TEXAS INSTRUMENTS

MSP430® Peripheral Driver Library for F5xx and F6xx Devices

USER'S GUIDE

Copyright

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
http://www.ti.com/msp430

Revision Information

This is version 1.90.00.65 of this document, last updated on 2014-06-25.

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 5xx/6xx 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:

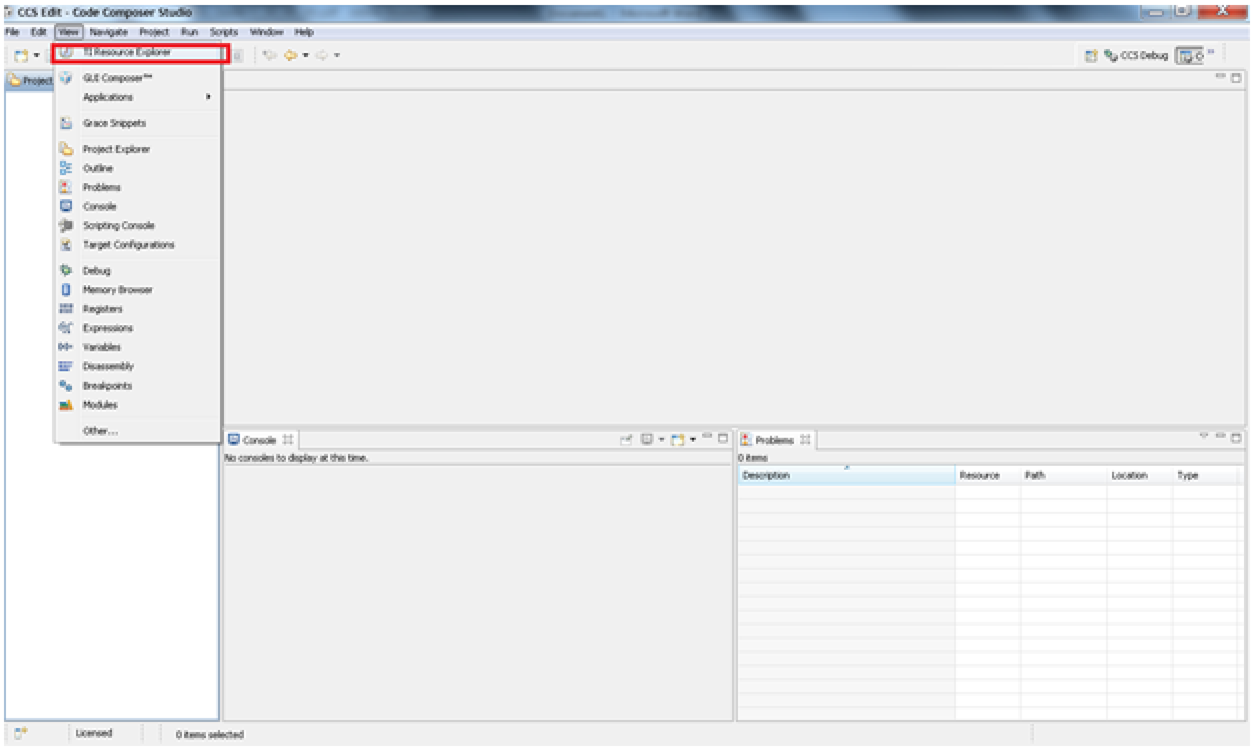
- IAR Embedded Workbench®
- Texas Instruments Code Composer Studio™

