used in conjunction with the DMA to store the converted data directly to memory.

ADC_getResults()

```
int16_t ADC_getResults (
            uint16_t baseAddress )
```

Returns the raw contents of the specified memory buffer.

Returns the raw contents of the specified memory buffer. The format of the content depends on the read-back format of the data: if the data is in signed 2's complement format then the contents in the memory buffer will be left-justified with the least-siginificant bits as 0's, whereas if the data is in unsigned format then the contents in the memory buffer will be right- justified with the most-significant bits as 0's.

Parameters

baseAddress	is the base address of the ADC module.

Returns

A Signed Integer of the contents of the specified memory buffer.

ADC_init()

```
void ADC_init (
            uint16_t baseAddress,
            uint16_t sampleHoldSignalSourceSelect,
            uint8_t clockSourceSelect,
            uint16_t clockSourceDivider )
```

Initializes the ADC Module.

This function initializes the ADC module to allow for analog-to-digital conversions. Specifically this function sets up the sample-and-hold signal and clock sources for the ADC core to use for conversions. Upon successful completion of the initialization all of the ADC control registers will be reset, excluding the memory controls and reference module bits, the given parameters will be set, and the ADC core will be turned on (Note, that the ADC core only draws power during conversions and remains off when not converting).Note that sample/hold signal sources are device dependent. Note that if re-initializing the ADC after starting a conversion with the startConversion() function, the disableConversion() must be called BEFORE this function can be called.

Parameters

baseAddress	is the base address of the ADC module.
sampleHoldSignalSourceSelect	is the signal that will trigger a sample-and-hold for an input signal to be converted. This parameter is device specific and sources should be found in the device's datasheet. Valid values are: ■ **ADC_SAMPLEHOLDSOURCE_SC** [Default] ■ **ADC_SAMPLEHOLDSOURCE_1** ■ **ADC_SAMPLEHOLDSOURCE_2** ■ **ADC_SAMPLEHOLDSOURCE_3** Modified bits are **ADCSHSx** of **ADCCTL1** register.

Parameters

clockSourceSelect	selects the clock that will be used by the ADC core and the sampling timer if a sampling pulse mode is enabled. Valid values are: ■ **ADC_CLOCKSOURCE_ADCOSC** [Default] - MODOSC 5 MHz oscillator from the clock system ■ **ADC_CLOCKSOURCE_ACLK** - The Auxilary Clock ■ **ADC_CLOCKSOURCE_SMCLK** - The Sub-Master Clock Modified bits are **ADCSSELx** of **ADCCTL1** register.
clockSourceDivider	selects the amount that the clock will be divided. Valid values are: ■ **ADC_CLOCKDIVIDER_1** [Default] ■ **ADC_CLOCKDIVIDER_2** ■ **ADC_CLOCKDIVIDER_3** ■ **ADC_CLOCKDIVIDER_4** ■ **ADC_CLOCKDIVIDER_5** ■ **ADC_CLOCKDIVIDER_6** ■ **ADC_CLOCKDIVIDER_7** ■ **ADC_CLOCKDIVIDER_8** ■ **ADC_CLOCKDIVIDER_12** ■ **ADC_CLOCKDIVIDER_16** ■ **ADC_CLOCKDIVIDER_20** ■ **ADC_CLOCKDIVIDER_24** ■ **ADC_CLOCKDIVIDER_28** ■ **ADC_CLOCKDIVIDER_32** ■ **ADC_CLOCKDIVIDER_64** ■ **ADC_CLOCKDIVIDER_128** ■ **ADC_CLOCKDIVIDER_192** ■ **ADC_CLOCKDIVIDER_256** ■ **ADC_CLOCKDIVIDER_320** ■ **ADC_CLOCKDIVIDER_384** ■ **ADC_CLOCKDIVIDER_448** ■ **ADC_CLOCKDIVIDER_512** Modified bits are **ADCDIVx** of **ADCCTL1** register; bits **ADCPDIVx** of **ADCCTL2** register.

Returns

None

ADC_isBusy()

```
uint8_t ADC_isBusy (
            uint16_t baseAddress )
```

Returns the busy status of the ADC core.

Returns the status of the ADC core if there is a conversion currently taking place.

Parameters

baseAddress	is the base address of the ADC module.

Returns

ADC_BUSY or ADC_NOTBUSY dependent if there is a conversion currently taking place. Return one of the following:

- **ADC_NOTBUSY**
- **ADC_BUSY**

ADC_setDataReadBackFormat()

```
void ADC_setDataReadBackFormat (
            uint16_t baseAddress,
            uint16_t readBackFormat )
```

Use to set the read-back format of the converted data.

Sets the format of the converted data: how it will be stored into the memory buffer, and how it should be read back. The format can be set as right-justified (default), which indicates that the number will be unsigned, or left-justified, which indicates that the number will be signed in 2's complement format. This change affects all memory buffers for subsequent conversions.

Parameters

baseAddress	is the base address of the ADC module.
readBackFormat	is the specified format to store the conversions in the memory buffer. Valid values are: ■ **ADC_UNSIGNED_BINARY** [Default] ■ **ADC_SIGNED_2SCOMPLEMENT** Modified bits are **ADCDF** of **ADCCTL2** register.

Returns

None

ADC_setReferenceBufferSamplingRate()

```
void ADC_setReferenceBufferSamplingRate (
```

```
            uint16_t baseAddress,
            uint16_t samplingRateSelect )
```

Use to set the reference buffer's sampling rate.

Sets the reference buffer's sampling rate to the selected sampling rate. The default sampling rate is maximum of 200-ksps, and can be reduced to a maximum of 50-ksps to conserve power.

Parameters

baseAddress	is the base address of the ADC module.
samplingRateSelect	is the specified maximum sampling rate. Valid values are: ■ **ADC_MAXSAMPLINGRATE_200KSPS** [Default] ■ **ADC_MAXSAMPLINGRATE_50KSPS** Modified bits are **ADCSR** of **ADCCTL2** register.

Modified bits of **ADCCTL2** register.

Returns

None

ADC_setResolution()

```
void ADC_setResolution (
            uint16_t baseAddress,
            uint16_t resolutionSelect )
```

Use to change the resolution of the converted data.

This function can be used to change the resolution of the converted data from the default of 10-bits. Refer to the device user's guide for available options.

Parameters

baseAddress	is the base address of the ADC module.
resolutionSelect	determines the resolution of the converted data. Valid values are: ■ **ADC_RESOLUTION_8BIT** ■ **ADC_RESOLUTION_10BIT** [Default] ■ **ADC_RESOLUTION_12BIT** - [Only available in some devices] Modified bits are **ADCRES** of **ADCCTL2** register.

Returns

None

ADC_setSampleHoldSignalInversion()

```
void ADC_setSampleHoldSignalInversion (
            uint16_t baseAddress,
            uint16_t invertedSignal )
```

Use to invert or un-invert the sample/hold signal.

This function can be used to invert or un-invert the sample/hold signal. Note that if a conversion has been started with the startConversion() function, then a call to disableConversions() is required before this function may be called.

Parameters

baseAddress	is the base address of the ADC module.
invertedSignal	set if the sample/hold signal should be inverted Valid values are: ■ **ADC_NONINVERTEDSIGNAL** [Default] - a sample-and-hold of an input signal for conversion will be started on a rising edge of the sample/hold signal. ■ **ADC_INVERTEDSIGNAL** - a sample-and-hold of an input signal for conversion will be started on a falling edge of the sample/hold signal. Modified bits are **ADCISSH** of **ADCCTL1** register.

Returns

None

ADC_setupSamplingTimer()

```
void ADC_setupSamplingTimer (
            uint16_t baseAddress,
            uint16_t clockCycleHoldCount,
            uint16_t multipleSamplesEnabled )
```

Sets up and enables the Sampling Timer Pulse Mode.

This function sets up the sampling timer pulse mode which allows the sample/hold signal to trigger a sampling timer to sample-and-hold an input signal for a specified number of clock cycles without having to hold the sample/hold signal for the entire period of sampling. Note that if a conversion has been started with the startConversion() function, then a call to disableConversions() is required before this function may be called.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

clockCycleHoldCount	sets the amount of clock cycles to sample-and- hold for the memory buffer. Valid values are: ■ **ADC_CYCLEHOLD_4_CYCLES** [Default] ■ **ADC_CYCLEHOLD_8_CYCLES** ■ **ADC_CYCLEHOLD_16_CYCLES** ■ **ADC_CYCLEHOLD_32_CYCLES** ■ **ADC_CYCLEHOLD_64_CYCLES** ■ **ADC_CYCLEHOLD_96_CYCLES** ■ **ADC_CYCLEHOLD_128_CYCLES** ■ **ADC_CYCLEHOLD_192_CYCLES** ■ **ADC_CYCLEHOLD_256_CYCLES** ■ **ADC_CYCLEHOLD_384_CYCLES** ■ **ADC_CYCLEHOLD_512_CYCLES** ■ **ADC_CYCLEHOLD_768_CYCLES** ■ **ADC_CYCLEHOLD_1024_CYCLES** Modified bits are **ADCSHTx** of **ADCCTL0** register.
multipleSamplesEnabled	allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are: ■ **ADC_MULTIPLESAMPLESDISABLE** - a timer trigger will be needed to start every ADC conversion. ■ **ADC_MULTIPLESAMPLESENABLE** - during a sequenced and/or repeated conversion mode, after the first conversion, no sample/hold signal is necessary to start subsequent samples. Modified bits are **ADCMSC** of **ADCCTL0** register.

Returns

None

ADC_setWindowComp()

```
void ADC_setWindowComp (
            uint16_t baseAddress,
            uint16_t highThreshold,
            uint16_t lowThreshold )
```

Sets the high and low threshold for the window comparator feature.

Sets the high and low threshold for the window comparator feature. Use the ADCHIIE, ADCINIE, ADCLOIE interrupts to utilize this feature.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

highThreshold	is the upper bound that could trip an interrupt for the window comparator.
lowThreshold	is the lower bound that could trip on interrupt for the window comparator.

Modified bits of **ADCLO** register and bits of **ADCHI** register.

Returns

None

ADC_startConversion()

```
void ADC_startConversion (
            uint16_t baseAddress,
            uint16_t conversionSequenceModeSelect )
```

Enables/Starts an Analog-to-Digital Conversion.

This function enables/starts the conversion process of the ADC. If the sample/hold signal source chosen during initialization was ADCOSC, then the conversion is started immediately, otherwise the chosen sample/hold signal source starts the conversion by a rising edge of the signal. Keep in mind when selecting conversion modes, that for sequenced and/or repeated modes, to keep the sample/hold-and-convert process continuing without a trigger from the sample/hold signal source, the multiple samples must be enabled using the ADC_setupSamplingTimer() function. Also note that when a sequence conversion mode is selected, the first input channel is the one mapped to the memory buffer, the next input channel selected for conversion is one less than the input channel just converted (i.e. A1 comes after A2), until A0 is reached, and if in repeating mode, then the next input channel will again be the one mapped to the memory buffer. Note that after this function is called, the ADC_stopConversions() has to be called to re-initialize the ADC, reconfigure a memory buffer control, enable/disable the sampling timer, or to change the internal reference voltage.

Parameters

baseAddress	is the base address of the ADC module.

Parameters

conversionSequenceModeSelect	determines the ADC operating mode. Valid values are: ■ **ADC_SINGLECHANNEL** [Default] - one-time conversion of a single channel into a single memory buffer ■ **ADC_SEQOFCHANNELS** - one time conversion of multiple channels into the specified starting memory buffer and each subsequent memory buffer up until the conversion is stored in a memory buffer dedicated as the end-of-sequence by the memory's control register ■ **ADC_REPEATED_SINGLECHANNEL** - repeated conversions of one channel into a single memory buffer ■ **ADC_REPEATED_SEQOFCHANNELS** - repeated conversions of multiple channels into the specified starting memory buffer and each subsequent memory buffer up until the conversion is stored in a memory buffer dedicated as the end-of-sequence by the memory's control register Modified bits are **ADCCONSEQx** of **ADCCTL1** register.

Returns

None

7.3 Programming Example

The following example shows how to initialize and use the ADC API to start a single channel, single conversion.

```
// Initialize ADC with ADC's built-in oscillator
ADC_init (ADC_BASE,
            ADC_SAMPLEHOLDSOURCE_SC,
            ADC_CLOCKSOURCE_ADCOSC,
            ADC_CLOCKDIVIDER_1);

//Switch ON ADC
ADC_enable(ADC_BASE);

// Setup sampling timer to sample-and-hold for 16 clock cycles
ADC_setupSamplingTimer (ADC_BASE,
                          ADC_CYCLEHOLD_16_CYCLES,
                          FALSE);

// Configure the Input to the Memory Buffer with the specified Reference Voltages
ADC_configureMemory(ADC_BASE,
                       ADC_INPUT_A0,
                       ADC_VREFPOS_AVCC, // Vref+ = AVcc
                       ADC_VREFNEG_AVSS  // Vref- = AVss
                       );
while (1)
{
    // Start a single conversion, no repeating or sequences.
```

```
    ADC_startConversion (ADC_BASE,
                         ADC_SINGLECHANNEL);

    // Wait for the Interrupt Flag to assert
    while( !(ADC_getInterruptStatus(ADC_BASE,ADC_COMPLETED_INTERRUPT_FLAG)) );

    // Clear the Interrupt Flag and start another conversion
    ADC_clearInterrupt(ADC_BASE,ADC_COMPLETED_INTERRUPT_FLAG);
}
```

8 Cyclical Redundancy Check (CRC)

8.1 Introduction

The Cyclic Redundancy Check (CRC) API provides a set of functions for using the MSP430Ware CRC 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 CRC module offers no interrupts and is used only to generate CRC signatures to verify against pre-made CRC signatures (Checksums).

8.2 API Functions

Functions

- void CRC_setSeed (uint16_t baseAddress, uint16_t seed)
  Sets the seed for the CRC.
- void CRC_set16BitData (uint16_t baseAddress, uint16_t dataIn)
  Sets the 16 bit data to add into the CRC module to generate a new signature.
- void CRC_set8BitData (uint16_t baseAddress, uint8_t dataIn)
  Sets the 8 bit data to add into the CRC module to generate a new signature.
- void CRC_set16BitDataReversed (uint16_t baseAddress, uint16_t dataIn)
  Translates the 16 bit data by reversing the bits in each byte and then sets this data to add into the CRC module to generate a new signature.
- void CRC_set8BitDataReversed (uint16_t baseAddress, uint8_t dataIn)
  Translates the 8 bit data by reversing the bits in each byte and then sets this data to add into the CRC module to generate a new signature.
- uint16_t CRC_getData (uint16_t baseAddress)
  Returns the value currently in the Data register.
- uint16_t CRC_getResult (uint16_t baseAddress)
  Returns the value pf the Signature Result.
- uint16_t CRC_getResultBitsReversed (uint16_t baseAddress)
  Returns the bit-wise reversed format of the Signature Result.

8.2.1 Detailed Description

The CRC API is one group that controls the CRC module. The APIs that are used to set the seed and data are

- CRC_setSeed()
- CRC_set16BitData()

- CRC_set8BitData()
- CRC_set16BitDataReversed()
- CRC_set8BitDataReversed()
- CRC_setSeed()

The APIs that are used to get the data and results are

- CRC_getData()
- CRC_getResult()
- CRC_getResultBitsReversed()

8.2.2 Function Documentation

CRC_getData()

```
uint16_t CRC_getData (
            uint16_t baseAddress )
```

Returns the value currently in the Data register.

This function returns the value currently in the data register. If set in byte bits reversed format, then the translated data would be returned.

Parameters

baseAddress	is the base address of the CRC module.

Returns

The value currently in the data register

CRC_getResult()

```
uint16_t CRC_getResult (
            uint16_t baseAddress )
```

Returns the value pf the Signature Result.

This function returns the value of the signature result generated by the CRC.

Parameters

baseAddress	is the base address of the CRC module.

Returns

The value currently in the data register

CRC_getResultBitsReversed()

```
uint16_t CRC_getResultBitsReversed (
            uint16_t baseAddress )
```

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

This function returns the bit-wise reversed format of the Signature Result.

Parameters

baseAddress	is the base address of the CRC module.

Returns

The bit-wise reversed format of the Signature Result

CRC_set16BitData()

```
void CRC_set16BitData (
            uint16_t baseAddress,
            uint16_t dataIn )
```

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

This function sets the given data into the CRC module to generate the new signature from the current signature and new data.

Parameters

baseAddress	is the base address of the CRC module.
dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRCDI** of **CRCDI** register.

Returns

None

CRC_set16BitDataReversed()

```
void CRC_set16BitDataReversed (
            uint16_t baseAddress,
            uint16_t dataIn )
```

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

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.

Parameters

baseAddress	is the base address of the CRC module.
dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRCDIRB** of **CRCDIRB** register.

Returns

None

CRC_set8BitData()

```
void CRC_set8BitData (
            uint16_t baseAddress,
            uint8_t dataIn )
```

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

This function sets the given data into the CRC module to generate the new signature from the current signature and new data.

Parameters

baseAddress	is the base address of the CRC module.
dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRCDI** of **CRCDI** register.

Returns

None

CRC_set8BitDataReversed()

```
void CRC_set8BitDataReversed (
            uint16_t baseAddress,
            uint8_t dataIn )
```

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

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.

Parameters

baseAddress	is the base address of the CRC module.
dataIn	is the data to be added, through the CRC module, to the signature. Modified bits are **CRCDIRB** of **CRCDIRB** register.

Returns

None

CRC_setSeed()

```
void CRC_setSeed (
            uint16_t baseAddress,
            uint16_t seed )
```

Sets the seed for the CRC.

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 CRC signature.

Parameters

baseAddress	is the base address of the CRC module.
seed	is the seed for the CRC to start generating a signature from. Modified bits are **CRCINIRES** of **CRCINIRES** register.

Returns

None

8.3 Programming Example

The following example shows how to initialize and use the CRC API to generate a CRC signature on an array of data.

```
unsigned int crcSeed = 0xBEEF;
unsigned int data[] = {0x0123,
                       0x4567,
                       0x8910,
                       0x1112,
                       0x1314};
unsigned int crcResult;
int i;

// Stop WDT
WDT_hold(WDT_A_BASE);

// Set P1.0 as an output
GPIO_setAsOutputPin(GPIO_PORT_P1,
                    GPIO_PIN0);

// Set the CRC seed
CRC_setSeed(CRC_BASE,
            crcSeed);

for (i = 0; i < 5; i++)
{
//Add all of the values into the CRC signature
CRC_set16BitData(CRC_BASE,
    data[i]);
}

// Save the current CRC signature checksum to be compared for later
crcResult = CRC_getResult(CRC_BASE);
```

9 Clock System (CS)

9.1 Introduction

The CS is based on five available clock sources (XT1, VLO, REFO, DCO and MOD) providing signals to three system clocks (MCLK, SMCLK, ACLK). Different low power modes are achieved by turning off the MCLK, SMCLK, ACLK, and integrated LDO.

- VLO - Internal very-low-power low-frequency oscillator. 10 kHz (?0.5%/?C, ?4%/V)
- REFO - Reference oscillator. 32 kHz (?1%, ?3% over full temp range)
- XT1 (LFXT1, HFXT1) - Ultra-low-power oscillator, compatible with low-frequency 32768-Hz watch crystals and with standard XT1 (LFXT1, HFXT1) crystals, resonators, or external clock sources in the 4-MHz to 32-MHz range, including digital inputs. Most commonly used as 32-kHz watch crystal oscillator.
- DCO - Internal digitally-controlled oscillator (DCO) that can be stabilized by a frequency lock loop (FLL) that sets the DCO to a specified multiple of a reference frequency.
- MOD - Internal high-frequency oscillator with 5-MHz typical frequency.

System Clocks and Functionality on the MSP430 MCLK Master Clock Services the CPU. Commonly sourced by DCO. Is available in Active mode only SMCLK Subsystem Master Clock Services 'fast' system peripherals. Commonly sourced by DCO. Is available in Active mode, LPM0 and LPM1 ACLK Auxiliary Clock Services 'slow' system peripherals. Commonly used for 32-kHz signal.Is available in Active mode, LPM0 to LPM3

System clocks of the MSP430FR2xx_4xx generation are automatically enabled, regardless of the LPM mode of operation, if they are required for the proper operation of the peripheral module that they source. This additional flexibility of the CS, along with improved fail-safe logic, provides a robust clocking scheme for all applications.

Fail-Safe logic The CS fail-safe logic plays an important part in providing a robust clocking scheme for MSP430FR2xx and MSP430FR4xx applications. This feature hinges on the ability to detect an oscillator fault for the XT1 in low-frequency mode and the DCO (DCOFFG). These flags are set and latched when the respective oscillator is enabled but not operating properly; therefore, they must be explicitly cleared in software.

The oscillator fault flags on previous MSP430 generations are not latched and are asserted only as long as the failing condition exists. Therefore, an important difference between the families is that the fail-safe behavior in a FR2xx_4xx-based MSP430 remains active until both the OFIFG and the respective fault flag are cleared in software.

This fail-safe behavior is implemented at the oscillator level, at the system clock level and, consequently, at the module level. Some notable highlights of this behavior are described below. For the full description of fail-safe behavior and conditions, see the MSP430FR2xx_4xx Family User?s Guide (SLAU445).

- Low-frequency crystal oscillator 1 (XT1) The low-frequency (32768 Hz) crystal oscillator is the default reference clock to the FLL. An asserted XT1LFOFFG switches the FLL reference from the failing XT1 to the internal 32-kHz REFO. This can influence the DCO accuracy, because the FLL crystal ppm specification is typically tighter than the REFO accuracy over temperature and voltage of ?3%.

- System Clocks (ACLK, SMCLK, MCLK) A fault on the oscillator that is sourcing a system clock switches the source from the failing oscillator to the DCO oscillator (DCOCLKDIV). This is true for all clock sources except the XT1. As previously described, a fault on the XT1 switches the source to the REFO. Since ACLK is the active clock in LPM3 there is a notable difference in the LPM3 current consumption when the REFO is the clock source (∼3 ?A active) versus the XT1 (∼300 nA active).
- Modules (WDT_A) In watchdog mode, when SMCLK or ACLK fails, the clock source defaults to the VLOCLK.

Please note that MCLK and SMCLK share the same clock source. Changes on selecting clock source on either system clock impact on clock source for both system clocks.

9.2 API Functions

Macros

- #define **CS_VLOCLK_FREQUENCY** 10000
- #define **CS_REFOCLK_FREQUENCY** 32768
- #define **CS_DCO_RANGE_1MHZ** 1000000
- #define **CS_DCO_RANGE_2MHZ** 2000000
- #define **CS_DCO_RANGE_4MHZ** 4000000
- #define **CS_DCO_RANGE_8MHZ** 8000000
- #define **CS_DCO_RANGE_12MHZ** 12000000
- #define **CS_DCO_RANGE_16MHZ** 16000000
- #define **CS_DCO_RANGE_20MHZ** 20000000
- #define **CS_DCO_RANGE_24MHZ** 24000000

Functions

- void CS_setExternalClockSource (uint32_t XT1CLK_frequency)
  Sets the external clock source.
- void CS_initClockSignal (uint8_t selectedClockSignal, uint16_t clockSource, uint16_t clockSourceDivider)
  Initializes a clock signal.
- void CS_turnOnXT1LF (uint16_t xt1Drive)
  Intializes the XT1 crystal oscillator in low frequency mode.
- void CS_bypassXT1 (void)
  Bypass the XT1 crystal oscillator.
- bool CS_turnOnXT1LFWithTimeout (uint16_t xt1Drive, uint16_t timeout)
  Initializes the XT1 crystal oscillator in low frequency mode with timeout.
- bool CS_bypassXT1WithTimeout (uint16_t timeout)
  Bypasses the XT1 crystal oscillator with time out.
- void CS_turnOffXT1 (void)
  Stops the XT1 oscillator using the XT1AUTOOFF bit.
- void CS_turnOnXT1HF (uint16_t xt1Drive, uint16_t xt1HFFreq)
  Intializes the XT1 crystal oscillator in high frequency mode.

- bool CS_turnOnXT1HFWithTimeout (uint16_t xt1Drive, uint16_t xt1HFFreq, uint16_t timeout)
  Initializes the XT1 crystal oscillator in high frequency mode with timeout.
- void CS_turnOnSMCLK (void)
  Turn On SMCLK.
- void CS_turnOffSMCLK (void)
  Turn Off SMCLK.
- void CS_enableVLOAutoOff (void)
  VLO is turned off when not used.
- void CS_disableVLOAutoOff (void)
  VLO is always on.
- bool CS_initFLLSettle (uint16_t fsystem, uint16_t ratio)
  Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL.
- bool CS_initFLL (uint16_t fsystem, uint16_t ratio)
  Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. This function performs DCO Factory Trim.
- bool CS_initFLLCalculateTrim (uint16_t fsystem, uint16_t ratio, CS_initFLLParam ∗param)
  Performs same function as initFLLSettle in addition to setting the proper DCOFTRIM according to clock frequency. This function performs DCO Software Trim and saves the trim value into initFLLParam.
- bool CS_initFLLLoadTrim (uint16_t fsystem, uint16_t ratio, CS_initFLLParam ∗param)
  Performs same function as initFLLCalculateTrim without the overhead of calculating the trim, but rather using the one specified in param. This function corresponds with the DCO Software Trim.
- void CS_enableClockRequest (uint8_t selectClock)
  Enables conditional module requests.
- void CS_disableClockRequest (uint8_t selectClock)
  Disables conditional module requests.
- uint8_t CS_getFaultFlagStatus (uint8_t mask)
  Gets the current CS fault flag status.
- void CS_clearFaultFlag (uint8_t mask)
  Clears the current CS fault flag status for the masked bit.
- uint32_t CS_getACLK (void)
  Get the current ACLK frequency.
- uint32_t CS_getSMCLK (void)
  Get the current SMCLK frequency.
- uint32_t CS_getMCLK (void)
  Get the current MCLK frequency.
- uint16_t CS_clearAllOscFlagsWithTimeout (uint16_t timeout)
  Clears all the Oscillator Flags.
- void CS_enableXT1AutomaticGainControl (void)
  Enables XT1 automatic gain control.
- void CS_disableXT1AutomaticGainControl (void)
  Disables XT1 automatic gain control.
- void CS_enableFLLUnlock (void)
  Enables FLL unlock interrupt.
- void CS_disableFLLUnlock (void)
  Disables FLL unlock interrupt.
- void CS_enableREFOLP (void)
  Enable low-power REFO.
- void CS_disableREFOLP (void)
  Disable low-power REFO.
- bool CS_getREFOLP (void)
  Get status of low-power REFO.
- void CS_enableXT1FaultOff (void)

Turns off switching from XT1 to REFO when XT1 fails.
- void CS_disableXT1FaultOff (void)
  Turns on switching from XT1 to REFO when XT1 fails.
- bool CS_getXT1FaultOff (void)
  Get status of XT1 fault switching.
- bool CS_getREFOReady (void)
  Get status indication of low-power REFO switching.

9.2.1 Detailed Description

The CS API is broken into three groups of functions: those that deal with clock configuration and control

General CS configuration and initialization is handled by

- CS_initClockSignal(),
- CS_initFLLSettle(),
- CS_initFLLCalculateTrim(),
- CS_initFLLLoadTrim(),
- CS_enableClockRequest(),
- CS_disableClockRequest(),
- CS_enableFLLUnlock(),
- CS_disableFLLUnlock(),
- CS_enableREFOLP(),
- CS_disableREFOLP(),

External crystal specific configuration and initialization is handled by

- CS_setExternalClockSource(),
- CS_turnOnXT1LF(),
- CS_turnOnXT1HF(),
- CS_bypassXT1(),
- CS_turnOnXT1LFWithTimeout(),
- CS_turnOnXT1HFWithTimeout(),
- CS_bypassXT1WithTimeout(),
- CS_turnOffXT1(),
- CS_clearAllOscFlagsWithTimeout(),
- CS_turnOffSMCLK(),
- CS_turnOnSMCLK(),
- CS_enableVLOAutoOff(),
- CS_disableVLOAutoOff()
- CS_enableXT1AutomaticGainControl(),
- CS_disableXT1AutomaticGainControl(),
- CS_enableXT1FaultOff(),

- CS_disableXT1FaultOff(),
- CS_getXT1FaultOff(),

CS_setExternalClockSource must be called if an external crystal XT1 is used and the user intends to call CS_getMCLK, CS_getSMCLK or CS_getACLK APIs. If not, it is not necessary to invoke this API.

Failure to invoke CS_initClockSignal() sets the clock signals to the default modes ACLK default mode - CS_XT1CLK_SELECT SMCLK default mode - CS_DCOCLKDIV_SELECT MCLK default mode - CS_DCOCLKDIV_SELECT

Also fail-safe mode behavior takes effect when a selected mode fails.

The status and configuration query are done by

- CS_getFaultFlagStatus(),
- CS_clearFaultFlag(),
- CS_getACLK(),
- CS_getSMCLK(),
- CS_getMCLK(),
- CS_getREFOLP(),
- CS_getREFOReady(),

9.2.2 Function Documentation

CS_bypassXT1()

```
void CS_bypassXT1 (
            void )
```

Bypass the XT1 crystal oscillator.

Bypasses the XT1 crystal oscillator. Loops until all oscillator fault flags are cleared, with no timeout.

Modified bits of **SFRIFG1** register, bits of **CSCTL7** register and bits of **CSCTL6** register.

Returns

None

CS_bypassXT1WithTimeout()

```
bool CS_bypassXT1WithTimeout (
            uint16_t timeout )
```

Bypasses the XT1 crystal oscillator with time out.

Bypasses the XT1 crystal oscillator with time out. Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero.

Parameters

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

Modified bits of **SFRIFG1** register, bits of **CSCTL7** register and bits of **CSCTL6** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

CS_clearAllOscFlagsWithTimeout()

```
uint16_t CS_clearAllOscFlagsWithTimeout (
            uint16_t timeout )
```

Clears all the Oscillator Flags.

Parameters

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

Returns

The mask of the oscillator flag status Return Logical OR of any of the following:
- **CS_XT1OFFG** XT1 oscillator fault flag
- **CS_DCOFFG** DCO fault flag
- **CS_FLLULIFG** FLL unlock interrupt flag
  indicating the status of the osciallator fault flags

CS_clearFaultFlag()

```
void CS_clearFaultFlag (
            uint8_t mask )
```

Clears the current CS fault flag status for the masked bit.

Parameters

mask	is the masked interrupt flag status to be returned. mask parameter can be any one of the following Valid values are: ■ **CS_XT1OFFG** - XT1 oscillator fault flag ■ **CS_DCOFFG** - DCO fault flag ■ **CS_FLLULIFG** - FLL unlock interrupt flag

Modified bits of **CSCTL7** register.

Returns

None

CS_disableClockRequest()

```
void CS_disableClockRequest (
            uint8_t selectClock )
```

Disables conditional module requests.

Parameters

selectClock	selects specific request disable Valid values are: ■ **CS_ACLK** ■ **CS_MCLK** ■ **CS_SMCLK** ■ **CS_MODOSC**

Modified bits of **CSCTL8** register.

Returns

None

CS_disableFLLUnlock()

```
void CS_disableFLLUnlock (
            void )
```

Disables FLL unlock interrupt.

Modified bits are **FLLULIE** of **CSCTL7** register.

Returns

None

CS_disableREFOLP()

```
void CS_disableREFOLP (
            void )
```

Disable low-power REFO.

Modified bits are **REFOLP** of **CSCTL3** register.

Returns

None

CS_disableVLOAutoOff()

```
void CS_disableVLOAutoOff (
            void )
```

VLO is always on.

Returns

None

CS_disableXT1AutomaticGainControl()

```
void CS_disableXT1AutomaticGainControl (
            void )
```

Disables XT1 automatic gain control.

Modified bits of **CSCTL6** register.

Returns

None

CS_disableXT1FaultOff()

```
void CS_disableXT1FaultOff (
            void )
```

Turns on switching from XT1 to REFO when XT1 fails.

Modified bits are **XT1FAULTOFF** of **CSCTL6** register.

Returns

None

CS_enableClockRequest()

```
void CS_enableClockRequest (
            uint8_t selectClock )
```

Enables conditional module requests.

Parameters

selectClock	selects specific request enables Valid values are: ■ **CS_ACLK** ■ **CS_MCLK** ■ **CS_SMCLK** ■ **CS_MODOSC**

Modified bits of **CSCTL8** register.

Returns

None

CS_enableFLLUnlock()

```
void CS_enableFLLUnlock (
            void )
```

Enables FLL unlock interrupt.

Modified bits are **FLLULIE** of **CSCTL7** register.

Returns

None

CS_enableREFOLP()

```
void CS_enableREFOLP (
            void )
```

Enable low-power REFO.

Modified bits are **REFOLP** of **CSCTL3** register.

Returns

None

CS_enableVLOAutoOff()

```
void CS_enableVLOAutoOff (
            void )
```

VLO is turned off when not used.

Returns

None

CS_enableXT1AutomaticGainControl()

```
void CS_enableXT1AutomaticGainControl (
            void )
```

Enables XT1 automatic gain control.

Modified bits of **CSCTL6** register.

Returns

None

CS_enableXT1FaultOff()

```
void CS_enableXT1FaultOff (
            void )
```

Turns off switching from XT1 to REFO when XT1 fails.

Modified bits are **XT1FAULTOFF** of **CSCTL6** register.

Returns

None

CS_getACLK()

```
uint32_t CS_getACLK (
            void )
```

Get the current ACLK frequency.

Get the current ACLK frequency. The user of this API must ensure that CS_setExternalClockSource API was invoked before in case XT1 is being used.

Returns

Current ACLK frequency in Hz

CS_getFaultFlagStatus()

```
uint8_t CS_getFaultFlagStatus (
            uint8_t mask )
```

Gets the current CS fault flag status.

Parameters

mask	is the masked interrupt flag status to be returned. Mask parameter can be either any of the following selection. Valid values are: ■ **CS_XT1OFFG** - XT1 oscillator fault flag ■ **CS_DCOFFG** - DCO fault flag ■ **CS_FLLULIFG** - FLL unlock interrupt flag

Modified bits of **CSCTL7** register.

Returns

The current flag status for the corresponding masked bit

CS_getMCLK()

```
uint32_t CS_getMCLK (
            void )
```

Get the current MCLK frequency.

Get the current MCLK frequency. The user of this API must ensure that CS_setExternalClockSource API was invoked before in case XT1 is being used.

Returns

Current MCLK frequency in Hz

CS_getREFOLP()

```
bool CS_getREFOLP (
            void )
```

Get status of low-power REFO.

Returns

Get status of low-power REFO.

CS_getREFOReady()

```
bool CS_getREFOReady (
            void )
```

Get status indication of low-power REFO switching.

Returns

Get status indication of low-power REFO switching.

CS_getSMCLK()

```
uint32_t CS_getSMCLK (
            void )
```

Get the current SMCLK frequency.

Get the current SMCLK frequency. The user of this API must ensure that CS_setExternalClockSource API was invoked before in case XT1 is being used.

Returns

Current SMCLK frequency in Hz

CS_getXT1FaultOff()

```
bool CS_getXT1FaultOff (
            void )
```

Get status of XT1 fault switching.

Returns

Get status of XT1 fault switching.

CS_initClockSignal()

```
void CS_initClockSignal (
            uint8_t selectedClockSignal,
            uint16_t clockSource,
            uint16_t clockSourceDivider )
```

Initializes a clock signal.

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. Please check the device specific data sheet for details on the following: Some devices do not support divider settings for **CS_FLLREF**. VLO is only a valid clock source for ACLK on some devices.

Parameters

selectedClockSignal	selected clock signal Valid values are: ■ **CS_ACLK** ■ **CS_MCLK** ■ **CS_SMCLK** ■ **CS_FLLREF**

Parameters

clockSource	is clock source for the selectedClockSignal Valid values are: ■ **CS_XT1CLK_SELECT** ■ **CS_VLOCLK_SELECT** ■ **CS_REFOCLK_SELECT** ■ **CS_DCOCLKDIV_SELECT**
clockSourceDivider	selected the clock divider to calculate clocksignal from clock source. Valid values are: ■ **CS_CLOCK_DIVIDER_1** [Default] - [Valid for CS_FLLREF, CS_MCLK, CS_ACLK, CS_SMCLK] ■ **CS_CLOCK_DIVIDER_2** - [Valid for CS_MCLK, CS_SMCLK] ■ **CS_CLOCK_DIVIDER_4** - [Valid for CS_MCLK, CS_SMCLK] ■ **CS_CLOCK_DIVIDER_8** - [Valid for CS_MCLK, CS_SMCLK] ■ **CS_CLOCK_DIVIDER_16** - [Valid for CS_MCLK, CS_ACLK] ■ **CS_CLOCK_DIVIDER_32** - [Valid for CS_FLLREF, CS_MCLK, CS_ACLK] ■ **CS_CLOCK_DIVIDER_64** - [Valid for CS_FLLREF, CS_MCLK, CS_ACLK] ■ **CS_CLOCK_DIVIDER_128** - [Valid for CS_FLLREF, CS_MCLK, CS_ACLK] ■ **CS_CLOCK_DIVIDER_256** - [Valid for CS_FLLREF, CS_ACLK] ■ **CS_CLOCK_DIVIDER_384** - [Valid for CS_FLLREF, CS_ACLK] ■ **CS_CLOCK_DIVIDER_512** - [Valid for CS_FLLREF, CS_ACLK] ■ **CS_CLOCK_DIVIDER_768** - [Valid for CS_FLLREF, CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 24 MHz preference] ■ **CS_CLOCK_DIVIDER_1024** - [Valid for CS_FLLREF, CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 32 MHz preference] ■ **CS_CLOCK_DIVIDER_108** - [Valid for CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 3.5712 MHz preference] ■ **CS_CLOCK_DIVIDER_338** - [Valid for CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 11.0592 MHz preference] ■ **CS_CLOCK_DIVIDER_414** - [Valid for CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 13.56 MHz preference] ■ **CS_CLOCK_DIVIDER_640** - [Valid for CS_FLLREF, CS_ACLK] [Only available in 24MHz clock system] [If CS_ACLK, 20.00 MHz preference]

Modified bits of **CSCTL3** register, bits of **CSCTL5** register and bits of **CSCTL4** register.

Returns

None

CS_initFLL()

```
bool CS_initFLL (
            uint16_t fsystem,
            uint16_t ratio )
```

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. This function performs DCO Factory Trim.

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. Loops until all oscillator fault flags are cleared, with a timeout. If the frequency is greater than clock system allows, the function sets the MCLK and SMCLK source to the undivided DCO frequency and returns false. Otherwise, the function sets the MCLK and SMCLK source to the DCOCLKDIV frequency.

Parameters

fsystem	is the target frequency for MCLK in kHz
ratio	is the ratio x/y, where x = fsystem and y = FLL reference frequency.

Modified bits of **CSCTL1** register, bits of **CSCTL0** register, bits of **CSCTL2** register, bits of **CSCTL4** register, bits of **CSCTL7** register and bits of **SFRIFG1** register.

Returns

True if successful, false if unsuccessful and resorted to undivided DCO frequency for MCLK and SMCLK source

Referenced by CS_initFLLSettle().

CS_initFLLCalculateTrim()

```
bool CS_initFLLCalculateTrim (
            uint16_t fsystem,
            uint16_t ratio,
            CS_initFLLParam * param )
```

Performs same function as initFLLSettle in addition to setting the proper DCOFTRIM according to clock frequency. This function performs DCO Software Trim and saves the trim value into initFLLParam.

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. Loops until all oscillator fault flags are cleared, with a timeout. If the frequency is greater than clock system allows, the function sets the MCLK and SMCLK source to the undivided DCO frequency and returns false. Otherwise, the function sets the MCLK and SMCLK source to the DCOCLKDIV frequency. This function executes a software delay that is proportional in length to the ratio of the target FLL frequency and the FLL reference. It also calibrates the DCOFTRIM value according to clock frequency. Lastly, it saves the DCOTAP and DCOFTRIM values for future use.

Parameters

fsystem	is the target frequency for MCLK in kHz
ratio	is the ratio x/y, where x = fsystem and y = FLL reference frequency.

Modified bits of **CSCTL1** register, bits of **CSCTL0** register, bits of **CSCTL2** register, bits of **CSCTL4** register, bits of **CSCTL7** register and bits of **SFRIFG1** register.

Returns

True if successful, false if unsuccessful and resorted to undivided DCO frequency for MCLK and SMCLK source

References CS_initFLLParam::fsystem.

CS_initFLLLoadTrim()

```
bool CS_initFLLLoadTrim (
            uint16_t fsystem,
            uint16_t ratio,
            CS_initFLLParam * param )
```

Performs same function as initFLLCalculateTrim without the overhead of calculating the trim, but rather using the one specified in param. This function corresponds with the DCO Software Trim.