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

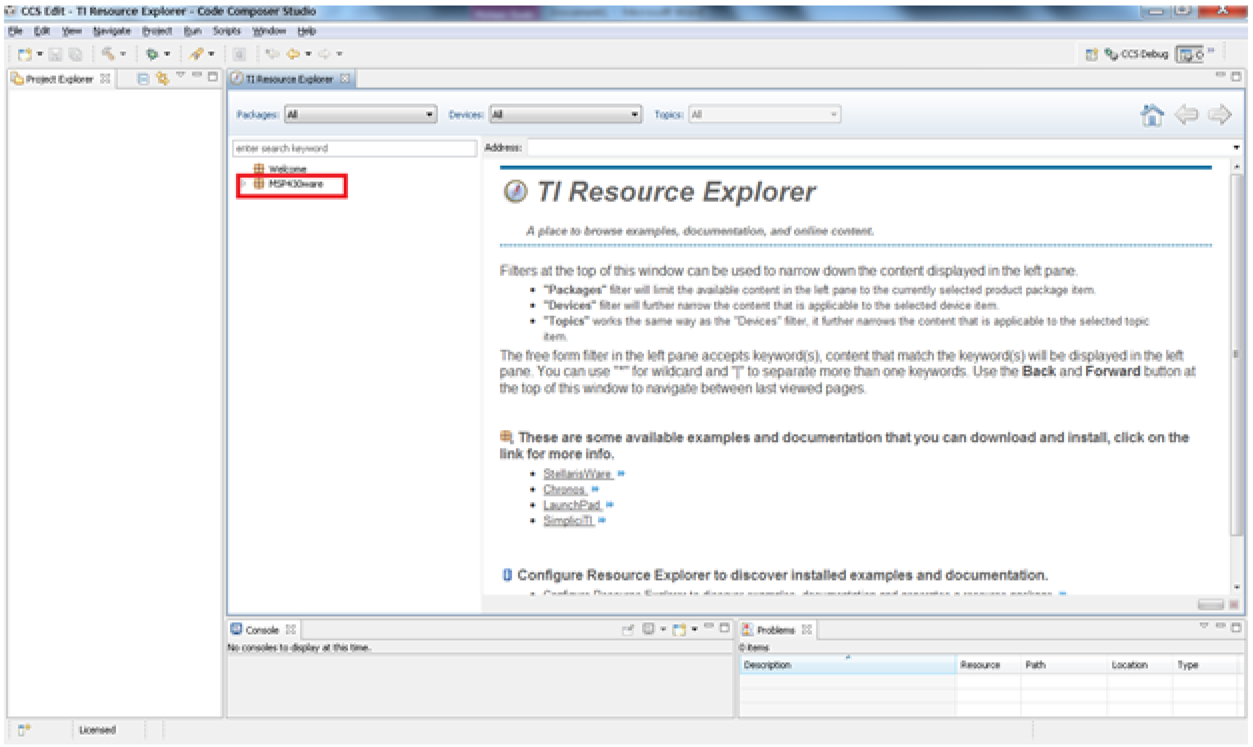


No consoles to display at this time.

0 items

0 items selected

In Resource Explorer View, click on MSP430ware



TI Resource Explorer

A place to browse examples, documentation, and online content.

Filters at the top of this window can be used to narrow down the content displayed in the left pane.

- **"Packages"** filter will limit the available content in the left pane to the currently selected product package item.
- **"Devices"** filter will further narrow the content that is applicable to the selected device item.
- **"Topics"** works the same way as the "Devices" filter, it further narrows the content that is applicable to the selected topic item.

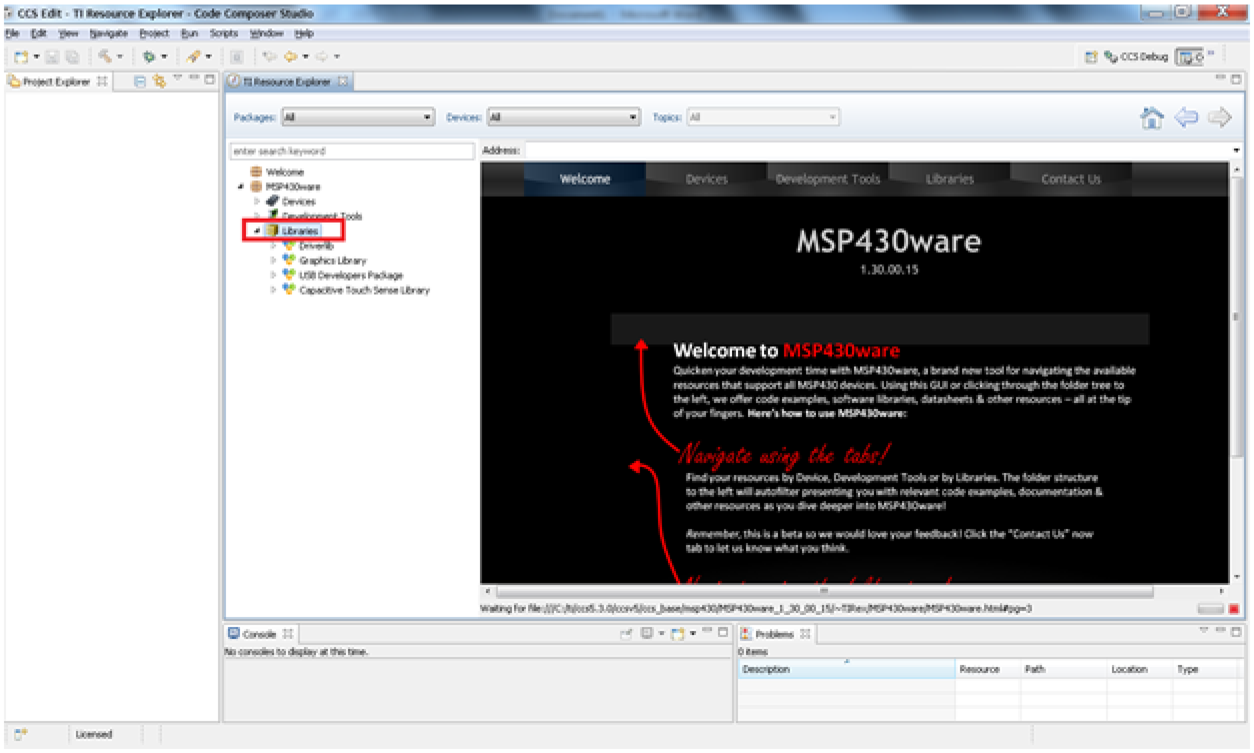
The free form filter in the left pane accepts keyword(s), content that match the keyword(s) will be displayed in the left pane. You can use "*" for wildcard and '|' to separate more than one keywords. Use the **Back** and **Forward** button at the top of this window to navigate between last viewed pages.