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. Loops until all oscillator fault flags are cleared, with a timeout. If the frequency is greater than clock system allows, the function sets the MCLK and SMCLK source to the undivided DCO frequency and returns false. Otherwise, the function sets the MCLK and SMCLK source to the DCOCLKDIV frequency. This function executes a software delay that is proportional in length to the ratio of the target FLL frequency and the FLL reference. Lastly, it uses the saved DCOTAP and DCOFTRIM values from the param to avoid overhead in recalculation.

Parameters

fsystem	is the target frequency for MCLK in kHz
ratio	is the ratio x/y, where x = fsystem and y = FLL reference frequency.

Modified bits of **CSCTL1** register, bits of **CSCTL0** register, bits of **CSCTL2** register, bits of **CSCTL4** register, bits of **CSCTL7** register and bits of **SFRIFG1** register.

Returns

True if initialization successful, false if saved DCOFTRIM value is not for the correct clock frequency combination or resorted to undivided DCO frequency for MCLK and SMCLK source

References CS_initFLLParam::csCtl0, CS_initFLLParam::csCtl1, and CS_initFLLParam::fsystem.

CS_initFLLSettle()

```
bool CS_initFLLSettle (
            uint16_t fsystem,
            uint16_t ratio )
```

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL.

Initializes the DCO to operate a frequency that is a multiple of the reference frequency into the FLL. Loops until all oscillator fault flags are cleared, with a timeout. If the frequency is greater than clock system allows, the function sets the MCLK and SMCLK source to the undivided DCO frequency and returns false. Otherwise, the function sets the MCLK and SMCLK source to the DCOCLKDIV frequency. This function executes a software delay that is proportional in length to the ratio of the target FLL frequency and the FLL reference.

Parameters

fsystem	is the target frequency for MCLK in kHz
ratio	is the ratio x/y, where x = fsystem and y = FLL reference frequency.

Modified bits of **CSCTL1** register, bits of **CSCTL0** register, bits of **CSCTL2** register, bits of **CSCTL4** register, bits of **CSCTL7** register and bits of **SFRIFG1** register.

Returns

True if successful, false if unsuccessful and resorted to undivided DCO frequency for MCLK and SMCLK source

References CS_initFLL().

CS_setExternalClockSource()

```
void CS_setExternalClockSource (
            uint32_t XT1CLK_frequency )
```

Sets the external clock source.

This function sets the external clock sources XT1 crystal oscillator frequency values. This function must be called if an external crystal XT1 is used and the user intends to call CS_getMCLK, CS_getSMCLK or CS_getACLK APIs. If not, it is not necessary to invoke this API.

Parameters

XT1CLK_frequency	is the XT1 crystal frequencies in Hz

Returns

None

CS_turnOffSMCLK()

```
void CS_turnOffSMCLK (
            void )
```

Turn Off SMCLK.

Returns

None

CS_turnOffXT1()

```
void CS_turnOffXT1 (
            void )
```

Stops the XT1 oscillator using the XT1AUTOOFF bit.

Modified bits are **XT1AUTOOFF** of **CSCTL6** register.

Returns

None

CS_turnOnSMCLK()

```
void CS_turnOnSMCLK (
            void )
```

Turn On SMCLK.

Returns

None

CS_turnOnXT1HF()

```
void CS_turnOnXT1HF (
            uint16_t xt1Drive,
            uint16_t xt1HFFreq )
```

Intializes the XT1 crystal oscillator in high frequency mode.

Initializes the XT1 crystal oscillator in high frequency mode. Loops until all oscillator fault flags are cleared, with no timeout. See the device- specific data sheet for appropriate drive settings.

Parameters

xt1Drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default] Modified bits are **XT1DRIVE** of **UCSCTL6** register.
xt1HFFreq	is the high frequency range selection. Valid values are: ■ **CS_XT1_HFFREQ_1MHZ_4MHZ** [Default] - 1 MHz to 4 MHz ■ **CS_XT1_HFFREQ_4MHZ_6MHZ** - Above 4 MHz to 6 MHz ■ **CS_XT1_HFFREQ_6MHZ_16MHZ** - Above 6 MHz to 16 MHz ■ **CS_XT1_HFFREQ_16MHZ_24MHZ** - Above 16 MHz to 24 MHz (Only available in 24MHz clock system)

Returns

None

CS_turnOnXT1HFWithTimeout()

```
bool CS_turnOnXT1HFWithTimeout (
            uint16_t xt1Drive,
            uint16_t xt1HFFreq,
            uint16_t timeout )
```

Initializes the XT1 crystal oscillator in high frequency mode with timeout.

Initializes the XT1 crystal oscillator in high frequency mode with timeout. Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero. See the device-specific datasheet for appropriate drive settings.

Parameters

xt1Drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default]

Parameters

xt1HFFreq	is the high frequency range selection. Valid values are: ■ **CS_XT1_HFFREQ_1MHZ_4MHZ** [Default] - 1 MHz to 4 MHz ■ **CS_XT1_HFFREQ_4MHZ_6MHZ** - Above 4 MHz to 6 MHz ■ **CS_XT1_HFFREQ_6MHZ_16MHZ** - Above 6 MHz to 16 MHz ■ **CS_XT1_HFFREQ_16MHZ_24MHZ** - Above 16 MHz to 24 MHz (Only available in 24MHz clock system)
timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Modified bits of **SFRIFG1** register, bits of **CSCTL7** register and bits of **CSCTL6** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

CS_turnOnXT1LF()

```
void CS_turnOnXT1LF (
            uint16_t xt1Drive )
```

Intializes the XT1 crystal oscillator in low frequency mode.

Initializes the XT1 crystal oscillator in low frequency mode. Loops until all oscillator fault flags are cleared, with no timeout. See the device- specific data sheet for appropriate drive settings.

Parameters

xt1Drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default] Modified bits are **XT1DRIVE** of **UCSCTL6** register.

Returns

None

CS_turnOnXT1LFWithTimeout()

```
bool CS_turnOnXT1LFWithTimeout (
            uint16_t xt1Drive,
            uint16_t timeout )
```

Initializes the XT1 crystal oscillator in low frequency mode with timeout.

Initializes the XT1 crystal oscillator in low frequency mode with timeout. Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero. See the device-specific datasheet for appropriate drive settings.

Parameters

xt1Drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default]
timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Modified bits of **SFRIFG1** register, bits of **CSCTL7** register and bits of **CSCTL6** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

9.3 Programming Example

The following example shows some CS operations using the APIs

```
//Target frequency for MCLK in kHz
#define CS_MCLK_DESIRED_FREQUENCY_IN_KHZ   12000
//MCLK/FLLRef Ratio
#define CS_MCLK_FLLREF_RATIO   366
//Variable to store current Clock values
uint32_t clockValue = 0;

  // Set DCO FLL reference = REFO
  CS_initClockSignal(CS_BASE,
                     CS_FLLREF,
                     CS_REFOCLK_SELECT,
                     CS_CLOCK_DIVIDER_1
                      );
  // Set ACLK = REFO
  CS_initClockSignal(CS_BASE,
                     CS_ACLK,
                     CS_REFOCLK_SELECT,
                     CS_CLOCK_DIVIDER_1
                      );

  // Set Ratio and Desired MCLK Frequency  and initialize DCO
  CS_initFLLSettle(CS_BASE,
                    CS_MCLK_DESIRED_FREQUENCY_IN_KHZ,
                    CS_MCLK_FLLREF_RATIO
                    );

  //Verify if the Clock settings are as expected
  clockValue = CS_getSMCLK (CS_BASE);

  while(1);
```

10 Enhanced Comparator (eCOMP)

10.1 Introduction

Enhanced Comparator (eCOMP) is an analog voltage comparator with internal reference DAC. The eCOMP supports up to 7 channels including 4 external inputs, 2 external inputs, and one reference from DAC output. It also implements programmable hysteresis and power modes.

The API provides a set of functions for using the eCOMP module. Functions are provided to initialize the eCOMP module, setup reference voltages for input, and manage interrupts for the eCOMP module.

10.2 API Functions

Functions

- void ECcomp_init (uint16_t baseAddress, EComp_initParam ∗param)
  Initializes the EComp.
- void EComp_selectHysteresisMode (uint16_t baseAddress, uint16_t hysteresisMode)
  Sets the hysteresis mode.
- void EComp_selectPowerMode (uint16_t baseAddress, uint16_t powerMode)
  Sets the power mode.
- void EComp_enable (uint16_t baseAddress)
  Turns on the EComp module.
- void EComp_disable (uint16_t baseAddress)
  Turns off the EComp module.
- void EComp_enableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Enables selected EComp interrupt sources.
- void EComp_disableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Disables selected EComp interrupt sources.
- void EComp_clearInterrupt (uint16_t baseAddress, uint16_t interruptFlagMask)
  Clears EComp interrupt flags.
- uint8_t EComp_getInterruptStatus (uint16_t baseAddress, uint16_t interruptFlagMask)
  Gets the current EComp interrupt status.
- void EComp_setInterruptEdgeDirection (uint16_t baseAddress, uint16_t edgeDirection)
  Explicitly sets the edge direction that would trigger an interrupt.
- void EComp_toggleInterruptEdgeDirection (uint16_t baseAddress)
  Toggles the edge direction that would trigger an interrupt.
- uint8_t EComp_outputValue (uint16_t baseAddress)
  Returns the output value of the EComp module.
- void EComp_configureDAC (uint16_t baseAddress, EComp_configureDACParam ∗param)
  Configures the built-in DAC for internal reference.
- void EComp_enableDAC (uint16_t baseAddress)

Enables DAC output.

- void EComp_disableDAC (uint16_t baseAddress)

  Disables DAC output.

10.2.1 Detailed Description

The API is broken into three groups of functions: those that deal with initialization and output, those that handle interrupts, and those that handle Auxiliary features of the eCOMP.

The eCOMP initialization and output functions are

- EComp_init()
- EComp_enable()
- EComp_disable()
- EComp_enableDAC()
- EComp_disableDAC()
- EComp_configurDAC()
- EComp_outputValue()

The eCOMP interrupts are handled by

- EComp_enableInterrupt()
- EComp_disableInterrupt()
- EComp_clearInterrupt()
- EComp_getInterruptStatus()
- EComp_setInterruptEdgeDirection()
- EComp_toggleInterruptEdgeDirection()

Auxiliary features of the eCOMP are handled by

- EComp_selectHysteresisMode()
- EComp_selectPowerMode()

10.2.2 Function Documentation

EComp_clearInterrupt()

```
void EComp_clearInterrupt (
            uint16_t baseAddress,
            uint16_t interruptFlagMask )
```

Clears EComp interrupt flags.

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

Parameters

baseAddress	is the base address of the ECOMP module.

Parameters

interruptFlagMask	Mask value is the logical OR of any of the following: ■ **ECOMP_OUTPUT_INTERRUPT_FLAG** - Output interrupt flag ■ **ECOMP_INVERTED_POLARITY_INTERRUPT_FLAG** - Output interrupt flag inverted polarity

Returns

None

EComp_configureDAC()

```
void EComp_configureDAC (
            uint16_t baseAddress,
            EComp_configureDACParam * param )
```

Configures the built-in DAC for internal reference.

This function will configure the built-in DAC register bits including reference voltage and DAC buffer source.

Parameters

baseAddress	is the base address of the ECOMP module.

Returns

None

References EComp_configureDACParam::bufferSource, EComp_configureDACParam::firstBufferData, EComp_configureDACParam::referenceVoltage, and EComp_configureDACParam::secondBufferData.

EComp_disable()

```
void EComp_disable (
            uint16_t baseAddress )
```

Turns off the EComp module.

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

Parameters

baseAddress	is the base address of the ECOMP module.

Modified bits are **CPEN** of **CPCTL1** register.

Returns

None

EComp_disableDAC()

```
void EComp_disableDAC (
            uint16_t baseAddress )
```

Disables DAC output.

This function will disable DAC output. When it is disabled, the DAC always output low.

Parameters

baseAddress	is the base address of the ECOMP module.

Returns

None

EComp_disableInterrupt()

```
void EComp_disableInterrupt (
            uint16_t baseAddress,
            uint16_t interruptMask )
```

Disables selected EComp interrupt sources.

Disables the indicated EComp 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 ECOMP module.
interruptMask	Mask value is the logical OR of any of the following: ■ **ECOMP_OUTPUT_INTERRUPT** - Output interrupt ■ **ECOMP_INVERTED_POLARITY_INTERRUPT** - Output interrupt inverted polarity

Returns

None

EComp_enable()

```
void EComp_enable (
```

```
uint16_t baseAddress )
```

Turns on the EComp module.

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

Parameters

baseAddress	is the base address of the ECOMP module.

Modified bits are **CPEN** of **CPCTL1** register.

Returns

None

EComp_enableDAC()

```
void EComp_enableDAC (
            uint16_t baseAddress )
```

Enables DAC output.

This function will enable DAC output.

Parameters

baseAddress	is the base address of the ECOMP module.

Returns

None

EComp_enableInterrupt()

```
void EComp_enableInterrupt (
            uint16_t baseAddress,
            uint16_t interruptMask )
```

Enables selected EComp interrupt sources.

Enables the indicated EComp 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 ECOMP module.

Parameters

interruptMask	Mask value is the logical OR of any of the following: ■ **ECOMP_OUTPUT_INTERRUPT** - Output interrupt ■ **ECOMP_INVERTED_POLARITY_INTERRUPT** - Output interrupt inverted polarity

Returns

None

EComp_getInterruptStatus()

```
uint8_t EComp_getInterruptStatus (
            uint16_t baseAddress,
            uint16_t interruptFlagMask )
```

Gets the current EComp interrupt status.

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

Parameters

baseAddress	is the base address of the ECOMP module.
interruptFlagMask	Mask value is the logical OR of any of the following: ■ **ECOMP_OUTPUT_INTERRUPT_FLAG** - Output interrupt flag ■ **ECOMP_INVERTED_POLARITY_INTERRUPT_FLAG** - Output interrupt flag inverted polarity

Returns

Logical OR of any of the following:

- **ECOMP_OUTPUT_INTERRUPT_FLAG** Output interrupt flag
- **ECOMP_INVERTED_POLARITY_INTERRUPT_FLAG** Output interrupt flag inverted polarity
  indicating the status of the masked flags

EComp_init()

```
void EComp_init (
            uint16_t baseAddress,
            EComp_initParam * param )
```

Initializes the EComp.

Upon successful initialization of the EComp module, this function will have reset all necessary register bits and set the given options in the registers. To actually use the EComp_E module, the

EComp_enable() function must be explicitly called before use. (If a Reference Voltage is set to a terminal, the Voltage should be set using the EComp_configureDAC() function.)

Parameters

baseAddress	is the base address of the ECOMP module.

Returns

None

References EComp_initParam::invertedOutputPolarity, EComp_initParam::negativeTerminalInput, EComp_initParam::outputFilterEnableAndDelayLevel, and EComp_initParam::positiveTerminalInput.

EComp_outputValue()

```
uint8_t EComp_outputValue (
            uint16_t baseAddress )
```

Returns the output value of the EComp module.

Returns the output value of the EComp module.

Parameters

baseAddress	is the base address of the ECOMP module.

Returns

indicating the output value of the EComp module Return one of the following:

- **ECOMP_LOW**
- **ECOMP_HIGH**
  indicating the output value of the EComp module

EComp_selectHysteresisMode()

```
void EComp_selectHysteresisMode (
            uint16_t baseAddress,
            uint16_t hysteresisMode )
```

Sets the hysteresis mode.

This function sets the hysteresis mode.

Parameters

baseAddress	is the base address of the ECOMP module.

Parameters

hysteresisMode	decides the hysteresis mode Valid values are: ■ **ECOMP_HYSTERESIS_MODE_DISABLE** [Default] ■ **ECOMP_HYSTERESIS_MODE_10MV** ■ **ECOMP_HYSTERESIS_MODE_20MV** ■ **ECOMP_HYSTERESIS_MODE_30MV**

Modified bits are **CPHSEL** of **CPCTL1** register.

Returns

None

EComp_selectPowerMode()

```
void EComp_selectPowerMode (
            uint16_t baseAddress,
            uint16_t powerMode )
```

Sets the power mode.

This function sets the power mode.

Parameters

baseAddress	is the base address of the ECOMP module.
powerMode	decides the power mode Valid values are: ■ **ECOMP_POWER_MODE_HIGH_POWER_HIGH_SPEED** [Default] ■ **ECOMP_POWER_MODE_LOW_POWER_LOW_SPEED**

Returns

None

EComp_setInterruptEdgeDirection()

```
void EComp_setInterruptEdgeDirection (
            uint16_t baseAddress,
            uint16_t edgeDirection )
```

Explicitly sets the edge direction that would trigger an interrupt.

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.

Parameters

baseAddress	is the base address of the ECOMP module.
edgeDirection	determines which direction the edge would have to go to generate an interrupt based on the non-inverted interrupt flag. Valid values are: ■ **ECOMP_OUTPUT_INTERRUPT_RISING_EDGE** [Default] - sets the bit to generate an interrupt when the output of the EComp rises from LOW to HIGH if the normal interrupt bit is set(and HIGH to LOW if the inverted interrupt enable bit is set). ■ **ECOMP_OUTPUT_INTERRUPT_FALLING_EDGE** - sets the bit to generate an interrupt when the output of the EComp falls from HIGH to LOW if the normal interrupt bit is set(and LOW to HIGH if the inverted interrupt enable bit is set). Modified bits are **CPIES** of **CPCTL1** register.

Returns

None

EComp_toggleInterruptEdgeDirection()

```
void EComp_toggleInterruptEdgeDirection (
            uint16_t baseAddress )
```

Toggles the edge direction that would trigger an interrupt.

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.

Parameters

baseAddress	is the base address of the ECOMP module.

Modified bits are **CPIES** of **CPCTL1** register.

Returns

None

10.3 Programming Example

The following example shows how to initialize eCOMP and DAC

```
EComp_initParam param = {0};
param.positiveTerminalInput = ECOMP_INPUT_0;
param.negativeTerminalInput = ECOMP_INPUT_DAC;
param.outputFilterEnableAndDelayLevel = ECOMP_FILTER_DELAY_OFF;
param.invertedOutputPolarity = ECOMP_NORMAL_OUTPUT_POLARITY;
```

```
EComp_init(ECOMP_BASE, &param);

//Set the reference voltage that is outputed by built-in DAC
//Vref' = Vref*(63/64)
EComp_configureDACParam dacParam = {0};
dacParam.referenceVoltage = ECOMP_DAC_REFERENCE_VOLTAGE_VREF;
dacParam.bufferSource = ECOMP_DAC_BUFFER_SOURCE_DUAL_BUFFER_1;
dacParam.firstBufferData = 63;
EComp_configureDAC(ECOMP_BASE, &dacParam);
EComp_enableDAC(ECOMP_BASE);

//Select low power low speed mode
EComp_selectPowerMode(ECOMP_BASE, ECOMP_POWER_MODE_LOW_POWER_LOW_SPEED);

EComp_clearInterrupt(ECOMP_BASE,
    ECOMP_OUTPUT_INTERRUPT_FLAG
    );

EComp_enableInterrupt(ECOMP_BASE,
    ECOMP_OUTPUT_INTERRUPT
    );

//Allow power to Comparator module
EComp_enable(ECOMP_BASE);

__bis_SR_register(LPM4_bits);          // Enter LPM4
__no_operation();                      // For debug
```

11 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)

11.1 Introduction

The MSP430Ware 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
- Receiver start-edge detection for auto wake up from LPMx modes
- Status flags for error detection and suppression
- Status flags for address detection
- Independent interrupt capability for receive and transmit

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.

11.2 API Functions

Functions

- bool EUSCI_A_UART_init (uint16_t baseAddress, EUSCI_A_UART_initParam *param)
  Advanced initialization routine for the UART block. The values to be written into the clockPrescalar, firstModReg, secondModReg and overSampling parameters should be pre-computed and passed into the initialization function.
- void EUSCI_A_UART_transmitData (uint16_t baseAddress, uint8_t transmitData)
  Transmits a byte from the UART Module.Please note that if TX interrupt is disabled, this function manually polls the TX IFG flag waiting for an indication that it is safe to write to the transmit buffer and does not time-out.
- uint8_t EUSCI_A_UART_receiveData (uint16_t baseAddress)
  Receives a byte that has been sent to the UART Module.
- void EUSCI_A_UART_enableInterrupt (uint16_t baseAddress, uint8_t mask)
  Enables individual UART interrupt sources.
- void EUSCI_A_UART_disableInterrupt (uint16_t baseAddress, uint8_t mask)
  Disables individual UART interrupt sources.
- uint8_t EUSCI_A_UART_getInterruptStatus (uint16_t baseAddress, uint8_t mask)

Gets the current UART interrupt status.
- void EUSCI_A_UART_clearInterrupt (uint16_t baseAddress, uint16_t mask)
  Clears UART interrupt sources.
- void EUSCI_A_UART_enable (uint16_t baseAddress)
  Enables the UART block.
- void EUSCI_A_UART_disable (uint16_t baseAddress)
  Disables the UART block.
- uint8_t EUSCI_A_UART_queryStatusFlags (uint16_t baseAddress, uint8_t mask)
  Gets the current UART status flags.
- void EUSCI_A_UART_setDormant (uint16_t baseAddress)
  Sets the UART module in dormant mode.
- void EUSCI_A_UART_resetDormant (uint16_t baseAddress)
  Re-enables UART module from dormant mode.
- void EUSCI_A_UART_transmitAddress (uint16_t baseAddress, uint8_t transmitAddress)
  Transmits the next byte to be transmitted marked as address depending on selected multiprocessor mode.
- void EUSCI_A_UART_transmitBreak (uint16_t baseAddress)
  Transmit break.
- uint32_t EUSCI_A_UART_getReceiveBufferAddress (uint16_t baseAddress)
  Returns the address of the RX Buffer of the UART for the DMA module.
- uint32_t EUSCI_A_UART_getTransmitBufferAddress (uint16_t baseAddress)
  Returns the address of the TX Buffer of the UART for the DMA module.
- void EUSCI_A_UART_selectDeglitchTime (uint16_t baseAddress, uint16_t deglitchTime)
  Sets the deglitch time.
- void EUSCI_A_UART_remapPins (uint16_t baseAddress, uint8_t pinsSelect)
  Remaps eUSCI_A GPIO pins.

11.2.1 Detailed Description

The EUSI_A_UART API provides the set of functions required to implement an interrupt driven EUSI_A_UART driver. The EUSI_A_UART initialization with the various modes and features is done by the EUSCI_A_UART_init(). At the end of this function EUSI_A_UART is initialized and stays disabled. EUSCI_A_UART_enable() enables the EUSI_A_UART and the module is now ready for transmit and receive. It is recommended to initialize the EUSI_A_UART via EUSCI_A_UART_init(), enable the required interrupts and then enable EUSI_A_UART via EUSCI_A_UART_enable().

The EUSI_A_UART API is broken into three groups of functions: those that deal with configuration and control of the EUSI_A_UART modules, those used to send and receive data, and those that deal with interrupt handling and those dealing with DMA.

Configuration and control of the EUSI_UART are handled by the

- EUSCI_A_UART_init()
- EUSCI_A_UART_initAdvance()
- EUSCI_A_UART_enable()
- EUSCI_A_UART_disable()
- EUSCI_A_UART_setDormant()
- EUSCI_A_UART_resetDormant()
- EUSCI_A_UART_selectDeglitchTime()

Sending and receiving data via the EUSI_UART is handled by the

- EUSCI_A_UART_transmitData()
- EUSCI_A_UART_receiveData()
- EUSCI_A_UART_transmitAddress()
- EUSCI_A_UART_transmitBreak()
- EUSCI_A_UART_getTransmitBufferAddress()
- EUSCI_A_UART_getTransmitBufferAddress()

Managing the EUSI_UART interrupts and status are handled by the

- EUSCI_A_UART_enableInterrupt()
- EUSCI_A_UART_disableInterrupt()
- EUSCI_A_UART_getInterruptStatus()
- EUSCI_A_UART_clearInterrupt()
- EUSCI_A_UART_queryStatusFlags()

11.2.2 Function Documentation

EUSCI_A_UART_clearInterrupt()

```
void EUSCI_A_UART_clearInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Clears UART interrupt sources.

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.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is a bit mask of the interrupt sources to be cleared. 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**

Modified bits of **UCAxIFG** register.

Returns

None

EUSCI_A_UART_disable()

```
void EUSCI_A_UART_disable (
            uint16_t baseAddress )
```

Disables the UART block.

This will disable operation of the UART block.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits are **UCSWRST** of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_disableInterrupt()

```
void EUSCI_A_UART_disableInterrupt (
            uint16_t baseAddress,
            uint8_t mask )
```

Disables individual UART interrupt sources.

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.

Parameters

baseAddress	is the base address of the EUSCI_A_UART 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_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 ■ **EUSCI_A_UART_STARTBIT_INTERRUPT** - Start bit received interrupt enable ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT** - Transmit complete interrupt enable

Modified bits of **UCAxCTL1** register and bits of **UCAxIE** register.

Returns

None

EUSCI_A_UART_enable()

```
void EUSCI_A_UART_enable (
            uint16_t baseAddress )
```

Enables the UART block.

This will enable operation of the UART block.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits are **UCSWRST** of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_enableInterrupt()

```
void EUSCI_A_UART_enableInterrupt (
            uint16_t baseAddress,
            uint8_t mask )
```

Enables individual UART interrupt sources.

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. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Parameters

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_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 ■ **EUSCI_A_UART_STARTBIT_INTERRUPT** - Start bit received interrupt enable ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT** - Transmit complete interrupt enable

Modified bits of **UCAxCTL1** register and bits of **UCAxIE** register.

Returns

None

EUSCI_A_UART_getInterruptStatus()

```
uint8_t EUSCI_A_UART_getInterruptStatus (
            uint16_t baseAddress,
            uint8_t mask )
```

Gets the current UART interrupt status.

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
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**

Modified bits of **UCAxIFG** register.

Returns

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**
  indicating the status of the masked flags

EUSCI_A_UART_getReceiveBufferAddress()

```
uint32_t EUSCI_A_UART_getReceiveBufferAddress (
            uint16_t baseAddress )
```

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

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.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Returns

Address of RX Buffer

EUSCI_A_UART_getTransmitBufferAddress()

```
uint32_t EUSCI_A_UART_getTransmitBufferAddress (
            uint16_t baseAddress )
```

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

Returns the address of the UART 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_UART module.

Returns

Address of TX Buffer

EUSCI_A_UART_init()

```
bool EUSCI_A_UART_init (
            uint16_t baseAddress,
            EUSCI_A_UART_initParam * param )
```

Advanced initialization routine for the UART block. The values to be written into the clockPrescalar, firstModReg, secondModReg and overSampling parameters should be pre-computed and passed into the initialization function.

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 EUSCI_A_UART_enable(). To calculate values for clockPrescalar, firstModReg, secondModReg and overSampling please use the link below.

`http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.html`

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
param	is the pointer to struct for initialization.

Modified bits are **UCPEN**, **UCPAR**, **UCMSB**, **UC7BIT**, **UCSPB**, **UCMODEx** and **UCSYNC** of **UCAxCTL0** register; bits **UCSSELx** and **UCSWRST** of **UCAxCTL1** register.

Returns

STATUS_SUCCESS or STATUS_FAIL of the initialization process

References EUSCI_A_UART_initParam::clockPrescalar, EUSCI_A_UART_initParam::firstModReg, EUSCI_A_UART_initParam::msborLsbFirst, EUSCI_A_UART_initParam::numberofStopBits, EUSCI_A_UART_initParam::overSampling, EUSCI_A_UART_initParam::parity, EUSCI_A_UART_initParam::secondModReg, EUSCI_A_UART_initParam::selectClockSource, and EUSCI_A_UART_initParam::uartMode.

EUSCI_A_UART_queryStatusFlags()

```
uint8_t EUSCI_A_UART_queryStatusFlags (
            uint16_t baseAddress,
            uint8_t mask )
```

Gets the current UART status flags.

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_LISTEN_ENABLE** ■ **EUSCI_A_UART_FRAMING_ERROR** ■ **EUSCI_A_UART_OVERRUN_ERROR** ■ **EUSCI_A_UART_PARITY_ERROR** ■ **EUSCI_A_UART_BREAK_DETECT** ■ **EUSCI_A_UART_RECEIVE_ERROR** ■ **EUSCI_A_UART_ADDRESS_RECEIVED** ■ **EUSCI_A_UART_IDLELINE** ■ **EUSCI_A_UART_BUSY**

Modified bits of **UCAxSTAT** register.

Returns

Logical OR of any of the following:

- **EUSCI_A_UART_LISTEN_ENABLE**
- **EUSCI_A_UART_FRAMING_ERROR**
- **EUSCI_A_UART_OVERRUN_ERROR**
- **EUSCI_A_UART_PARITY_ERROR**
- **EUSCI_A_UART_BREAK_DETECT**
- **EUSCI_A_UART_RECEIVE_ERROR**
- **EUSCI_A_UART_ADDRESS_RECEIVED**
- **EUSCI_A_UART_IDLELINE**
- **EUSCI_A_UART_BUSY**
  indicating the status of the masked interrupt flags

EUSCI_A_UART_receiveData()

```
uint8_t EUSCI_A_UART_receiveData (
            uint16_t baseAddress )
```

Receives a byte that has been sent to the UART Module.

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits of **UCAxRXBUF** register.

Returns

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

EUSCI_A_UART_remapPins()

```
void EUSCI_A_UART_remapPins (
            uint16_t baseAddress,
            uint8_t pinsSelect )
```

Remaps eUSCI_A GPIO pins.

Remaps eUSCI_A GPIO pins. After calling this function, GPIO_setAsPeripheralModuleFunctionInputPin() or GPIO_setAsPeripheralModuleFunctionInputPin() still needs to be invoked to set peripheral functions. Caution: this will also remap eusci_a_spi GPIO pins.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Parameters

pinsSelect	remapping pins to select. Please refer to device specific datasheet for remapping pins details. Valid values are: ■ **EUSCI_A_UART_REMAP_PINS_FALSE** [Default] ■ **EUSCI_A_UART_REMAP_PINS_TRUE**

Returns

None

EUSCI_A_UART_resetDormant()

```
void EUSCI_A_UART_resetDormant (
            uint16_t baseAddress )
```

Re-enables UART module from dormant mode.

Not dormant. All received characters set UCRXIFG.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits are **UCDORM** of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_selectDeglitchTime()

```
void EUSCI_A_UART_selectDeglitchTime (
            uint16_t baseAddress,
            uint16_t deglitchTime )
```

Sets the deglitch time.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
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

None

EUSCI_A_UART_setDormant()

```
void EUSCI_A_UART_setDormant (
            uint16_t baseAddress )
```

Sets the UART module in dormant 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 sync field sets UCRXIFG.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_transmitAddress()

```
void EUSCI_A_UART_transmitAddress (
            uint16_t baseAddress,
            uint8_t transmitAddress )
```

Transmits the next byte to be transmitted marked as address depending on selected multiprocessor mode.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
transmitAddress	is the next byte to be transmitted

Modified bits of **UCAxTXBUF** register and bits of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_transmitBreak()

```
void EUSCI_A_UART_transmitBreak (
            uint16_t baseAddress )
```

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/sync fields. Otherwise, DEFAULT_SYNC(0x00) must be written into the transmit buffer. Also ensures module is ready for transmitting the next data.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits of **UCAxTXBUF** register and bits of **UCAxCTL1** register.

Returns

None

EUSCI_A_UART_transmitData()

```
void EUSCI_A_UART_transmitData (
            uint16_t baseAddress,
            uint8_t transmitData )
```

Transmits a byte from the UART Module.Please note that if TX interrupt is disabled, this function manually polls the TX IFG flag waiting for an indication that it is safe to write to the transmit buffer and does not time-out.

This function will place the supplied data into UART transmit data register to start transmission

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
transmitData	data to be transmitted from the UART module

Modified bits of **UCAxTXBUF** register.

Returns

None

11.3 Programming Example

The following example shows how to use the EUSI_UART API to initialize the EUSI_UART, transmit characters, and receive characters.

```
// Configure UART
  EUSCI_A_UART_initParam param = {0};
  param.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_ACLK;
  param.clockPrescalar = 15;
  param.firstModReg = 0;
```

```
param.secondModReg = 68;
param.parity = EUSCI_A_UART_NO_PARITY;
param.msborLsbFirst = EUSCI_A_UART_LSB_FIRST;
param.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT;
param.uartMode = EUSCI_A_UART_MODE;
param.overSampling = EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION;

if (STATUS_FAIL == EUSCI_A_UART_init(EUSCI_A0_BASE, &param)) {
    return;
}

EUSCI_A_UART_enable(EUSCI_A0_BASE);

// Enable USCI_A0 RX interrupt
EUSCI_A_UART_enableInterrupt(EUSCI_A0_BASE,
      EUSCI_A_UART_RECEIVE_INTERRUPT);
```

12 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)

12.1 Introduction

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.

This library provides the API for handling a SPI communication using EUSCI.

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 module's input clock.

12.2 Functions

Functions

- void EUSCI_A_SPI_initMaster (uint16_t baseAddress, EUSCI_A_SPI_initMasterParam *param)
  Initializes the SPI Master block.
- void EUSCI_A_SPI_select4PinFunctionality (uint16_t baseAddress, uint16_t select4PinFunctionality)
  Selects 4Pin Functionality.
- void EUSCI_A_SPI_changeMasterClock (uint16_t baseAddress, EUSCI_A_SPI_changeMasterClockParam *param)
  Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.
- void EUSCI_A_SPI_initSlave (uint16_t baseAddress, EUSCI_A_SPI_initSlaveParam *param)
  Initializes the SPI Slave block.
- void EUSCI_A_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
  Changes the SPI clock phase and polarity. At the end of this function call, SPI module is left enabled.
- void EUSCI_A_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)
  Transmits a byte from the SPI Module.
- uint8_t EUSCI_A_SPI_receiveData (uint16_t baseAddress)
  Receives a byte that has been sent to the SPI Module.
- void EUSCI_A_SPI_enableInterrupt (uint16_t baseAddress, uint16_t mask)
  Enables individual SPI interrupt sources.
- void EUSCI_A_SPI_disableInterrupt (uint16_t baseAddress, uint16_t mask)
  Disables individual SPI interrupt sources.

- uint8_t EUSCI_A_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
  Gets the current SPI interrupt status.
- void EUSCI_A_SPI_clearInterrupt (uint16_t baseAddress, uint16_t mask)
  Clears the selected SPI interrupt status flag.
- void EUSCI_A_SPI_enable (uint16_t baseAddress)
  Enables the SPI block.
- void EUSCI_A_SPI_disable (uint16_t baseAddress)
  Disables the SPI block.
- uint32_t EUSCI_A_SPI_getReceiveBufferAddress (uint16_t baseAddress)
  Returns the address of the RX Buffer of the SPI for the DMA module.
- uint32_t EUSCI_A_SPI_getTransmitBufferAddress (uint16_t baseAddress)
  Returns the address of the TX Buffer of the SPI for the DMA module.
- uint16_t EUSCI_A_SPI_isBusy (uint16_t baseAddress)
  Indicates whether or not the SPI bus is busy.
- void EUSCI_A_SPI_remapPins (uint16_t baseAddress, uint8_t pinsSelect)
  Remaps eUSCI_A GPIO pins.

12.2.1 Detailed Description

To use the module as a master, the user must call EUSCI_A_SPI_initMaster() to configure the SPI Master. This is followed by enabling the SPI module using EUSCI_A_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 EUSCI_A_SPI_transmitData() and then when the receive flag is set, the received data is read using EUSCI_A_SPI_receiveData() and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using EUSCI_A_SPI_initSlave() and this is followed by enabling the module using EUSCI_A_SPI_enable(). Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using EUSCI_A_SPI_transmitData() and this is followed by a data reception by EUSCI_A_SPI_receiveData()

The SPI API is broken into 3 groups of functions: those that deal with status and initialization, those that handle data, and those that manage interrupts.

The status and initialization of the SPI module are managed by

- EUSCI_A_SPI_initMaster()
- EUSCI_A_SPI_initSlave()
- EUSCI_A_SPI_disable()
- EUSCI_A_SPI_enable()
- EUSCI_A_SPI_masterChangeClock()
- EUSCI_A_SPI_isBusy()
- EUSCI_A_SPI_select4PinFunctionality()
- EUSCI_A_SPI_changeClockPhasePolarity()

Data handling is done by

- EUSCI_A_SPI_transmitData()
- EUSCI_A_SPI_receiveData()

Interrupts from the SPI module are managed using

- EUSCI_A_SPI_disableInterrupt()
- EUSCI_A_SPI_enableInterrupt()
- EUSCI_A_SPI_getInterruptStatus()
- EUSCI_A_SPI_clearInterrupt()

DMA related

- EUSCI_A_SPI_getReceiveBufferAddressForDMA()
- EUSCI_A_SPI_getTransmitBufferAddressForDMA()

12.2.2 Function Documentation

EUSCI_A_SPI_changeClockPhasePolarity()

```
void EUSCI_A_SPI_changeClockPhasePolarity (
            uint16_t baseAddress,
            uint16_t clockPhase,
            uint16_t clockPolarity )
```

Changes the SPI clock 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 **UCCKPL**, **UCCKPH** and **UCSWRST** of **UCAxCTLW0** register.

Returns

None

EUSCI_A_SPI_changeMasterClock()

```
void EUSCI_A_SPI_changeMasterClock (
```

```
uint16_t baseAddress,
EUSCI_A_SPI_changeMasterClockParam * param )
```

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.
param	is the pointer to struct for master clock setting.

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns

None

References EUSCI_A_SPI_changeMasterClockParam::clockSourceFrequency, and EUSCI_A_SPI_changeMasterClockParam::desiredSpiClock.

EUSCI_A_SPI_clearInterrupt()

```
void EUSCI_A_SPI_clearInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_A_SPI_disable()

```
void EUSCI_A_SPI_disable (
            uint16_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

EUSCI_A_SPI_disableInterrupt()

```
void EUSCI_A_SPI_disableInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_A_SPI_enable()

```
void EUSCI_A_SPI_enable (
            uint16_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

EUSCI_A_SPI_enableInterrupt()

```
void EUSCI_A_SPI_enableInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_A_SPI_getInterruptStatus()

```
uint8_t EUSCI_A_SPI_getInterruptStatus (
            uint16_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

EUSCI_A_SPI_getReceiveBufferAddress()

```
uint32_t EUSCI_A_SPI_getReceiveBufferAddress (
            uint16_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

EUSCI_A_SPI_getTransmitBufferAddress()

```
uint32_t EUSCI_A_SPI_getTransmitBufferAddress (
            uint16_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

EUSCI_A_SPI_initMaster()

```
void EUSCI_A_SPI_initMaster (
            uint16_t baseAddress,
            EUSCI_A_SPI_initMasterParam * param )
```

Initializes the SPI Master block.

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 EUSCI_A_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_A_SPI Master module.
param	is the pointer to struct for master initialization.

Modified bits are **UCCKPH**, **UCCKPL**, **UC7BIT**, **UCMSB**, **UCSSELx** and **UCSWRST** of **UCAxCTLW0** register.

Returns

STATUS_SUCCESS

References EUSCI_A_SPI_initMasterParam::clockPhase, EUSCI_A_SPI_initMasterParam::clockPolarity, EUSCI_A_SPI_initMasterParam::clockSourceFrequency, EUSCI_A_SPI_initMasterParam::desiredSpiClock, EUSCI_A_SPI_initMasterParam::msbFirst, EUSCI_A_SPI_initMasterParam::selectClockSource, and EUSCI_A_SPI_initMasterParam::spiMode.

EUSCI_A_SPI_initSlave()

```
void EUSCI_A_SPI_initSlave (
            uint16_t baseAddress,
            EUSCI_A_SPI_initSlaveParam * param )
```

Initializes the SPI Slave block.

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 EUSCI_A_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_A_SPI Slave module.
param	is the pointer to struct for slave initialization.

Modified bits are **UCMSB**, **UCMST**, **UC7BIT**, **UCCKPL**, **UCCKPH**, **UCMODE** and **UCSWRST** of **UCAxCTLW0** register.

Returns

STATUS_SUCCESS

References EUSCI_A_SPI_initSlaveParam::clockPhase, EUSCI_A_SPI_initSlaveParam::clockPolarity, EUSCI_A_SPI_initSlaveParam::msbFirst, and EUSCI_A_SPI_initSlaveParam::spiMode.

EUSCI_A_SPI_isBusy()

```
uint16_t EUSCI_A_SPI_isBusy (
            uint16_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

One of the following:

- **EUSCI_A_SPI_BUSY**
- **EUSCI_A_SPI_NOT_BUSY**
  indicating if the EUSCI_A_SPI is busy

EUSCI_A_SPI_receiveData()

```
uint8_t EUSCI_A_SPI_receiveData (
            uint16_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.

EUSCI_A_SPI_remapPins()

```
void EUSCI_A_SPI_remapPins (
            uint16_t baseAddress,
            uint8_t pinsSelect )
```

Remaps eUSCI_A GPIO pins.

Remaps eUSCI_A GPIO pins. After calling this function, GPIO_setAsPeripheralModuleFunctionInputPin() or GPIO_setAsPeripheralModuleFunctionInputPin() still needs to be invoked to set peripheral functions. Caution: this will also remap eusci_a_uart GPIO pins.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
pinsSelect	remapping pins to select. Please refer to device specific datasheet for remapping pins details. Valid values are: ■ **EUSCI_A_SPI_REMAP_PINS_FALSE** [Default] ■ **EUSCI_A_SPI_REMAP_PINS_TRUE**

Returns

None

EUSCI_A_SPI_select4PinFunctionality()

```
void EUSCI_A_SPI_select4PinFunctionality (
            uint16_t baseAddress,
            uint16_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_MA↩ STERS** ■ **EUSCI_A_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

Modified bits are **UCSTEM** of **UCAxCTLW0** register.

Returns

None

EUSCI_A_SPI_transmitData()

```
void EUSCI_A_SPI_transmitData (
            uint16_t baseAddress,
            uint8_t transmitData )
```

Transmits a byte from the SPI Module.

This function will place the supplied data into SPI transmit 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

12.3 Programming Example

The following example shows how to use the SPI API to configure the SPI module as a master device, and how to do a simple send of data.

```
//Initialize slave to MSB first, inactive high clock polarity and 3 wire SPI
EUSCI_A_SPI_initSlaveParam param = {0};
param.msbFirst = EUSCI_A_SPI_MSB_FIRST;
param.clockPhase = EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT;
param.clockPolarity = EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH;
param.spiMode = EUSCI_A_SPI_3PIN;
EUSCI_A_SPI_initSlave(EUSCI_A0_BASE, &param);


//Enable SPI Module
EUSCI_A_SPI_enable(EUSCI_A0_BASE);

//Enable Receive interrupt
EUSCI_A_SPI_enableInterrupt(EUSCI_A0_BASE,
        EUSCI_A_SPI_RECEIVE_INTERRUPT
        );
```

13 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)

13.1 Introduction

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.

This library provides the API for handling a SPI communication using EUSCI.

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 module's input clock.

13.2 Functions

Functions

- void EUSCI_B_SPI_initMaster (uint16_t baseAddress, EUSCI_B_SPI_initMasterParam ∗param)
  Initializes the SPI Master block.
- void EUSCI_B_SPI_select4PinFunctionality (uint16_t baseAddress, uint16_t select4PinFunctionality)
  Selects 4Pin Functionality.
- void EUSCI_B_SPI_changeMasterClock (uint16_t baseAddress, EUSCI_B_SPI_changeMasterClockParam ∗param)
  Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.
- void EUSCI_B_SPI_initSlave (uint16_t baseAddress, EUSCI_B_SPI_initSlaveParam ∗param)
  Initializes the SPI Slave block.
- void EUSCI_B_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
  Changes the SPI clock phase and polarity. At the end of this function call, SPI module is left enabled.
- void EUSCI_B_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)
  Transmits a byte from the SPI Module.
- uint8_t EUSCI_B_SPI_receiveData (uint16_t baseAddress)
  Receives a byte that has been sent to the SPI Module.
- void EUSCI_B_SPI_enableInterrupt (uint16_t baseAddress, uint16_t mask)
  Enables individual SPI interrupt sources.
- void EUSCI_B_SPI_disableInterrupt (uint16_t baseAddress, uint16_t mask)
  Disables individual SPI interrupt sources.

- uint8_t EUSCI_B_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
  Gets the current SPI interrupt status.
- void EUSCI_B_SPI_clearInterrupt (uint16_t baseAddress, uint16_t mask)
  Clears the selected SPI interrupt status flag.
- void EUSCI_B_SPI_enable (uint16_t baseAddress)
  Enables the SPI block.
- void EUSCI_B_SPI_disable (uint16_t baseAddress)
  Disables the SPI block.
- uint32_t EUSCI_B_SPI_getReceiveBufferAddress (uint16_t baseAddress)
  Returns the address of the RX Buffer of the SPI for the DMA module.
- uint32_t EUSCI_B_SPI_getTransmitBufferAddress (uint16_t baseAddress)
  Returns the address of the TX Buffer of the SPI for the DMA module.
- uint16_t EUSCI_B_SPI_isBusy (uint16_t baseAddress)
  Indicates whether or not the SPI bus is busy.
- void EUSCI_B_SPI_remapPins (uint16_t baseAddress, uint8_t pinsSelect)
  Remaps eUSCI_B GPIO pins.

13.2.1 Detailed Description

To use the module as a master, the user must call EUSCI_B_SPI_masterInit() to configure the SPI Master. This is followed by enabling the SPI module using EUSCI_B_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 EUSCI_B_SPI_transmitData() and then when the receive flag is set, the received data is read using EUSCI_B_SPI_receiveData() and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using EUSCI_B_SPI_slaveInit() and this is followed by enabling the module using EUSCI_B_SPI_enable(). Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using EUSCI_B_SPI_transmitData() and this is followed by a data reception by EUSCI_B_SPI_receiveData()

The SPI API is broken into 3 groups of functions: those that deal with status and initialization, those that handle data, and those that manage interrupts.

The status and initialization of the SPI module are managed by

- EUSCI_B_SPI_masterInit()
- EUSCI_B_SPI_slaveInit()
- EUSCI_B_SPI_disable()
- EUSCI_B_SPI_enable()
- EUSCI_B_SPI_masterChangeClock()
- EUSCI_B_SPI_isBusy()
- EUSCI_B_SPI_select4PinFunctionality()
- EUSCI_B_SPI_changeClockPhasePolarity()

Data handling is done by

- EUSCI_B_SPI_transmitData()
- EUSCI_B_SPI_receiveData()

Interrupts from the SPI module are managed using

- EUSCI_B_SPI_disableInterrupt()
- EUSCI_B_SPI_enableInterrupt()
- EUSCI_B_SPI_getInterruptStatus()
- EUSCI_B_SPI_clearInterrupt()

DMA related

- EUSCI_B_SPI_getReceiveBufferAddressForDMA()
- EUSCI_B_SPI_getTransmitBufferAddressForDMA()

13.2.2 Function Documentation

EUSCI_B_SPI_changeClockPhasePolarity()

```
void EUSCI_B_SPI_changeClockPhasePolarity (
            uint16_t baseAddress,
            uint16_t clockPhase,
            uint16_t clockPolarity )
```

Changes the SPI clock 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_O↩ N_NEXT** [Default] ■ **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_O↩ N_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 **UCCKPL**, **UCCKPH** and **UCSWRST** of **UCAxCTLW0** register.

Returns

None

EUSCI_B_SPI_changeMasterClock()

```
void EUSCI_B_SPI_changeMasterClock (
```

```
uint16_t baseAddress,
EUSCI_B_SPI_changeMasterClockParam * param )
```

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.
param	is the pointer to struct for master clock setting.

Modified bits are **UCSWRST** of **UCAxCTLW0** register.

Returns

None

References EUSCI_B_SPI_changeMasterClockParam::clockSourceFrequency, and EUSCI_B_SPI_changeMasterClockParam::desiredSpiClock.

EUSCI_B_SPI_clearInterrupt()

```
void EUSCI_B_SPI_clearInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_B_SPI_disable()

```
void EUSCI_B_SPI_disable (
            uint16_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 **UCAxCTLW0** register.

Returns

None

EUSCI_B_SPI_disableInterrupt()

```
void EUSCI_B_SPI_disableInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_B_SPI_enable()

```
void EUSCI_B_SPI_enable (
            uint16_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 **UCAxCTLW0** register.

Returns

None

EUSCI_B_SPI_enableInterrupt()

```
void EUSCI_B_SPI_enableInterrupt (
            uint16_t baseAddress,
            uint16_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

EUSCI_B_SPI_getInterruptStatus()

```
uint8_t EUSCI_B_SPI_getInterruptStatus (
            uint16_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

EUSCI_B_SPI_getReceiveBufferAddress()

```
uint32_t EUSCI_B_SPI_getReceiveBufferAddress (
            uint16_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

EUSCI_B_SPI_getTransmitBufferAddress()

```
uint32_t EUSCI_B_SPI_getTransmitBufferAddress (
            uint16_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

EUSCI_B_SPI_initMaster()

```
void EUSCI_B_SPI_initMaster (
            uint16_t baseAddress,
            EUSCI_B_SPI_initMasterParam * param )
```

Initializes the SPI Master block.

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 EUSCI_B_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_B_SPI Master module.
param	is the pointer to struct for master initialization.

Modified bits are **UCCKPH**, **UCCKPL**, **UC7BIT**, **UCMSB**, **UCSSELx** and **UCSWRST** of **UCAxCTLW0** register.

Returns

STATUS_SUCCESS

References EUSCI_B_SPI_initMasterParam::clockPhase, EUSCI_B_SPI_initMasterParam::clockPolarity, EUSCI_B_SPI_initMasterParam::clockSourceFrequency, EUSCI_B_SPI_initMasterParam::desiredSpiClock, EUSCI_B_SPI_initMasterParam::msbFirst, EUSCI_B_SPI_initMasterParam::selectClockSource, and EUSCI_B_SPI_initMasterParam::spiMode.

EUSCI_B_SPI_initSlave()

```
void EUSCI_B_SPI_initSlave (
            uint16_t baseAddress,
            EUSCI_B_SPI_initSlaveParam * param )
```

Initializes the SPI Slave block.

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 EUSCI_B_SPI_enable()

Parameters

baseAddress	is the base address of the EUSCI_B_SPI Slave module.
param	is the pointer to struct for slave initialization.

Modified bits are **UCMSB**, **UCMST**, **UC7BIT**, **UCCKPL**, **UCCKPH**, **UCMODE** and **UCSWRST** of **UCAxCTLW0** register.

Returns

STATUS_SUCCESS

References EUSCI_B_SPI_initSlaveParam::clockPhase, EUSCI_B_SPI_initSlaveParam::clockPolarity, EUSCI_B_SPI_initSlaveParam::msbFirst, and EUSCI_B_SPI_initSlaveParam::spiMode.

EUSCI_B_SPI_isBusy()

```
uint16_t EUSCI_B_SPI_isBusy (
            uint16_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

One of the following:

- **EUSCI_B_SPI_BUSY**
- **EUSCI_B_SPI_NOT_BUSY**
  indicating if the EUSCI_B_SPI is busy

EUSCI_B_SPI_receiveData()

```
uint8_t EUSCI_B_SPI_receiveData (
            uint16_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.

EUSCI_B_SPI_remapPins()

```
void EUSCI_B_SPI_remapPins (
            uint16_t baseAddress,
            uint8_t pinsSelect )
```

Remaps eUSCI_B GPIO pins.

Remaps eUSCI_B GPIO pins. After calling this function, GPIO_setAsPeripheralModuleFunctionInputPin() or GPIO_setAsPeripheralModuleFunctionInputPin() still needs to be invoked to set peripheral functions. Caution: this will also remap eusci_b_i2c GPIO pins.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
pinsSelect	remapping pins to select. Please refer to device specific datasheet for remapping pins details. Valid values are: ■ **EUSCI_B_SPI_REMAP_PINS_FALSE** [Default] ■ **EUSCI_B_SPI_REMAP_PINS_TRUE**

Returns

None

EUSCI_B_SPI_select4PinFunctionality()

```
void EUSCI_B_SPI_select4PinFunctionality (
            uint16_t baseAddress,
            uint16_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_MA↩ STERS** ■ **EUSCI_B_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

Modified bits are **UCSTEM** of **UCAxCTLW0** register.

Returns

None

EUSCI_B_SPI_transmitData()

```
void EUSCI_B_SPI_transmitData (
            uint16_t baseAddress,
            uint8_t transmitData )
```

Transmits a byte from the SPI Module.

This function will place the supplied data into SPI transmit 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

13.3 Programming Example

The following example shows how to use the SPI API to configure the SPI module as a master device, and how to do a simple send of data.

```
//Initialize slave to MSB first, inactive high clock polarity and 3 wire SPI
EUSCI_B_SPI_initSlaveParam param = {0};
param.msbFirst = EUSCI_B_SPI_MSB_FIRST;
param.clockPhase = EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT;
param.clockPolarity = EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH;
param.spiMode = EUSCI_B_SPI_3PIN;
EUSCI_B_SPI_initSlave(EUSCI_B0_BASE, &param);

//Enable SPI Module
EUSCI_B_SPI_enable(EUSCI_B0_BASE);

//Enable Receive interrupt
EUSCI_B_SPI_enableInterrupt(EUSCI_B0_BASE,
        EUSCI_B_SPI_RECEIVE_INTERRUPT
        );
```

14 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)

14.1 Introduction

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 MSP430Ware 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.

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 MSP430Ware 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.

14.2 Master Operations

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

- **EUSCI_B_I2C_initMaster**
- **EUSCI_B_I2C_setSlaveAddress**
- **EUSCI_B_I2C_setMode**
- **EUSCI_B_I2C_enable**
- **EUSCI_B_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 EUSCI_B_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 EUSCI_B_I2C_setSlaveAddress. Then the mode of operation (transmit or receive) is chosen using EUSCI_B_I2C_setMode. The I2C module may now be enabled using EUSCI_B_I2C_enable. It is recommended to enable the EUSCI_B_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

- EUSCI_B_I2C_masterSendSingleByte()

Master Multiple Byte Transmission

- EUSCI_B_I2C_masterSendMultiByteStart()
- EUSCI_B_I2C_masterSendMultiByteNext()
- EUSCI_B_I2C_masterSendMultiByteStop()

Master Single Byte Reception

- EUSCI_B_I2C_masterReceiveSingleByte()

Master Multiple Byte Reception

- EUSCI_B_I2C_masterMultiByteReceiveStart()
- EUSCI_B_I2C_masterReceiveMultiByteNext()
- EUSCI_B_I2C_masterReceiveMultiByteFinish()
- EUSCI_B_I2C_masterReceiveMultiByteStop()

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

14.3 Slave Operations

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

- **EUSCI_B_I2C_initSlave()**
- **EUSCI_B_I2C_setMode()**
- **EUSCI_B_I2C_enable()**
- **EUSCI_B_I2C_enableInterrupt()** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first call the EUSCI_B_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 EUSCI_B_I2C_enable. 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

- EUSCI_B_I2C_slavePutData()

Slave Reception API

- EUSCI_B_I2C_slaveGetData()

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

14.4 API Functions

Functions

- void EUSCI_B_I2C_initMaster (uint16_t baseAddress, EUSCI_B_I2C_initMasterParam ∗param)
  Initializes the I2C Master block.
- void EUSCI_B_I2C_initSlave (uint16_t baseAddress, EUSCI_B_I2C_initSlaveParam ∗param)
  Initializes the I2C Slave block.
- void EUSCI_B_I2C_enable (uint16_t baseAddress)
  Enables the I2C block.
- void EUSCI_B_I2C_disable (uint16_t baseAddress)
  Disables the I2C block.
- void EUSCI_B_I2C_setSlaveAddress (uint16_t baseAddress, uint8_t slaveAddress)
  Sets the address that the I2C Master will place on the bus.
- void EUSCI_B_I2C_setMode (uint16_t baseAddress, uint16_t mode)
  Sets the mode of the I2C device.
- uint8_t EUSCI_B_I2C_getMode (uint16_t baseAddress)
  Gets the mode of the I2C device.
- void EUSCI_B_I2C_slavePutData (uint16_t baseAddress, uint8_t transmitData)
  Transmits a byte from the I2C Module.
- uint8_t EUSCI_B_I2C_slaveGetData (uint16_t baseAddress)
  Receives a byte that has been sent to the I2C Module.
- uint16_t EUSCI_B_I2C_isBusBusy (uint16_t baseAddress)
  Indicates whether or not the I2C bus is busy.
- uint16_t EUSCI_B_I2C_masterIsStopSent (uint16_t baseAddress)
  Indicates whether STOP got sent.
- uint16_t EUSCI_B_I2C_masterIsStartSent (uint16_t baseAddress)
  Indicates whether Start got sent.
- void EUSCI_B_I2C_enableInterrupt (uint16_t baseAddress, uint16_t mask)
  Enables individual I2C interrupt sources.
- void EUSCI_B_I2C_disableInterrupt (uint16_t baseAddress, uint16_t mask)
  Disables individual I2C interrupt sources.
- void EUSCI_B_I2C_clearInterrupt (uint16_t baseAddress, uint16_t mask)
  Clears I2C interrupt sources.
- uint16_t EUSCI_B_I2C_getInterruptStatus (uint16_t baseAddress, uint16_t mask)
  Gets the current I2C interrupt status.
- void EUSCI_B_I2C_masterSendSingleByte (uint16_t baseAddress, uint8_t txData)
  Does single byte transmission from Master to Slave.
- uint8_t EUSCI_B_I2C_masterReceiveSingleByte (uint16_t baseAddress)
  Does single byte reception from Slave.
- bool EUSCI_B_I2C_masterSendSingleByteWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
  Does single byte transmission from Master to Slave with timeout.
- void EUSCI_B_I2C_masterSendMultiByteStart (uint16_t baseAddress, uint8_t txData)

Starts multi-byte transmission from Master to Slave.
- bool EUSCI_B_I2C_masterSendMultiByteStartWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
  Starts multi-byte transmission from Master to Slave with timeout.
- void EUSCI_B_I2C_masterSendMultiByteNext (uint16_t baseAddress, uint8_t txData)
  Continues multi-byte transmission from Master to Slave.
- bool EUSCI_B_I2C_masterSendMultiByteNextWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
  Continues multi-byte transmission from Master to Slave with timeout.
- void EUSCI_B_I2C_masterSendMultiByteFinish (uint16_t baseAddress, uint8_t txData)
  Finishes multi-byte transmission from Master to Slave.
- bool EUSCI_B_I2C_masterSendMultiByteFinishWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
  Finishes multi-byte transmission from Master to Slave with timeout.
- void EUSCI_B_I2C_masterSendStart (uint16_t baseAddress)
  This function is used by the Master module to initiate START.
- void EUSCI_B_I2C_masterSendMultiByteStop (uint16_t baseAddress)
  Send STOP byte at the end of a multi-byte transmission from Master to Slave.
- bool EUSCI_B_I2C_masterSendMultiByteStopWithTimeout (uint16_t baseAddress, uint32_t timeout)
  Send STOP byte at the end of a multi-byte transmission from Master to Slave with timeout.
- void EUSCI_B_I2C_masterReceiveStart (uint16_t baseAddress)
  Starts reception at the Master end.
- uint8_t EUSCI_B_I2C_masterReceiveMultiByteNext (uint16_t baseAddress)
  Starts multi-byte reception at the Master end one byte at a time.
- uint8_t EUSCI_B_I2C_masterReceiveMultiByteFinish (uint16_t baseAddress)
  Finishes multi-byte reception at the Master end.
- bool EUSCI_B_I2C_masterReceiveMultiByteFinishWithTimeout (uint16_t baseAddress, uint8_t *txData, uint32_t timeout)
  Finishes multi-byte reception at the Master end with timeout.
- void EUSCI_B_I2C_masterReceiveMultiByteStop (uint16_t baseAddress)
  Sends the STOP at the end of a multi-byte reception at the Master end.
- void EUSCI_B_I2C_enableMultiMasterMode (uint16_t baseAddress)
  Enables Multi Master Mode.
- void EUSCI_B_I2C_disableMultiMasterMode (uint16_t baseAddress)
  Disables Multi Master Mode.
- uint8_t EUSCI_B_I2C_masterReceiveSingle (uint16_t baseAddress)
  receives a byte that has been sent to the I2C Master Module.
- uint32_t EUSCI_B_I2C_getReceiveBufferAddress (uint16_t baseAddress)
  Returns the address of the RX Buffer of the I2C for the DMA module.
- uint32_t EUSCI_B_I2C_getTransmitBufferAddress (uint16_t baseAddress)
  Returns the address of the TX Buffer of the I2C for the DMA module.
- void EUSCI_B_I2C_remapPins (uint16_t baseAddress, uint8_t pinsSelect)
  Remaps eUSCI_B GPIO pins.
- void EUSCI_B_I2C_setTimeout (uint16_t baseAddress, uint16_t timeout)
  Enforces a timeout if the I2C clock is held low longer than a defined time.

14.4.1 Detailed Description

The eUSCI I2C API is broken into three groups of functions: those that deal with interrupts, those that handle status and initialization, and those that deal with sending and receiving data.

The I2C master and slave interrupts are handled by

- EUSCI_B_I2C_enableInterrupt
- EUSCI_B_I2C_disableInterrupt
- EUSCI_B_I2C_clearInterrupt
- EUSCI_B_I2C_getInterruptStatus

Status and initialization functions for the I2C modules are

- EUSCI_B_I2C_initMaster
- EUSCI_B_I2C_enable
- EUSCI_B_I2C_disable
- EUSCI_B_I2C_isBusBusy
- EUSCI_B_I2C_isBusy
- EUSCI_B_I2C_initSlave
- EUSCI_B_I2C_interruptStatus
- EUSCI_B_I2C_setSlaveAddress
- EUSCI_B_I2C_setMode
- EUSCI_B_I2C_masterIsStopSent
- EUSCI_B_I2C_masterIsStartSent
- EUSCI_B_I2C_selectMasterEnvironmentSelect

Sending and receiving data from the I2C slave module is handled by

- EUSCI_B_I2C_slavePutData
- EUSCI_B_I2C_slaveGetData

Sending and receiving data from the I2C slave module is handled by

- EUSCI_B_I2C_masterSendSingleByte
- EUSCI_B_I2C_masterSendStart
- EUSCI_B_I2C_masterSendMultiByteStart
- EUSCI_B_I2C_masterSendMultiByteNext
- EUSCI_B_I2C_masterSendMultiByteFinish
- EUSCI_B_I2C_masterSendMultiByteStop
- EUSCI_B_I2C_masterReceiveMultiByteNext
- EUSCI_B_I2C_masterReceiveMultiByteFinish
- EUSCI_B_I2C_masterReceiveMultiByteStop
- EUSCI_B_I2C_masterReceiveStart
- EUSCI_B_I2C_masterReceiveSingle

14.4.2 Function Documentation

EUSCI_B_I2C_clearInterrupt()

```
void EUSCI_B_I2C_clearInterrupt (
```

```
uint16_t baseAddress,
uint16_t mask )
```

Clears I2C interrupt sources.

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.

Parameters

baseAddress	is the base address of the I2C module.
mask	is a bit mask of the interrupt sources to be cleared. 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

Modified bits of **UCBxIFG** register.

Returns

None

EUSCI_B_I2C_disable()

```
void EUSCI_B_I2C_disable (
            uint16_t baseAddress )
```

Disables the I2C block.

This will disable operation of the I2C block.

Parameters

baseAddress	is the base address of the USCI I2C module.

Modified bits are **UCSWRST** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_disableInterrupt()

```
void EUSCI_B_I2C_disableInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Disables individual I2C interrupt sources.

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.

Parameters

baseAddress	is the base address of the I2C 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_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

Modified bits of **UCBxIE** register.

Returns

None

EUSCI_B_I2C_disableMultiMasterMode()

```
void EUSCI_B_I2C_disableMultiMasterMode (
            uint16_t baseAddress )
```

Disables Multi Master Mode.

At the end of this function, the I2C module is still disabled till EUSCI_B_I2C_enable is invoked

Parameters

baseAddress	is the base address of the I2C module.

Modified bits are **UCSWRST** and **UCMM** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_enable()

```
void EUSCI_B_I2C_enable (
            uint16_t baseAddress )
```

Enables the I2C block.

This will enable operation of the I2C block.

Parameters

baseAddress	is the base address of the USCI I2C module.

Modified bits are **UCSWRST** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_enableInterrupt()

```
void EUSCI_B_I2C_enableInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Enables individual I2C interrupt sources.

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.

Parameters

baseAddress	is the base address of the I2C 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_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

Modified bits of **UCBxIE** register.

Returns

None

EUSCI_B_I2C_enableMultiMasterMode()

```
void EUSCI_B_I2C_enableMultiMasterMode (
            uint16_t baseAddress )
```

Enables Multi Master Mode.

At the end of this function, the I2C module is still disabled till EUSCI_B_I2C_enable is invoked

Parameters

baseAddress	is the base address of the I2C module.

Modified bits are **UCSWRST** and **UCMM** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_getInterruptStatus()

```
uint16_t EUSCI_B_I2C_getInterruptStatus (
            uint16_t baseAddress,
            uint16_t mask )
```

Gets the current I2C interrupt status.

This returns the interrupt status for the I2C module based on which flag is passed.

Parameters

baseAddress	is the base address of the I2C module.
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

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
  indicating the status of the masked interrupts

EUSCI_B_I2C_getMode()

```
uint8_t EUSCI_B_I2C_getMode (
            uint16_t baseAddress )
```

Gets the mode of the I2C device.

Current I2C transmit/receive mode.

Parameters

baseAddress	is the base address of the I2C module.

Modified bits are **UCTR** of **UCBxCTLW0** register.

Returns

One of the following:

- **EUSCI_B_I2C_TRANSMIT_MODE**
- **EUSCI_B_I2C_RECEIVE_MODE**
  indicating the current mode

EUSCI_B_I2C_getReceiveBufferAddress()

```
uint32_t EUSCI_B_I2C_getReceiveBufferAddress (
            uint16_t baseAddress )
```

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

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.

Parameters

baseAddress	is the base address of the I2C module.

Returns

The address of the I2C RX Buffer

EUSCI_B_I2C_getTransmitBufferAddress()

```
uint32_t EUSCI_B_I2C_getTransmitBufferAddress (
            uint16_t baseAddress )
```

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

Returns the address of the I2C 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 I2C module.

Returns

The address of the I2C TX Buffer

EUSCI_B_I2C_initMaster()

```
void EUSCI_B_I2C_initMaster (
            uint16_t baseAddress,
            EUSCI_B_I2C_initMasterParam * param )
```

Initializes the I2C Master block.

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 EUSCI_B_I2C_enable is invoked.

Parameters

baseAddress	is the base address of the I2C Master module.
param	is the pointer to the struct for master initialization.

Returns

None

References EUSCI_B_I2C_initMasterParam::autoSTOPGeneration, EUSCI_B_I2C_initMasterParam::byteCounterThreshold, EUSCI_B_I2C_initMasterParam::dataRate, EUSCI_B_I2C_initMasterParam::i2cClk, and EUSCI_B_I2C_initMasterParam::selectClockSource.

EUSCI_B_I2C_initSlave()

```
void EUSCI_B_I2C_initSlave (
            uint16_t baseAddress,
            EUSCI_B_I2C_initSlaveParam * param )
```

Initializes the I2C Slave block.

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 EUSCI_B_I2C_enable is invoked.

Parameters

baseAddress	is the base address of the I2C Slave module.
param	is the pointer to the struct for slave initialization.

Returns

None

References EUSCI_B_I2C_initSlaveParam::slaveAddress, EUSCI_B_I2C_initSlaveParam::slaveAddressOffset, and EUSCI_B_I2C_initSlaveParam::slaveOwnAddressEnable.

EUSCI_B_I2C_isBusBusy()

```
uint16_t EUSCI_B_I2C_isBusBusy (
            uint16_t baseAddress )
```

Indicates whether or not the I2C bus is busy.

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.

Parameters

baseAddress	is the base address of the I2C module.

Returns

One of the following:

- **EUSCI_B_I2C_BUS_BUSY**

- **EUSCI_B_I2C_BUS_NOT_BUSY**
  indicating whether the bus is busy

EUSCI_B_I2C_masterIsStartSent()

```
uint16_t EUSCI_B_I2C_masterIsStartSent (
            uint16_t baseAddress )
```

Indicates whether Start got sent.

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.

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

One of the following:

- **EUSCI_B_I2C_START_SEND_COMPLETE**
- **EUSCI_B_I2C_SENDING_START**
  indicating whether the start was sent

EUSCI_B_I2C_masterIsStopSent()

```
uint16_t EUSCI_B_I2C_masterIsStopSent (
            uint16_t baseAddress )
```

Indicates whether STOP got sent.

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.

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

One of the following:

- **EUSCI_B_I2C_STOP_SEND_COMPLETE**
- **EUSCI_B_I2C_SENDING_STOP**
  indicating whether the stop was sent

EUSCI_B_I2C_masterReceiveMultiByteFinish()

```
uint8_t EUSCI_B_I2C_masterReceiveMultiByteFinish (
```

```
                uint16_t baseAddress )
```

Finishes multi-byte reception at the Master end.

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.

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits are **UCTXSTP** of **UCBxCTLW0** register.

Returns

Received byte at Master end.

EUSCI_B_I2C_masterReceiveMultiByteFinishWithTimeout()

```
bool EUSCI_B_I2C_masterReceiveMultiByteFinishWithTimeout (
                uint16_t baseAddress,
                uint8_t * txData,
                uint32_t timeout )
```

Finishes multi-byte reception at the Master end with timeout.

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.

Parameters

baseAddress	is the base address of the I2C Master module.
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

Modified bits are **UCTXSTP** of **UCBxCTLW0** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the reception process

EUSCI_B_I2C_masterReceiveMultiByteNext()

```
uint8_t EUSCI_B_I2C_masterReceiveMultiByteNext (
                uint16_t baseAddress )
```

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

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

Received byte at Master end.

EUSCI_B_I2C_masterReceiveMultiByteStop()

```
void EUSCI_B_I2C_masterReceiveMultiByteStop (
            uint16_t baseAddress )
```

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

This function is used by the Master module to initiate STOP

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits are **UCTXSTP** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_masterReceiveSingle()

```
uint8_t EUSCI_B_I2C_masterReceiveSingle (
            uint16_t baseAddress )
```

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

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

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

EUSCI_B_I2C_masterReceiveSingleByte()

```
uint8_t EUSCI_B_I2C_masterReceiveSingleByte (
            uint16_t baseAddress )
```

Does single byte reception from Slave.

This function is used by the Master module to receive a single byte. This function sends start and stop, waits for data reception and then receives the data from the slave

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits of **UCBxTXBUF** register, bits of **UCBxCTLW0** register, bits of **UCBxIE** register and bits of **UCBxIFG** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterReceiveStart()

```
void EUSCI_B_I2C_masterReceiveStart (
            uint16_t baseAddress )
```

Starts reception at the Master end.

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

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits are **UCTXSTT** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_masterSendMultiByteFinish()

```
void EUSCI_B_I2C_masterSendMultiByteFinish (
            uint16_t baseAddress,
            uint8_t txData )
```

Finishes multi-byte transmission from Master to Slave.

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 and then sends a stop.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the last data byte to be transmitted in a multi-byte transmission

Modified bits of **UCBxTXBUF** register and bits of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_masterSendMultiByteFinishWithTimeout()

```
bool EUSCI_B_I2C_masterSendMultiByteFinishWithTimeout (
            uint16_t baseAddress,
            uint8_t txData,
            uint32_t timeout )
```

Finishes multi-byte transmission from Master to Slave with timeout.

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 and then sends a stop.

Parameters

baseAddress	is the base address of the I2C Master module.
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

Modified bits of **UCBxTXBUF** register and bits of **UCBxCTLW0** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterSendMultiByteNext()

```
void EUSCI_B_I2C_masterSendMultiByteNext (
            uint16_t baseAddress,
            uint8_t txData )
```

Continues multi-byte transmission from Master to Slave.

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.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the next data byte to be transmitted

Modified bits of **UCBxTXBUF** register.

Returns

None

EUSCI_B_I2C_masterSendMultiByteNextWithTimeout()

```
bool EUSCI_B_I2C_masterSendMultiByteNextWithTimeout (
            uint16_t baseAddress,
            uint8_t txData,
            uint32_t timeout )
```

Continues multi-byte transmission from Master to Slave with timeout.

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.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the next data byte to be transmitted
timeout	is the amount of time to wait until giving up

Modified bits of **UCBxTXBUF** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterSendMultiByteStart()

```
void EUSCI_B_I2C_masterSendMultiByteStart (
            uint16_t baseAddress,
            uint8_t txData )
```

Starts multi-byte transmission from Master to Slave.

This function is used by the master module to start a multi byte transaction.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the first data byte to be transmitted

Modified bits of **UCBxTXBUF** register, bits of **UCBxCTLW0** register, bits of **UCBxIE** register and bits of **UCBxIFG** register.

Returns

None

EUSCI_B_I2C_masterSendMultiByteStartWithTimeout()

```
bool EUSCI_B_I2C_masterSendMultiByteStartWithTimeout (
            uint16_t baseAddress,
            uint8_t txData,
```

```
uint32_t timeout )
```

Starts multi-byte transmission from Master to Slave with timeout.

This function is used by the master module to start a multi byte transaction.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the first data byte to be transmitted
timeout	is the amount of time to wait until giving up

Modified bits of **UCBxTXBUF** register, bits of **UCBxCTLW0** register, bits of **UCBxIE** register and bits of **UCBxIFG** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterSendMultiByteStop()

```
void EUSCI_B_I2C_masterSendMultiByteStop (
            uint16_t baseAddress )
```

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

This function is used by the Master module send STOP at the end of a multi- byte transmission. This function sends a stop after current transmission is complete.

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits are **UCTXSTP** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_masterSendMultiByteStopWithTimeout()

```
bool EUSCI_B_I2C_masterSendMultiByteStopWithTimeout (
            uint16_t baseAddress,
            uint32_t timeout )
```

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

This function is used by the Master module send STOP at the end of a multi- byte transmission. This function sends a stop after current transmission is complete.

Parameters

baseAddress	is the base address of the I2C Master module.
timeout	is the amount of time to wait until giving up

Modified bits are **UCTXSTP** of **UCBxCTLW0** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterSendSingleByte()

```
void EUSCI_B_I2C_masterSendSingleByte (
            uint16_t baseAddress,
            uint8_t txData )
```

Does single byte transmission from Master to Slave.

This function is used by the Master module to send a single byte. This function sends a start, then transmits the byte to the slave and then sends a stop.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the data byte to be transmitted

Modified bits of **UCBxTXBUF** register, bits of **UCBxCTLW0** register, bits of **UCBxIE** register and bits of **UCBxIFG** register.

Returns

None

EUSCI_B_I2C_masterSendSingleByteWithTimeout()

```
bool EUSCI_B_I2C_masterSendSingleByteWithTimeout (
            uint16_t baseAddress,
            uint8_t txData,
            uint32_t timeout )
```

Does single byte transmission from Master to Slave with timeout.

This function is used by the Master module to send a single byte. This function sends a start, then transmits the byte to the slave and then sends a stop.

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the data byte to be transmitted
timeout	is the amount of time to wait until giving up

Modified bits of **UCBxTXBUF** register, bits of **UCBxCTLW0** register, bits of **UCBxIE** register and bits of **UCBxIFG** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

EUSCI_B_I2C_masterSendStart()

```
void EUSCI_B_I2C_masterSendStart (
            uint16_t baseAddress )
```

This function is used by the Master module to initiate START.

This function is used by the Master module to initiate START

Parameters

baseAddress	is the base address of the I2C Master module.

Modified bits are **UCTXSTT** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_remapPins()

```
void EUSCI_B_I2C_remapPins (
            uint16_t baseAddress,
            uint8_t pinsSelect )
```

Remaps eUSCI_B GPIO pins.

Remaps eUSCI_B GPIO pins. After calling this function, GPIO_setAsPeripheralModuleFunctionInputPin() or GPIO_setAsPeripheralModuleFunctionInputPin() still needs to be invoked to set peripheral functions. Caution: this will also remap eusci_b_spi GPIO pins.

Parameters

baseAddress	is the base address of the I2C module.
pinsSelect	remapping pins to select. Please refer to device specific datasheet for remapping pins details. Valid values are: ■ **EUSCI_B_I2C_REMAP_PINS_FALSE** [Default] ■ **EUSCI_B_I2C_REMAP_PINS_TRUE**

Returns

None

EUSCI_B_I2C_setMode()

```
void EUSCI_B_I2C_setMode (
            uint16_t baseAddress,
            uint16_t mode )
```

Sets the mode of the I2C device.

When the mode parameter is set to EUSCI_B_I2C_TRANSMIT_MODE, the address will indicate that the I2C module is in send mode; otherwise, the I2C module is in receive mode.

Parameters

baseAddress	is the base address of the USCI I2C module.
mode	Mode for the EUSCI_B_I2C module Valid values are: ■ **EUSCI_B_I2C_TRANSMIT_MODE** [Default] ■ **EUSCI_B_I2C_RECEIVE_MODE**

Modified bits are **UCTR** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_setSlaveAddress()

```
void EUSCI_B_I2C_setSlaveAddress (
            uint16_t baseAddress,
            uint8_t slaveAddress )
```

Sets the address that the I2C Master will place on the bus.

This function will set the address that the I2C Master will place on the bus when initiating a transaction.

Parameters

baseAddress	is the base address of the USCI I2C module.
slaveAddress	7-bit slave address

Modified bits of **UCBxI2CSA** register.

Returns

None

EUSCI_B_I2C_setTimeout()

```
void EUSCI_B_I2C_setTimeout (
            uint16_t baseAddress,
            uint16_t timeout )
```

Enforces a timeout if the I2C clock is held low longer than a defined time.

By using this function, the UCCLTOIFG interrupt will trigger if the clock is held low longer than this defined time. It is possible to detect the situation, when a clock is stretched by a master or slave for too long. The user can then handle this issue by, for example, resetting the eUSCI_B module. It is possible to select one of three predefined times for the clock low timeout.

Parameters

baseAddress	is the base address of the I2C module.
timeout	how long the clock can be low before a timeout triggers. Enables generation of the UCCLTOIFG interrupt. Valid values are: ■ **EUSCI_B_I2C_TIMEOUT_DISABLE** [Default] ■ **EUSCI_B_I2C_TIMEOUT_28_MS** ■ **EUSCI_B_I2C_TIMEOUT_31_MS** ■ **EUSCI_B_I2C_TIMEOUT_34_MS**

Modified bits are **UCCLTO** of **UCBxCTLW1** register; bits **UCSWRST** of **UCBxCTLW0** register.

Returns

None

EUSCI_B_I2C_slaveGetData()

```
uint8_t EUSCI_B_I2C_slaveGetData (
            uint16_t baseAddress )
```

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

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

Parameters

baseAddress	is the base address of the I2C Slave module.

Returns

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

EUSCI_B_I2C_slavePutData()

```
void EUSCI_B_I2C_slavePutData (
```

```
uint16_t baseAddress,
uint8_t transmitData )
```

Transmits a byte from the I2C Module.

This function will place the supplied data into I2C transmit data register to start transmission.

Parameters

baseAddress	is the base address of the I2C Slave module.
transmitData	data to be transmitted from the I2C module

Modified bits of **UCBxTXBUF** register.

Returns

None

14.5 Programming Example

The following example shows how to use the I2C API to send data as a master.