These are some available examples and documentation that you can download and install, click on the link for more info.

- StellarisWare
- Chronos
- LaunchPad
- SimpliciTI

Configure Resource Explorer to discover installed examples and documentation.

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.



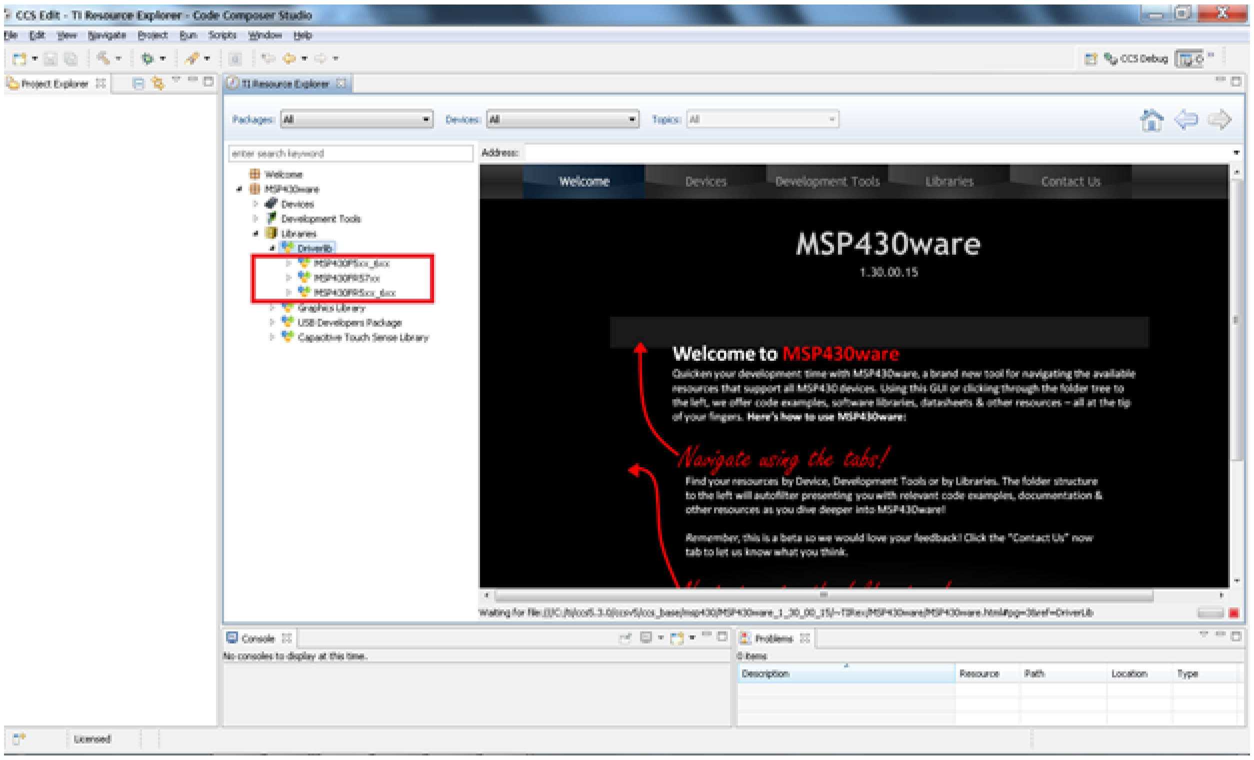
Welcome to MSP430ware

Quicken your development time with MSP430ware, a brand new tool for navigating the available resources that support all MSP430 devices. Using this GUI or clicking through the folder tree to the left, we offer code examples, software libraries, datasheets & other resources – all at the tip of your fingers. **Here's how to use MSP430ware:**

Find your resources by Device, Development Tools or by Libraries. The folder structure to the left will autofilter presenting you with relevant code examples, documentation & other resources as you dive deeper into MSP430ware!

Remember, this is a beta so we would love your feedback! Click the "Contact Us" now tab to let us know what you think.

Waiting for file:///C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/~TIRex/MSP430ware/MSP430ware.html#pg=3



Welcome to MSP430ware

Quicken your development time with MSP430ware, a brand new tool for navigating the available resources that support all MSP430 devices. Using this GUI or clicking through the folder tree to the left, we offer code examples, software libraries, datasheets & other resources – all at the tip of your fingers. **Here's how to use MSP430ware:**