```
//Initialize Slave
 EUSCI_B_I2C_initSlaveParam param = {0};
 param.slaveAddress = 0x48;
 param.slaveAddressOffset = EUSCI_B_I2C_OWN_ADDRESS_OFFSET0;
 param.slaveOwnAddressEnable = EUSCI_B_I2C_OWN_ADDRESS_ENABLE;
 EUSCI_B_I2C_initSlave(EUSCI_B0_BASE, &param);

 EUSCI_B_I2C_enable(EUSCI_B0_BASE);

 EUSCI_B_I2C_enableInterrupt(EUSCI_B0_BASE,
         EUSCI_B_I2C_TRANSMIT_INTERRUPT0 +
         EUSCI_B_I2C_STOP_INTERRUPT);
```

15 FRAMCtl - FRAM Controller

15.1 Introduction

FRAM memory is a non-volatile memory that reads and writes like standard SRAM. The MSP430 FRAM memory features include:

- Byte or word write access
- Automatic and programmable wait state control with independent wait state settings for access and cycle times
- Error Correction Code with bit error correction, extended bit error detection and flag indicators
- Cache for fast read
- Power control for disabling FRAM on non-usage

15.2 API Functions

Functions

- void FRAMCtl_write8 (uint8_t *dataPtr, uint8_t *framPtr, uint16_t numberOfBytes)
  Write data into the fram memory in byte format.
- void FRAMCtl_write16 (uint16_t *dataPtr, uint16_t *framPtr, uint16_t numberOfWords)
  Write data into the fram memory in word format.
- void FRAMCtl_write32 (uint32_t *dataPtr, uint32_t *framPtr, uint16_t count)
  Write data into the fram memory in long format, pass by reference.
- void FRAMCtl_fillMemory32 (uint32_t value, uint32_t *framPtr, uint16_t count)
  Write data into the fram memory in long format, pass by value.
- void FRAMCtl_enableInterrupt (uint16_t interruptMask)
  Enables selected FRAMCtl interrupt sources.
- uint8_t FRAMCtl_getInterruptStatus (uint16_t interruptFlagMask)
  Returns the status of the selected FRAMCtl interrupt flags.
- void FRAMCtl_disableInterrupt (uint16_t interruptMask)
  Disables selected FRAMCtl interrupt sources.
- void FRAMCtl_configureWaitStateControl (uint8_t waitState)
  Configures the access time of the FRAMCtl module.
- void FRAMCtl_delayPowerUpFromLPM (uint8_t delayStatus)
  Configures when the FRAMCtl module will power up after LPM exit.

15.2.1 Detailed Description

FRAMCtl_enableInterrupt enables selected FRAM interrupt sources.

FRAMCtl_getInterruptStatus returns the status of the selected FRAM interrupt flags.

FRAMCtl_disableInterrupt disables selected FRAM interrupt sources.

Depending on the kind of writes being performed to the FRAM, this library provides APIs for FRAM writes.

FRAMCtl_write8 facilitates writing into the FRAM memory in byte format. FRAMCtl_write16 facilitates writing into the FRAM memory in word format. FRAMCtl_write32 facilitates writing into the FRAM memory in long format, pass by reference. FRAMCtl_fillMemory32 facilitates writing into the FRAM memory in long format, pass by value.

Please note the FRAM writing behavior is different in the family MSP430FR2xx_4xx since it needs to clear FRAM write protection bits before writing. The Driverlib FRAM functions already take care of this protection for users. It is the user's responsiblity to clear protection bits if they don't use Driverlib functions.

The FRAM API is broken into 3 groups of functions: those that write into FRAM, those that handle interrupts, and those that configure the wait state and power-up delay after LPM.

FRAM writes are managed by

- FRAMCtl_write8()
- FRAMCtl_write16()
- FRAMCtl_write32()
- FRAMCtl_fillMemory32()

The FRAM interrupts are handled by

- FRAMCtl_enableInterrupt()
- FRAMCtl_getInterruptStatus()
- FRAMCtl_disableInterrupt()

The FRAM wait state and power-up delay after LPM are handled by

- FRAMCtl_configureWaitStateControl()
- FRAMCtl_delayPowerUpFromLPM()

15.2.2 Function Documentation

FRAMCtl_configureWaitStateControl()

```
void FRAMCtl_configureWaitStateControl (
            uint8_t waitState )
```

Configures the access time of the FRAMCtl module.

Configures the access time of the FRAMCtl module.

Parameters

waitState	defines the number of CPU cycles required for access time defined in the datasheet Valid values are: ■ **FRAMCTL_ACCESS_TIME_CYCLES_0** ■ **FRAMCTL_ACCESS_TIME_CYCLES_1** ■ **FRAMCTL_ACCESS_TIME_CYCLES_2** ■ **FRAMCTL_ACCESS_TIME_CYCLES_3** ■ **FRAMCTL_ACCESS_TIME_CYCLES_4** ■ **FRAMCTL_ACCESS_TIME_CYCLES_5** ■ **FRAMCTL_ACCESS_TIME_CYCLES_6** ■ **FRAMCTL_ACCESS_TIME_CYCLES_7**

Modified bits are **NWAITS** of **GCCTL0** register.

Returns

None

FRAMCtl_delayPowerUpFromLPM()

```
void FRAMCtl_delayPowerUpFromLPM (
            uint8_t delayStatus )
```

Configures when the FRAMCtl module will power up after LPM exit.

Configures when the FRAMCtl module will power up after LPM exit. The module can either wait until the first FRAMCtl access to power up or power up immediately after leaving LPM. If FRAMCtl power is disabled, a memory access will automatically insert wait states to ensure sufficient timing for the FRAMCtl power-up and access.

Parameters

delayStatus	chooses if FRAMCTL should power up instantly with LPM exit or to wait until first FRAMCTL access after LPM exit Valid values are: ■ **FRAMCTL_DELAY_FROM_LPM_ENABLE** ■ **FRAMCTL_DELAY_FROM_LPM_DISABLE**

Returns

None

FRAMCtl_disableInterrupt()

```
void FRAMCtl_disableInterrupt (
```

```
uint16_t interruptMask )
```

Disables selected FRAMCtl interrupt sources.

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

Parameters

interruptMask	is the bit mask of the memory buffer interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **FRAMCTL_PUC_ON_UNCORRECTABLE_BIT** - Enable PUC reset if FRAMCtl uncorrectable bit error detected. ■ **FRAMCTL_UNCORRECTABLE_BIT_INTERRUPT** - Interrupts when an uncorrectable bit error is detected. ■ **FRAMCTL_CORRECTABLE_BIT_INTERRUPT** - Interrupts when a correctable bit error is detected. ■ **FRAMCTL_ACCESS_TIME_ERROR_INTERRUPT** - Interrupts when an access time error occurs.

Returns

None

FRAMCtl_enableInterrupt()

```
void FRAMCtl_enableInterrupt (
            uint16_t interruptMask )
```

Enables selected FRAMCtl interrupt sources.

Enables the indicated FRAMCtl 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

interruptMask	is the bit mask of the memory buffer interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **FRAMCTL_PUC_ON_UNCORRECTABLE_BIT** - Enable PUC reset if FRAMCtl uncorrectable bit error detected. ■ **FRAMCTL_UNCORRECTABLE_BIT_INTERRUPT** - Interrupts when an uncorrectable bit error is detected. ■ **FRAMCTL_CORRECTABLE_BIT_INTERRUPT** - Interrupts when a correctable bit error is detected. ■ **FRAMCTL_ACCESS_TIME_ERROR_INTERRUPT** - Interrupts when an access time error occurs.

Modified bits of **GCCTL0** register and bits of **FRCTL0** register.

Returns

None

FRAMCtl_fillMemory32()

```
void FRAMCtl_fillMemory32 (
            uint32_t value,
            uint32_t * framPtr,
            uint16_t count )
```

Write data into the fram memory in long format, pass by value.

Parameters

value	is the value to written to FRAMCTL memory
framPtr	is the pointer into which to write the data
count	is the number of 32 bit addresses to fill

Returns

None

FRAMCtl_getInterruptStatus()

```
uint8_t FRAMCtl_getInterruptStatus (
            uint16_t interruptFlagMask )
```

Returns the status of the selected FRAMCtl interrupt flags.

Parameters

interruptFlagMask	is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following: ■ **FRAMCTL_ACCESS_TIME_ERROR_FLAG** - Interrupt flag is set if a wrong setting for NPRECHG and NACCESS is set and FRAMCtl access time is not hold. ■ **FRAMCTL_UNCORRECTABLE_BIT_FLAG** - Interrupt flag is set if an uncorrectable bit error has been detected in the FRAMCtl memory error detection logic. ■ **FRAMCTL_CORRECTABLE_BIT_FLAG** - Interrupt flag is set if a correctable bit error has been detected and corrected in the FRAMCtl memory error detection logic.

Returns

Logical OR of any of the following:
- **FRAMCTL_ACCESS_TIME_ERROR_FLAG** Interrupt flag is set if a wrong setting for NPRECHG and NACCESS is set and FRAMCtl access time is not hold.
- **FRAMCTL_UNCORRECTABLE_BIT_FLAG** Interrupt flag is set if an uncorrectable bit error has been detected in the FRAMCtl memory error detection logic.
- **FRAMCTL_CORRECTABLE_BIT_FLAG** Interrupt flag is set if a correctable bit error has been detected and corrected in the FRAMCtl memory error detection logic. indicating the status of the masked flags

FRAMCtl_write16()

```
void FRAMCtl_write16 (
            uint16_t * dataPtr,
            uint16_t * framPtr,
            uint16_t numberOfWords )
```

Write data into the fram memory in word format.

Parameters

dataPtr	is the pointer to the data to be written
framPtr	is the pointer into which to write the data
numberOfWords	is the number of words to be written

Returns

None

FRAMCtl_write32()

```
void FRAMCtl_write32 (
            uint32_t * dataPtr,
            uint32_t * framPtr,
            uint16_t count )
```

Write data into the fram memory in long format, pass by reference.

Parameters

dataPtr	is the pointer to the data to be written
framPtr	is the pointer into which to write the data
count	is the number of 32 bit words to be written

Returns

None

FRAMCtl_write8()

```
void FRAMCtl_write8 (
            uint8_t * dataPtr,
            uint8_t * framPtr,
            uint16_t numberOfBytes )
```

Write data into the fram memory in byte format.

Parameters

dataPtr	is the pointer to the data to be written
framPtr	is the pointer into which to write the data
numberOfBytes	is the number of bytes to be written

Returns

None

15.3 Programming Example

The following example shows some FRAM operations using the APIs

```
//Writes the value of "data", 128 times to FRAM
FRAMCtl_fillMemory32(FRAM_BASE,data,
                     (unsigned long *)FRAMCTL_TEST_START,128);
```

16 GPIO

16.1 Introduction

The Digital I/O (GPIO) API provides a set of functions for using the MSP430Ware 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.

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.

Ports P1 and P2 always have interrupt capability. Each interrupt for the P1 and P2 I/O lines can be individually enabled and configured to provide an interrupt on a rising or falling edge of an input signal. All P1 I/O lines source a single interrupt vector P1IV, and all P2 I/O lines source a different, single interrupt vector P2IV. On some devices, additional ports with interrupt capability may be available (see the device-specific data sheet for details) and contain their own respective interrupt vectors. 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 except for the interrupt vector registers, P1IV and P2IV; that is, PAIV does not exist. 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().

16.2 API Functions

Functions

- void GPIO_setAsOutputPin (uint8_t selectedPort, uint16_t selectedPins)
  This function configures the selected Pin as output pin.
- void GPIO_setAsInputPin (uint8_t selectedPort, uint16_t selectedPins)
  This function configures the selected Pin as input pin.
- void GPIO_setAsPeripheralModuleFunctionOutputPin (uint8_t selectedPort, uint16_t selectedPins, uint8_t mode)
  This function configures the peripheral module function in the output direction for the selected pin.
- void GPIO_setAsPeripheralModuleFunctionInputPin (uint8_t selectedPort, uint16_t selectedPins, uint8_t mode)
  This function configures the peripheral module function in the input direction for the selected pin.
- void GPIO_setOutputHighOnPin (uint8_t selectedPort, uint16_t selectedPins)
  This function sets output HIGH on the selected Pin.
- void GPIO_setOutputLowOnPin (uint8_t selectedPort, uint16_t selectedPins)
  This function sets output LOW on the selected Pin.
- void GPIO_toggleOutputOnPin (uint8_t selectedPort, uint16_t selectedPins)
  This function toggles the output on the selected Pin.
- void GPIO_setAsInputPinWithPullDownResistor (uint8_t selectedPort, uint16_t selectedPins)
  This function sets the selected Pin in input Mode with Pull Down resistor.
- void GPIO_setAsInputPinWithPullUpResistor (uint8_t selectedPort, uint16_t selectedPins)
  This function sets the selected Pin in input Mode with Pull Up resistor.
- uint8_t GPIO_getInputPinValue (uint8_t selectedPort, uint16_t selectedPins)
  This function gets the input value on the selected pin.
- void GPIO_enableInterrupt (uint8_t selectedPort, uint16_t selectedPins)
  This function enables the port interrupt on the selected pin.
- void GPIO_disableInterrupt (uint8_t selectedPort, uint16_t selectedPins)
  This function disables the port interrupt on the selected pin.
- uint16_t GPIO_getInterruptStatus (uint8_t selectedPort, uint16_t selectedPins)
  This function gets the interrupt status of the selected pin.
- void GPIO_clearInterrupt (uint8_t selectedPort, uint16_t selectedPins)
  This function clears the interrupt flag on the selected pin.
- void GPIO_selectInterruptEdge (uint8_t selectedPort, uint16_t selectedPins, uint8_t edgeSelect)
  This function selects on what edge the port interrupt flag should be set for a transition.

16.2.1 Detailed Description

The GPIO API is broken into three groups of functions: those that deal with configuring the GPIO pins, those that deal with interrupts, and those that access the pin value.

The GPIO pins are configured with

- GPIO_setAsOutputPin()
- GPIO_setAsInputPin()
- GPIO_setAsInputPinWithPullDownResistor()
- GPIO_setAsInputPinWithPullUpResistor()
- GPIO_setAsPeripheralModuleFunctionOutputPin()
- GPIO_setAsPeripheralModuleFunctionInputPin()

The GPIO interrupts are handled with

- GPIO_enableInterrupt()
- GPIO_disbleInterrupt()
- GPIO_clearInterrupt()
- GPIO_getInterruptStatus()
- GPIO_selectInterruptEdge()

The GPIO pin state is accessed with

- GPIO_setOutputHighOnPin()
- GPIO_setOutputLowOnPin()
- GPIO_toggleOutputOnPin()
- GPIO_getInputPinValue()

16.2.2 Function Documentation

GPIO_clearInterrupt()

```
void GPIO_clearInterrupt (
            uint8_t selectedPort,
            uint16_t selectedPins )
```

This function clears the interrupt flag on the selected pin.

This function clears the interrupt flag on the selected pin. Please refer to family user's guide for available ports with interrupt capability.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ▪ **GPIO_PIN_ALL8** ▪ **GPIO_PIN_ALL16**

Modified bits of **PxIFG** register.

Returns

None

GPIO_disableInterrupt()

```
void GPIO_disableInterrupt (
            uint8_t selectedPort,
            uint16_t selectedPins )
```

This function disables the port interrupt on the selected pin.

This function disables the port interrupt on the selected pin. Please refer to family user's guide for available ports with interrupt capability.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Modified bits of **PxIE** register.

Returns

None

GPIO_enableInterrupt()

```
void GPIO_enableInterrupt (
            uint8_t selectedPort,
            uint16_t selectedPins )
```

This function enables the port interrupt on the selected pin.

This function enables the port interrupt on the selected pin. Please refer to family user's guide for available ports with interrupt capability.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Modified bits of **PxIE** register.

Returns

None

GPIO_getInputPinValue()

```
uint8_t GPIO_getInputPinValue (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Returns

One of the following:
- **GPIO_INPUT_PIN_HIGH**
- **GPIO_INPUT_PIN_LOW**
  indicating the status of the pin

GPIO_getInterruptStatus()

```
uint16_t GPIO_getInterruptStatus (
            uint8_t selectedPort,
            uint16_t selectedPins )
```

This function gets the interrupt status of the selected pin.

This function gets the interrupt status of the selected pin. Please refer to family user's guide for available ports with interrupt capability.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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**
- **GPIO_PIN_ALL8**

- **GPIO_PIN_ALL16**
  indicating the interrupt status of the selected pins [Default: 0]

GPIO_selectInterruptEdge()

```
void GPIO_selectInterruptEdge (
            uint8_t selectedPort,
            uint16_t selectedPins,
            uint8_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. Please refer to family user's guide for available ports with interrupt capability.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**
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

GPIO_setAsInputPin()

```
void GPIO_setAsInputPin (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

Returns

None

GPIO_setAsInputPinWithPullDownResistor()

```
void GPIO_setAsInputPinWithPullDownResistor (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

Returns

None

GPIO_setAsInputPinWithPullUpResistor()

```
void GPIO_setAsInputPinWithPullUpResistor (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

Returns

None

GPIO_setAsOutputPin()

```
void GPIO_setAsOutputPin (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

Returns

None

GPIO_setAsPeripheralModuleFunctionInputPin()

```
void GPIO_setAsPeripheralModuleFunctionInputPin (
            uint8_t selectedPort,
            uint16_t selectedPins,
            uint8_t mode )
```

This function configures the peripheral module function in the input direction for the selected pin.

This function configures the peripheral module function in the input direction for the selected pin for either primary, secondary or ternary module function modes. Note that MSP430F5xx/6xx family doesn't support these function modes.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**
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_TERNARY_MODULE_FUNCTION**

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

Returns

None

GPIO_setAsPeripheralModuleFunctionOutputPin()

```
void GPIO_setAsPeripheralModuleFunctionOutputPin (
            uint8_t selectedPort,
            uint16_t selectedPins,
            uint8_t mode )
```

This function configures the peripheral module function in the output direction for the selected pin.

This function configures the peripheral module function in the output direction for the selected pin for either primary, secondary or ternary module function modes. Note that MSP430F5xx/6xx family doesn't support these function modes.

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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**
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_TERNARY_MODULE_FUNCTION**

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

Returns

None

GPIO_setOutputHighOnPin()

```
void GPIO_setOutputHighOnPin (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Modified bits of **PxOUT** register.

Returns

None

GPIO_setOutputLowOnPin()

```
void GPIO_setOutputLowOnPin (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Modified bits of **PxOUT** register.

Returns

None

GPIO_toggleOutputOnPin()

```
void GPIO_toggleOutputOnPin (
            uint8_t selectedPort,
            uint16_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_PA** ■ **GPIO_PORT_PB** ■ **GPIO_PORT_PC** ■ **GPIO_PORT_PD** ■ **GPIO_PORT_PE** ■ **GPIO_PORT_PF** ■ **GPIO_PORT_PJ**

Parameters

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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

Modified bits of **PxOUT** register.

Returns

None

16.3 Programming Example

The following example shows how to use the GPIO API. A trigger is generated on a hi "TO" low transition on P1.4 (pulled-up input pin), which will generate P1_ISR. In the ISR, we toggle P1.0 (output pin).

```
    //Set P1.0 to output direction
GPIO_setAsOutputPin(
    GPIO_PORT_P1,
    GPIO_PIN0
    );


//Enable P1.4 internal resistance as pull-Up resistance
GPIO_setAsInputPinWithPullUpresistor(
    GPIO_PORT_P1,
    GPIO_PIN4
```

```
        );

    //P1.4 interrupt enabled
    GPIO_enableInterrupt(
        GPIO_PORT_P1,
        GPIO_PIN4
        );

    //P1.4 Hi/Lo edge
    GPIO_selectInterruptEdge(
        GPIO_PORT_P1,
        GPIO_PIN4,
        GPIO_HIGH_TO_LOW_TRANSITION
        );


    //P1.4 IFG cleared
    GPIO_clearInterrupt(
        GPIO_PORT_P1,
        GPIO_PIN4
        );

    //Enter LPM4 w/interrupt
    __bis_SR_register(LPM4_bits + GIE);

    //For debugger
    __no_operation();
}

//******************************************************************************
//
//This is the PORT1_VECTOR interrupt vector service routine
//
//******************************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void Port_1 (void)
{
    //P1.0 = toggle
    GPIO_toggleOutputOnPin(
        GPIO_PORT_P1,
        GPIO_PIN0
        );


    //P1.4 IFG cleared
    GPIO_clearInterrupt(
        GPIO_PORT_P1,
        GPIO_PIN4
        );
}
```

17 Interrupt Compare Controller (ICC)

17.1 Introduction

Interrupt Compare Controller (ICC) provides a way to nest hardware interrupts. It allows all maskable interrupt sources to be scheduled in a preemptive mechanism. When ICC module is enabled, the ISR in lower software priority can be interrupted by higher priority. It is required to enable GIE in ISR for proper ICC operation.

The API provides a set of functions for using the ICC. Functions are provided to adjust interrupt vector nesting priority level, and examine the stack of pending interrupts.

17.2 API Functions

Functions

- void ICC_enable (void)
  Enables ICC module.
- void ICC_disable (void)
  Disables ICC module.
- void ICC_setInterruptLevel (uint32_t ILSRmask, uint8_t interruptLevel)
  Sets ICC interrupt level for selected source.
- uint8_t ICC_getInterruptLevel (uint32_t interruptSource)
  Gets ICC interrupt level for selected source.
- bool ICC_isVirtualStackEmpty (void)
  Returns true if virtual stack is empty, false if not.
- bool ICC_isVirtualStackFull (void)
  Returns true if virtual stack is full, false if not.
- uint8_t ICC_getCurrentICM (void)
  Gets the current interrupt compare mask.
- uint8_t ICC_getMVSStackPointer (void)
  Gets the ICC Mask Virtual Stack stack pointer.
- uint8_t ICC_getICM3 (void)
  Gets the interrupt level at Interrupt compare mask virtual stack position 3.
- uint8_t ICC_getICM2 (void)
  Gets the interrupt level at Interrupt compare mask virtual stack position 2.
- uint8_t ICC_getICM1 (void)
  Gets the interrupt level at Interrupt compare mask virtual stack position 1.
- uint8_t ICC_getICM0 (void)
  Gets the interrupt level at Interrupt compare mask virtual stack position 0.

17.2.1 Detailed Description

- ICC_enable()
- ICC_disable()
- ICC_setInterruptLevel()
- ICC_getInterruptLevel()
- ICC_isVirtualStackEmpty()
- ICC_isVirtualStackFull()
- ICC_getCurrentICM()
- ICC_getMVSStackPointer()
- ICC_getICM3()
- ICC_getICM2()
- ICC_getICM1()
- ICC_getICM0()

17.2.2 Function Documentation

ICC_disable()

```
void ICC_disable (
            void  )
```

Disables ICC module.

This function disables ICC.

Returns

None

ICC_enable()

```
void ICC_enable (
            void  )
```

Enables ICC module.

This function enables ICC.

Returns

None

ICC_getCurrentICM()

```
uint8_t ICC_getCurrentICM (
            void  )
```

Gets the current interrupt compare mask.

Returns a 2-bit value that specifies the minimum interrupt priority level that will be sent for service. If ICM[1:0] is less than the priority level (ILSRx[1:0]) of the new interrupt, the corresponding source is sent to the CPU. Note that the ICMC is the element stack that the stack pointer is pointing to.

Returns

Minimum ICC interrupt priority level that will be serviced. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**
- **ICC_LEVEL_ERROR**
  returns minimum interrupt level.

ICC_getICM0()

```
uint8_t ICC_getICM0 (
            void  )
```

Gets the interrupt level at Interrupt compare mask virtual stack position 0.

Returns interrupt level of ICM virtual stack position 0 as a ready-to-read 8-bit integer value (automatically adjusted from original bit position.)

Returns

Interrupt level of ICM virtual stack position 0. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**
- **ICC_LEVEL_ERROR**
  returns interrupt level of ICM virtual stack position 0.

ICC_getICM1()

```
uint8_t ICC_getICM1 (
            void  )
```

Gets the interrupt level at Interrupt compare mask virtual stack position 1.

Returns interrupt level of ICM virtual stack position 1 as a ready-to-read 8-bit integer value (automatically adjusted from original bit position.)

Returns

Interrupt level of ICM virtual stack position 1. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**

- **ICC_LEVEL_ERROR**
  returns interrupt level of ICM virtual stack position 1.

ICC_getICM2()

```
uint8_t ICC_getICM2 (
            void )
```

Gets the interrupt level at Interrupt compare mask virtual stack position 2.

Returns interrupt level of ICM virtual stack position 2 as a ready-to-read 8-bit integer value (automatically adjusted from original bit position.)

Returns

Interrupt level of ICM virtual stack position 2. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**
- **ICC_LEVEL_ERROR**
  returns interrupt level of ICM virtual stack position 2.

ICC_getICM3()

```
uint8_t ICC_getICM3 (
            void )
```

Gets the interrupt level at Interrupt compare mask virtual stack position 3.

Returns interrupt level of ICM virtual stack position 3 as a ready-to-read 8-bit integer value (automatically adjusted from original bit position.)

Returns

Interrupt level of ICM virtual stack position 3. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**
- **ICC_LEVEL_ERROR**
  returns interrupt level of ICM virtual stack position 3.

ICC_getInterruptLevel()

```
uint8_t ICC_getInterruptLevel (
            uint32_t interruptSource )
```

Gets ICC interrupt level for selected source.

This function gets ICC interrupt level given a maskable interrupt source.

Parameters

interruptSource	is a 32-bit unsigned integer in which bit position (31 to 0) determines which interrupt source to read level from. Valid values are: ■ **ICC_ILSR_P4** ■ **ICC_ILSR_P3** ■ **ICC_ILSR_P2** ■ **ICC_ILSR_P1** ■ **ICC_ILSR_SAC3DAC_SAC1DAC** ■ **ICC_ILSR_SAC2DAC_SAC0DAC** ■ **ICC_ILSR_ECOMP1_ECOMP0** ■ **ICC_ILSR_ADC** ■ **ICC_ILSR_EUSCI_B1** ■ **ICC_ILSR_EUSCI_B0** ■ **ICC_ILSR_EUSCI_A1** ■ **ICC_ILSR_EUSCI_A0** ■ **ICC_ILSR_WDT_INT** ■ **ICC_ILSR_RTC_COUNTER** ■ **ICC_ILSR_TIMER3_B1** ■ **ICC_ILSR_TIMER3_B0** ■ **ICC_ILSR_TIMER2_B1** ■ **ICC_ILSR_TIMER2_B0** ■ **ICC_ILSR_TIMER1_B1** ■ **ICC_ILSR_TIMER1_B0** ■ **ICC_ILSR_TIMER0_B1** ■ **ICC_ILSR_TIMER0_B0**

Returns

Interrupt level of a given maskable interrupt source. Return one of the following:

- **ICC_LEVEL_0**
- **ICC_LEVEL_1**
- **ICC_LEVEL_2**
- **ICC_LEVEL_3**
- **ICC_LEVEL_ERROR**
  returns interrupt level of given interrupt source.

ICC_getMVSStackPointer()

```
uint8_t ICC_getMVSStackPointer (
            void )
```

Gets the ICC Mask Virtual Stack stack pointer.

Returns the stack pointer of the ICC Mask Virtual Stack.

Returns

0 if stack is empty, 1 if ICM0 occupied, 2 if ICM0/ICM1, 3 if ICM0/ICM1/ICM2, 4 if full. Return one of the following:

- **ICC_MVS_STACK_EMPTY**
- **ICC_MVS_STACK_ICM0**
- **ICC_MVS_STACK_ICM0_ICM1**
- **ICC_MVS_STACK_ICM0_ICM1_ICM2**
- **ICC_MVS_STACK_FULL**
  Determines how full MVS stack is

ICC_isVirtualStackEmpty()

```
bool ICC_isVirtualStackEmpty (
            void  )
```

Returns true if virtual stack is empty, false if not.

This function returns true(1) if virtual stack is empty, false(0) if not.

Returns

1 if virtual stack is empty, 0 if not

ICC_isVirtualStackFull()

```
bool ICC_isVirtualStackFull (
            void  )
```

Returns true if virtual stack is full, false if not.

This function returns true(1) if virtual stack is full, false(0) if not.

Returns

1 if virtual stack is full, 0 if not

ICC_setInterruptLevel()

```
void ICC_setInterruptLevel (
            uint32_t ILSRmask,
            uint8_t interruptLevel )
```

Sets ICC interrupt level for selected source.

This function sets ICC interrupt level given a maskable interrupt source.

Parameters

ILSRmask	is a 32-bit unsigned integer in which bit position (31 to 0) determines which interrupt source is set. Mask value is the logical OR of any of the following: ■ **ICC_ILSR_P4** ■ **ICC_ILSR_P3** ■ **ICC_ILSR_P2** ■ **ICC_ILSR_P1** ■ **ICC_ILSR_SAC3DAC_SAC1DAC** ■ **ICC_ILSR_SAC2DAC_SAC0DAC** ■ **ICC_ILSR_ECOMP1_ECOMP0** ■ **ICC_ILSR_ADC** ■ **ICC_ILSR_EUSCI_B1** ■ **ICC_ILSR_EUSCI_B0** ■ **ICC_ILSR_EUSCI_A1** ■ **ICC_ILSR_EUSCI_A0** ■ **ICC_ILSR_WDT_INT** ■ **ICC_ILSR_RTC_COUNTER** ■ **ICC_ILSR_TIMER3_B1** ■ **ICC_ILSR_TIMER3_B0** ■ **ICC_ILSR_TIMER2_B1** ■ **ICC_ILSR_TIMER2_B0** ■ **ICC_ILSR_TIMER1_B1** ■ **ICC_ILSR_TIMER1_B0** ■ **ICC_ILSR_TIMER0_B1** ■ **ICC_ILSR_TIMER0_B0**
interruptLevel	determines what interrupt level to set to. Valid values are: ■ **ICC_LEVEL_0** ■ **ICC_LEVEL_1** ■ **ICC_LEVEL_2** ■ **ICC_LEVEL_3** ■ **ICC_LEVEL_ERROR**

Returns

None

17.3 Programming Example

The following example shows how to initialize and use ICC.

```
// ...Enable your interrupts prior to this
ICC_setInterruptLevel(ICC_ILSR_ADC, ICC_LEVEL_1);
ICC_setInterruptLevel(ICC_ILSR_P1, ICC_LEVEL_0);
ICC_enable();

//Enter LPM3 mode, enable interrupts
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

18 LCD_E Controller

18.1 Introduction

The LCD_E Controller APIs provides a set of functions for using the LCD_E module. Main functions include initialization, LCD enable/disable, charge pump config, voltage settings and memory/blink memory writing.

LCD_E is same as LCD_C which supports 5-mux $\sim$ 8-mux and low power waveform. Besides that, all the LCD drive pins can be configured as COM. LCD_E also supports LPM 3.5 by using separated power domain.

18.2 API Functions

Functions

- void LCD_E_init (uint16_t baseAddress, LCD_E_initParam *initParams)
  Initializes the LCD_E Module.
- void LCD_E_on (uint16_t baseAddress)
  Turns on the LCD_E module.
- void LCD_E_off (uint16_t baseAddress)
  Turns the LCD_E off.
- void LCD_E_clearInterrupt (uint16_t baseAddress, uint16_t mask)
  Clears the LCD_E selected interrupt flags.
- uint16_t LCD_E_getInterruptStatus (uint16_t baseAddress, uint16_t mask)
  Returns the status of the selected interrupt flags.
- void LCD_E_enableInterrupt (uint16_t baseAddress, uint16_t mask)
  Enables selected LCD_E interrupt sources.
- void LCD_E_disableInterrupt (uint16_t baseAddress, uint16_t mask)
  Disables selected LCD_E interrupt sources.
- void LCD_E_clearAllMemory (uint16_t baseAddress)
  Clears all LCD_E memory registers.
- void LCD_E_clearAllBlinkingMemory (uint16_t baseAddress)
  Clears all LCD_E blinking memory registers.
- void LCD_E_selectDisplayMemory (uint16_t baseAddress, uint16_t displayMemory)
  Selects display memory.
- void LCD_E_setBlinkingControl (uint16_t baseAddress, uint16_t clockPrescalar, uint16_t mode)
  Sets the blinking control register.
- void LCD_E_enableChargePump (uint16_t baseAddress)
  Enables the charge pump.
- void LCD_E_disableChargePump (uint16_t baseAddress)
  Disables the charge pump.

- void LCD_E_setChargePumpFreq (uint16_t baseAddress, uint16_t freq)
  Sets the charge pump frequency.
- void LCD_E_setVLCDSource (uint16_t baseAddress, uint16_t r13Source, uint16_t r33Source)
  Sets LCD_E voltage source.
- void LCD_E_setVLCDVoltage (uint16_t baseAddress, uint16_t voltage)
  Sets LCD_E internal voltage for R13.
- void LCD_E_setReferenceMode (uint16_t baseAddress, uint16_t mode)
  Sets the reference mode for R13.
- void LCD_E_setPinAsLCDFunction (uint16_t baseAddress, uint8_t pin)
  Sets the LCD_E pins as LCD function pin.
- void LCD_E_setPinAsPortFunction (uint16_t baseAddress, uint8_t pin)
  Sets the LCD_E pins as port function pin.
- void LCD_E_setPinAsLCDFunctionEx (uint16_t baseAddress, uint8_t startPin, uint8_t endPin)
  Sets the LCD_E pins as LCD function pin.
- void LCD_E_setPinAsCOM (uint16_t baseAddress, uint8_t pin, uint8_t com)
  Sets the LCD_E pin as a common line.
- void LCD_E_setPinAsSEG (uint16_t baseAddress, uint8_t pin)
  Sets the LCD_E pin as a segment line.
- void LCD_E_setMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Sets the LCD_E memory register.
- void LCD_E_updateMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Updates the LCD_E memory register.
- void LCD_E_toggleMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Toggles the LCD_E memory register.
- void LCD_E_clearMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Clears the LCD_E memory register.
- void LCD_E_setBlinkingMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Sets the LCD_E blinking memory register.
- void LCD_E_updateBlinkingMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Updates the LCD_E blinking memory register.
- void LCD_E_toggleBlinkingMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Toggles the LCD_E blinking memory register.
- void LCD_E_clearBlinkingMemory (uint16_t baseAddress, uint8_t memory, uint8_t mask)
  Clears the LCD_E blinking memory register.

Variables

- const LCD_E_initParam LCD_E_INIT_PARAM

18.2.1 Detailed Description

The LCD_E API is broken into four groups of functions: those that deal with the basic setup and pin config, those that handle change pump, VLCD voltage and source, those that set memory and blink memory, and those auxiliary functions.

The LCD_E setup and pin config functions are

- LCD_E_init()

- LCD_E_on()
- LCD_E_off()
- LCD_E_setPinAsLCDFunction()
- LCD_E_setPinAsPortFunction()
- LCD_E_setPinAsLCDFunctionEx()
- LCD_E_setPinAsCOM()
- LCD_E_setPinAsSEG()

The LCD_E charge pump, VLCD voltage/source functions are

- LCD_E_enableChargePump()
- LCD_E_disableChargePump()
- LCD_E_setChargePumpFreq()
- LCD_E_setVLCDSource()
- LCD_E_setVLCDVoltage()
- LCD_E_setReferenceMode()

The LCD_E memory/blinking memory setting funtions are

- LCD_E_clearAllMemory()
- LCD_E_clearAllBlinkingMemory()
- LCD_E_selectDisplayMemory()
- LCD_E_setBlinkingControl()
- LCD_E_setMemory()
- LCD_E_updateMemory()
- LCD_E_toggleMemory()
- LCD_E_clearMemory()
- LCD_E_setBlinkingMemory()
- LCD_E_updateBlinkingMemory()
- LCD_E_toggleBlinkingMemory()
- LCD_E_clearBlinkingMemory()

The LCD_E auxiliary functions are

- LCD_E_clearInterrupt()
- LCD_E_getInterruptStatus()
- LCD_E_enableInterrupt()
- LCD_E_disableInterrupt()

18.2.2 Function Documentation

LCD_E_clearAllBlinkingMemory()

```
void LCD_E_clearAllBlinkingMemory (
```

```
uint16_t baseAddress )
```

Clears all LCD_E blinking memory registers.

This function clears all LCD_E blinking memory registers.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDCLRBM** of **LCDMEMCTL** register.

Returns

None

LCD_E_clearAllMemory()

```
void LCD_E_clearAllMemory (
            uint16_t baseAddress )
```

Clears all LCD_E memory registers.

This function clears all LCD_E memory registers.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDCLRM** of **LCDMEMCTL** register.

Returns

None

LCD_E_clearBlinkingMemory()

```
void LCD_E_clearBlinkingMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Clears the LCD_E blinking memory register.

This function clears the specific bits in the LCD_E blinking memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select blinking memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding blinking memory.

Modified bits are **MBITx** of **LCDBMx** register.

Returns

None

LCD_E_clearInterrupt()

```
void LCD_E_clearInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Clears the LCD_E selected interrupt flags.

This function clears the specified interrupt flags.

Parameters

baseAddress	is the base address of the LCD_E module.
mask	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: ■ **LCD_E_BLINKING_SEGMENTS_ON_INTERRUPT** ■ **LCD_E_BLINKING_SEGMENTS_OFF_INTERRUPT** ■ **LCD_E_FRAME_INTERRUPT** Modified bits are **LCDBLKONIFG**, **LCDBLKOFFIFG** and **LCDFRMIFG** of **LCDCTL1** register.

Returns

None

LCD_E_clearMemory()

```
void LCD_E_clearMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Clears the LCD_E memory register.

This function clears the specific bits in the LCD_E memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding memory.

Modified bits are **MBITx** of **LCDMx** register.

Returns

None

LCD_E_disableChargePump()

```
void LCD_E_disableChargePump (
            uint16_t baseAddress )
```

Disables the charge pump.

This function disables the charge pump.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDCPEN** of **LCDVCTL** register.

Returns

None

LCD_E_disableInterrupt()

```
void LCD_E_disableInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Disables selected LCD_E interrupt sources.

This function disables the indicated LCD_E interrupt sources.

Parameters

baseAddress	is the base address of the LCD_E module.
mask	is the interrupts to be disabled. Mask value is the logical OR of any of the following: ■ **LCD_E_BLINKING_SEGMENTS_ON_INTERRUPT** ■ **LCD_E_BLINKING_SEGMENTS_OFF_INTERRUPT** ■ **LCD_E_FRAME_INTERRUPT** Modified bits are **LCDBLKONIE**, **LCDBLKOFFIE** and **LCDFRMIE** of **LCDCTL1** register.

Returns

None

LCD_E_enableChargePump()

```
void LCD_E_enableChargePump (
            uint16_t baseAddress )
```

Enables the charge pump.

This function enables the charge pump and config the charge pump frequency.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDCPEN** of **LCDVCTL** register.

Returns

None

LCD_E_enableInterrupt()

```
void LCD_E_enableInterrupt (
            uint16_t baseAddress,
            uint16_t mask )
```

Enables selected LCD_E interrupt sources.

This function enables the indicated LCD_E interrupt sources.

Parameters

baseAddress	is the base address of the LCD_E module.
mask	is the interrupts to be enabled. Mask value is the logical OR of any of the following: ■ **LCD_E_BLINKING_SEGMENTS_ON_INTERRUPT** ■ **LCD_E_BLINKING_SEGMENTS_OFF_INTERRUPT** ■ **LCD_E_FRAME_INTERRUPT** Modified bits are **LCDBLKONIE**, **LCDBLKOFFIE** and **LCDFRMIE** of **LCDCTL1** register.

Returns

None

LCD_E_getInterruptStatus()

```
uint16_t LCD_E_getInterruptStatus (
            uint16_t baseAddress,
            uint16_t mask )
```

Returns the status of the selected interrupt flags.

This function returns the status of the selected interrupt flags.

Parameters

baseAddress	is the base address of the LCD_E module.
mask	is the masked interrupt flags. Mask value is the logical OR of any of the following: ■ **LCD_E_BLINKING_SEGMENTS_ON_INTERRUPT** ■ **LCD_E_BLINKING_SEGMENTS_OFF_INTERRUPT** ■ **LCD_E_FRAME_INTERRUPT**

Returns

The current interrupt flag status for the corresponding mask. Return Logical OR of any of the following:

- **LCD_E_BLINKING_SEGMENTS_ON_INTERRUPT**
- **LCD_E_BLINKING_SEGMENTS_OFF_INTERRUPT**
- **LCD_E_FRAME_INTERRUPT**
  indicating the status of the masked interrupts

LCD_E_init()

```
void LCD_E_init (
            uint16_t baseAddress,
            LCD_E_initParam * initParams )
```

Initializes the LCD_E Module.

This function initializes the LCD_E but without turning on. It bascially setup the clock source, clock divider, mux rate, low-power waveform and segments on/off. After calling this function, user can enable/disable charge pump, internal reference voltage, or pin SEG/COM configurations.

Parameters

baseAddress	is the base address of the LCD_E module.
initParams	is the pointer to LCD_InitParam structure. See the following parameters for each field.

Returns

None

References LCD_E_initParam::clockDivider, LCD_E_initParam::clockSource, LCD_E_initParam::muxRate, LCD_E_initParam::segments, and LCD_E_initParam::waveforms.

LCD_E_off()

```
void LCD_E_off (
            uint16_t baseAddress )
```

Turns the LCD_E off.

This function turns the LCD_E off.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDPCTL** of **SYSCFG2** register; bits **LCDON** of **LCDCTL0** register.

Returns

None

LCD_E_on()

```
void LCD_E_on (
            uint16_t baseAddress )
```

Turns on the LCD_E module.

This function turns the LCD_E on.

Parameters

baseAddress	is the base address of the LCD_E module.

Modified bits are **LCDPCTL** of **SYSCFG2** register; bits **LCDON** of **LCDCTL0** register.

Returns

None

LCD_E_selectDisplayMemory()

```
void LCD_E_selectDisplayMemory (
            uint16_t baseAddress,
            uint16_t displayMemory )
```

Selects display memory.

This function selects display memory either from memory or blinking memory. Please note if the blinking mode is selected as LCD_E_BLINKMODE_INDIVIDUALSEGMENTS or LCD_E_BLINKMODE_ALLSEGMENTS or mux rate >=5, display memory can not be changed. If LCD_E_BLINKMODE_SWITCHDISPLAYCONTENTS is selected, display memory bit reflects current displayed memory.

Parameters

baseAddress	is the base address of the LCD_E module.
displayMemory	is the desired displayed memory. Valid values are: ■ **LCD_E_DISPLAYSOURCE_MEMORY** [Default] ■ **LCD_E_DISPLAYSOURCE_BLINKINGMEMORY** Modified bits are **LCDDISP** of **LCDMEMCTL** register.

Returns

None

LCD_E_setBlinkingControl()

```
void LCD_E_setBlinkingControl (
            uint16_t baseAddress,
            uint16_t clockPrescalar,
            uint16_t mode )
```

Sets the blinking control register.

This function sets the blink control related parameter, including blink clock frequency prescalar and blink mode.

Parameters

baseAddress	is the base address of the LCD_E module.
clockPrescalar	is the clock pre-scalar for blinking frequency. Valid values are: ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_4** [Default] ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_8** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_16** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_32** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_128** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_256** ■ **LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_512** Modified bits are **LCDBLKPREx** of **LCDBLKCTL** register.

Parameters

mode	is the select for blinking mode. Valid values are: ■ **LCD_E_BLINK_MODE_DISABLED** [Default] ■ **LCD_E_BLINK_MODE_INDIVIDUAL_SEGMENTS** ■ **LCD_E_BLINK_MODE_ALL_SEGMENTS** ■ **LCD_E_BLINK_MODE_SWITCHING_BETWEEN_DISPLAY_CONTE↩ NTS** Modified bits are **LCDBLKMODx** of **LCDBLKCTL** register.

Returns

None

LCD_E_setBlinkingMemory()

```
void LCD_E_setBlinkingMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Sets the LCD_E blinking memory register.