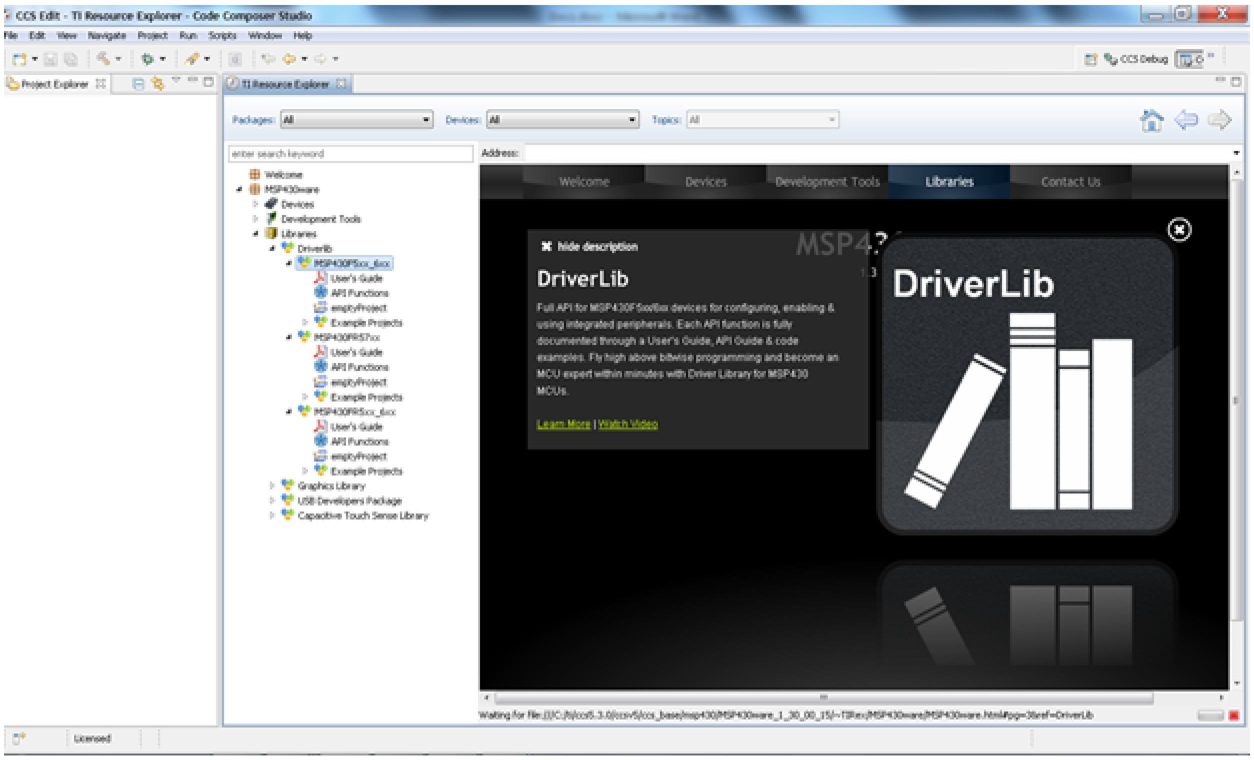
Navigate using the tabs!

Find your resources by Device, Development Tools or by Libraries. The folder structure to the left will autofilter presenting you with relevant code examples, documentation & other resources as you dive deeper into MSP430ware!

Remember, this is a beta so we would love your feedback! Click the "Contact Us" now tab to let us know what you think.

Waiting for file:///C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/~TIRex/MSP430ware/MSP430ware.html#pg=3&ref=DriverLib