This function sets the entire one LCD_E blinking memory register.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select blinking memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding blinking memory.

Modified bits are **MBITx** of **LCDBMx** register.

Returns

None

LCD_E_setChargePumpFreq()

```
void LCD_E_setChargePumpFreq (
            uint16_t baseAddress,
            uint16_t freq )
```

Sets the charge pump frequency.

This function sets the charge pump frequency. It takes effect once charge pump is enabled by LCD_E_enableChargePump().

Parameters

baseAddress	is the base address of the LCD_E module.
freq	is the charge pump frequency to select. Valid values are: ■ **LCD_E_CHARGEPUMP_FREQ_1** [Default] ■ **LCD_E_CHARGEPUMP_FREQ_2** ■ **LCD_E_CHARGEPUMP_FREQ_3** ■ **LCD_E_CHARGEPUMP_FREQ_4** ■ **LCD_E_CHARGEPUMP_FREQ_5** ■ **LCD_E_CHARGEPUMP_FREQ_6** ■ **LCD_E_CHARGEPUMP_FREQ_7** ■ **LCD_E_CHARGEPUMP_FREQ_8** ■ **LCD_E_CHARGEPUMP_FREQ_9** ■ **LCD_E_CHARGEPUMP_FREQ_10** ■ **LCD_E_CHARGEPUMP_FREQ_11** ■ **LCD_E_CHARGEPUMP_FREQ_12** ■ **LCD_E_CHARGEPUMP_FREQ_13** ■ **LCD_E_CHARGEPUMP_FREQ_14** ■ **LCD_E_CHARGEPUMP_FREQ_15** ■ **LCD_E_CHARGEPUMP_FREQ_16** Modified bits are **LCDCPFSELx** of **LCDVCTL** register.

Returns

None

LCD_E_setMemory()

```
void LCD_E_setMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Sets the LCD_E memory register.

This function sets the entire one LCD_E memory register.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding memory.

Modified bits are **MBITx** of **LCDMx** register.

Returns

None

LCD_E_setPinAsCOM()

```
void LCD_E_setPinAsCOM (
            uint16_t baseAddress,
            uint8_t pin,
            uint8_t com )
```

Sets the LCD_E pin as a common line.

This function sets the LCD_E pin as a common line and assigns the corresponding memory pin to a specific COM line.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

pin	is the selected pin to be configed as common line. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40** ■ **LCD_E_SEGMENT_LINE_41**

Parameters

com	is the selected COM number for the common line. Valid values are: ■ **LCD_E_MEMORY_COM0** ■ **LCD_E_MEMORY_COM1** ■ **LCD_E_MEMORY_COM2** ■ **LCD_E_MEMORY_COM3** ■ **LCD_E_MEMORY_COM4** - only for 5-Mux/6-Mux/7-Mux/8-Mux ■ **LCD_E_MEMORY_COM5** - only for 5-Mux/6-Mux/7-Mux/8-Mux ■ **LCD_E_MEMORY_COM6** - only for 5-Mux/6-Mux/7-Mux/8-Mux ■ **LCD_E_MEMORY_COM7** - only for 5-Mux/6-Mux/7-Mux/8-Mux

Modified bits are **LCDCSSx** of **LCDSSELx** register; bits **MBITx** of **LCDBMx** register; bits **MBITx** of **LCDMx** register.

Returns

None

LCD_E_setPinAsLCDFunction()

```
void LCD_E_setPinAsLCDFunction (
            uint16_t baseAddress,
            uint8_t pin )
```

Sets the LCD_E pins as LCD function pin.

This function sets the LCD_E pins as LCD function pin.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

pin	is the select pin set as LCD function. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40**

Parameters

Modified bits are **LCDSx** of **LCDPCTLx** register.

Returns

None

LCD_E_setPinAsLCDFunctionEx()

```
void LCD_E_setPinAsLCDFunctionEx (
            uint16_t baseAddress,
            uint8_t startPin,
            uint8_t endPin )
```

Sets the LCD_E pins as LCD function pin.

This function sets the LCD_E pins as LCD function pin. Instead of passing the all the possible pins, it just requires the start pin and the end pin.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

startPin	is the starting pin to be configed as LCD function pin. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40** ■ **LCD_E_SEGMENT_LINE_41**

Parameters

endPin	is the ending pin to be configed as LCD function pin. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40** ■ **LCD_E_SEGMENT_LINE_41**

Parameters

Modified bits are **LCDSx** of **LCDPCTLx** register.

Returns

None

LCD_E_setPinAsPortFunction()

```
void LCD_E_setPinAsPortFunction (
            uint16_t baseAddress,
            uint8_t pin )
```

Sets the LCD_E pins as port function pin.

This function sets the LCD_E pins as port function pin.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

pin	is the select pin set as Port function. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40**

Parameters

Modified bits are **LCDSx** of **LCDPCTLx** register.

Returns

None

LCD_E_setPinAsSEG()

```
void LCD_E_setPinAsSEG (
            uint16_t baseAddress,
            uint8_t pin )
```

Sets the LCD_E pin as a segment line.

This function sets the LCD_E pin as segment line.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

pin	is the selected pin to be configed as segment line. Valid values are: ■ **LCD_E_SEGMENT_LINE_0** ■ **LCD_E_SEGMENT_LINE_1** ■ **LCD_E_SEGMENT_LINE_2** ■ **LCD_E_SEGMENT_LINE_3** ■ **LCD_E_SEGMENT_LINE_4** ■ **LCD_E_SEGMENT_LINE_5** ■ **LCD_E_SEGMENT_LINE_6** ■ **LCD_E_SEGMENT_LINE_7** ■ **LCD_E_SEGMENT_LINE_8** ■ **LCD_E_SEGMENT_LINE_9** ■ **LCD_E_SEGMENT_LINE_10** ■ **LCD_E_SEGMENT_LINE_11** ■ **LCD_E_SEGMENT_LINE_12** ■ **LCD_E_SEGMENT_LINE_13** ■ **LCD_E_SEGMENT_LINE_14** ■ **LCD_E_SEGMENT_LINE_15** ■ **LCD_E_SEGMENT_LINE_16** ■ **LCD_E_SEGMENT_LINE_17** ■ **LCD_E_SEGMENT_LINE_18** ■ **LCD_E_SEGMENT_LINE_19** ■ **LCD_E_SEGMENT_LINE_20** ■ **LCD_E_SEGMENT_LINE_21** ■ **LCD_E_SEGMENT_LINE_22** ■ **LCD_E_SEGMENT_LINE_23** ■ **LCD_E_SEGMENT_LINE_24** ■ **LCD_E_SEGMENT_LINE_25** ■ **LCD_E_SEGMENT_LINE_26** ■ **LCD_E_SEGMENT_LINE_27** ■ **LCD_E_SEGMENT_LINE_28** ■ **LCD_E_SEGMENT_LINE_29** ■ **LCD_E_SEGMENT_LINE_30** ■ **LCD_E_SEGMENT_LINE_31** ■ **LCD_E_SEGMENT_LINE_32** ■ **LCD_E_SEGMENT_LINE_33** ■ **LCD_E_SEGMENT_LINE_34** ■ **LCD_E_SEGMENT_LINE_35** ■ **LCD_E_SEGMENT_LINE_36** ■ **LCD_E_SEGMENT_LINE_37** ■ **LCD_E_SEGMENT_LINE_38** ■ **LCD_E_SEGMENT_LINE_39** ■ **LCD_E_SEGMENT_LINE_40** ■ **LCD_E_SEGMENT_LINE_41**

Parameters

Modified bits are **LCDCSSx** of **LCDSSELx** register.

Returns

None

LCD_E_setReferenceMode()

```
void LCD_E_setReferenceMode (
            uint16_t baseAddress,
            uint16_t mode )
```

Sets the reference mode for R13.

This function sets the reference mode for R13. In the switch mode, the Bias Voltage Generator is on for 1 clock and off for 256 clock cycles to save power. In the static mode, the Bias Voltage Generator is able to drive larger LCD panels.

Parameters

baseAddress	is the base address of the LCD_E module.
mode	is the reference mode on R13. Valid values are: ■ **LCD_E_REFERENCE_MODE_STATIC** [Default] ■ **LCD_E_REFERENCE_MODE_SWITCHED** Modified bits are **LCDREFMODE** of **LCDVCTL** register.

Returns

None

LCD_E_setVLCDSource()

```
void LCD_E_setVLCDSource (
            uint16_t baseAddress,
            uint16_t r13Source,
            uint16_t r33Source )
```

Sets LCD_E voltage source.

Two voltage sources are set in this function: R13 and R33. For the R13, the voltage source can be either internal reference voltage or non internal reference voltage (Vext or Vdd). For the R33, it can be external supply voltage (Vext) or internal supply voltage (Vdd).

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

r13Source	is the voltage source for R13. Valid values are: ■ **LCD_E_NON_INTERNAL_REFERENCE_VOLTAGE** [Default] ■ **LCD_E_INTERNAL_REFERENCE_VOLTAGE** Modified bits are **LCDREFEN** of **LCDVCTL** register.
r33Source	is the voltage source for R33. Valid values are: ■ **LCD_E_EXTERNAL_SUPPLY_VOLTAGE** [Default] ■ **LCD_E_INTERNAL_SUPPLY_VOLTAGE** Modified bits are **LCDSELVDD** of **LCDVCTL** register.

Returns

None

LCD_E_setVLCDVoltage()

```
void LCD_E_setVLCDVoltage (
            uint16_t baseAddress,
            uint16_t voltage )
```

Sets LCD_E internal voltage for R13.

This function sets the internal voltage for R13. The voltage is only valuable when R13 voltage source is using internal reference voltage and charge pump is enabled.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

voltage	is the charge pump select. Valid values are: ■ **LCD_E_REFERENCE_VOLTAGE_2_60V** [Default] ■ **LCD_E_REFERENCE_VOLTAGE_2_66V** ■ **LCD_E_REFERENCE_VOLTAGE_2_72V** ■ **LCD_E_REFERENCE_VOLTAGE_2_78V** ■ **LCD_E_REFERENCE_VOLTAGE_2_84V** ■ **LCD_E_REFERENCE_VOLTAGE_2_90V** ■ **LCD_E_REFERENCE_VOLTAGE_2_96V** ■ **LCD_E_REFERENCE_VOLTAGE_3_02V** ■ **LCD_E_REFERENCE_VOLTAGE_3_08V** ■ **LCD_E_REFERENCE_VOLTAGE_3_14V** ■ **LCD_E_REFERENCE_VOLTAGE_3_20V** ■ **LCD_E_REFERENCE_VOLTAGE_3_26V** ■ **LCD_E_REFERENCE_VOLTAGE_3_32V** ■ **LCD_E_REFERENCE_VOLTAGE_3_38V** ■ **LCD_E_REFERENCE_VOLTAGE_3_44V** ■ **LCD_E_REFERENCE_VOLTAGE_3_50V** Modified bits are **VLCDx** of **LCDVCTL** register.

Returns

None

LCD_E_toggleBlinkingMemory()

```
void LCD_E_toggleBlinkingMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Toggles the LCD_E blinking memory register.

This function toggles the specific bits in the LCD_E blinking memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select blinking memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding blinking memory.

Modified bits are **MBITx** of **LCDBMx** register.

Returns

None

LCD_E_toggleMemory()

```
void LCD_E_toggleMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Toggles the LCD_E memory register.

This function toggles the specific bits in the LCD_E memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding memory.

Modified bits are **MBITx** of **LCDMx** register.

Returns

None

LCD_E_updateBlinkingMemory()

```
void LCD_E_updateBlinkingMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Updates the LCD_E blinking memory register.

This function updates the specific bits in the LCD_E blinking memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select blinking memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding blinking memory.

Modified bits are **MBITx** of **LCDBMx** register.

Returns

None

LCD_E_updateMemory()

```
void LCD_E_updateMemory (
            uint16_t baseAddress,
            uint8_t memory,
            uint8_t mask )
```

Updates the LCD_E memory register.

This function updates the specific bits in the LCD_E memory register according to the mask.

Parameters

baseAddress	is the base address of the LCD_E module.

Parameters

memory	is the select memory for setting value. Valid values are: ■ **LCD_E_MEMORY_BLINKINGMEMORY_0** ■ **LCD_E_MEMORY_BLINKINGMEMORY_1** ■ **LCD_E_MEMORY_BLINKINGMEMORY_2** ■ **LCD_E_MEMORY_BLINKINGMEMORY_3** ■ **LCD_E_MEMORY_BLINKINGMEMORY_4** ■ **LCD_E_MEMORY_BLINKINGMEMORY_5** ■ **LCD_E_MEMORY_BLINKINGMEMORY_6** ■ **LCD_E_MEMORY_BLINKINGMEMORY_7** ■ **LCD_E_MEMORY_BLINKINGMEMORY_8** ■ **LCD_E_MEMORY_BLINKINGMEMORY_9** ■ **LCD_E_MEMORY_BLINKINGMEMORY_10** ■ **LCD_E_MEMORY_BLINKINGMEMORY_11** ■ **LCD_E_MEMORY_BLINKINGMEMORY_12** ■ **LCD_E_MEMORY_BLINKINGMEMORY_13** ■ **LCD_E_MEMORY_BLINKINGMEMORY_14** ■ **LCD_E_MEMORY_BLINKINGMEMORY_15** ■ **LCD_E_MEMORY_BLINKINGMEMORY_16** ■ **LCD_E_MEMORY_BLINKINGMEMORY_17** ■ **LCD_E_MEMORY_BLINKINGMEMORY_18** ■ **LCD_E_MEMORY_BLINKINGMEMORY_19** ■ **LCD_E_MEMORY_BLINKINGMEMORY_20** ■ **LCD_E_MEMORY_BLINKINGMEMORY_21** ■ **LCD_E_MEMORY_BLINKINGMEMORY_22** ■ **LCD_E_MEMORY_BLINKINGMEMORY_23** ■ **LCD_E_MEMORY_BLINKINGMEMORY_24** ■ **LCD_E_MEMORY_BLINKINGMEMORY_25** ■ **LCD_E_MEMORY_BLINKINGMEMORY_26** ■ **LCD_E_MEMORY_BLINKINGMEMORY_27** ■ **LCD_E_MEMORY_BLINKINGMEMORY_28** ■ **LCD_E_MEMORY_BLINKINGMEMORY_29** ■ **LCD_E_MEMORY_BLINKINGMEMORY_30** ■ **LCD_E_MEMORY_BLINKINGMEMORY_31** ■ **LCD_E_MEMORY_BLINKINGMEMORY_32** ■ **LCD_E_MEMORY_BLINKINGMEMORY_33** ■ **LCD_E_MEMORY_BLINKINGMEMORY_34** ■ **LCD_E_MEMORY_BLINKINGMEMORY_35** ■ **LCD_E_MEMORY_BLINKINGMEMORY_36** ■ **LCD_E_MEMORY_BLINKINGMEMORY_37** ■ **LCD_E_MEMORY_BLINKINGMEMORY_38** ■ **LCD_E_MEMORY_BLINKINGMEMORY_39**

Parameters

mask	is the designated value for the corresponding memory.

Modified bits are **MBITx** of **LCDMx** register.

Returns

None

18.2.3 Variable Documentation

LCD_E_INIT_PARAM

```
const LCD_E_initParam LCD_E_INIT_PARAM
```

Initial value:

```
= {
        LCD_E_CLOCKSOURCE_XTCLK,
        LCD_E_CLOCKDIVIDER_1,
        LCD_E_STATIC,
        LCD_E_STANDARD_WAVEFORMS,
        LCD_E_SEGMENTS_DISABLED
}
```

Initialization parameter instance

Parameters

clockSource	selects the clock that will be used by the LCD_E. Valid values are: ■ **LCD_E_CLOCKSOURCE_XTCLK** [Default] - The external oscillator clock. ■ **LCD_E_CLOCKSOURCE_ACLK** - The Auxilary Clock. ■ **LCD_E_CLOCKSOURCE_VLOCLK** - The internal low power and low frequency clock. Modified bits are **LCDSSEL** of **LCDCTL0** register.

Parameters

clockDivider	selects the divider for LCD_E frequency. Valid values are: ■ **LCD_E_CLOCKDIVIDER_1** [Default] ■ **LCD_E_CLOCKDIVIDER_2** ■ **LCD_E_CLOCKDIVIDER_3** ■ **LCD_E_CLOCKDIVIDER_4** ■ **LCD_E_CLOCKDIVIDER_5** ■ **LCD_E_CLOCKDIVIDER_6** ■ **LCD_E_CLOCKDIVIDER_7** ■ **LCD_E_CLOCKDIVIDER_8** ■ **LCD_E_CLOCKDIVIDER_9** ■ **LCD_E_CLOCKDIVIDER_10** ■ **LCD_E_CLOCKDIVIDER_11** ■ **LCD_E_CLOCKDIVIDER_12** ■ **LCD_E_CLOCKDIVIDER_13** ■ **LCD_E_CLOCKDIVIDER_14** ■ **LCD_E_CLOCKDIVIDER_15** ■ **LCD_E_CLOCKDIVIDER_16** ■ **LCD_E_CLOCKDIVIDER_17** ■ **LCD_E_CLOCKDIVIDER_18** ■ **LCD_E_CLOCKDIVIDER_19** ■ **LCD_E_CLOCKDIVIDER_20** ■ **LCD_E_CLOCKDIVIDER_21** ■ **LCD_E_CLOCKDIVIDER_22** ■ **LCD_E_CLOCKDIVIDER_23** ■ **LCD_E_CLOCKDIVIDER_24** ■ **LCD_E_CLOCKDIVIDER_25** ■ **LCD_E_CLOCKDIVIDER_26** ■ **LCD_E_CLOCKDIVIDER_27** ■ **LCD_E_CLOCKDIVIDER_28** ■ **LCD_E_CLOCKDIVIDER_29** ■ **LCD_E_CLOCKDIVIDER_30** ■ **LCD_E_CLOCKDIVIDER_31** ■ **LCD_E_CLOCKDIVIDER_32** Modified bits are **LCDDIVx** of **LCDCTL0** register.

Parameters

muxRate	selects LCD_E mux rate. Valid values are: ■ **LCD_E_STATIC** [Default] ■ **LCD_E_2_MUX** ■ **LCD_E_3_MUX** ■ **LCD_E_4_MUX** ■ **LCD_E_5_MUX** ■ **LCD_E_6_MUX** ■ **LCD_E_7_MUX** ■ **LCD_E_8_MUX** Modified bits are **LCDMXx** of **LCDCTL0** register.
waveforms	selects LCD_E waveform mode. Valid values are: ■ **LCD_E_STANDARD_WAVEFORMS** [Default] ■ **LCD_E_LOW_POWER_WAVEFORMS** Modified bits are **LCDLP** of **LCDCTL0** register.
segments	sets LCD_E segment on/off. Valid values are: ■ **LCD_E_SEGMENTS_DISABLED** [Default] ■ **LCD_E_SEGMENTS_ENABLED** Modified bits are **LCDSON** of **LCDCTL0** register.

18.3 Programming Example

The following example shows how to initialize a 4-mux LCD and display "123456" on the LCD screen.

```
// L0~L26 & L36~L39 pins selected
LCD_E_setPinAsLCDFunctionEx(LCD_E_BASE, LCD_E_SEGMENT_LINE_0,
      LCD_E_SEGMENT_LINE_26);
LCD_E_setPinAsLCDFunctionEx(LCD_E_BASE, LCD_E_SEGMENT_LINE_36,
      LCD_E_SEGMENT_LINE_39);

LCD_E_initParam initParams = {0};
initParams.clockSource = LCD_E_CLOCKSOURCE_XTCLK;
initParams.clockDivider = LCD_E_CLOLKDIVIDER_8;
initParams.muxRate = LCD_E_4_MUX;
initParams.waveforms = LCD_E_STANDARD_WAVEFORMS;
initParams.segments = LCD_E_SEGMENTS_ENABLED;

// Init LCD as 4-mux mode
LCD_E_init(LCD_E_BASE, &initParams);

// LCD Operation - Mode 3, internal 3.08v, charge pump 256Hz
LCD_E_setVLCDSource(LCD_E_BASE, LCD_E_INTERNAL_REFERENCE_VOLTAGE,
      LCD_E_EXTERNAL_SUPPLY_VOLTAGE);
LCD_E_setVLCDVoltage(LCD_E_BASE, LCD_E_REFERENCE_VOLTAGE_3_08V);

LCD_E_enableChargePump(LCD_E_BASE);
LCD_E_setChargePumpFreq(LCD_E_BASE, LCD_E_CHARGEPUMP_FREQ_16);
```

```
// Clear LCD memory
LCD_E_clearAllMemory(LCD_E_BASE);

// Configure COMs and SEGs
// L0, L1, L2, L3: COM pins
// L0 = COM0, L1 = COM1, L2 = COM2, L3 = COM3
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_0, LCD_E_MEMORY_COM0);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_1, LCD_E_MEMORY_COM1);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_2, LCD_E_MEMORY_COM2);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_3, LCD_E_MEMORY_COM3);

// Display "123456"
// LCD Pin8-Pin9 for '1'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_4, 0x60);

// LCD Pin12-Pin13 for '2'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_6, 0xDB);

// LCD Pin16-Pin17 for '3'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_8, 0xF3);

// LCD Pin20-Pin21 for '4'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_10, 0x67);

// LCD Pin4-Pin5 for '5'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_2, 0xB7);

// LCD Pin36-Pin37 for '6'
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_18, 0xBF);

// Turn on LCD
LCD_E_on(LCD_E_BASE);
```

19 Power Management Module (PMM)

19.1 Introduction

The PMM manages all functions related to the power supply and its supervision for the device. Its primary functions are first to generate a supply voltage for the core logic, and second, provide several mechanisms for the supervision of the voltage applied to the device (DVCC).

The PMM uses an integrated low-dropout voltage regulator (LDO) to produce a secondary core voltage (VCORE) from the primary one applied to the device (DVCC). In general, VCORE supplies the CPU, memories, and the digital modules, while DVCC supplies the I/Os and analog modules. The VCORE output is maintained using a dedicated voltage reference. The input or primary side of the regulator is referred to as its high side. The output or secondary side is referred to as its low side.

19.2 API Functions

Functions

- void PMM_enableSVSH (void)
  Enables the high-side SVS circuitry.
- void PMM_disableSVSH (void)
  Disables the high-side SVS circuitry.
- void PMM_turnOnRegulator (void)
  Makes the low-dropout voltage regulator (LDO) remain ON when going into LPM 3/4.
- void PMM_turnOffRegulator (void)
  Turns OFF the low-dropout voltage regulator (LDO) when going into LPM3/4, thus the system will enter LPM3.5 or LPM4.5 respectively.
- void PMM_trigPOR (void)
  Calling this function will trigger a software Power On Reset (POR).
- void PMM_trigBOR (void)
  Calling this function will trigger a software Brown Out Rest (BOR).
- void PMM_clearInterrupt (uint16_t mask)
  Clears interrupt flags for the PMM.
- uint16_t PMM_getInterruptStatus (uint16_t mask)
  Returns interrupt status.
- void PMM_unlockLPM5 (void)
  Unlock LPM5.
- uint16_t PMM_getBandgapMode (void)
  Returns the bandgap mode of the PMM module.
- uint16_t PMM_isBandgapActive (void)
  Returns the active status of the bandgap in the PMM module.
- uint16_t PMM_isRefGenActive (void)
  Returns the active status of the reference generator in the PMM module.

- uint16_t PMM_getBufferedBandgapVoltageStatus (void)
  Returns the active status of the reference generator in the PMM module.
- uint16_t PMM_getVariableReferenceVoltageStatus (void)
  Returns the busy status of the variable reference voltage in the PMM module.
- void PMM_disableTempSensor (void)
  Disables the internal temperature sensor to save power consumption.
- void PMM_enableTempSensor (void)
  Enables the internal temperature sensor.
- void PMM_disableExternalReference (void)
  Disables the external reference output.
- void PMM_enableExternalReference (void)
  Enables the external reference output.
- void PMM_disableInternalReference (void)
  Disables the internal reference output.
- void PMM_enableInternalReference (void)
  Enables the internal reference output.
- void PMM_selectVoltageReference (uint16_t refV)
  Selects reference voltage level.
- void PMM_setPowerMode (uint8_t mode)
  Selects power supply in multi-power supply systems.

19.2.1 Detailed Description

PMM_enableLowPowerReset() / **PMM_disableLowPowerReset()** If enabled, SVSH does not reset device but triggers a system NMI. If disabled, SVSH resets device.

PMM_enableSVSH() / **PMM_disableSVSH()** If disabled on FR58xx/FR59xx, High-side SVS (SVSH) is disabled in LPM2, LPM3, LPM4, LPM3.5 and LPM4.5. SVSH is always enabled in active mode, LPM0, and LPM1. If enabled, SVSH is always enabled. Note: this API has different functionality depending on the part.

PMM_turnOffRegulator() / **PMM_turnOnRegulator()** If off, Regulator is turned off when going to LPM3/4. System enters LPM3.5 or LPM4.5, respectively. If on, Regulator remains on when going into LPM3/4

PMM_clearInterrupt() Clear selected or all interrupt flags for the PMM

PMM_getInterruptStatus() Returns interrupt status of the selected flag in the PMM module

PMM_lockLPM5() / **PMM_unlockLPM5()** If unlocked, LPMx.5 configuration is not locked and defaults to its reset condition. if locked, LPMx.5 configuration remains locked. Pin state is held during LPMx.5 entry and exit.

PMM_getBandgapMode() / **PMM_isBandgapActive()** Return the banggap mode or check its activity.

PMM_isRefGenActive() Check the active status of the reference generator.

PMM_getBufferedBandgapVoltageStatus() / **PMM_getVariableReferenceVoltageStatus()** Check the ready-status for buffered bandgap voltage or variable refernece voltage.

PMM_enableTempSensor() / **PMM_disableTempSensor()** Enable or disable temperature sensor.

PMM_enableExternalReference() / **PMM_disableExternalReference()** Enable or disable external reference.

PMM_enableInternalReference() / **PMM_disableInternalReference()** Enable or disable internal reference.

PMM_selectVoltageReference()

PMM_setPowerMode()

19.2.2 Function Documentation

PMM_clearInterrupt()

```
void PMM_clearInterrupt (
            uint16_t mask )
```

Clears interrupt flags for the PMM.

Parameters

mask	is the mask for specifying the required flag Mask value is the logical OR of any of the following: ■ **PMM_BOR_INTERRUPT** - Software BOR interrupt ■ **PMM_RST_INTERRUPT** - RESET pin interrupt ■ **PMM_POR_INTERRUPT** - Software POR interrupt ■ **PMM_SVSH_INTERRUPT** - SVS high side interrupt ■ **PMM_LPM5_INTERRUPT** - LPM5 indication ■ **PMM_ALL** - All interrupts

Modified bits of **PMMCTL0** register and bits of **PMMIFG** register.

Returns

None

PMM_disableExternalReference()

```
void PMM_disableExternalReference (
            void )
```

Disables the external reference output.

This function is used to disable the external reference output. The external reference is connected to a given external ADC channel. The external reference is disabled by default.

Modified bits are **EXTREFEN** of **PMMCTL2** register.

Returns

None

PMM_disableInternalReference()

```
void PMM_disableInternalReference (
            void  )
```

Disables the internal reference output.

This function is used to disable the internal reference output. The internal reference is internally connected to the ADC channel. The internal reference is disabled by default.

Modified bits are **INTREFEN** of **PMMCTL2** register.

Returns

None

PMM_disableSVSH()

```
void PMM_disableSVSH (
            void  )
```

Disables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

PMM_disableTempSensor()

```
void PMM_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 disabled by default.

Modified bits are **TSENSOREN** of **PMMCTL2** register.

Returns

None

PMM_enableExternalReference()

```
void PMM_enableExternalReference (
            void  )
```

Enables the external reference output.

This function is used to enable the external reference output. The external reference is connected to a given external ADC channel. The external reference is disabled by default.

Modified bits are **EXTREFEN** of **PMMCTL2** register.

Returns

None

PMM_enableInternalReference()

```
void PMM_enableInternalReference (
            void )
```

Enables the internal reference output.

This function is used to enable the internal reference output. The internal reference is internally connected to the ADC channel. The internal reference is disabled by default.

Modified bits are **INTREFEN** of **PMMCTL2** register.

Returns

None

PMM_enableSVSH()

```
void PMM_enableSVSH (
            void )
```

Enables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

PMM_enableTempSensor()

```
void PMM_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 disabled by default.

Modified bits are **TSENSOREN** of **PMMCTL2** register.

Returns

None

PMM_getBandgapMode()

```
uint16_t PMM_getBandgapMode (
            void )
```

Returns the bandgap mode of the PMM module.

This function is used to return the bandgap mode of the PMM 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 PMM module: Return Logical OR of any of the following:

- **PMM_STATICMODE** if the bandgap is operating in static mode
- **PMM_SAMPLEMODE** if the bandgap is operating in sample mode

PMM_getBufferedBandgapVoltageStatus()

```
uint16_t PMM_getBufferedBandgapVoltageStatus (
            void  )
```

Returns the active status of the reference generator in the PMM module.

This function is used to return the ready status of the buffered bandgap voltage in the PMM module. If the buffered bandgap voltage is ready to use, the ready status will be returned.

Returns

The buffered bandgap voltage ready status of the PMM module: Return Logical OR of any of the following:

- **PMM_REFBG_NOTREADY** if buffered bandgap voltage is NOT ready to be used
- **PMM_REFBG_READY** if buffered bandgap voltage ready to be used

PMM_getInterruptStatus()

```
uint16_t PMM_getInterruptStatus (
            uint16_t mask )
```

Returns interrupt status.

Parameters

mask	is the mask for specifying the required flag Mask value is the logical OR of any of the following: ■ **PMM_BOR_INTERRUPT** - Software BOR interrupt ■ **PMM_RST_INTERRUPT** - RESET pin interrupt ■ **PMM_POR_INTERRUPT** - Software POR interrupt ■ **PMM_SVSH_INTERRUPT** - SVS high side interrupt ■ **PMM_LPM5_INTERRUPT** - LPM5 indication ■ **PMM_ALL** - All interrupts

Returns

Logical OR of any of the following:
- **PMM_BOR_INTERRUPT** Software BOR interrupt
- **PMM_RST_INTERRUPT** RESET pin interrupt
- **PMM_POR_INTERRUPT** Software POR interrupt
- **PMM_SVSH_INTERRUPT** SVS high side interrupt
- **PMM_LPM5_INTERRUPT** LPM5 indication
- **PMM_ALL** All interrupts
  indicating the status of the selected interrupt flags

PMM_getVariableReferenceVoltageStatus()

```
uint16_t PMM_getVariableReferenceVoltageStatus (
            void )
```

Returns the busy status of the variable reference voltage in the PMM module.

This function is used to return the ready status of the variable reference voltage in the REFPMM module. If the reference generator is on and ready to use, then the ready status will be returned.

Returns

The variable reference voltage active status of the PMM module: Return Logical OR of any of the following:
- **PMM_REFGEN_NOTREADY** if variable reference voltage is NOT ready to be used
- **PMM_REFGEN_READY** if variable reference voltage ready to be used

PMM_isBandgapActive()

```
uint16_t PMM_isBandgapActive (
            void )
```

Returns the active status of the bandgap in the PMM module.

This function is used to return the active status of the bandgap in the PMM module. If the bandgap is in use by a peripheral, then the status will be seen as active.

Returns

The bandgap active status of the PMM module: Return Logical OR of any of the following:
- **PMM_REFBG_INACTIVE** if the bandgap is not being used at the time of query
- **PMM_REFBG_ACTIVE** if the bandgap is being used at the time of query

PMM_isRefGenActive()

```
uint16_t PMM_isRefGenActive (
            void )
```

Returns the active status of the reference generator in the PMM module.

This function is used to return the active status of the reference generator in the PMM module. If the reference generator is on and ready to use, then the status will be seen as active.

Returns

The reference generator active status of the PMM module: Return Logical OR of any of the following:
- **PMM_REFGEN_INACTIVE** if the reference generator is off and not operating
- **PMM_REFGEN_ACTIVE** if the reference generator is on and ready to be used

PMM_selectVoltageReference()

```
void PMM_selectVoltageReference (
            uint16_t refV )
```

Selects reference voltage level.

This function selects the reference voltage level.

Parameters

refV	is the reference voltage Valid values are: ■ **PMM_REFVSEL_1_5V** [Default] ■ **PMM_REFVSEL_2_0V** ■ **PMM_REFVSEL_2_5V**

Modified bits are **REFVSEL** of **PMMCTL2** register.

Returns

None

PMM_setPowerMode()

```
void PMM_setPowerMode (
            uint8_t mode )
```

Selects power supply in multi-power supply systems.

This function selects power supply in multi power supply systems. A single power supply system is not affected by the bits.

Parameters

mode	is the power mode

Modified bits are **PWRMODE** of **PMMCTL2** register.

Returns

None

PMM_trigBOR()

```
void PMM_trigBOR (
            void )
```

Calling this function will trigger a software Brown Out Rest (BOR).

Modified bits of **PMMCTL0** register.

Returns

None

PMM_trigPOR()

```
void PMM_trigPOR (
            void )
```

Calling this function will trigger a software Power On Reset (POR).

Modified bits of **PMMCTL0** register.

Returns

None

PMM_turnOffRegulator()

```
void PMM_turnOffRegulator (
            void )
```

Turns OFF the low-dropout voltage regulator (LDO) when going into LPM3/4, thus the system will enter LPM3.5 or LPM4.5 respectively.

Modified bits of **PMMCTL0** register.

Returns

None

PMM_turnOnRegulator()

```
void PMM_turnOnRegulator (
            void )
```

Makes the low-dropout voltage regulator (LDO) remain ON when going into LPM 3/4.

Modified bits of **PMMCTL0** register.

Returns

None

PMM_unlockLPM5()

```
void PMM_unlockLPM5 (
            void )
```

Unlock LPM5.

LPMx.5 configuration is not locked and defaults to its reset condition. Disable the GPIO power-on default high-impedance mode to activate previously configured port settings.

Returns

None

19.3 Programming Example

```
  /*
   * Base Address of PMM,
   * By default, the pins are unlocked unless waking
   * up from an LPMx.5 state in which case all GPIO
   * are previously locked.

   */
PMM_unlockLPM5();

if (PMM_getInterruptStatus(PMM_RST_INTERRUPT)) // Was this reset triggered by the
       Reset flag?
{
    PMM_clearInterrupt(PMM_RST_INTERRUPT); // Clear reset flag

    //Trigger a software Brown Out Reset (BOR)
    /*
     * Forces the devices to perform a BOR.
     */
    PMM_trigBOR();                 // Software trigger a BOR.
}

if (PMM_getInterruptStatus(PMM_BOR_INTERRUPT)) // Was this reset triggered by the BOR
       flag?
{
    PMM_clearInterrupt(PMM_BOR_INTERRUPT);     // Clear BOR flag

 //Disable Regulator
/*
 * Regulator is turned off when going to LPM3/4.
 * System enters LPM3.5 or LPM4.5, respectively.
 */
    PMM_turnOffRegulator();
    __bis_SR_register(LPM4_bits); // Enter LPM4.5, This automatically locks
                    // (if not locked already) all GPIO pins.
                    //  and will set the LPM5 flag and set the LOCKLPM5 bit
                    //  in the PM5CTL0 register upon wake up.
}

while (1)
{
  __no_operation();            // Don't sleep
}
```

20 Real-Time Clock (RTC)

20.1 Introduction

The Real Time Clock Counter (RTC) is a 16-bit counter that is functional in active mode(AM) and several low-power modes (LPMs). RTC counter accepts multiple clock sources, which are selected by control register settings to generate timing from less than 1us up to many hours.

The API provides a set of functions for using the RTC modules. Functions are provided to calibrate the clock, initialize the RTC modules in counter mode, enable/disable interrupts for the RTC modules.

The RTC module generates one interrupt in counter mode for counter overflow.

20.2 API Functions

Functions

- void RTC_init (uint16_t baseAddress, uint16_t modulo, uint16_t clockPredivider)
  Initializes the RTC.
- void RTC_start (uint16_t baseAddress, uint16_t clockSource)
  Starts RTC running.
- void RTC_stop (uint16_t baseAddress)
  Stops RTC running.
- void RTC_setModulo (uint16_t baseAddress, uint16_t modulo)
  Sets the modulo value.
- void RTC_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
  Enables selected RTC interrupt sources.
- void RTC_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
  Disables selected RTC interrupt sources.
- uint8_t RTC_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
  Returns the status of the selected interrupts flags.
- void RTC_clearInterrupt (uint16_t baseAddress, int8_t interruptFlagMask)
  Clears selected RTC interrupt flags.

20.2.1 Detailed Description

The RTC API is broken into 2 groups of functions: RTC setup and interrupt functions.

The RTC Calender Mode is initialized and setup by

- RTC_init()
- RTC_start()

- RTC_stop()
- RTC_setModulo()

The RTC interrupts are handled by

- RTC_enableInterrupt()
- RTC_disableInterrupt()
- RTC_getInterruptStatus()
- RTC_clearInterrupt()

20.2.2 Function Documentation

RTC_clearInterrupt()

```
void RTC_clearInterrupt (
            uint16_t baseAddress,
            int8_t interruptFlagMask )
```

Clears selected RTC interrupt flags.

This function clears the RTC interrupt flag is cleared, so that it no longer asserts.

Parameters

baseAddress	is the base address of the RTC module.
interruptFlagMask	is a bit mask of the interrupt flags to clear Valid values are: ■ **RTC_OVERFLOW_INTERRUPT_FLAG** - asserts when counter overflows

Modified bits are **RTCIF** of **RTCCTL** register.

Returns

None

RTC_disableInterrupt()

```
void RTC_disableInterrupt (
            uint16_t baseAddress,
            uint8_t interruptMask )
```

Disables selected RTC interrupt sources.

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.

Parameters

baseAddress	is the base address of the RTC module.

Parameters

interruptMask	is a bit mask of the interrupts to disable. Valid values are: ■ **RTC_OVERFLOW_INTERRUPT** - counter overflow interrupt

Modified bits are **RTCIE** of **RTCCTL** register.

Returns

None

RTC_enableInterrupt()

```
void RTC_enableInterrupt (
            uint16_t baseAddress,
            uint8_t interruptMask )
```

Enables selected RTC interrupt sources.

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. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the RTC module.
interruptMask	is a bit mask of the interrupts to enable. Valid values are: ■ **RTC_OVERFLOW_INTERRUPT** - counter overflow interrupt

Modified bits are **RTCIE** of **RTCCTL** register.

Returns

None

RTC_getInterruptStatus()

```
uint8_t RTC_getInterruptStatus (
            uint16_t baseAddress,
            uint8_t interruptFlagMask )
```

Returns the status of the selected interrupts flags.

This function returns the status of the interrupt flag for the selected channel.

Parameters

baseAddress	is the base address of the RTC module.