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 MSP430FR5xx_6xx MSP430i2xx

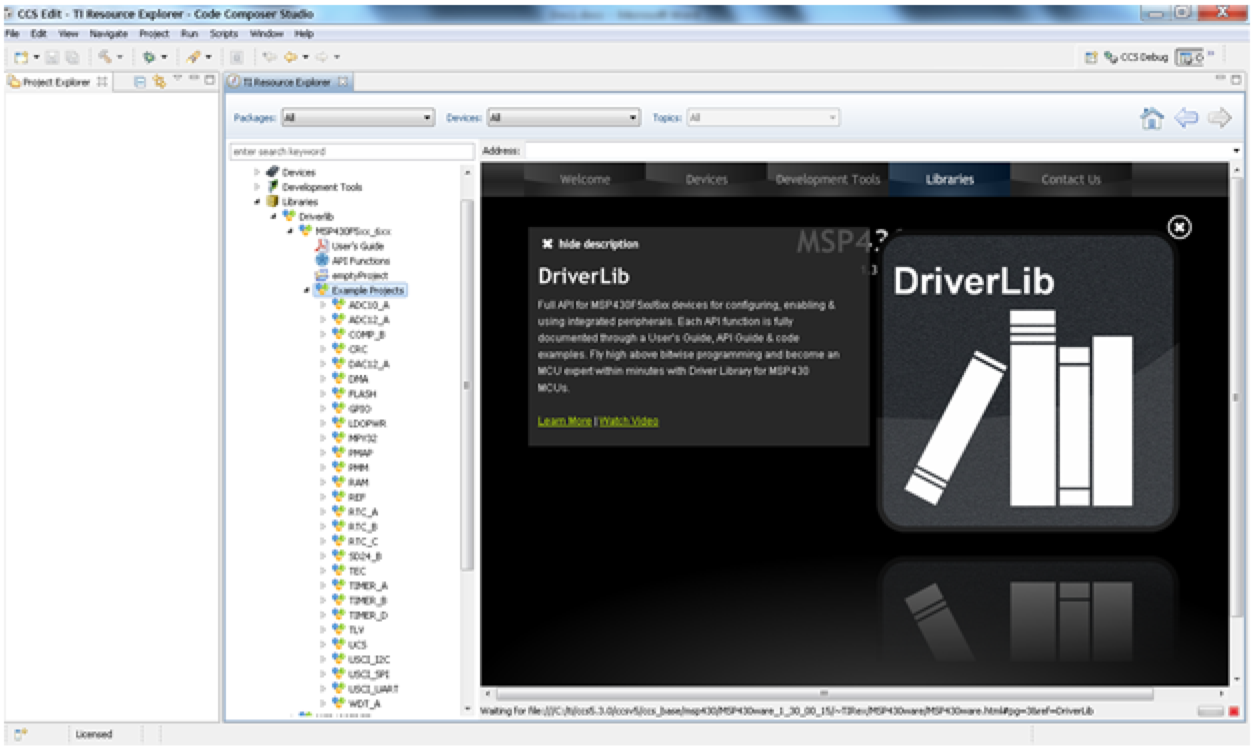


DriverLib

Full API for MSP430F5xx/6xx devices for configuring, enabling & using integrated peripherals. Each API function is fully documented through a User's Guide, API Guide & code examples. Fly high above bitwise programming and become an MCU expert within minutes with Driver Library for MSP430 MCUs.

Waiting for file:///C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/~TIRex/MSP430ware/MSP430ware.html#pg=3&ref=DriverLib

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

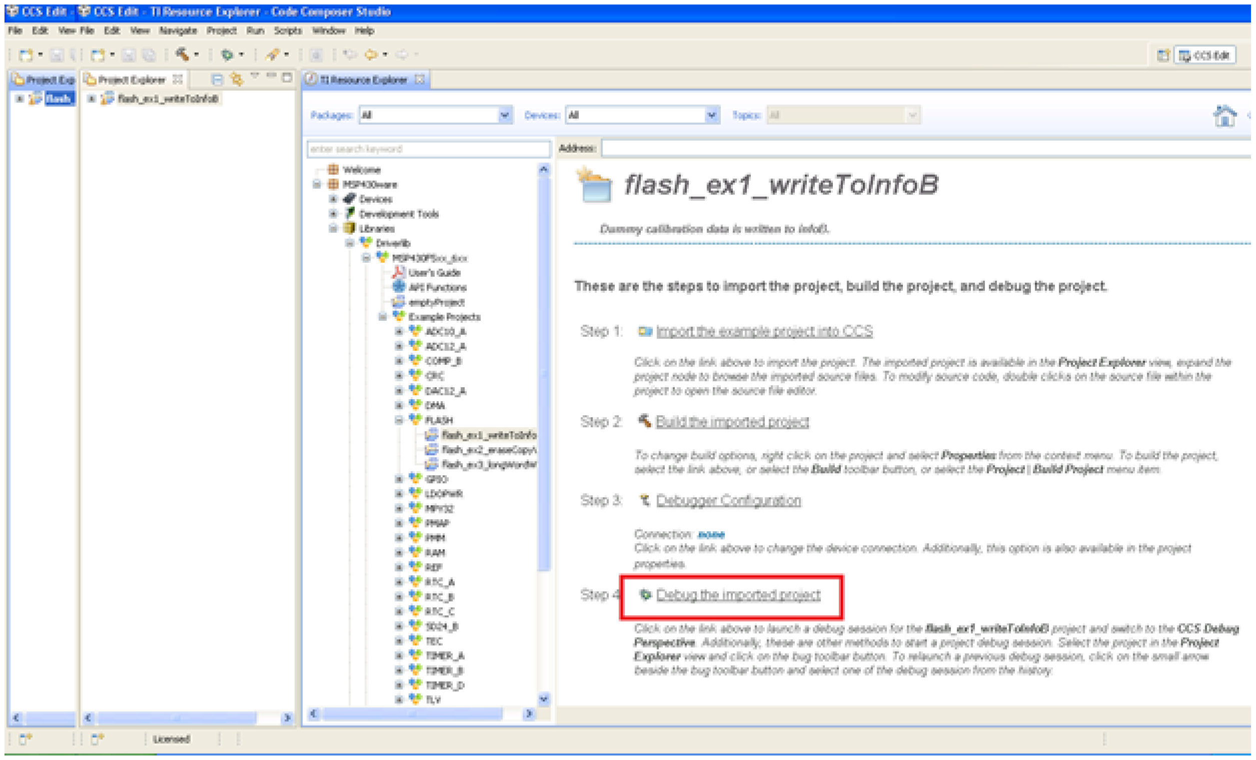


DriverLib

Full API for MSP430F5xx/6xx devices for configuring, enabling & using integrated peripherals. Each API function is fully documented through a User's Guide, API Guide & code examples. Fly high above bitwise programming and become an MCU expert within minutes with Driver Library for MSP430 MCUs.

Waiting for file:///C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/~TIRex/MSP430ware/MSP430ware.html#pg=3&ref=DriverLib

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.



flash_ex1_writeToInfoB

Dummy calibration data is written to infoB.

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

*Click on the link above to import the project. The imported project is available in the **Project Explorer** view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.*

Step 2: Build the imported project

*To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** toolbar button, or select the **Project | Build Project** menu item.*

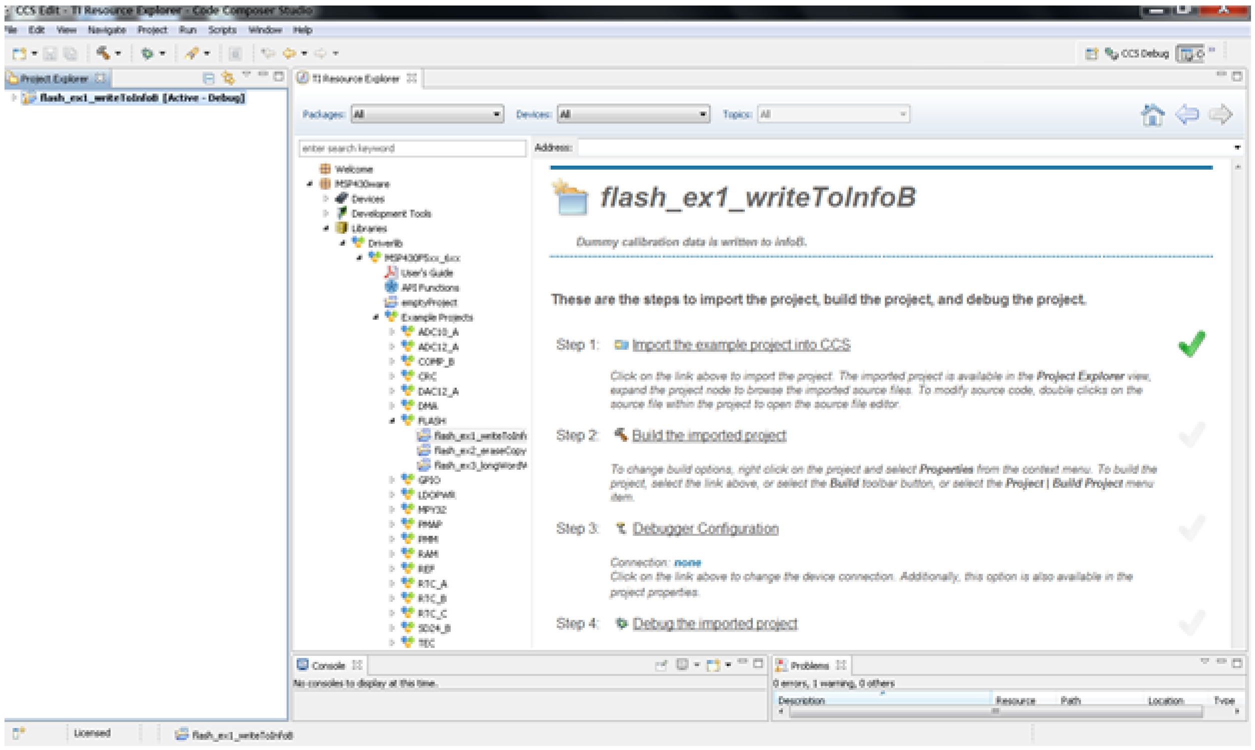
Step 3: Debugger Configuration

Connection: **none**
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: Debug the imported project

*Click on the link above to launch a debug session for the **flash_ex1_writeToInfoB** project and switch to the **CCS Debug Perspective**. Additionally, these are other methods to start a project debug session. Select the project in the **Project Explorer** view and click on the bug toolbar button. To relaunch a previous debug session, click on the small arrow beside the bug toolbar button and select one of the debug session from the history.*

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"



flash_ex1_writeToInfoB

Dummy calibration data is written to infoB.

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS ✓

*Click on the link above to import the project. The imported project is available in the **Project Explorer** view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.*

Step 2: Build the imported project

*To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** toolbar button, or select the **Project | Build Project** menu item.*