Parameters

interruptFlagMask	is a bit mask of the interrupt flags to return the status of. Valid values are: ■ **RTC_OVERFLOW_INTERRUPT_FLAG** - asserts when counter overflows

Returns

A bit mask of the selected interrupt flag's status.

RTC_init()

```
void RTC_init (
            uint16_t baseAddress,
            uint16_t modulo,
            uint16_t clockPredivider )
```

Initializes the RTC.

This function initializes the RTC for clock source and clock pre-divider.

Parameters

baseAddress	is the base address of the RTC module.
modulo	is the modulo value to set to RTC. Modified bits of **RTCMOD** register.
clockPredivider	is the clock pre-divider select for RTC. Valid values are: ■ **RTC_CLOCKPREDIVIDER_1** [Default] ■ **RTC_CLOCKPREDIVIDER_10** ■ **RTC_CLOCKPREDIVIDER_100** ■ **RTC_CLOCKPREDIVIDER_1000** ■ **RTC_CLOCKPREDIVIDER_16** ■ **RTC_CLOCKPREDIVIDER_64** ■ **RTC_CLOCKPREDIVIDER_256** ■ **RTC_CLOCKPREDIVIDER_1024** Modified bits are **RTCPS** of **RTCCTL** register.

Returns

None

RTC_setModulo()

```
void RTC_setModulo (
```

```
                uint16_t baseAddress,
                uint16_t modulo )
```

Sets the modulo value.

This function does software reset for RTC.

Parameters

baseAddress	is the base address of the RTC module.
modulo	is the modulo value to set to RTC. Modified bits of **RTCMOD** register.

Returns

None

RTC_start()

```
void RTC_start (
                uint16_t baseAddress,
                uint16_t clockSource )
```

Starts RTC running.

This function starts the RTC by setting the clock source field (RTCSS). When started, the RTC counter will begin counting at the rate described by the clock source and pre-divider value. When the RTC counter reaches the value in the modulo register, the RTC hardware sets the RTC's interrupt flag bit (RTCIF). Please note, that the RTC actually compares the RTC counter to the modulo shadow register. Since the RTC_start() function sets the RTCSR (RTC software reset) bit, this forces the RTC to copy the value from the Modulo register into the shadow register.

Parameters

baseAddress	is the base address of the RTC module.
clockSource	is the clock source select for RTC. Valid values are: ■ **RTC_CLOCKSOURCE_DISABLED** [Default] ■ **RTC_CLOCKSOURCE_SMCLK** ■ **RTC_CLOCKSOURCE_XT1CLK** ■ **RTC_CLOCKSOURCE_VLOCLK** ■ **RTC_CLOCKSOURCE_ACLK** Modified bits are **RTCSS** of **RTCCTL** register.

Modified bits are **RTCSR** of **RTCCTL** register.

Returns

None

RTC_stop()

```
void RTC_stop (
            uint16_t baseAddress )
```

Stops RTC running.

This function does software reset for RTC.

Parameters

baseAddress	is the base address of the RTC module.

Returns

None

20.3 Programming Example

The following example shows how to initialize and use the RTC API to setup Calender Mode with the current time and various interrupts.

```
//Initialize RTC with modulo value 32768 and pre-divider /1
RTC_init(RTC_BASE,
    32768,
    RTC_CLOCKPREDIVIDER_1);

//Clear interrupt for RTC overflow
RTC_clearInterrupt(RTC_BASE,
    RTC_OVERFLOW_INTERRUPT_FLAG);

//Enable interrupt for RTC overflow
RTC_enableInterrupt(RTC_BASE,
    RTC_OVERFLOW_INTERRUPT);

//Start RTC Clock with clock source XT1
RTC_start(RTC_BASE, RTC_CLOCKSOURCE_XT1CLK);

//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

21 Smart Analog Combo (SAC)

21.1 Introduction

Smart Analog Combo (SAC) integrates a high performance, low-power operational amplifier, up to 33x gain PGA, a 12-bit Digital-to-Analog converter, and a fast Sample and Hold (S&H) circuitry.

The API provides a set of functions for using the SAC operational amplifier. Functions are provided to select positive inputs/negative inputs, select power modes and enable/disable SAC op-amp module.

21.2 API Functions

Functions

- void SAC_OA_init (uint16_t baseAddress, uint16_t positiveInput, uint16_t negativeInput)
  Initializes SAC OA with positive input and negative input. Available to at least SAC-L1.
- void SAC_OA_selectPowerMode (uint16_t baseAddress, uint16_t powerMode)
  Selects power mode for OA. Available to at least SAC-L1.
- void SAC_OA_enable (uint16_t baseAddress)
  Enables OA. Available to at least SAC-L1.
- void SAC_OA_disable (uint16_t baseAddress)
  Disables OA. Available to at least SAC-L1.
- void SAC_enable (uint16_t baseAddress)
  Enables the SAC all modules. Available to at least SAC-L1.
- void SAC_disable (uint16_t baseAddress)
  Disables the SAC all modules. Available to at least SAC-L1.
- void SAC_PGA_setMode (uint16_t baseAddress, uint16_t mode)
  SAC PGA mode selection. Only available to at least SAC-L2.
- void SAC_PGA_setGain (uint16_t baseAddress, uint16_t gain)
  SAC PGA gain configuration. Only available to at least SAC-L2.
- void SAC_DAC_enable (uint16_t baseAddress)
  SAC DAC enable. Only available to at least SAC-L3.
- void SAC_DAC_disable (uint16_t baseAddress)
  SAC DAC disable. Only available to at least SAC-L3.
- void SAC_DAC_interruptEnable (uint16_t baseAddress)
  SAC DAC interrupt enable. Only available to at least SAC-L3.
- void SAC_DAC_interruptDisable (uint16_t baseAddress)
  SAC DAC interrupt disable. Only available to at least SAC-L3.
- void SAC_DAC_DMARequestEnable (uint16_t baseAddress)
  SAC DAC DMA request enable. Only available to at least SAC-L3.
- void SAC_DAC_DMARequestDisable (uint16_t baseAddress)
  SAC DAC DMA request disable. Only available to at least SAC-L3.
- void SAC_DAC_selectLoad (uint16_t baseAddress, uint16_t load)

SAC DAC load select. Only available to at least SAC-L3.

- void SAC_DAC_selectRefVoltage (uint16_t baseAddress, uint16_t reference)
  SAC DAC select reference voltage. Only available to at least SAC-L3.
- uint16_t SAC_DAC_getData (uint16_t baseAddress)
  Get SAC DAC data. Only available to at least SAC-L3.
- void SAC_DAC_setData (uint16_t baseAddress, uint16_t data)
  Set SAC DAC data. Only available to at least SAC-L3.
- bool SAC_DAC_getIFG (uint16_t baseAddress)
  Get SAC DAC data update flag. Only available to at least SAC-L3.
- void SAC_DAC_clearIFG (uint16_t baseAddress)
  Clears SAC DAC data update flag. Only available to at least SAC-L3.
- uint16_t SAC_getInterruptVector (uint16_t baseAddress)
  Get SAC DAC interrupt vector value. Only available to at least SAC-L3.

21.2.1 Detailed Description

- SAC_OA_init()
- SAC_OA_selectPowerMode()
- SAC_OA_enable()
- SAC_OA_disable()
- SAC_enable()
- SAC_disable()
- SAC_PGA_setMode()
- SAC_PGA_setGain()
- SAC_DAC_enable()
- SAC_DAC_disable()
- SAC_DAC_interruptEnable()
- SAC_DAC_interruptDisable()
- SAC_DAC_DMARequestEnable()
- SAC_DAC_DMARequestDisable()
- SAC_DAC_selectLoad()
- SAC_DAC_selectRefVoltage()
- SAC_DAC_getData()
- SAC_DAC_setData()
- SAC_DAC_getIFG()
- SAC_DAC_clearIFG()
- SAC_getInterruptVector()

21.2.2 Function Documentation

SAC_DAC_clearIFG()

```
void SAC_DAC_clearIFG (
            uint16_t baseAddress )
```

Clears SAC DAC data update flag. Only available to at least SAC-L3.

Clears SAC DAC data update flag by writing 1. It could also be cleared by reading SACxIV register. If DMA is enabled, this flag is automatically cleared by DMA when a new data request is accepted. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_disable()

```
void SAC_DAC_disable (
            uint16_t baseAddress )
```

SAC DAC disable. Only available to at least SAC-L3.

Disables SAC DAC. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_DMARequestDisable()

```
void SAC_DAC_DMARequestDisable (
            uint16_t baseAddress )
```

SAC DAC DMA request disable. Only available to at least SAC-L3.

Disables SAC DAC DMA request. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_DMARequestEnable()

```
void SAC_DAC_DMARequestEnable (
            uint16_t baseAddress )
```

SAC DAC DMA request enable. Only available to at least SAC-L3.

Enables SAC DAC DMA request. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_enable()

```
void SAC_DAC_enable (
            uint16_t baseAddress )
```

SAC DAC enable. Only available to at least SAC-L3.

Enables SAC DAC. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_getData()

```
uint16_t SAC_DAC_getData (
            uint16_t baseAddress )
```

Get SAC DAC data. Only available to at least SAC-L3.

Gets from SAC DAC data. Bit 11 represents the MSB. Only word access to SACxDAT register is allowed. Byte operation may cause unexpected results. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

12-bit value from SAC DAC data.

SAC_DAC_getIFG()

```
bool SAC_DAC_getIFG (
            uint16_t baseAddress )
```

Get SAC DAC data update flag. Only available to at least SAC-L3.

Gets flag of SAC DAC update status. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

True or false whether DAC latch data register updated

SAC_DAC_interruptDisable()

```
void SAC_DAC_interruptDisable (
            uint16_t baseAddress )
```

SAC DAC interrupt disable. Only available to at least SAC-L3.

Disables SAC DAC interrupt. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_interruptEnable()

```
void SAC_DAC_interruptEnable (
            uint16_t baseAddress )
```

SAC DAC interrupt enable. Only available to at least SAC-L3.

Enables SAC DAC interrupt. Can be modified only when DACEN = 0. Asynchronously enable the SAC and the SAC DAC interrupt to prevent unexpected results. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_DAC_selectLoad()

```
void SAC_DAC_selectLoad (
            uint16_t baseAddress,
            uint16_t load )
```

SAC DAC load select. Only available to at least SAC-L3.

Selects the load trigger for the DAC latch. DACENC must be set for the DAC to update, except when DACLSEL = 0. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
load	selects DAC load. Valid values are: ■ **SAC_DAC_LOAD_DACDAT_WRITTEN** [Default] ■ **SAC_DAC_LOAD_DEVICE_SPECIFIC_0** ■ **SAC_DAC_LOAD_DEVICE_SPECIFIC_1** Modified bits are **DACLSEL** of **SACxDAC** register.

Returns

None

SAC_DAC_selectRefVoltage()

```
void SAC_DAC_selectRefVoltage (
            uint16_t baseAddress,
            uint16_t reference )
```

SAC DAC select reference voltage. Only available to at least SAC-L3.

Selects SAC DAC select reference voltage, primary or secondary. Can be modified only when DACEN = 0. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
reference	selects DAC reference voltage. Valid values are: ■ **SAC_DAC_PRIMARY_REFERENCE** [Default] ■ **SAC_DAC_SECONDARY_REFERENCE** Modified bits are **DACSREF** of **SACxDAC** register.

Returns

None

SAC_DAC_setData()

```
void SAC_DAC_setData (
            uint16_t baseAddress,
            uint16_t data )
```

Set SAC DAC data. Only available to at least SAC-L3.

Sets data to SAC DAC. Bit 11 represents the MSB. Only word access to SACxDAT register is allowed. Byte operation may cause unexpected results. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
data	sends DAC data. Mask value is the logical OR of any of the following: ■ **SAC_DAC_DATA_BIT0** [Default] ■ **SAC_DAC_DATA_BIT1** ■ **SAC_DAC_DATA_BIT2** ■ **SAC_DAC_DATA_BIT3** ■ **SAC_DAC_DATA_BIT4** ■ **SAC_DAC_DATA_BIT5** ■ **SAC_DAC_DATA_BIT6** ■ **SAC_DAC_DATA_BIT7** ■ **SAC_DAC_DATA_BIT8** ■ **SAC_DAC_DATA_BIT9** ■ **SAC_DAC_DATA_BIT10** ■ **SAC_DAC_DATA_BIT11** Modified bits are **DACDATA** of **SACxDAT** register.

Returns

None

SAC_disable()

```
void SAC_disable (
            uint16_t baseAddress )
```

Disables the SAC all modules. Available to at least SAC-L1.

This will disable SAC all modules. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_enable()

```
void SAC_enable (
            uint16_t baseAddress )
```

Enables the SAC all modules. Available to at least SAC-L1.

This will enable SAC all modules. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_getInterruptVector()

```
uint16_t SAC_getInterruptVector (
            uint16_t baseAddress )
```

Get SAC DAC interrupt vector value. Only available to at least SAC-L3.

Get SAC DAC interrupt vector value. Only word access to the SACIVx register is recommended. Only available to at least SAC-L3. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

SAC DAC interrupt vector value

SAC_OA_disable()

```
void SAC_OA_disable (
            uint16_t baseAddress )
```

Disables OA. Available to at least SAC-L1.

This will disable OA and OA outputs high impedance. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_OA_enable()

```
void SAC_OA_enable (
            uint16_t baseAddress )
```

Enables OA. Available to at least SAC-L1.

This will enables OA for normal mode. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.

Returns

None

SAC_OA_init()

```
void SAC_OA_init (
            uint16_t baseAddress,
            uint16_t positiveInput,
```

```
            uint16_t negativeInput )
```

Initializes SAC OA with positive input and negative input. Available to at least SAC-L1.

This function initializes SAC OA with positive input and negative input. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
positiveInput	selects the positive input source Valid values are: ■ **SAC_OA_POSITIVE_INPUT_SOURCE_EXTERNAL** [Default] ■ **SAC_OA_POSITIVE_INPUT_SOURCE_DAC** ■ **SAC_OA_POSITIVE_INPUT_SOURCE_PAIR_OA** ■ **SAC_OA_POSITIVE_INPUT_SOURCE_DISCONNECTED** Modified bits are **PSEL** and **PMUXEN** of **SACxOA** register.
negativeInput	selects the negative input source Valid values are: ■ **SAC_OA_NEGATIVE_INPUT_SOURCE_EXTERNAL** [Default] ■ **SAC_OA_NEGATIVE_INPUT_SOURCE_PGA** ■ **SAC_OA_NEGATIVE_INPUT_SOURCE_DISCONNECTED** Modified bits are **NSEL** and **NMUXEN** of **SACxOA** register.

Returns

None

SAC_OA_selectPowerMode()

```
void SAC_OA_selectPowerMode (
            uint16_t baseAddress,
            uint16_t powerMode )
```

Selects power mode for OA. Available to at least SAC-L1.

This function selects power mode for OA. Available to at least SAC-L1. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
powerMode	selects OA power mode. Valid values are: ■ **SAC_OA_POWER_MODE_HIGH_SPEED_HIGH_POWER** [Default] ■ **SAC_OA_POWER_MODE_LOW_SPEED_LOW_POWER** Modified bits are **OAPM** of **SACxOA** register.

Returns

None

SAC_PGA_setGain()

```
void SAC_PGA_setGain (
            uint16_t baseAddress,
            uint16_t gain )
```

SAC PGA gain configuration. Only available to at least SAC-L2.

Allows different SAC PGA gain configurations. Only available to at least SAC-L2. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
gain	selects PGA gain configuration. Mask value is the logical OR of any of the following: ■ **SAC_PGA_GAIN_BIT0** [Default] ■ **SAC_PGA_GAIN_BIT1** ■ **SAC_PGA_GAIN_BIT2** Modified bits are **GAIN** of **SACxPGA** register.

Returns

None

SAC_PGA_setMode()

```
void SAC_PGA_setMode (
            uint16_t baseAddress,
            uint16_t mode )
```

SAC PGA mode selection. Only available to at least SAC-L2.

Allows selection of different SAC PGA modes. Only available to at least SAC-L2. Please consult your device-specific datasheet to see what SAC level is available.

Parameters

baseAddress	is the base address of the SAC module.
mode	selects PGA mode. Valid values are: ■ **SAC_PGA_MODE_INVERTING** ■ **SAC_PGA_MODE_BUFFER** [Default] ■ **SAC_PGA_MODE_NONINVERTING** ■ **SAC_PGA_MODE_CASCADE_OA_INVERTING** Modified bits are **MSEL** of **SACxPGA** register.

Returns

None

21.3 Programming Example

The following example shows how to initialize SAC inputs and configure with low speed low power mode.

```
//Select external source for both positive and negative inputs
SAC_OA_init(SAC0_BASE, SAC_OA_POSITIVE_INPUT_SOURCE_EXTERNAL,
  SAC_OA_NEGATIVE_INPUT_SOURCE_EXTERNAL);

//Select low speed and low power mode
SAC_OA_selectPowerMode(SAC0_BASE, SAC_OA_POWER_MODE_LOW_SPEED_LOW_POWER);
SAC_OA_enable(SAC0_BASE);                  // Enable OA
SAC_enable(SAC0_BASE);                     // Enable SAC

//Enter LPM3 mode
__bis_SR_register(LPM3_bits);
__no_operation();
```

22 SFR Module

22.1 Introduction

The Special Function Registers API provides a set of functions for using the MSP430Ware SFR module. Functions are provided to enable and disable interrupts and control the ∼RST/NMI pin

The SFR module can enable interrupts to be generated from other peripherals of the device.

22.2 API Functions

Functions

- void SFR_enableInterrupt (uint8_t interruptMask)
  Enables selected SFR interrupt sources.
- void SFR_disableInterrupt (uint8_t interruptMask)
  Disables selected SFR interrupt sources.
- uint8_t SFR_getInterruptStatus (uint8_t interruptFlagMask)
  Returns the status of the selected SFR interrupt flags.
- void SFR_clearInterrupt (uint8_t interruptFlagMask)
  Clears the selected SFR interrupt flags.
- void SFR_setResetPinPullResistor (uint16_t pullResistorSetup)
  Sets the pull-up/down resistor on the ∼RST/NMI pin.
- void SFR_setNMIEdge (uint16_t edgeDirection)
  Sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if NMI function is active.
- void SFR_setResetNMIPinFunction (uint8_t resetPinFunction)
  Sets the function of the ∼RST/NMI pin.

22.2.1 Detailed Description

The SFR API is broken into 2 groups: the SFR interrupts and the SFR ∼RST/NMI pin control

The SFR interrupts are handled by

- SFR_enableInterrupt()
- SFR_disableInterrupt()
- SFR_getInterruptStatus()
- SFR_clearInterrupt()

The SFR ∼RST/NMI pin is controlled by

- SFR_setResetPinPullResistor()
- SFR_setNMIEdge()
- SFR_setResetNMIPinFunction()

22.2.2 Function Documentation

SFR_clearInterrupt()

```
void SFR_clearInterrupt (
            uint8_t interruptFlagMask )
```

Clears the selected SFR interrupt flags.

This function clears the status of the selected SFR interrupt flags.

Parameters

interruptFlagMask	is the bit mask of interrupt flags that will be cleared. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

SFR_disableInterrupt()

```
void SFR_disableInterrupt (
            uint8_t interruptMask )
```

Disables selected SFR interrupt sources.

This function disables the selected SFR interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Parameters

interruptMask	is the bit mask of interrupts that will be disabled. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

SFR_enableInterrupt()

```
void SFR_enableInterrupt (
            uint8_t interruptMask )
```

Enables selected SFR interrupt sources.

This function enables the selected SFR 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

interruptMask	is the bit mask of interrupts that will be enabled. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

SFR_getInterruptStatus()

```
uint8_t SFR_getInterruptStatus (
            uint8_t interruptFlagMask )
```

Returns the status of the selected SFR interrupt flags.

This function returns the status of the selected SFR interrupt flags in a bit mask format matching that passed into the interruptFlagMask parameter.

Parameters

interruptFlagMask	is the bit mask of interrupt flags that the status of should be returned. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

A bit mask of the status of the selected interrupt flags. Return Logical OR of any of the following:

- **SFR_JTAG_OUTBOX_INTERRUPT** JTAG outbox interrupt
- **SFR_JTAG_INBOX_INTERRUPT** JTAG inbox interrupt
- **SFR_NMI_PIN_INTERRUPT** NMI pin interrupt, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** Vacant memory access interrupt
- **SFR_OSCILLATOR_FAULT_INTERRUPT** Oscillator fault interrupt
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** Watchdog interval timer interrupt indicating the status of the masked interrupts

SFR_setNMIEdge()

```
void SFR_setNMIEdge (
            uint16_t edgeDirection )
```

Sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if NMI function is active.

This function sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if the NMI function is active. To activate the NMI function of the ∼RST/NMI use the SFR_setResetNMIPinFunction() passing SFR_RESETPINFUNC_NMI into the resetPinFunction parameter.

Parameters

edgeDirection	is the direction that the signal on the ∼RST/NMI pin should go to signal an interrupt, if enabled. Valid values are: ■ **SFR_NMI_RISINGEDGE** [Default] ■ **SFR_NMI_FALLINGEDGE** Modified bits are **SYSNMIIES** of **SFRRPCR** register.

Returns

None

SFR_setResetNMIPinFunction()

```
void SFR_setResetNMIPinFunction (
            uint8_t resetPinFunction )
```

Sets the function of the ∼RST/NMI pin.

This function sets the functionality of the ∼RST/NMI pin, whether in reset mode which will assert a reset if a low signal is observed on that pin, or an NMI which will assert an interrupt from an edge of the signal dependent on the setting of the edgeDirection parameter in SFR_setNMIEdge().

Parameters

resetPinFunction	is the function that the ∼RST/NMI pin should take on. Valid values are: ■ **SFR_RESETPINFUNC_RESET** [Default] ■ **SFR_RESETPINFUNC_NMI** Modified bits are **SYSNMI** of **SFRRPCR** register.

Returns

None

SFR_setResetPinPullResistor()

```
void SFR_setResetPinPullResistor (
            uint16_t pullResistorSetup )
```

Sets the pull-up/down resistor on the ∼RST/NMI pin.

This function sets the pull-up/down resistors on the ∼RST/NMI pin to the settings from the pullResistorSetup parameter.

Parameters

pullResistorSetup	is the selection of how the pull-up/down resistor on the ∼RST/NMI pin should be setup or disabled. Valid values are: ■ **SFR_RESISTORDISABLE** ■ **SFR_RESISTORENABLE_PULLUP** [Default] ■ **SFR_RESISTORENABLE_PULLDOWN** Modified bits are **SYSRSTUP** and **SYSRSTRE** of **SFRRPCR** register.

Returns

None

22.3 Programming Example

The following example shows how to initialize and use the SFR API

```
do
  {

    // Clear SFR Fault Flag
    SFR_clearInterrupt(SFR_BASE,
                       OFIFG);

    // Test oscillator fault flag
  }while (SFR_getInterruptStatus(SFR_BASE,OFIFG));
```

23 System Control Module

23.1 Introduction

The System Control (SYS) API provides a set of functions for using the MSP430Ware SYS module. Functions are provided to control various SYS controls, setup the BSL, control the JTAG Mailbox, control the protection bits for FRAM data/program write and configure the infrared data.

23.2 API Functions

Functions

- void SysCtl_enableDedicatedJTAGPins (void)
  Sets the JTAG pins to be exclusively for JTAG until a BOR occurs.
- uint8_t SysCtl_getBSLEntryIndication (void)
  Returns the indication of a BSL entry sequence from the Spy-Bi-Wire.
- void SysCtl_enablePMMAccessProtect (void)
  Enables PMM Access Protection.
- void SysCtl_enableRAMBasedInterruptVectors (void)
  Enables RAM-based Interrupt Vectors.
- void SysCtl_disableRAMBasedInterruptVectors (void)
  Disables RAM-based Interrupt Vectors.
- void SysCtl_enableBSLProtect (void)
  Enables BSL memory protection.
- void SysCtl_disableBSLProtect (void)
  Disables BSL memory protection.
- void SysCtl_enableBSLMemory (void)
  Enables BSL memory.
- void SysCtl_disableBSLMemory (void)
  Disables BSL memory.
- void SysCtl_setRAMAssignedToBSL (uint8_t BSLRAMAssignment)
  Sets RAM assignment to BSL area.
- void SysCtl_initJTAGMailbox (uint8_t mailboxSizeSelect, uint8_t autoClearInboxFlagSelect)
  Initializes JTAG Mailbox with selected properties.
- uint8_t SysCtl_getJTAGMailboxFlagStatus (uint8_t mailboxFlagMask)
  Returns the status of the selected JTAG Mailbox flags.
- void SysCtl_clearJTAGMailboxFlagStatus (uint8_t mailboxFlagMask)
  Clears the status of the selected JTAG Mailbox flags.
- uint16_t SysCtl_getJTAGInboxMessage16Bit (uint8_t inboxSelect)
  Returns the contents of the selected JTAG Inbox in a 16 bit format.
- uint32_t SysCtl_getJTAGInboxMessage32Bit (void)
  Returns the contents of JTAG Inboxes in a 32 bit format.

- void SysCtl_setJTAGOutgoingMessage16Bit (uint8_t outboxSelect, uint16_t outgoingMessage)
  Sets a 16 bit outgoing message in to the selected JTAG Outbox.
- void SysCtl_setJTAGOutgoingMessage32Bit (uint32_t outgoingMessage)
  Sets a 32 bit message in to both JTAG Outboxes.
- void SysCtl_protectFRAMWrite (uint8_t writeProtect)
  Sets write protected for data FRAM and program FRAM.
- void SysCtl_enableFRAMWrite (uint8_t writeEnable)
  Sets write enable for data FRAM and program FRAM.
- void SysCtl_setInfraredConfig (uint8_t dataSource, uint8_t mode, uint8_t polarity)
  Sets infrared configuration bits.
- void SysCtl_enableInfrared (void)
  Enables infrared function.
- void SysCtl_disableInfrared (void)
  Disables infrared function.
- uint8_t SysCtl_getInfraredData (void)
  This function returns the infrared data if the infrared data source is configured as from IRDATA bit.
- void SysCtl_setFRWPOA (uint8_t offsetAddress)
  This function sets the Program FRAM write protection offset address from the beginning of Program FRAM. The offset increases by 1 kB resolution.

23.2.1 Detailed Description

The SYS API is broken into 5 groups: the various SYS controls, the BSL controls, the JTAG mailbox controls, the FRAM write protection controls and infrared data configuration.

The various SYS controls are handled by

- SysCtl_enableDedicatedJTAGPins()
- SysCtl_getBSLEntryIndication()
- SysCtl_enablePMMAccessProtect()
- SysCtl_enableRAMBasedInterruptVectors()
- SysCtl_disableRAMBasedInterruptVectors()

The BSL controls are handled by

- SysCtl_enableBSLProtect()
- SysCtl_disableBSLProtect()
- SysCtl_disableBSLMemory()
- SysCtl_enableBSLMemory()
- SysCtl_setRAMAssignedToBSL()

The JTAG Mailbox controls are handled by

- SysCtl_initJTAGMailbox()
- SysCtl_getJTAGMailboxFlagStatus()
- SysCtl_getJTAGInboxMessage16Bit()
- SysCtl_getJTAGInboxMessage32Bit()
- SysCtl_setJTAGOutgoingMessage16Bit()

- SysCtl_setJTAGOutgoingMessage32Bit()
- SysCtl_clearJTAGMailboxFlagStatus()

The FRAM write protection controls are handled by

- SysCtl_protectFRAMWrite()
- SysCtl_enableFRAMWrite()
- SysCtl_setFRWPOA()

The infrared data configuration are handled by

- SysCtl_setInfraredConfig()
- SysCtl_enableInfrared()
- SysCtl_disableInfrared()
- SysCtl_getInfraredData()

23.2.2 Function Documentation

SysCtl_clearJTAGMailboxFlagStatus()

```
void SysCtl_clearJTAGMailboxFlagStatus (
            uint8_t mailboxFlagMask )
```

Clears the status of the selected JTAG Mailbox flags.

This function clears the selected JTAG Mailbox flags.

Parameters

mailboxFlagMask	is the bit mask of JTAG mailbox flags that the status of should be cleared. Mask value is the logical OR of any of the following: - **SYSCTL_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0 - **SYSCTL_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1 - **SYSCTL_JTAGINBOX_FLAG0** - flag for JTAG inbox 0 - **SYSCTL_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns

None

SysCtl_disableBSLMemory()

```
void SysCtl_disableBSLMemory (
            void )
```

Disables BSL memory.

This function disables BSL memory, which makes BSL memory act like vacant memory.

Returns

None

SysCtl_disableBSLProtect()

```
void SysCtl_disableBSLProtect (
            void )
```

Disables BSL memory protection.

This function disables protection on the BSL memory.

Returns

None

SysCtl_disableInfrared()

```
void SysCtl_disableInfrared (
            void )
```

Disables infrared function.

Returns

None

SysCtl_disableRAMBasedInterruptVectors()

```
void SysCtl_disableRAMBasedInterruptVectors (
            void )
```

Disables RAM-based Interrupt Vectors.

This function disables the interrupt vectors from being generated at the top of the RAM.

Returns

None

SysCtl_enableBSLMemory()

```
void SysCtl_enableBSLMemory (
            void )
```

Enables BSL memory.

This function enables BSL memory, which allows BSL memory to be addressed

Returns

None

SysCtl_enableBSLProtect()

```
void SysCtl_enableBSLProtect (
            void )
```

Enables BSL memory protection.

This function enables protection on the BSL memory, which prevents any reading, programming, or erasing of the BSL memory.

Returns

None

SysCtl_enableDedicatedJTAGPins()

```
void SysCtl_enableDedicatedJTAGPins (
            void )
```

Sets the JTAG pins to be exclusively for JTAG until a BOR occurs.

This function sets the JTAG pins to be exclusively used for the JTAG, and not to be shared with the GPIO pins. This setting can only be cleared when a BOR occurs.

Returns

None

SysCtl_enableFRAMWrite()

```
void SysCtl_enableFRAMWrite (
            uint8_t writeEnable )
```

Sets write enable for data FRAM and program FRAM.

Parameters

writeEnable	is the value setting data FRAM and program write enabled. Mask value is the logical OR of any of the following: ■ **SYSCTL_FRAMWRITEPROTECTION_DATA** - data FRAM write protected ■ **SYSCTL_FRAMWRITEPROTECTION_PROGRAM** - program FRAM write protected

Returns

None

SysCtl_enableInfrared()

```
void SysCtl_enableInfrared (
```

```
void )
```

Enables infrared function.

Returns

None

SysCtl_enablePMMAccessProtect()

```
void SysCtl_enablePMMAccessProtect (
          void )
```

Enables PMM Access Protection.

This function enables the PMM Access Protection, which will lock any changes on the PMM control registers until a BOR occurs.

Returns

None

SysCtl_enableRAMBasedInterruptVectors()

```
void SysCtl_enableRAMBasedInterruptVectors (
          void )
```

Enables RAM-based Interrupt Vectors.

This function enables RAM-base Interrupt Vectors, which means that interrupt vectors are generated with the end address at the top of RAM, instead of the top of the lower 64kB of flash.

Returns

None

SysCtl_getBSLEntryIndication()

```
uint8_t SysCtl_getBSLEntryIndication (
          void )
```

Returns the indication of a BSL entry sequence from the Spy-Bi-Wire.

This function returns the indication of a BSL entry sequence from the Spy- Bi-Wire.

Returns

One of the following:
- **SYSCTL_BSLENTRY_INDICATED**
- **SYSCTL_BSLENTRY_NOTINDICATED**
  indicating if a BSL entry sequence was detected

SysCtl_getInfraredData()

```
uint8_t SysCtl_getInfraredData (
            void )
```

This function returns the infrared data if the infrared data source is configured as from IRDATA bit.

Returns

the infrared logic data '0' or '1'

SysCtl_getJTAGInboxMessage16Bit()

```
uint16_t SysCtl_getJTAGInboxMessage16Bit (
            uint8_t inboxSelect )
```

Returns the contents of the selected JTAG Inbox in a 16 bit format.

This function returns the message contents of the selected JTAG inbox. If the auto clear settings for the Inbox flags were set, then using this function will automatically clear the corresponding JTAG inbox flag.

Parameters

inboxSelect	is the chosen JTAG inbox that the contents of should be returned Valid values are: ■ **SYSCTL_JTAGINBOX_0** - return contents of JTAG inbox 0 ■ **SYSCTL_JTAGINBOX_1** - return contents of JTAG inbox 1

Returns

The contents of the selected JTAG inbox in a 16 bit format.

SysCtl_getJTAGInboxMessage32Bit()

```
uint32_t SysCtl_getJTAGInboxMessage32Bit (
            void )
```

Returns the contents of JTAG Inboxes in a 32 bit format.

This function returns the message contents of both JTAG inboxes in a 32 bit format. This function should be used if 32-bit messaging has been set in the SYS_initJTAGMailbox() function. If the auto clear settings for the Inbox flags were set, then using this function will automatically clear both JTAG inbox flags.

Returns

The contents of both JTAG messages in a 32 bit format.

SysCtl_getJTAGMailboxFlagStatus()

```
uint8_t SysCtl_getJTAGMailboxFlagStatus (
            uint8_t mailboxFlagMask )
```

Returns the status of the selected JTAG Mailbox flags.

This function will return the status of the selected JTAG Mailbox flags in bit mask format matching that passed into the mailboxFlagMask parameter.

Parameters

mailboxFlagMask	is the bit mask of JTAG mailbox flags that the status of should be returned. Mask value is the logical OR of any of the following: ■ **SYSCTL_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0 ■ **SYSCTL_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1 ■ **SYSCTL_JTAGINBOX_FLAG0** - flag for JTAG inbox 0 ■ **SYSCTL_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns

A bit mask of the status of the selected mailbox flags.

SysCtl_initJTAGMailbox()

```
void SysCtl_initJTAGMailbox (
            uint8_t mailboxSizeSelect,
            uint8_t autoClearInboxFlagSelect )
```

Initializes JTAG Mailbox with selected properties.

This function sets the specified settings for the JTAG Mailbox system. The settings that can be set are the size of the JTAG messages, and the auto- clearing of the inbox flags. If the inbox flags are set to auto-clear, then the inbox flags will be cleared upon reading of the inbox message buffer, otherwise they will have to be reset by software using the SYS_clearJTAGMailboxFlagStatus() function.

Parameters

mailboxSizeSelect	is the size of the JTAG Mailboxes, whether 16- or 32-bits. Valid values are: ■ **SYSCTL_JTAGMBSIZE_16BIT** [Default] - the JTAG messages will take up only one JTAG mailbox (i. e. an outgoing message will take up only 1 outbox of the JTAG mailboxes) ■ **SYSCTL_JTAGMBSIZE_32BIT** - the JTAG messages will be contained within both JTAG mailboxes (i. e. an outgoing message will take up both Outboxes of the JTAG mailboxes) Modified bits are **JMBMODE** of **SYSJMBC** register.

Parameters

autoClearInboxFlagSelect	decides how the JTAG inbox flags should be cleared, whether automatically after the corresponding outbox has been written to, or manually by software. Valid values are: ■ **SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1AUTO** [Default] - both JTAG inbox flags will be reset automatically when the corresponding inbox is read from. ■ **SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1SW** - only JTAG inbox 0 flag is reset automatically, while JTAG inbox 1 is reset with the ■ **SYSCTL_JTAGINBOX0SW_JTAGINBOX1AUTO** - only JTAG inbox 1 flag is reset automatically, while JTAG inbox 0 is reset with the ■ **SYSCTL_JTAGINBOX0SW_JTAGINBOX1SW** - both JTAG inbox flags will need to be reset manually by the Modified bits are **JMBCLR0OFF** and **JMBCLR1OFF** of **SYSJMBC** register.

Returns

None

SysCtl_protectFRAMWrite()

```
void SysCtl_protectFRAMWrite (
            uint8_t writeProtect )
```

Sets write protected for data FRAM and program FRAM.

Parameters

writeProtect	is the value setting data FRAM and program write protection. Mask value is the logical OR of any of the following: ■ **SYSCTL_FRAMWRITEPROTECTION_DATA** - data FRAM write protected ■ **SYSCTL_FRAMWRITEPROTECTION_PROGRAM** - program FRAM write protected

Returns

None

SysCtl_setFRWPOA()

```
void SysCtl_setFRWPOA (
            uint8_t offsetAddress )
```

This function sets the Program FRAM write protection offset address from the beginning of Program FRAM. The offset increases by 1 kB resolution.

Parameters

offsetAddress	is the Program FRAM write protection offset address from the beginning of Program FRAM, with offset increases of 1KB resolution. Mask value is the logical OR of any of the following: ■ **SYSCTL_FRWPOA0** ■ **SYSCTL_FRWPOA1** ■ **SYSCTL_FRWPOA2** ■ **SYSCTL_FRWPOA3** ■ **SYSCTL_FRWPOA4** ■ **SYSCTL_FRWPOA5**

Returns

None

SysCtl_setInfraredConfig()

```
void SysCtl_setInfraredConfig (
            uint8_t dataSource,
            uint8_t mode,
            uint8_t polarity )
```

Sets infrared configuration bits.

Parameters

dataSource	is the value setting infrared data source. Valid values are: ■ **SYSCTL_INFRAREDDATASOURCE_CONFIG** - infrared data from hardware peripherals upon device configuration ■ **SYSCTL_INFRAREDDATASOURCE_IRDATA** - infrared data from IRDATA bit
mode	is the value setting infrared mode. Valid values are: ■ **SYSCTL_INFRAREDMODE_ASK** - infrared ASK mode ■ **SYSCTL_INFRAREDMODE_FSK** - infrared FSK mode
polarity	is the value setting infrared polarity. Valid values are: ■ **SYSCTL_INFRAREDPOLARITY_NORMAL** - infrared normal polarity ■ **SYSCTL_INFRAREDPOLARITY_INVERTED** - infrared inverted polarity

Returns

None

SysCtl_setJTAGOutgoingMessage16Bit()

```
void SysCtl_setJTAGOutgoingMessage16Bit (
            uint8_t outboxSelect,
            uint16_t outgoingMessage )
```

Sets a 16 bit outgoing message in to the selected JTAG Outbox.

This function sets the outgoing message in the selected JTAG outbox. The corresponding JTAG outbox flag is cleared after this function, and set after the JTAG has read the message.

Parameters

outboxSelect	is the chosen JTAG outbox that the message should be set it. Valid values are: ■ **SYSCTL_JTAGOUTBOX_0** - set the contents of JTAG outbox 0 ■ **SYSCTL_JTAGOUTBOX_1** - set the contents of JTAG outbox 1
outgoingMessage	is the message to send to the JTAG. Modified bits are **MSGHI** and **MSGLO** of **SYSJMBOx** register.

Returns

None

SysCtl_setJTAGOutgoingMessage32Bit()

```
void SysCtl_setJTAGOutgoingMessage32Bit (
            uint32_t outgoingMessage )
```

Sets a 32 bit message in to both JTAG Outboxes.

This function sets the 32-bit outgoing message in both JTAG outboxes. The JTAG outbox flags are cleared after this function, and set after the JTAG has read the message.

Parameters

outgoingMessage	is the message to send to the JTAG. Modified bits are **MSGHI** and **MSGLO** of **SYSJMBOx** register.

Returns

None

SysCtl_setRAMAssignedToBSL()

```
void SysCtl_setRAMAssignedToBSL (
            uint8_t BSLRAMAssignment )
```

Sets RAM assignment to BSL area.

This function allows RAM to be assigned to BSL, based on the selection of the BSLRAMAssignment parameter.

Parameters

BSLRAMAssignment	is the selection of if the BSL should be placed in RAM or not. Valid values are: ■ **SYSCTL_BSLRAMASSIGN_NORAM** [Default] ■ **SYSCTL_BSLRAMASSIGN_LOWEST16BYTES** Modified bits are **SYSBSLR** of **SYSBSLC** register.

Returns

None

23.3 Programming Example

The following example shows how to initialize and use the SYS API

```
SysCtl_enableBSLProtect();
```

24 16-Bit Timer_A (TIMER_A)

24.1 Introduction

TIMER_A is a 16-bit timer/counter with multiple capture/compare registers. TIMER_A can support multiple capture/compares, PWM outputs, and interval timing. TIMER_A 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.

TIMER_A 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

TIMER_A can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

TIMER_A Interrupts may be generated on counter overflow conditions and during capture compare events.

The TIMER_A may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TIMER_A_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 Timer_A_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 Timer_A_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The TIMER_A API provides a set of functions for dealing with the TIMER_A 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.

24.2 API Functions

Functions

- void Timer_A_startCounter (uint16_t baseAddress, uint16_t timerMode)
  Starts Timer_A counter.
- void Timer_A_initContinuousMode (uint16_t baseAddress, Timer_A_initContinuousModeParam *param)
  Configures Timer_A in continuous mode.
- void Timer_A_initUpMode (uint16_t baseAddress, Timer_A_initUpModeParam *param)
  Configures Timer_A in up mode.
- void Timer_A_initUpDownMode (uint16_t baseAddress, Timer_A_initUpDownModeParam *param)
  Configures Timer_A in up down mode.
- void Timer_A_initCaptureMode (uint16_t baseAddress, Timer_A_initCaptureModeParam *param)
  Initializes Capture Mode.
- void Timer_A_initCompareMode (uint16_t baseAddress, Timer_A_initCompareModeParam *param)
  Initializes Compare Mode.
- void Timer_A_enableInterrupt (uint16_t baseAddress)
  Enable timer interrupt.
- void Timer_A_disableInterrupt (uint16_t baseAddress)
  Disable timer interrupt.
- uint32_t Timer_A_getInterruptStatus (uint16_t baseAddress)
  Get timer interrupt status.
- void Timer_A_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Enable capture compare interrupt.
- void Timer_A_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Disable capture compare interrupt.
- uint32_t Timer_A_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
  Return capture compare interrupt status.
- void Timer_A_clear (uint16_t baseAddress)
  Reset/Clear the timer clock divider, count direction, count.
- uint8_t Timer_A_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
  Get synchronized capturecompare input.
- uint8_t Timer_A_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get output bit for output mode.
- uint16_t Timer_A_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get current capturecompare count.
- void Timer_A_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
  Set output bit for output mode.
- void Timer_A_outputPWM (uint16_t baseAddress, Timer_A_outputPWMParam *param)
  Generate a PWM with timer running in up mode.
- void Timer_A_stop (uint16_t baseAddress)

Stops the timer.
- void Timer_A_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
  Sets the value of the capture-compare register.
- void Timer_A_setOutputMode (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareOutputMode)
  Sets the output mode.
- void Timer_A_clearTimerInterrupt (uint16_t baseAddress)
  Clears the Timer TAIFG interrupt flag.
- void Timer_A_clearCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Clears the capture-compare interrupt flag.
- uint16_t Timer_A_getCounterValue (uint16_t baseAddress)
  Reads the current timer count value.

24.2.1 Detailed Description

The TIMER_A API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TIMER_A configuration and initialization is handled by

- Timer_A_startCounter()
- Timer_A_initUpMode()
- Timer_A_initUpDownMode()
- Timer_A_initContinuousMode()
- Timer_A_initCaptureMode()
- Timer_A_initCompareMode()
- Timer_A_clear()
- Timer_A_stop()

TIMER_A outputs are handled by

- Timer_A_getSynchronizedCaptureCompareInput()
- Timer_A_getOutputForOutputModeOutBitValue()
- Timer_A_setOutputForOutputModeOutBitValue()
- Timer_A_outputPWM()
- Timer_A_getCaptureCompareCount()
- Timer_A_setCompareValue()
- Timer_A_getCounterValue()

The interrupt handler for the TIMER_A interrupt is managed with

- Timer_A_enableInterrupt()
- Timer_A_disableInterrupt()
- Timer_A_getInterruptStatus()
- Timer_A_enableCaptureCompareInterrupt()

- Timer_A_disableCaptureCompareInterrupt()
- Timer_A_getCaptureCompareInterruptStatus()
- Timer_A_clearCaptureCompareInterrupt()
- Timer_A_clearTimerInterrupt()

24.2.2 Function Documentation

Timer_A_clear()

```
void Timer_A_clear (
            uint16_t baseAddress )
```

Reset/Clear the timer clock divider, count direction, count.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

References Timer_A_getSynchronizedCaptureCompareInput().

Timer_A_clearCaptureCompareInterrupt()

```
void Timer_A_clearCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Clears the capture-compare interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_A module.
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**

Modified bits are **CCIFG** of **TAxCCTLn** register.