Step 3: Debugger Configuration

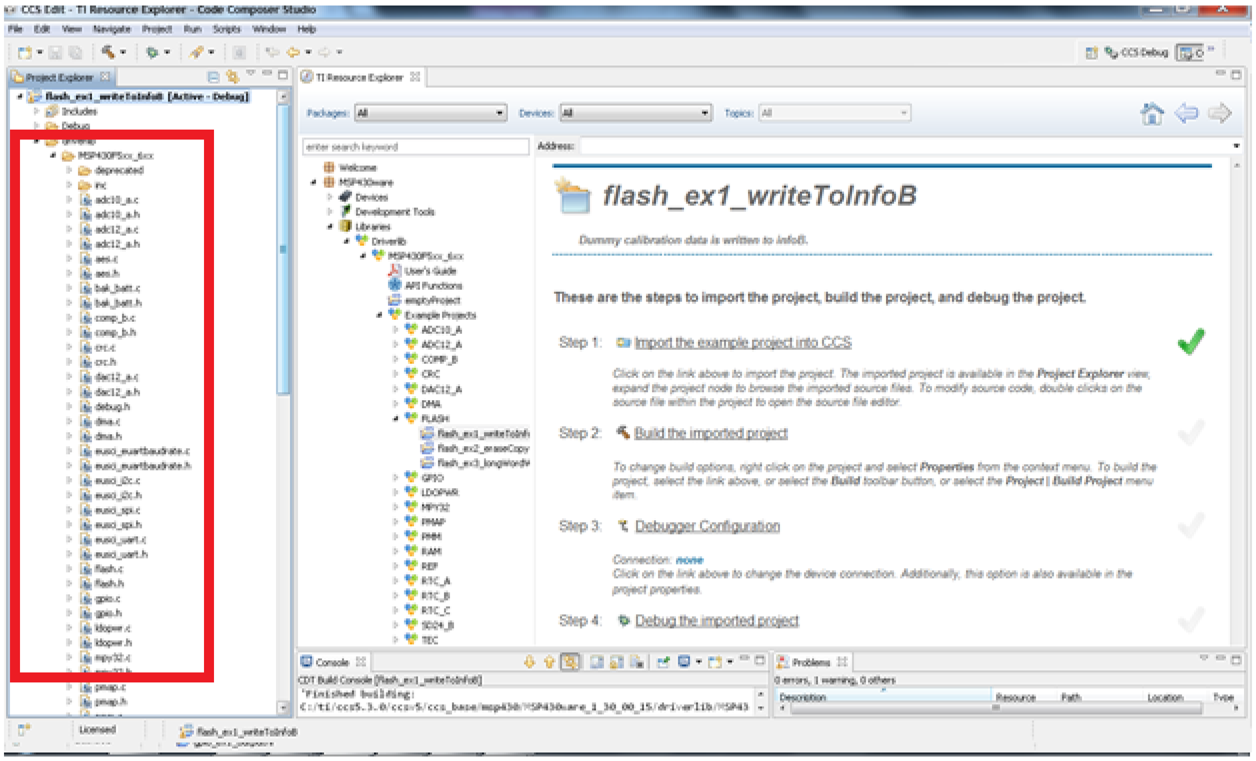
Connection: **none**
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: Debug the imported project

No consoles to display at this time.

0 errors, 1 warning, 0 others

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.



flash_ex1_writeToInfoB

Dummy calibration data is written to infoB.

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

Click on the link above to import the project. The imported project is available in the ***Project Explorer*** *view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.*

Step 2: Build the imported project

To change build options, right click on the project and select ***Properties*** *from the context menu. To build the project, select the link above, or select the* ***Build*** *toolbar button, or select the* ***Project | Build Project*** *menu item.*