Returns

None

Timer_A_clearTimerInterrupt()

```
void Timer_A_clearTimerInterrupt (
            uint16_t baseAddress )
```

Clears the Timer TAIFG interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits are **TAIFG** of **TAxCTL** register.

Returns

None

Timer_A_disableCaptureCompareInterrupt()

```
void Timer_A_disableCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Disable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_A module.
captureCompareRegister	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Modified bits of **TAxCCTLn** register.

Returns

None

Timer_A_disableInterrupt()

```
void Timer_A_disableInterrupt (
```

```
uint16_t baseAddress )
```

Disable timer interrupt.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

Timer_A_enableCaptureCompareInterrupt()

```
void Timer_A_enableCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Enable capture compare interrupt.

Does not clear interrupt flags

Parameters

baseAddress	is the base address of the TIMER_A module.
captureCompareRegister	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Modified bits of **TAxCCTLn** register.

Returns

None

Timer_A_enableInterrupt()

```
void Timer_A_enableInterrupt (
            uint16_t baseAddress )
```

Enable timer interrupt.

Does not clear interrupt flags

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

Timer_A_getCaptureCompareCount()

```
uint16_t Timer_A_getCaptureCompareCount (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Get current capturecompare count.

Parameters

baseAddress	is the base address of the TIMER_A module.
captureCompareRegister	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER←_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER←_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER←_2**

Returns

Current count as an uint16_t

References Timer_A_setOutputForOutputModeOutBitValue().

Referenced by Timer_A_getOutputForOutputModeOutBitValue().

Timer_A_getCaptureCompareInterruptStatus()

```
uint32_t Timer_A_getCaptureCompareInterruptStatus (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint16_t mask )
```

Return capture compare interrupt status.

Parameters

baseAddress	is the base address of the TIMER_A module.

Parameters

captureCompareRegister	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
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

Logical OR of any of the following:
- **TIMER_A_CAPTURE_OVERFLOW**
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG**
  indicating the status of the masked interrupts

Timer_A_getCounterValue()

```
uint16_t Timer_A_getCounterValue (
            uint16_t baseAddress )
```

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The TIMER_A_THRESHOLD #define in the corresponding header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

baseAddress	is the base address of the TIMER_A module.

Returns

Majority vote of timer count value

Timer_A_getInterruptStatus()

```
uint32_t Timer_A_getInterruptStatus (
            uint16_t baseAddress )
```

Get timer interrupt status.

Parameters

baseAddress	is the base address of the TIMER_A module.

Returns

One of the following:
- **TIMER_A_INTERRUPT_NOT_PENDING**
- **TIMER_A_INTERRUPT_PENDING**
indicating the Timer_A interrupt status

Timer_A_getOutputForOutputModeOutBitValue()

```
uint8_t Timer_A_getOutputForOutputModeOutBitValue (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Get output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
captureCompareRegister	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Returns

One of the following:
- **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH**
- **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

References Timer_A_getCaptureCompareCount().

Referenced by Timer_A_getSynchronizedCaptureCompareInput().

Timer_A_getSynchronizedCaptureCompareInput()

```
uint8_t Timer_A_getSynchronizedCaptureCompareInput (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint16_t synchronized )
```

Get synchronized capturecompare input.

Parameters

baseAddress	is the base address of the TIMER_A module.

Parameters

captureCompareRegister	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
synchronized	Valid values are: ■ **TIMER_A_READ_SYNCHRONIZED_CAPTURECOMPAR↩ EINPUT** ■ **TIMER_A_READ_CAPTURE_COMPARE_INPUT**

Returns

One of the following:

- **TIMER_A_CAPTURECOMPARE_INPUT_HIGH**
- **TIMER_A_CAPTURECOMPARE_INPUT_LOW**

References Timer_A_getOutputForOutputModeOutBitValue().

Referenced by Timer_A_clear().

Timer_A_initCaptureMode()

```
void Timer_A_initCaptureMode (
            uint16_t baseAddress,
            Timer_A_initCaptureModeParam * param )
```

Initializes Capture Mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for capture mode initialization.

Modified bits of **TAxCCTLn** register.

Returns

None

References Timer_A_initCaptureModeParam::captureInputSelect, Timer_A_initCaptureModeParam::captureInterruptEnable, Timer_A_initCaptureModeParam::captureMode, Timer_A_initCaptureModeParam::captureOutputMode, Timer_A_initCaptureModeParam::captureRegister, and Timer_A_initCaptureModeParam::synchronizeCaptureSource.

Timer_A_initCompareMode()

```
void Timer_A_initCompareMode (
            uint16_t baseAddress,
            Timer_A_initCompareModeParam * param )
```

Initializes Compare Mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for compare mode initialization.

Modified bits of **TAxCCRn** register and bits of **TAxCCTLn** register.

Returns

None

References Timer_A_initCompareModeParam::compareInterruptEnable, Timer_A_initCompareModeParam::compareOutputMode, Timer_A_initCompareModeParam::compareRegister, and Timer_A_initCompareModeParam::compareValue.

Timer_A_initContinuousMode()

```
void Timer_A_initContinuousMode (
            uint16_t baseAddress,
            Timer_A_initContinuousModeParam * param )
```

Configures Timer_A in continuous mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for continuous mode initialization.

Modified bits of **TAxCTL** register.

Returns

None

References Timer_A_initContinuousModeParam::clockSource, Timer_A_initContinuousModeParam::clockSourceDivider, Timer_A_initContinuousModeParam::startTimer, Timer_A_initContinuousModeParam::timerClear, and Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE.

Timer_A_initUpDownMode()

```
void Timer_A_initUpDownMode (
```

```
uint16_t baseAddress,
Timer_A_initUpDownModeParam * param )
```

Configures Timer_A in up down mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for up-down mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_A_initUpDownModeParam::clockSource, Timer_A_initUpDownModeParam::clockSourceDivider, Timer_A_initUpDownModeParam::startTimer, Timer_A_initUpDownModeParam::timerClear, Timer_A_initUpDownModeParam::timerInterruptEnable_TAIE, and Timer_A_initUpDownModeParam::timerPeriod.

Timer_A_initUpMode()

```
void Timer_A_initUpMode (
        uint16_t baseAddress,
        Timer_A_initUpModeParam * param )
```

Configures Timer_A in up mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for up mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_A_initUpModeParam::clockSource, Timer_A_initUpModeParam::clockSourceDivider, Timer_A_initUpModeParam::startTimer, Timer_A_initUpModeParam::timerClear, Timer_A_initUpModeParam::timerInterruptEnable_TAIE, and Timer_A_initUpModeParam::timerPeriod.

Timer_A_outputPWM()

```
void Timer_A_outputPWM (
```

```
uint16_t baseAddress,
Timer_A_outputPWMParam * param )
```

Generate a PWM with timer running in up mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for PWM configuration.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register, bits of **TAxCCR0** register and bits of **TAxCCTLn** register.

Returns

None

References Timer_A_outputPWMParam::clockSource, Timer_A_outputPWMParam::clockSourceDivider, Timer_A_outputPWMParam::compareOutputMode, Timer_A_outputPWMParam::compareRegister, Timer_A_outputPWMParam::dutyCycle, and Timer_A_outputPWMParam::timerPeriod.

Timer_A_setCompareValue()

```
void Timer_A_setCompareValue (
            uint16_t baseAddress,
            uint16_t compareRegister,
            uint16_t compareValue )
```

Sets the value of the capture-compare register.

Parameters

baseAddress	is the base address of the TIMER_A module.
compareRegister	selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
compareValue	is the count to be compared with in compare mode

Modified bits of **TAxCCRn** register.

Returns

None

Timer_A_setOutputForOutputModeOutBitValue()

```
void Timer_A_setOutputForOutputModeOutBitValue (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint8_t outputModeOutBitValue )
```

Set output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
captureCompareRegister	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
outputModeOutBitValue	is the value to be set for out bit Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TAxCCTLn** register.

Returns

None

Referenced by Timer_A_getCaptureCompareCount().

Timer_A_setOutputMode()

```
void Timer_A_setOutputMode (
            uint16_t baseAddress,
            uint16_t compareRegister,
            uint16_t compareOutputMode )
```

Sets the output mode.

Sets the output mode for the timer even the timer is already running.

Parameters

baseAddress	is the base address of the TIMER_A module.

Parameters

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**
compareOutputMode	specifies the output mode. Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default] ■ **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**

Modified bits are **OUTMOD** of **TAxCCTLn** register.

Returns

None

Timer_A_startCounter()

```
void Timer_A_startCounter (
            uint16_t baseAddress,
            uint16_t timerMode )
```

Starts Timer_A counter.

This function assumes that the timer has been previously configured using Timer_A_initContinuousMode, Timer_A_initUpMode or Timer_A_initUpDownMode.

Parameters

baseAddress	is the base address of the TIMER_A module.
timerMode	mode to put the timer in Valid values are: ■ **TIMER_A_STOP_MODE** ■ **TIMER_A_UP_MODE** ■ **TIMER_A_CONTINUOUS_MODE** [Default] ■ **TIMER_A_UPDOWN_MODE**

Modified bits of **TAxCTL** register.

Returns

None

Timer_A_stop()

```
void Timer_A_stop (
            uint16_t baseAddress )
```

Stops the timer.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

24.3 Programming Example

The following example shows some TIMER_A operations using the APIs

```
{   //Start TIMER_A
    Timer_A_initContinuousModeParam initContParam = {0};
    initContParam.clockSource = TIMER_A_CLOCKSOURCE_SMCLK;
    initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
    initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_DISABLE;
    initContParam.timerClear = TIMER_A_DO_CLEAR;
    initContParam.startTimer = false;
    Timer_A_initContinuousMode(TIMER_A1_BASE, &initContParam);

    //Initiaze compare mode
    Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_0
        );

    Timer_A_initCompareModeParam initCompParam = {0};
    initCompParam.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_0;
    initCompParam.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE;
    initCompParam.compareOutputMode = TIMER_A_OUTPUTMODE_OUTBITVALUE;
    initCompParam.compareValue = COMPARE_VALUE;
    Timer_A_initCompareMode(TIMER_A1_BASE, &initCompParam);

    Timer_A_startCounter( TIMER_A1_BASE,
            TIMER_A_CONTINUOUS_MODE
                );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

    //For debugger
    __no_operation();
}
```

25 16-Bit Timer_B (TIMER_B)

25.1 Introduction

TIMER_B is a 16-bit timer/counter with multiple capture/compare registers. TIMER_B can support multiple capture/compares, PWM outputs, and interval timing. TIMER_B 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 B hardware peripheral.

TIMER_B 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_B interrupts

Differences From Timer_A Timer_B is identical to Timer_A with the following exceptions:

- The length of Timer_B is programmable to be 8, 10, 12, or 16 bits
- Timer_B TBxCCRn registers are double-buffered and can be grouped
- All Timer_B outputs can be put into a high-impedance state
- The SCCI bit function is not implemented in Timer_B

TIMER_B can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

TIMER_B Interrupts may be generated on counter overflow conditions and during capture compare events.

The TIMER_B may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TIMER_B_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 TIMER_B_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 TIMER_B_generatePWM() or a combination of Timer_initCompare() and timer start APIs

The TIMER_B API provides a set of functions for dealing with the TIMER_B 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.

25.2 API Functions

Functions

- void Timer_B_startCounter (uint16_t baseAddress, uint16_t timerMode)
  Starts Timer_B counter.
- void Timer_B_initContinuousMode (uint16_t baseAddress, Timer_B_initContinuousModeParam *param)
  Configures Timer_B in continuous mode.
- void Timer_B_initUpMode (uint16_t baseAddress, Timer_B_initUpModeParam *param)
  Configures Timer_B in up mode.
- void Timer_B_initUpDownMode (uint16_t baseAddress, Timer_B_initUpDownModeParam *param)
  Configures Timer_B in up down mode.
- void Timer_B_initCaptureMode (uint16_t baseAddress, Timer_B_initCaptureModeParam *param)
  Initializes Capture Mode.
- void Timer_B_initCompareMode (uint16_t baseAddress, Timer_B_initCompareModeParam *param)
  Initializes Compare Mode.
- void Timer_B_enableInterrupt (uint16_t baseAddress)
  Enable Timer_B interrupt.
- void Timer_B_disableInterrupt (uint16_t baseAddress)
  Disable Timer_B interrupt.
- uint32_t Timer_B_getInterruptStatus (uint16_t baseAddress)
  Get Timer_B interrupt status.
- void Timer_B_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Enable capture compare interrupt.
- void Timer_B_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Disable capture compare interrupt.
- uint32_t Timer_B_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
  Return capture compare interrupt status.
- void Timer_B_clear (uint16_t baseAddress)
  Reset/Clear the Timer_B clock divider, count direction, count.
- uint8_t Timer_B_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
  Get synchronized capturecompare input.
- uint8_t Timer_B_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get output bit for output mode.

- uint16_t Timer_B_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get current capturecompare count.
- void Timer_B_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t outputModeOutBitValue)
  Set output bit for output mode.
- void Timer_B_outputPWM (uint16_t baseAddress, Timer_B_outputPWMParam *param)
  Generate a PWM with Timer_B running in up mode.
- void Timer_B_stop (uint16_t baseAddress)
  Stops the Timer_B.
- void Timer_B_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
  Sets the value of the capture-compare register.
- void Timer_B_clearTimerInterrupt (uint16_t baseAddress)
  Clears the Timer_B TBIFG interrupt flag.
- void Timer_B_clearCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Clears the capture-compare interrupt flag.
- void Timer_B_selectCounterLength (uint16_t baseAddress, uint16_t counterLength)
  Selects Timer_B counter length.
- void Timer_B_selectLatchingGroup (uint16_t baseAddress, uint16_t groupLatch)
  Selects Timer_B Latching Group.
- void Timer_B_initCompareLatchLoadEvent (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
  Selects Compare Latch Load Event.
- uint16_t Timer_B_getCounterValue (uint16_t baseAddress)
  Reads the current timer count value.
- void Timer_B_setOutputMode (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareOutputMode)
  Sets the output mode.
- void Timer_B_selectOutputHighImpedanceTrigger (uint16_t baseAddress, uint8_t triggerSelect)
  Selects the trigger source to output high impedance.
- void Timer_B_remapPins (uint16_t baseAddress, uint8_t pinsSelect)
  Remaps Timer_B GPIO pins.

25.2.1 Detailed Description

The TIMER_B API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TIMER_B configuration and initialization is handled by

- Timer_B_startCounter()
- Timer_B_initUpMode()
- Timer_B_initUpDownMode()
- Timer_B_initContinuousMode()
- Timer_B_initCapture()
- Timer_B_initCompare()
- Timer_B_clear()

- Timer_B_stop()
- Timer_B_initCompareLatchLoadEvent()
- Timer_B_selectLatchingGroup()
- Timer_B_selectCounterLength()

TIMER_B outputs are handled by

- Timer_B_getSynchronizedCaptureCompareInput()
- Timer_B_getOutputForOutputModeOutBitValue()
- Timer_B_setOutputForOutputModeOutBitValue()
- Timer_B_generatePWM()
- Timer_B_getCaptureCompareCount()
- Timer_B_setCompareValue()
- Timer_B_getCounterValue()

The interrupt handler for the TIMER_B interrupt is managed with

- Timer_B_enableInterrupt()
- Timer_B_disableInterrupt()
- Timer_B_getInterruptStatus()
- Timer_B_enableCaptureCompareInterrupt()
- Timer_B_disableCaptureCompareInterrupt()
- Timer_B_getCaptureCompareInterruptStatus()
- Timer_B_clearCaptureCompareInterrupt()
- Timer_B_clearTimerInterrupt()

25.2.2 Function Documentation

Timer_B_clear()

```
void Timer_B_clear (
            uint16_t baseAddress )
```

Reset/Clear the Timer_B clock divider, count direction, count.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

References Timer_B_getSynchronizedCaptureCompareInput().

Timer_B_clearCaptureCompareInterrupt()

```
void Timer_B_clearCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Clears the capture-compare interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits are **CCIFG** of **TBxCCTLn** register.

Returns

None

Timer_B_clearTimerInterrupt()

```
void Timer_B_clearTimerInterrupt (
            uint16_t baseAddress )
```

Clears the Timer_B TBIFG interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits are **TBIFG** of **TBxCTL** register.

Returns

None

Timer_B_disableCaptureCompareInterrupt()

```
void Timer_B_disableCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Disable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TBxCCTLn** register.

Returns

None

Timer_B_disableInterrupt()

```
void Timer_B_disableInterrupt (
            uint16_t baseAddress )
```

Disable Timer_B interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

Timer_B_enableCaptureCompareInterrupt()

```
void Timer_B_enableCaptureCompareInterrupt (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Enable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TBxCCTLn** register.

Returns

None

Timer_B_enableInterrupt()

```
void Timer_B_enableInterrupt (
            uint16_t baseAddress )
```

Enable Timer_B interrupt.

Enables Timer_B interrupt. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

Timer_B_getCaptureCompareCount()

```
uint16_t Timer_B_getCaptureCompareCount (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Get current capturecompare count.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Returns

Current count as uint16_t

References Timer_B_setOutputForOutputModeOutBitValue().

Referenced by Timer_B_getOutputForOutputModeOutBitValue().

Timer_B_getCaptureCompareInterruptStatus()

```
uint32_t Timer_B_getCaptureCompareInterruptStatus (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint16_t mask )
```

Return capture compare interrupt status.

Parameters

baseAddress	is the base address of the TIMER_B module.

Parameters

captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
mask	is the mask for the interrupt status Mask value is the logical OR of any of the following: ■ **TIMER_B_CAPTURE_OVERFLOW** ■ **TIMER_B_CAPTURECOMPARE_INTERRUPT_FLAG**

Returns

Logical OR of any of the following:
- **TIMER_B_CAPTURE_OVERFLOW**
- **TIMER_B_CAPTURECOMPARE_INTERRUPT_FLAG**
  indicating the status of the masked interrupts

Timer_B_getCounterValue()

```
uint16_t Timer_B_getCounterValue (
            uint16_t baseAddress )
```

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The Timer_B_THRESHOLD #define in the associated header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

baseAddress	is the base address of the Timer module.

Returns

Majority vote of timer count value

Timer_B_getInterruptStatus()

```
uint32_t Timer_B_getInterruptStatus (
            uint16_t baseAddress )
```

Get Timer_B interrupt status.

Parameters

baseAddress	is the base address of the TIMER_B module.

Returns

One of the following:
- **TIMER_B_INTERRUPT_NOT_PENDING**
- **TIMER_B_INTERRUPT_PENDING**
  indicating the status of the Timer_B interrupt

Timer_B_getOutputForOutputModeOutBitValue()

```
uint8_t Timer_B_getOutputForOutputModeOutBitValue (
            uint16_t baseAddress,
            uint16_t captureCompareRegister )
```

Get output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Returns

One of the following:
- **TIMER_B_OUTPUTMODE_OUTBITVALUE_HIGH**
- **TIMER_B_OUTPUTMODE_OUTBITVALUE_LOW**

References Timer_B_getCaptureCompareCount().

Referenced by Timer_B_getSynchronizedCaptureCompareInput().

Timer_B_getSynchronizedCaptureCompareInput()

```
uint8_t Timer_B_getSynchronizedCaptureCompareInput (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint16_t synchronized )
```

Get synchronized capturecompare input.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
synchronized	selects the type of capture compare input Valid values are: ■ **TIMER_B_READ_SYNCHRONIZED_CAPTURECOMPAR↩ EINPUT** ■ **TIMER_B_READ_CAPTURE_COMPARE_INPUT**

Returns

One of the following:
- **TIMER_B_CAPTURECOMPARE_INPUT_HIGH**
- **TIMER_B_CAPTURECOMPARE_INPUT_LOW**

References Timer_B_getOutputForOutputModeOutBitValue().

Referenced by Timer_B_clear().

Timer_B_initCaptureMode()

```
void Timer_B_initCaptureMode (
            uint16_t baseAddress,
            Timer_B_initCaptureModeParam * param )
```

Initializes Capture Mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for capture mode initialization.

Modified bits of **TBxCCTLn** register.

Returns

None

References Timer_B_initCaptureModeParam::captureInputSelect, Timer_B_initCaptureModeParam::captureInterruptEnable, Timer_B_initCaptureModeParam::captureMode, Timer_B_initCaptureModeParam::captureOutputMode, Timer_B_initCaptureModeParam::captureRegister, and Timer_B_initCaptureModeParam::synchronizeCaptureSource.

Timer_B_initCompareLatchLoadEvent()

```
void Timer_B_initCompareLatchLoadEvent (
            uint16_t baseAddress,
            uint16_t compareRegister,
            uint16_t compareLatchLoadEvent )
```

Selects Compare Latch Load Event.

Parameters

baseAddress	is the base address of the TIMER_B module.
compareRegister	selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Parameters

compareLatchLoadEvent	selects the latch load event Valid values are: ■ **TIMER_B_LATCH_ON_WRITE_TO_TBxCCRn_COMPAR↩E_REGISTER** [Default] ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_I↩N_UP_OR_CONT_MODE** ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_I↩N_UPDOWN_MODE** ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_CU↩RRENT_COMPARE_LATCH_VALUE**

Modified bits are **CLLD** of **TBxCCTLn** register.

Returns

None

Timer_B_initCompareMode()

```
void Timer_B_initCompareMode (
            uint16_t baseAddress,
            Timer_B_initCompareModeParam * param )
```

Initializes Compare Mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for compare mode initialization.

Modified bits of **TBxCCTLn** register and bits of **TBxCCRn** register.

Returns

None

References Timer_B_initCompareModeParam::compareInterruptEnable, Timer_B_initCompareModeParam::compareOutputMode, Timer_B_initCompareModeParam::compareRegister, and Timer_B_initCompareModeParam::compareValue.

Timer_B_initContinuousMode()

```
void Timer_B_initContinuousMode (
            uint16_t baseAddress,
            Timer_B_initContinuousModeParam * param )
```

Configures Timer_B in continuous mode.

This API does not start the timer. Timer needs to be started when required using the Timer_B_startCounter API.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for continuous mode initialization.

Modified bits of **TBxCTL** register.

Returns

None

References Timer_B_initContinuousModeParam::clockSource, Timer_B_initContinuousModeParam::clockSourceDivider, Timer_B_initContinuousModeParam::startTimer, Timer_B_initContinuousModeParam::timerClear, and Timer_B_initContinuousModeParam::timerInterruptEnable_TBIE.

Timer_B_initUpDownMode()

```
void Timer_B_initUpDownMode (
            uint16_t baseAddress,
            Timer_B_initUpDownModeParam * param )
```

Configures Timer_B in up down mode.

This API does not start the timer. Timer needs to be started when required using the Timer_B_startCounter API.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for up-down mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_B_initUpDownModeParam::clockSource, Timer_B_initUpDownModeParam::clockSourceDivider, Timer_B_initUpDownModeParam::startTimer, Timer_B_initUpDownModeParam::timerClear, Timer_B_initUpDownModeParam::timerInterruptEnable_TBIE, and Timer_B_initUpDownModeParam::timerPeriod.

Timer_B_initUpMode()

```
void Timer_B_initUpMode (
            uint16_t baseAddress,
            Timer_B_initUpModeParam * param )
```

Configures Timer_B in up mode.

This API does not start the timer. Timer needs to be started when required using the Timer_B_startCounter API.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for up mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_B_initUpModeParam::clockSource, Timer_B_initUpModeParam::clockSourceDivider, Timer_B_initUpModeParam::startTimer, Timer_B_initUpModeParam::timerClear, Timer_B_initUpModeParam::timerInterruptEnable_TBIE, and Timer_B_initUpModeParam::timerPeriod.

Timer_B_outputPWM()

```
void Timer_B_outputPWM (
            uint16_t baseAddress,
            Timer_B_outputPWMParam * param )
```

Generate a PWM with Timer_B running in up mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for PWM configuration.

Modified bits of **TBxCCTLn** register, bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_outputPWMParam::clockSource, Timer_B_outputPWMParam::clockSourceDivider, Timer_B_outputPWMParam::compareOutputMode, Timer_B_outputPWMParam::compareRegister, Timer_B_outputPWMParam::dutyCycle, and Timer_B_outputPWMParam::timerPeriod.

Timer_B_remapPins()

```
void Timer_B_remapPins (
            uint16_t baseAddress,
            uint8_t pinsSelect )
```

Remaps Timer_B GPIO pins.

Remaps Timer_B GPIO pins. After calling this function, GPIO_setAsPeripheralModuleFunctionInputPin() or GPIO_setAsPeripheralModuleFunctionInputPin() still needs to be invoked to set peripheral functions.

Parameters

baseAddress	is the base address of the TIMER_B module.
pinsSelect	remapping pins to select. Please refer to device specific datasheet for remapping pins details. Valid values are: ■ **TIMER_B_REMAP_PINS_1** [Default] ■ **TIMER_B_REMAP_PINS_2**

Returns

None

Timer_B_selectCounterLength()

```
void Timer_B_selectCounterLength (
            uint16_t baseAddress,
            uint16_t counterLength )
```

Selects Timer_B counter length.

Parameters

baseAddress	is the base address of the TIMER_B module.

Parameters

counterLength	selects the value of counter length. Valid values are: ■ **TIMER_B_COUNTER_16BIT** [Default] ■ **TIMER_B_COUNTER_12BIT** ■ **TIMER_B_COUNTER_10BIT** ■ **TIMER_B_COUNTER_8BIT**

Modified bits are **CNTL** of **TBxCTL** register.

Returns

None

Timer_B_selectLatchingGroup()

```
void Timer_B_selectLatchingGroup (
            uint16_t baseAddress,
            uint16_t groupLatch )
```

Selects Timer_B Latching Group.

Parameters

baseAddress	is the base address of the TIMER_B module.
groupLatch	selects the latching group. Valid values are: ■ **TIMER_B_GROUP_NONE** [Default] ■ **TIMER_B_GROUP_CL12_CL23_CL56** ■ **TIMER_B_GROUP_CL123_CL456** ■ **TIMER_B_GROUP_ALL**

Modified bits are **TBCLGRP** of **TBxCTL** register.

Returns

None

Timer_B_selectOutputHighImpedanceTrigger()

```
void Timer_B_selectOutputHighImpedanceTrigger (
            uint16_t baseAddress,
            uint8_t triggerSelect )
```

Selects the trigger source to output high impedance.

Timer_B output can be triggered to output high impedance. The trigger source can be selected either internal or external.

Parameters

baseAddress	is the base address of the TIMER_B module.
triggerSelect	trigger to output high impedance Valid values are: ■ **TIMER_B_OUTPUTHIGH_TRIGGER_INTERNALSOURCE** [Default] ■ **TIMER_B_OUTPUTHIGH_TRIGGER_EXTERNALSOURCE**

Returns

None

Timer_B_setCompareValue()

```
void Timer_B_setCompareValue (
            uint16_t baseAddress,
            uint16_t compareRegister,
            uint16_t compareValue )
```

Sets the value of the capture-compare register.

Parameters

baseAddress	is the base address of the TIMER_B module.
compareRegister	selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
compareValue	is the count to be compared with in compare mode

Modified bits of **TBxCCRn** register.

Returns

None

Timer_B_setOutputForOutputModeOutBitValue()

```
void Timer_B_setOutputForOutputModeOutBitValue (
            uint16_t baseAddress,
            uint16_t captureCompareRegister,
            uint16_t outputModeOutBitValue )
```

Set output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
outputModeOutBitValue	the value to be set for out bit Valid values are: ■ **TIMER_B_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_B_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TBxCCTLn** register.

Returns

None

Referenced by Timer_B_getCaptureCompareCount().

Timer_B_setOutputMode()

```
void Timer_B_setOutputMode (
            uint16_t baseAddress,
            uint16_t compareRegister,
            uint16_t compareOutputMode )
```

Sets the output mode.

Sets the output mode for the timer even the timer is already running.

Parameters

baseAddress	is the base address of the TIMER_B module.
compareRegister	selects the compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
compareOutputMode	specifies the output mode. Valid values are: ■ **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default] ■ **TIMER_B_OUTPUTMODE_SET** ■ **TIMER_B_OUTPUTMODE_TOGGLE_RESET** ■ **TIMER_B_OUTPUTMODE_SET_RESET** ■ **TIMER_B_OUTPUTMODE_TOGGLE** ■ **TIMER_B_OUTPUTMODE_RESET** ■ **TIMER_B_OUTPUTMODE_TOGGLE_SET** ■ **TIMER_B_OUTPUTMODE_RESET_SET**

Modified bits are **OUTMOD** of **TBxCCTLn** register.

Returns

None

Timer_B_startCounter()

```
void Timer_B_startCounter (
            uint16_t baseAddress,
            uint16_t timerMode )
```

Starts Timer_B counter.

This function assumes that the timer has been previously configured using Timer_B_initContinuousMode, Timer_B_initUpMode or Timer_B_initUpDownMode.

Parameters

baseAddress	is the base address of the TIMER_B module.

Parameters

timerMode	selects the mode of the timer Valid values are: ■ **TIMER_B_STOP_MODE** ■ **TIMER_B_UP_MODE** ■ **TIMER_B_CONTINUOUS_MODE** [Default] ■ **TIMER_B_UPDOWN_MODE**

Modified bits of **TBxCTL** register.

Returns

None

Timer_B_stop()

```
void Timer_B_stop (
            uint16_t baseAddress )
```

Stops the Timer_B.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

25.3 Programming Example

The following example shows some TIMER_B operations using the APIs

```
{   //Start timer in continuous mode sourced by SMCLK
    Timer_B_initContinuousModeParam initContParam = {0};
    initContParam.clockSource = TIMER_B_CLOCKSOURCE_SMCLK;
    initContParam.clockSourceDivider = TIMER_B_CLOCKSOURCE_DIVIDER_1;
    initContParam.timerInterruptEnable_TBIE = TIMER_B_TBIE_INTERRUPT_DISABLE;
    initContParam.timerClear = TIMER_B_DO_CLEAR;
    initContParam.startTimer = false;
    Timer_B_initContinuousMode(TIMER_B0_BASE, &initContParam);

     //Initiaze compare mode
    Timer_B_clearCaptureCompareInterrupt(TIMER_B0_BASE,
        TIMER_B_CAPTURECOMPARE_REGISTER_0);

    Timer_B_initCompareModeParam initCompParam = {0};
    initCompParam.compareRegister = TIMER_B_CAPTURECOMPARE_REGISTER_0;
    initCompParam.compareInterruptEnable = TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE;
    initCompParam.compareOutputMode = TIMER_B_OUTPUTMODE_OUTBITVALUE;
```

```
    initCompParam.compareValue = COMPARE_VALUE;
    Timer_B_initCompareMode(TIMER_B0_BASE, &initCompParam);

    Timer_B_startCounter( TIMER_B0_BASE,
        TIMER_B_CONTINUOUS_MODE
        );

}
```

26 TIA

26.1 Introduction

TIA module is a transimpedance amplifier that converts current to voltage. It works in AM through LPM4, and fully shut down in LPM5. It has two kinds of power modes: high power with high speed and low power with low speed.

The API provides a set of functions for using the TIA module. Functions are provided to select positive inputs, power modes and enable/disable TIA module.

26.2 API Functions

Functions

- void TIA_selectPositiveInput (uint16_t baseAddress, uint16_t positiveInput)
  Selects TIA positive input.
- void TIA_selectPowerMode (uint16_t baseAddress, uint16_t powerMode)
  Selects TIA power mode.
- void TIA_enable (uint16_t baseAddress)
  Enables TIA module.
- void TIA_disable (uint16_t baseAddress)
  Disables TIA module.

26.2.1 Detailed Description

- TIA_selectPositiveInput()
- TIA_selectPowerMode()
- TIA_enable()
- TIA_disable()

26.2.2 Function Documentation

TIA_disable()

```
void TIA_disable (
            uint16_t baseAddress )
```

Disables TIA module.

This function disables TIA module.

Parameters

baseAddress	is the base address of the TIA module.

Returns

None

TIA_enable()

```
void TIA_enable (
            uint16_t baseAddress )
```

Enables TIA module.

This function enables TIA module.

Parameters

baseAddress	is the base address of the TIA module.

Returns

None

TIA_selectPositiveInput()

```
void TIA_selectPositiveInput (
            uint16_t baseAddress,
            uint16_t positiveInput )
```

Selects TIA positive input.

This function selects TIA positive input.

Parameters

baseAddress	is the base address of the TIA module.
positiveInput	selects positive input. Valid values are: ■ **TIA_POSITIVE_INPUT_EXTERNAL_SOURCE** [Default] ■ **TIA_POSITIVE_INPUT_DEVICE_SPECIFIC_1** ■ **TIA_POSITIVE_INPUT_DEVICE_SPECIFIC_2** ■ **TIA_POSITIVE_INPUT_DEVICE_SPECIFIC_3** Modified bits are **TRIPSEL** of **TRICTL** register.

Returns

None

TIA_selectPowerMode()

```
void TIA_selectPowerMode (
            uint16_t baseAddress,
            uint16_t powerMode )
```

Selects TIA power mode.

This function selects TIA power mode.

Parameters

baseAddress	is the base address of the TIA module.
powerMode	selects TIA power mode. Valid values are: ■ **TIA_HIGH_SPEED_HIGH_POWER** [Default] ■ **TIA_LOW_SPEED_LOW_POWER** Modified bits are **TRIPM** of **TRICTL** register.

Returns

None

26.3 Programming Example

The following example shows how to use TIA in low speed and low power mode.

```
//Select external source as the positive input
TIA_selectPositiveInput(TIA0_BASE, TIA_POSITIVE_INPUT_EXTERNAL_SOURCE);

//Configure TIA low speed low power mode
TIA_selectPowerMode(TIA0_BASE, TIA_LOW_SPEED_LOW_POWER);

//Enable TIA
TIA_enable(TIA0_BASE);

//Enter LPM3 mode
__bis_SR_register(LPM3_bits);
__no_operation();
```

27 WatchDog Timer (WDT_A)

27.1 Introduction

The Watchdog Timer (WDT_A) API provides a set of functions for using the MSP430Ware WDT_A modules. Functions are provided to initialize the Watchdog in either timer interval mode, or watchdog mode, with selectable clock sources and dividers to define the timer interval.

The WDT_A module can generate only 1 kind of interrupt in timer interval mode. If in watchdog mode, then the WDT_A module will assert a reset once the timer has finished.

27.2 API Functions

Functions

- void WDT_A_hold (uint16_t baseAddress)
  Holds the Watchdog Timer.
- void WDT_A_start (uint16_t baseAddress)
  Starts the Watchdog Timer.
- void WDT_A_resetTimer (uint16_t baseAddress)
  Resets the timer counter of the Watchdog Timer.
- void WDT_A_initWatchdogTimer (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
  Sets the clock source for the Watchdog Timer in watchdog mode.
- void WDT_A_initIntervalTimer (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
  Sets the clock source for the Watchdog Timer in timer interval mode.

27.2.1 Detailed Description

The WDT_A API is one group that controls the WDT_A module.

- WDT_A_hold()
- WDT_A_start()
- WDT_A_clearCounter()
- WDT_A_initWatchdogTimer()
- WDT_A_initIntervalTimer()

27.2.2 Function Documentation

WDT_A_hold()

```
void WDT_A_hold (
            uint16_t baseAddress )
```

Holds the Watchdog Timer.

This function stops the watchdog timer from running, that way no interrupt or PUC is asserted.

Parameters

baseAddress	is the base address of the WDT_A module.

Returns

None

WDT_A_initIntervalTimer()

```
void WDT_A_initIntervalTimer (
            uint16_t baseAddress,
            uint8_t clockSelect,
            uint8_t clockDivider )
```

Sets the clock source for the Watchdog Timer in timer interval mode.

This function sets the watchdog timer as timer interval mode, which will assert an interrupt without causing a PUC.

Parameters

baseAddress	is the base address of the WDT_A module.
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_XCLK** Modified bits are **WDTSSEL** of **WDTCTL** register.

Parameters

clockDivider	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: ■ **WDT_A_CLOCKDIVIDER_2G** ■ **WDT_A_CLOCKDIVIDER_128M** ■ **WDT_A_CLOCKDIVIDER_8192K** ■ **WDT_A_CLOCKDIVIDER_512K** ■ **WDT_A_CLOCKDIVIDER_32K** [Default] ■ **WDT_A_CLOCKDIVIDER_8192** ■ **WDT_A_CLOCKDIVIDER_512** ■ **WDT_A_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

WDT_A_initWatchdogTimer()

```
void WDT_A_initWatchdogTimer (
            uint16_t baseAddress,
            uint8_t clockSelect,
            uint8_t clockDivider )
```

Sets the clock source for the Watchdog Timer in watchdog mode.

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.

Parameters

baseAddress	is the base address of the WDT_A module.
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_XCLK** Modified bits are **WDTSSEL** of **WDTCTL** register.

Parameters

clockDivider	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: ■ **WDT_A_CLOCKDIVIDER_2G** ■ **WDT_A_CLOCKDIVIDER_128M** ■ **WDT_A_CLOCKDIVIDER_8192K** ■ **WDT_A_CLOCKDIVIDER_512K** ■ **WDT_A_CLOCKDIVIDER_32K** [Default] ■ **WDT_A_CLOCKDIVIDER_8192** ■ **WDT_A_CLOCKDIVIDER_512** ■ **WDT_A_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

WDT_A_resetTimer()

```
void WDT_A_resetTimer (
            uint16_t baseAddress )
```

Resets the timer counter of the Watchdog Timer.

This function resets the watchdog timer to 0x0000h.

Parameters

baseAddress	is the base address of the WDT_A module.

Returns

None

WDT_A_start()

```
void WDT_A_start (
            uint16_t baseAddress )
```

Starts the Watchdog Timer.

This function starts the watchdog timer functionality to start counting again.

Parameters

baseAddress	is the base address of the WDT_A module.

Returns

None

27.3 Programming Example

The following example shows how to initialize and use the WDT_A API to interrupt about every 32 ms, toggling the LED in the ISR.

```
//Initialize WDT_A module in timer interval mode,
  //with SMCLK as source at an interval of 32 ms.
  WDT_A_initIntervalTimer(WDT_A_BASE,
      WDT_A_CLOCKSOURCE_SMCLK,
      WDT_A_CLOCKDIVIDER_32K);

  //Enable Watchdog Interrupt
  SFR_enableInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);

  //Set P1.0 to output direction
  GPIO_setAsOutputPin(
      GPIO_PORT_P1,
      GPIO_PIN0
      );

  //Enter LPM0, enable interrupts
  __bis_SR_register(LPM0_bits + GIE);
  //For debugger
  __no_operation();
```

28 Data Structure Documentation

28.1 Data Structures

Here are the data structures with brief descriptions:

28.2 Timer_B_initContinuousModeParam Struct Reference

Used in the Timer_B_initContinuousMode() function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerInterruptEnable_TBIE
- uint16_t timerClear
- bool startTimer
  Whether to start the timer immediately.

28.2.1 Detailed Description

Used in the Timer_B_initContinuousMode() function as the param parameter.

28.2.2 Field Documentation

clockSource

```
uint16_t Timer_B_initContinuousModeParam::clockSource
```

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_B_initContinuousMode().

clockSourceDivider

`uint16_t Timer_B_initContinuousModeParam::clockSourceDivider`

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by Timer_B_initContinuousMode().

timerClear

`uint16_t Timer_B_initContinuousModeParam::timerClear`

Decides if Timer_B clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by Timer_B_initContinuousMode().

timerInterruptEnable_TBIE

`uint16_t Timer_B_initContinuousModeParam::timerInterruptEnable_TBIE`

Is to enable or disable Timer_B interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_B_initContinuousMode().

The documentation for this struct was generated from the following file:

- timer_b.h

28.3 EComp_configureDACParam Struct Reference

Used in the EComp_configureDAC() function as the param parameter.

`#include <ecomp.h>`

Data Fields

- uint8_t referenceVoltage
- uint8_t bufferSource
- uint16_t firstBufferData
  Sets the first DAC buffer data (0∼63).
- uint16_t secondBufferData

28.3.1 Detailed Description

Used in the EComp_configureDAC() function as the param parameter.

28.3.2 Field Documentation

bufferSource

`uint8_t EComp_configureDACParam::bufferSource`

Selects the built-in DAC buffer controlled source.
Valid values are:

- **ECOMP_DAC_BUFFER_SOURCE_COMP_OUTPUT**
- **ECOMP_DAC_BUFFER_SOURCE_DUAL_BUFFER_1** [Default]
- **ECOMP_DAC_BUFFER_SOURCE_DUAL_BUFFER_2**

Referenced by EComp_configureDAC().

referenceVoltage

`uint8_t EComp_configureDACParam::referenceVoltage`

Selects the built-in DAC reference voltage.
Valid values are:

- **ECOMP_DAC_REFERENCE_VOLTAGE_VDD** [Default]
- **ECOMP_DAC_REFERENCE_VOLTAGE_VREF**

Referenced by EComp_configureDAC().

secondBufferData

`uint16_t EComp_configureDACParam::secondBufferData`

Sets the second DAC buffer data (0∼63). The reset value for the second DAC buffer is 0x1.

Referenced by EComp_configureDAC().

The documentation for this struct was generated from the following file:

- ecomp.h

28.4 Timer_A_initUpModeParam Struct Reference

Used in the Timer_A_initUpMode() function as the param parameter.

`#include <timer_a.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod
- uint16_t timerInterruptEnable_TAIE
- uint16_t captureCompareInterruptEnable_CCR0_CCIE
- uint16_t timerClear
- bool startTimer

    Whether to start the timer immediately.

28.4.1 Detailed Description

Used in the Timer_A_initUpMode() function as the param parameter.

28.4.2 Field Documentation

captureCompareInterruptEnable_CCR0_CCIE

`uint16_t Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE`

Is to enable or disable Timer_A CCR0 captureComapre interrupt.
Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpMode().

clockSource

`uint16_t Timer_A_initUpModeParam::clockSource`

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_initUpMode().

clockSourceDivider

`uint16_t Timer_A_initUpModeParam::clockSourceDivider`

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **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**

Referenced by Timer_A_initUpMode().

timerClear

`uint16_t Timer_A_initUpModeParam::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]

Referenced by Timer_A_initUpMode().

timerInterruptEnable_TAIE

`uint16_t Timer_A_initUpModeParam::timerInterruptEnable_TAIE`

Is to enable or disable Timer_A interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpMode().

timerPeriod

`uint16_t Timer_A_initUpModeParam::timerPeriod`

Is the specified Timer_A period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

Referenced by Timer_A_initUpMode().

The documentation for this struct was generated from the following file:

- timer_a.h

28.5 EUSCI_B_SPI_initSlaveParam Struct Reference

Used in the EUSCI_B_SPI_initSlave() function as the param parameter.

`#include <eusci_b_spi.h>`

Data Fields

- uint16_t msbFirst
- uint16_t clockPhase
- uint16_t clockPolarity
- uint16_t spiMode

28.5.1 Detailed Description

Used in the EUSCI_B_SPI_initSlave() function as the param parameter.

28.5.2 Field Documentation

clockPhase

`uint16_t EUSCI_B_SPI_initSlaveParam::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**

Referenced by EUSCI_B_SPI_initSlave().

clockPolarity

`uint16_t EUSCI_B_SPI_initSlaveParam::clockPolarity`

Is clock polarity select
Valid values are:

- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by EUSCI_B_SPI_initSlave().

msbFirst

`uint16_t EUSCI_B_SPI_initSlaveParam::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]

Referenced by EUSCI_B_SPI_initSlave().

spiMode

`uint16_t EUSCI_B_SPI_initSlaveParam::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**

Referenced by EUSCI_B_SPI_initSlave().

The documentation for this struct was generated from the following file:

- eusci_b_spi.h

28.6 Timer_A_initCompareModeParam Struct Reference

Used in the Timer_A_initCompareMode() function as the param parameter.

`#include <timer_a.h>`

Data Fields

- uint16_t compareRegister
- uint16_t compareInterruptEnable
- uint16_t compareOutputMode
- uint16_t compareValue
  Is the count to be compared with in compare mode.

28.6.1 Detailed Description

Used in the Timer_A_initCompareMode() function as the param parameter.

28.6.2 Field Documentation

compareInterruptEnable

`uint16_t Timer_A_initCompareModeParam::compareInterruptEnable`

Is to enable or disable timer captureComapre interrupt.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_A_initCompareMode().

compareOutputMode

`uint16_t Timer_A_initCompareModeParam::compareOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **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**

Referenced by Timer_A_initCompareMode().

compareRegister

`uint16_t Timer_A_initCompareModeParam::compareRegister`

Selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_REGISTER_0**
- **TIMER_A_CAPTURECOMPARE_REGISTER_1**
- **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Referenced by Timer_A_initCompareMode().

The documentation for this struct was generated from the following file:

- timer_a.h

28.7 EUSCI_B_SPI_changeMasterClockParam Struct Reference

Used in the EUSCI_B_SPI_changeMasterClock() function as the param parameter.

`#include <eusci_b_spi.h>`

Data Fields

- uint32_t clockSourceFrequency
  Is the frequency of the selected clock source in Hz.
- uint32_t desiredSpiClock
  Is the desired clock rate in Hz for SPI communication.

28.7.1 Detailed Description

Used in the EUSCI_B_SPI_changeMasterClock() function as the param parameter.

The documentation for this struct was generated from the following file:

- eusci_b_spi.h

28.8 Timer_B_initUpDownModeParam Struct Reference

Used in the Timer_B_initUpDownMode() function as the param parameter.

`#include <timer_b.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod
  Is the specified Timer_B period.
- uint16_t timerInterruptEnable_TBIE
- uint16_t captureCompareInterruptEnable_CCR0_CCIE
- uint16_t timerClear
- bool startTimer
  Whether to start the timer immediately.

28.8.1 Detailed Description

Used in the Timer_B_initUpDownMode() function as the param parameter.

28.8.2 Field Documentation

captureCompareInterruptEnable_CCR0_CCIE

`uint16_t Timer_B_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE`

Is to enable or disable Timer_B CCR0 capture compare interrupt.
Valid values are:

- **TIMER_B_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_B_initUpDownMode().

clockSource

`uint16_t Timer_B_initUpDownModeParam::clockSource`

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_B_initUpDownMode().

clockSourceDivider

`uint16_t Timer_B_initUpDownModeParam::clockSourceDivider`

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**

- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by Timer_B_initUpDownMode().

timerClear

`uint16_t Timer_B_initUpDownModeParam::timerClear`

Decides if Timer_B clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by Timer_B_initUpDownMode().

timerInterruptEnable_TBIE

`uint16_t Timer_B_initUpDownModeParam::timerInterruptEnable_TBIE`

Is to enable or disable Timer_B interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_B_initUpDownMode().

The documentation for this struct was generated from the following file:

- timer_b.h

28.9 Timer_A_initContinuousModeParam Struct Reference

Used in the Timer_A_initContinuousMode() function as the param parameter.

`#include <timer_a.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerInterruptEnable_TAIE
- uint16_t timerClear
- bool startTimer

    Whether to start the timer immediately.

28.9.1 Detailed Description

Used in the Timer_A_initContinuousMode() function as the param parameter.

28.9.2 Field Documentation

clockSource

`uint16_t Timer_A_initContinuousModeParam::clockSource`

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_initContinuousMode().

clockSourceDivider

`uint16_t Timer_A_initContinuousModeParam::clockSourceDivider`

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **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**

Referenced by Timer_A_initContinuousMode().

timerClear

`uint16_t Timer_A_initContinuousModeParam::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]

Referenced by Timer_A_initContinuousMode().

timerInterruptEnable_TAIE

`uint16_t Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE`

Is to enable or disable Timer_A interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initContinuousMode().

The documentation for this struct was generated from the following file:

- timer_a.h

28.10 EUSCI_B_I2C_initSlaveParam Struct Reference

Used in the EUSCI_B_I2C_initSlave() function as the param parameter.

```
#include <eusci_b_i2c.h>
```

Data Fields

- uint8_t slaveAddress
  7-bit slave address
- uint8_t slaveAddressOffset
- uint32_t slaveOwnAddressEnable

28.10.1 Detailed Description

Used in the EUSCI_B_I2C_initSlave() function as the param parameter.

28.10.2 Field Documentation

slaveAddressOffset

`uint8_t EUSCI_B_I2C_initSlaveParam::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**

Referenced by EUSCI_B_I2C_initSlave().

slaveOwnAddressEnable

`uint32_t EUSCI_B_I2C_initSlaveParam::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**

Referenced by EUSCI_B_I2C_initSlave().

The documentation for this struct was generated from the following file:

- eusci_b_i2c.h

28.11 Timer_A_initCaptureModeParam Struct Reference

Used in the Timer_A_initCaptureMode() function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- uint16_t captureRegister
- uint16_t captureMode
- uint16_t captureInputSelect
- uint16_t synchronizeCaptureSource
- uint16_t captureInterruptEnable
- uint16_t captureOutputMode

28.11.1 Detailed Description

Used in the Timer_A_initCaptureMode() function as the param parameter.

28.11.2 Field Documentation

captureInputSelect

```
uint16_t Timer_A_initCaptureModeParam::captureInputSelect
```

Decides the Input Select
Valid values are:

- **TIMER_A_CAPTURE_INPUTSELECT_CCIxA**
- **TIMER_A_CAPTURE_INPUTSELECT_CCIxB**
- **TIMER_A_CAPTURE_INPUTSELECT_GND**
- **TIMER_A_CAPTURE_INPUTSELECT_Vcc**

Referenced by Timer_A_initCaptureMode().

captureInterruptEnable

```
uint16_t Timer_A_initCaptureModeParam::captureInterruptEnable
```

Is to enable or disable timer captureComapre interrupt.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_A_initCaptureMode().

captureMode

```
uint16_t Timer_A_initCaptureModeParam::captureMode
```

Is the capture mode selected.
Valid values are:

- **TIMER_A_CAPTUREMODE_NO_CAPTURE** [Default]

- **TIMER_A_CAPTUREMODE_RISING_EDGE**
- **TIMER_A_CAPTUREMODE_FALLING_EDGE**
- **TIMER_A_CAPTUREMODE_RISING_AND_FALLING_EDGE**

Referenced by Timer_A_initCaptureMode().

captureOutputMode

`uint16_t Timer_A_initCaptureModeParam::captureOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **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**

Referenced by Timer_A_initCaptureMode().

captureRegister

`uint16_t Timer_A_initCaptureModeParam::captureRegister`

Selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_REGISTER_0**
- **TIMER_A_CAPTURECOMPARE_REGISTER_1**
- **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Referenced by Timer_A_initCaptureMode().

synchronizeCaptureSource

`uint16_t Timer_A_initCaptureModeParam::synchronizeCaptureSource`

Decides if capture source should be synchronized with timer clock
Valid values are:

- **TIMER_A_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_A_CAPTURE_SYNCHRONOUS**

Referenced by Timer_A_initCaptureMode().

The documentation for this struct was generated from the following file:

- timer_a.h

28.12 EUSCI_A_UART_initParam Struct Reference

Used in the EUSCI_A_UART_init() function as the param parameter.

`#include <eusci_a_uart.h>`

Data Fields

- uint8_t selectClockSource
- uint16_t clockPrescalar
  Is the value to be written into UCBRx bits.
- uint8_t firstModReg
- uint8_t secondModReg
- uint8_t parity
- uint16_t msborLsbFirst
- uint16_t numberofStopBits
- uint16_t uartMode
- uint8_t overSampling

28.12.1 Detailed Description

Used in the EUSCI_A_UART_init() function as the param parameter.

28.12.2 Field Documentation

firstModReg

`uint8_t EUSCI_A_UART_initParam::firstModReg`

Is First modulation stage register setting. This value is a pre- calculated value which can be obtained from the Device Users Guide. This value is written into UCBRFx bits of UCAxMCTLW.

Referenced by EUSCI_A_UART_init().

msborLsbFirst

`uint16_t EUSCI_A_UART_initParam::msborLsbFirst`

Controls direction of receive and transmit shift register.
Valid values are:

- **EUSCI_A_UART_MSB_FIRST**

- **EUSCI_A_UART_LSB_FIRST** [Default]

Referenced by EUSCI_A_UART_init().

numberofStopBits

`uint16_t EUSCI_A_UART_initParam::numberofStopBits`

Indicates one/two STOP bits
Valid values are:

- **EUSCI_A_UART_ONE_STOP_BIT** [Default]
- **EUSCI_A_UART_TWO_STOP_BITS**

Referenced by EUSCI_A_UART_init().

overSampling

`uint8_t EUSCI_A_UART_initParam::overSampling`

Indicates low frequency or oversampling baud generation
Valid values are:

- **EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION**
- **EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

Referenced by EUSCI_A_UART_init().

parity

`uint8_t EUSCI_A_UART_initParam::parity`

Is the desired parity.
Valid values are:

- **EUSCI_A_UART_NO_PARITY** [Default]
- **EUSCI_A_UART_ODD_PARITY**
- **EUSCI_A_UART_EVEN_PARITY**

Referenced by EUSCI_A_UART_init().

secondModReg

`uint8_t EUSCI_A_UART_initParam::secondModReg`

Is Second modulation stage register setting. This value is a pre- calculated value which can be obtained from the Device Users Guide. This value is written into UCBRSx bits of UCAxMCTLW.

Referenced by EUSCI_A_UART_init().

selectClockSource

`uint8_t EUSCI_A_UART_initParam::selectClockSource`

Selects Clock source. Refer to device specific datasheet for available options.
Valid values are:

- **EUSCI_A_UART_CLOCKSOURCE_UCLK**
- **EUSCI_A_UART_CLOCKSOURCE_SMCLK**
- **EUSCI_A_UART_CLOCKSOURCE_ACLK**
- **EUSCI_A_UART_CLOCKSOURCE_MODCLK**

Referenced by EUSCI_A_UART_init().

uartMode

`uint16_t EUSCI_A_UART_initParam::uartMode`

Selects the mode of operation
Valid values are:

- **EUSCI_A_UART_MODE** [Default]
- **EUSCI_A_UART_IDLE_LINE_MULTI_PROCESSOR_MODE**
- **EUSCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR_MODE**
- **EUSCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION_MODE**

Referenced by EUSCI_A_UART_init().

The documentation for this struct was generated from the following file:

- eusci_a_uart.h

28.13 Timer_B_outputPWMParam Struct Reference

Used in the Timer_B_outputPWM() function as the param parameter.

`#include <timer_b.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod
  Selects the desired Timer_B period.
- uint16_t compareRegister
- uint16_t compareOutputMode
- uint16_t dutyCycle
  Specifies the dutycycle for the generated waveform.

28.13.1 Detailed Description

Used in the Timer_B_outputPWM() function as the param parameter.

28.13.2 Field Documentation

clockSource

```
uint16_t Timer_B_outputPWMParam::clockSource
```

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_B_outputPWM().

clockSourceDivider

```
uint16_t Timer_B_outputPWMParam::clockSourceDivider
```

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**

- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by Timer_B_outputPWM().

compareOutputMode

`uint16_t Timer_B_outputPWMParam::compareOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_B_OUTPUTMODE_SET**
- **TIMER_B_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_B_OUTPUTMODE_SET_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE**
- **TIMER_B_OUTPUTMODE_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE_SET**
- **TIMER_B_OUTPUTMODE_RESET_SET**

Referenced by Timer_B_outputPWM().

compareRegister

`uint16_t Timer_B_outputPWMParam::compareRegister`

Selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by Timer_B_outputPWM().

The documentation for this struct was generated from the following file:

- timer_b.h

28.14 EUSCI_B_I2C_initMasterParam Struct Reference

Used in the EUSCI_B_I2C_initMaster() function as the param parameter.

`#include <eusci_b_i2c.h>`

Data Fields

- uint8_t selectClockSource
- uint32_t i2cClk
- uint32_t dataRate
- uint8_t byteCounterThreshold
  Sets threshold for automatic STOP or UCSTPIFG.
- uint8_t autoSTOPGeneration

28.14.1 Detailed Description

Used in the EUSCI_B_I2C_initMaster() function as the param parameter.

28.14.2 Field Documentation

autoSTOPGeneration

`uint8_t EUSCI_B_I2C_initMasterParam::autoSTOPGeneration`

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**

Referenced by EUSCI_B_I2C_initMaster().

dataRate

`uint32_t EUSCI_B_I2C_initMasterParam::dataRate`

Setup for selecting data transfer rate.
Valid values are:

- **EUSCI_B_I2C_SET_DATA_RATE_400KBPS**
- **EUSCI_B_I2C_SET_DATA_RATE_100KBPS**

Referenced by EUSCI_B_I2C_initMaster().

i2cClk

`uint32_t EUSCI_B_I2C_initMasterParam::i2cClk`

Is the rate of the clock supplied to the I2C module (the frequency in Hz of the clock source specified in selectClockSource).

Referenced by EUSCI_B_I2C_initMaster().

selectClockSource

`uint8_t EUSCI_B_I2C_initMasterParam::selectClockSource`

Selects the clocksource. Refer to device specific datasheet for available options.
Valid values are:

- **EUSCI_B_I2C_CLOCKSOURCE_UCLK**
- **EUSCI_B_I2C_CLOCKSOURCE_ACLK**
- **EUSCI_B_I2C_CLOCKSOURCE_MODCLK**
- **EUSCI_B_I2C_CLOCKSOURCE_SMCLK**

Referenced by EUSCI_B_I2C_initMaster().

The documentation for this struct was generated from the following file:

- eusci_b_i2c.h

28.15 EUSCI_A_SPI_changeMasterClockParam Struct Reference

Used in the EUSCI_A_SPI_changeMasterClock() function as the param parameter.

`#include <eusci_a_spi.h>`

Data Fields

- uint32_t clockSourceFrequency
  Is the frequency of the selected clock source in Hz.
- uint32_t desiredSpiClock
  Is the desired clock rate in Hz for SPI communication.

28.15.1 Detailed Description

Used in the EUSCI_A_SPI_changeMasterClock() function as the param parameter.

The documentation for this struct was generated from the following file:

- eusci_a_spi.h

28.16 Timer_B_initUpModeParam Struct Reference

Used in the Timer_B_initUpMode() function as the param parameter.

`#include <timer_b.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod
- uint16_t timerInterruptEnable_TBIE
- uint16_t captureCompareInterruptEnable_CCR0_CCIE
- uint16_t timerClear
- bool startTimer

  Whether to start the timer immediately.

28.16.1 Detailed Description

Used in the Timer_B_initUpMode() function as the param parameter.

28.16.2 Field Documentation

captureCompareInterruptEnable_CCR0_CCIE

`uint16_t Timer_B_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE`

Is to enable or disable Timer_B CCR0 capture compare interrupt.
Valid values are:

- **TIMER_B_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_B_initUpMode().

clockSource

`uint16_t Timer_B_initUpModeParam::clockSource`

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_B_initUpMode().

clockSourceDivider

`uint16_t Timer_B_initUpModeParam::clockSourceDivider`

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by Timer_B_initUpMode().

timerClear

`uint16_t Timer_B_initUpModeParam::timerClear`

Decides if Timer_B clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by Timer_B_initUpMode().

timerInterruptEnable_TBIE

`uint16_t Timer_B_initUpModeParam::timerInterruptEnable_TBIE`

Is to enable or disable Timer_B interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_B_initUpMode().

timerPeriod

`uint16_t Timer_B_initUpModeParam::timerPeriod`

Is the specified Timer_B period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

Referenced by Timer_B_initUpMode().

The documentation for this struct was generated from the following file:

- timer_b.h

28.17 Timer_B_initCompareModeParam Struct Reference

Used in the Timer_B_initCompareMode() function as the param parameter.

`#include <timer_b.h>`

Data Fields

- uint16_t compareRegister
- uint16_t compareInterruptEnable
- uint16_t compareOutputMode
- uint16_t compareValue
  Is the count to be compared with in compare mode.

28.17.1 Detailed Description

Used in the Timer_B_initCompareMode() function as the param parameter.

28.17.2 Field Documentation

compareInterruptEnable

`uint16_t Timer_B_initCompareModeParam::compareInterruptEnable`

Is to enable or disable Timer_B capture compare interrupt.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_B_initCompareMode().

compareOutputMode

`uint16_t Timer_B_initCompareModeParam::compareOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_B_OUTPUTMODE_SET**
- **TIMER_B_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_B_OUTPUTMODE_SET_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE**
- **TIMER_B_OUTPUTMODE_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE_SET**
- **TIMER_B_OUTPUTMODE_RESET_SET**

Referenced by Timer_B_initCompareMode().

compareRegister

`uint16_t Timer_B_initCompareModeParam::compareRegister`

Selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by Timer_B_initCompareMode().

The documentation for this struct was generated from the following file:

- timer_b.h

28.18 EUSCI_A_SPI_initMasterParam Struct Reference

Used in the EUSCI_A_SPI_initMaster() function as the param parameter.

`#include <eusci_a_spi.h>`

Data Fields

- uint8_t selectClockSource
- uint32_t clockSourceFrequency
  Is the frequency of the selected clock source in Hz.
- uint32_t desiredSpiClock
  Is the desired clock rate in Hz for SPI communication.
- uint16_t msbFirst
- uint16_t clockPhase
- uint16_t clockPolarity
- uint16_t spiMode

28.18.1 Detailed Description

Used in the EUSCI_A_SPI_initMaster() function as the param parameter.

28.18.2 Field Documentation

clockPhase

`uint16_t EUSCI_A_SPI_initMasterParam::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**

Referenced by EUSCI_A_SPI_initMaster().

clockPolarity

`uint16_t EUSCI_A_SPI_initMasterParam::clockPolarity`

Is clock polarity select
Valid values are:

- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by EUSCI_A_SPI_initMaster().

msbFirst

`uint16_t EUSCI_A_SPI_initMasterParam::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]

Referenced by EUSCI_A_SPI_initMaster().

selectClockSource

`uint8_t EUSCI_A_SPI_initMasterParam::selectClockSource`

Selects Clock source. Refer to device specific datasheet for available options.
Valid values are:

- **EUSCI_A_SPI_CLOCKSOURCE_UCLK**
- **EUSCI_A_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_A_SPI_CLOCKSOURCE_MODCLK**
- **EUSCI_A_SPI_CLOCKSOURCE_SMCLK**

Referenced by EUSCI_A_SPI_initMaster().

spiMode

`uint16_t EUSCI_A_SPI_initMasterParam::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**

Referenced by EUSCI_A_SPI_initMaster().

The documentation for this struct was generated from the following file:

- eusci_a_spi.h

28.19 Timer_B_initCaptureModeParam Struct Reference

Used in the Timer_B_initCaptureMode() function as the param parameter.

`#include <timer_b.h>`

Data Fields

- uint16_t captureRegister
- uint16_t captureMode
- uint16_t captureInputSelect
- uint16_t synchronizeCaptureSource
- uint16_t captureInterruptEnable
- uint16_t captureOutputMode

28.19.1 Detailed Description

Used in the Timer_B_initCaptureMode() function as the param parameter.

28.19.2 Field Documentation

captureInputSelect

`uint16_t Timer_B_initCaptureModeParam::captureInputSelect`

Decides the Input Select
Valid values are:

- **TIMER_B_CAPTURE_INPUTSELECT_CCIxA** [Default]
- **TIMER_B_CAPTURE_INPUTSELECT_CCIxB**
- **TIMER_B_CAPTURE_INPUTSELECT_GND**
- **TIMER_B_CAPTURE_INPUTSELECT_Vcc**

Referenced by Timer_B_initCaptureMode().

captureInterruptEnable

`uint16_t Timer_B_initCaptureModeParam::captureInterruptEnable`

Is to enable or disable Timer_B capture compare interrupt.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_B_initCaptureMode().

captureMode

`uint16_t Timer_B_initCaptureModeParam::captureMode`

Is the capture mode selected.
Valid values are:

- **TIMER_B_CAPTUREMODE_NO_CAPTURE** [Default]

- **TIMER_B_CAPTUREMODE_RISING_EDGE**
- **TIMER_B_CAPTUREMODE_FALLING_EDGE**
- **TIMER_B_CAPTUREMODE_RISING_AND_FALLING_EDGE**

Referenced by Timer_B_initCaptureMode().

captureOutputMode

`uint16_t Timer_B_initCaptureModeParam::captureOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_B_OUTPUTMODE_SET**
- **TIMER_B_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_B_OUTPUTMODE_SET_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE**
- **TIMER_B_OUTPUTMODE_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE_SET**
- **TIMER_B_OUTPUTMODE_RESET_SET**

Referenced by Timer_B_initCaptureMode().

captureRegister

`uint16_t Timer_B_initCaptureModeParam::captureRegister`

Selects the capture register being used. Refer to datasheet to ensure the device has the capture register being used.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by Timer_B_initCaptureMode().

synchronizeCaptureSource

`uint16_t Timer_B_initCaptureModeParam::synchronizeCaptureSource`

Decides if capture source should be synchronized with Timer_B clock
Valid values are:

- **TIMER_B_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_B_CAPTURE_SYNCHRONOUS**

Referenced by Timer_B_initCaptureMode().

The documentation for this struct was generated from the following file:

- timer_b.h

28.20 EUSCI_B_SPI_initMasterParam Struct Reference

Used in the EUSCI_B_SPI_initMaster() function as the param parameter.

```
#include <eusci_b_spi.h>
```

Data Fields

- uint8_t selectClockSource
- uint32_t clockSourceFrequency
  Is the frequency of the selected clock source in Hz.
- uint32_t desiredSpiClock
  Is the desired clock rate in Hz for SPI communication.
- uint16_t msbFirst
- uint16_t clockPhase
- uint16_t clockPolarity
- uint16_t spiMode

28.20.1 Detailed Description

Used in the EUSCI_B_SPI_initMaster() function as the param parameter.

28.20.2 Field Documentation

clockPhase

```
uint16_t EUSCI_B_SPI_initMasterParam::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**

Referenced by EUSCI_B_SPI_initMaster().

clockPolarity

`uint16_t EUSCI_B_SPI_initMasterParam::clockPolarity`

Is clock polarity select
Valid values are:

- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by EUSCI_B_SPI_initMaster().

msbFirst

`uint16_t EUSCI_B_SPI_initMasterParam::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]

Referenced by EUSCI_B_SPI_initMaster().

selectClockSource

`uint8_t EUSCI_B_SPI_initMasterParam::selectClockSource`

Selects Clock source. Refer to device specific datasheet for available options.
Valid values are:

- **EUSCI_B_SPI_CLOCKSOURCE_UCLK**
- **EUSCI_B_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_B_SPI_CLOCKSOURCE_MODCLK**
- **EUSCI_B_SPI_CLOCKSOURCE_SMCLK**

Referenced by EUSCI_B_SPI_initMaster().

spiMode

`uint16_t EUSCI_B_SPI_initMasterParam::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**

Referenced by EUSCI_B_SPI_initMaster().

The documentation for this struct was generated from the following file:

- eusci_b_spi.h

28.21 LCD_E_initParam Struct Reference

Used in the LCD_E_init() function as the initParams parameter.

```
#include <lcd_e.h>
```

Data Fields

- uint16_t clockSource
- uint16_t clockDivider
- uint16_t muxRate
- uint16_t waveforms
- uint16_t segments

28.21.1 Detailed Description

Used in the LCD_E_init() function as the initParams parameter.

28.21.2 Field Documentation

clockDivider

```
uint16_t LCD_E_initParam::clockDivider
```

Selects the divider for LCD_E frequency.
Valid values are:

- **LCD_E_CLOCKDIVIDER_1** [Default]
- **LCD_E_CLOCKDIVIDER_2**
- **LCD_E_CLOCKDIVIDER_3**
- **LCD_E_CLOCKDIVIDER_4**
- **LCD_E_CLOCKDIVIDER_5**
- **LCD_E_CLOCKDIVIDER_6**
- **LCD_E_CLOCKDIVIDER_7**
- **LCD_E_CLOCKDIVIDER_8**
- **LCD_E_CLOCKDIVIDER_9**
- **LCD_E_CLOCKDIVIDER_10**
- **LCD_E_CLOCKDIVIDER_11**

- **LCD_E_CLOCKDIVIDER_12**
- **LCD_E_CLOCKDIVIDER_13**
- **LCD_E_CLOCKDIVIDER_14**
- **LCD_E_CLOCKDIVIDER_15**
- **LCD_E_CLOCKDIVIDER_16**
- **LCD_E_CLOCKDIVIDER_17**
- **LCD_E_CLOCKDIVIDER_18**
- **LCD_E_CLOCKDIVIDER_19**
- **LCD_E_CLOCKDIVIDER_20**
- **LCD_E_CLOCKDIVIDER_21**
- **LCD_E_CLOCKDIVIDER_22**
- **LCD_E_CLOCKDIVIDER_23**
- **LCD_E_CLOCKDIVIDER_24**
- **LCD_E_CLOCKDIVIDER_25**
- **LCD_E_CLOCKDIVIDER_26**
- **LCD_E_CLOCKDIVIDER_27**
- **LCD_E_CLOCKDIVIDER_28**
- **LCD_E_CLOCKDIVIDER_29**
- **LCD_E_CLOCKDIVIDER_30**
- **LCD_E_CLOCKDIVIDER_31**
- **LCD_E_CLOCKDIVIDER_32**

Referenced by LCD_E_init().

clockSource

`uint16_t LCD_E_initParam::clockSource`

Selects the clock that will be used by the LCD_E.
Valid values are:

- **LCD_E_CLOCKSOURCE_XTCLK** [Default]
- **LCD_E_CLOCKSOURCE_ACLK** [Default]
- **LCD_E_CLOCKSOURCE_VLOCLK**

Referenced by LCD_E_init().

muxRate

`uint16_t LCD_E_initParam::muxRate`

Selects LCD_E mux rate.
Valid values are:

- **LCD_E_STATIC** [Default]

- **LCD_E_2_MUX**
- **LCD_E_3_MUX**
- **LCD_E_4_MUX**
- **LCD_E_5_MUX**
- **LCD_E_6_MUX**
- **LCD_E_7_MUX**
- **LCD_E_8_MUX**

Referenced by LCD_E_init().

segments

`uint16_t LCD_E_initParam::segments`

Sets LCD segment on/off.
Valid values are:

- **LCD_E_SEGMENTS_DISABLED** [Default]
- **LCD_E_SEGMENTS_ENABLED**

Referenced by LCD_E_init().

waveforms

`uint16_t LCD_E_initParam::waveforms`

Selects LCD waveform mode.
Valid values are:

- **LCD_E_STANDARD_WAVEFORMS** [Default]
- **LCD_E_LOW_POWER_WAVEFORMS**

Referenced by LCD_E_init().

The documentation for this struct was generated from the following file:

- lcd_e.h

28.22 Timer_A_initUpDownModeParam Struct Reference

Used in the Timer_A_initUpDownMode() function as the param parameter.

`#include <timer_a.h>`

Data Fields

- uint16_t clockSource

- uint16_t clockSourceDivider
- uint16_t timerPeriod

  Is the specified Timer_A period.
- uint16_t timerInterruptEnable_TAIE
- uint16_t captureCompareInterruptEnable_CCR0_CCIE
- uint16_t timerClear
- bool startTimer

  Whether to start the timer immediately.

28.22.1 Detailed Description

Used in the Timer_A_initUpDownMode() function as the param parameter.

28.22.2 Field Documentation

captureCompareInterruptEnable_CCR0_CCIE

`uint16_t Timer_A_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE`

Is to enable or disable Timer_A CCR0 captureComapre interrupt.
Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpDownMode().

clockSource

`uint16_t Timer_A_initUpDownModeParam::clockSource`

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_initUpDownMode().

clockSourceDivider

`uint16_t Timer_A_initUpDownModeParam::clockSourceDivider`

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **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**

Referenced by Timer_A_initUpDownMode().

timerClear

`uint16_t Timer_A_initUpDownModeParam::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]

Referenced by Timer_A_initUpDownMode().

timerInterruptEnable_TAIE

`uint16_t Timer_A_initUpDownModeParam::timerInterruptEnable_TAIE`

Is to enable or disable Timer_A interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpDownMode().

The documentation for this struct was generated from the following file:

- timer_a.h

28.23 EComp_initParam Struct Reference

Used in the EComp_init() function as the param parameter.

```
#include <ecomp.h>
```

Data Fields

- uint8_t positiveTerminalInput
- uint8_t negativeTerminalInput
- uint16_t outputFilterEnableAndDelayLevel
- uint16_t invertedOutputPolarity

28.23.1 Detailed Description

Used in the EComp_init() function as the param parameter.

28.23.2 Field Documentation

invertedOutputPolarity

```
uint16_t EComp_initParam::invertedOutputPolarity
```

Controls if the output will be inverted or not
Valid values are:

- **ECOMP_NORMAL_OUTPUT_POLARITY** [Default]
- **ECOMP_INVERTED_OUTPUT_POLARITY**

Referenced by EComp_init().

negativeTerminalInput

```
uint8_t EComp_initParam::negativeTerminalInput
```

Selects the input to the negative terminal
Valid values are:

- **ECOMP_INPUT_0**
- **ECOMP_INPUT_1**
- **ECOMP_INPUT_2**
- **ECOMP_INPUT_3**
- **ECOMP_INPUT_DEVICE_SPECIFIC_0**
- **ECOMP_INPUT_DEVICE_SPECIFIC_1**
- **ECOMP_INPUT_DAC**
- **ECOMP_INPUT_DISABLED**

Referenced by EComp_init().

outputFilterEnableAndDelayLevel

`uint16_t EComp_initParam::outputFilterEnableAndDelayLevel`

Controls the output filter delay state, which is either off or enabled with a specified delay level. This parameter is device specific and delay levels should be found in the device's datasheet.
Valid values are:

- **ECOMP_FILTER_DELAY_OFF** [Default]
- **ECOMP_FILTER_DELAY_450NS**
- **ECOMP_FILTER_DELAY_900NS**
- **ECOMP_FILTER_DELAY_1800NS**
- **ECOMP_FILTER_DELAY_3600NS**

Referenced by EComp_init().

positiveTerminalInput

`uint8_t EComp_initParam::positiveTerminalInput`

Selects the input to the positive terminal
Valid values are:

- **ECOMP_INPUT_0**
- **ECOMP_INPUT_1**
- **ECOMP_INPUT_2**
- **ECOMP_INPUT_3**
- **ECOMP_INPUT_DEVICE_SPECIFIC_0**
- **ECOMP_INPUT_DEVICE_SPECIFIC_1**
- **ECOMP_INPUT_DAC**
- **ECOMP_INPUT_DISABLED**

Referenced by EComp_init().

The documentation for this struct was generated from the following file:

- ecomp.h

28.24 CS_initFLLParam Struct Reference

Used in the CS_initFLLCalculateTrim(), CS_initFLLLoadTrim() functions as the param parameter.

`#include <cs.h>`

Data Fields

- uint16_t csCtl0

Contains software trim value for DCOTAP.

- uint16_t csCtl1

  Contains software trim value for DCOFTRIM.

- uint16_t fsystem

  Is the target frequency for MCLK in kHz.

28.24.1 Detailed Description

Used in the CS_initFLLCalculateTrim(), CS_initFLLLoadTrim() functions as the param parameter.

The documentation for this struct was generated from the following file:

- cs.h

28.25 EUSCI_A_SPI_initSlaveParam Struct Reference

Used in the EUSCI_A_SPI_initSlave() function as the param parameter.

```
#include <eusci_a_spi.h>
```

Data Fields

- uint16_t msbFirst
- uint16_t clockPhase
- uint16_t clockPolarity
- uint16_t spiMode

28.25.1 Detailed Description

Used in the EUSCI_A_SPI_initSlave() function as the param parameter.

28.25.2 Field Documentation

clockPhase

```
uint16_t EUSCI_A_SPI_initSlaveParam::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**

Referenced by EUSCI_A_SPI_initSlave().

clockPolarity

`uint16_t EUSCI_A_SPI_initSlaveParam::clockPolarity`

Is clock polarity select
Valid values are:

- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by EUSCI_A_SPI_initSlave().

msbFirst

`uint16_t EUSCI_A_SPI_initSlaveParam::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]

Referenced by EUSCI_A_SPI_initSlave().

spiMode

`uint16_t EUSCI_A_SPI_initSlaveParam::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**

Referenced by EUSCI_A_SPI_initSlave().

The documentation for this struct was generated from the following file:

- eusci_a_spi.h

28.26 Timer_A_outputPWMParam Struct Reference

Used in the Timer_A_outputPWM() function as the param parameter.

`#include <timer_a.h>`

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod

  Selects the desired timer period.
- uint16_t compareRegister
- uint16_t compareOutputMode
- uint16_t dutyCycle

  Specifies the dutycycle for the generated waveform.

28.26.1 Detailed Description

Used in the Timer_A_outputPWM() function as the param parameter.

28.26.2 Field Documentation

clockSource

```
uint16_t Timer_A_outputPWMParam::clockSource
```

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_outputPWM().

clockSourceDivider

```
uint16_t Timer_A_outputPWMParam::clockSourceDivider
```

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**

- **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**

Referenced by Timer_A_outputPWM().

compareOutputMode

`uint16_t Timer_A_outputPWMParam::compareOutputMode`

Specifies the output mode.
Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **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**

Referenced by Timer_A_outputPWM().

compareRegister

`uint16_t Timer_A_outputPWMParam::compareRegister`

Selects the compare register being used. Refer to datasheet to ensure the device has the capture compare register being used.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_REGISTER_0**
- **TIMER_A_CAPTURECOMPARE_REGISTER_1**
- **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Referenced by Timer_A_outputPWM().

The documentation for this struct was generated from the following file:

- timer_a.h

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265