Step 3: Debugger Configuration

Connection: none

Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

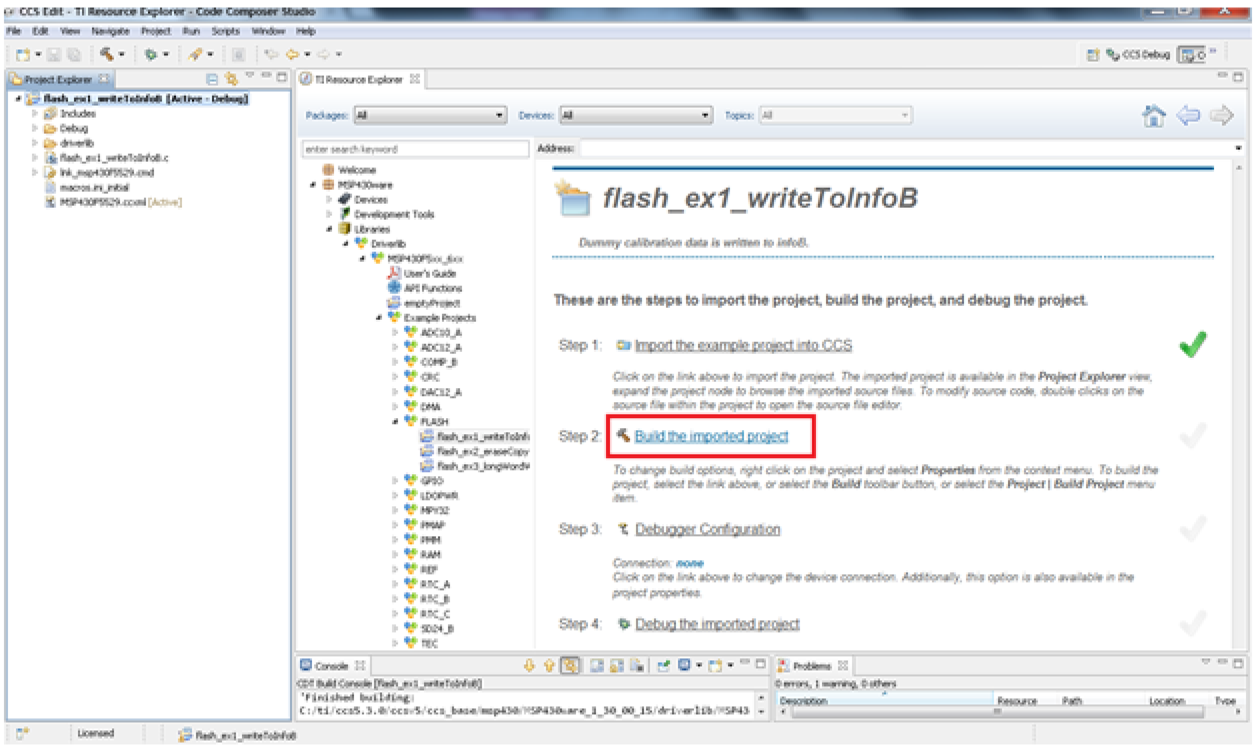
Step 4: Debug the imported project

CDT Build Console [flash_ex1_writeToInfoB]

```
'Finished building:
C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/driverlib/MSP43
```

0 errors, 1 warning, 0 others

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


CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
TI Resource Explorer
Packages: All
Devices: All
Topics: All
Address:

flash_ex1_writeToInfoB

Dummy calibration data is written to infoB.

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

Click on the link above to import the project. The imported project is available in the **Project Explorer** view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.

Step 2: Build the imported project

To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** toolbar button, or select the **Project | Build Project** menu item.

Step 3: Debugger Configuration

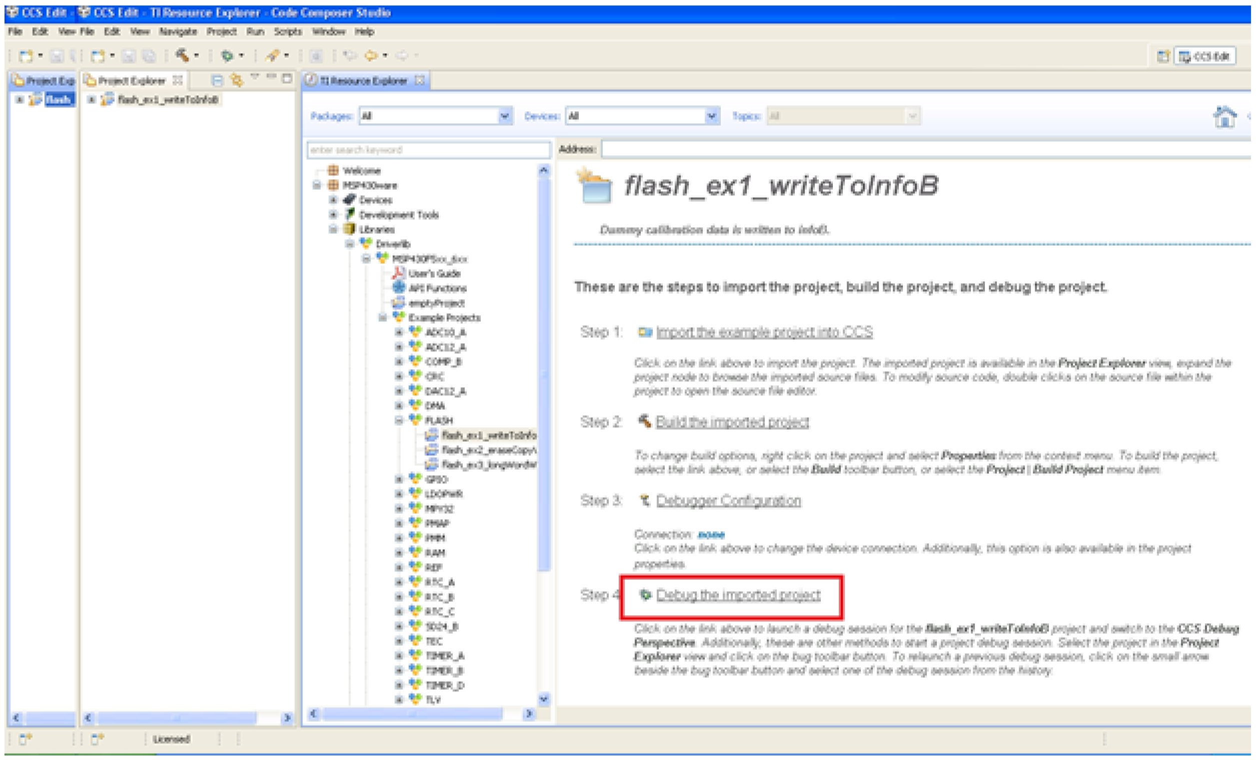
Connection: none
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: Debug the imported project

```
CDT Build Console [flash_ex1_writeToInfoB]
'Finished building:
C:/ti/ccs5.3.0/ccsv5/ccs_base/msp430/MSP430ware_1_30_00_15/driverlib/MSP43
```

0 errors, 1 warning, 0 others

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



flash_ex1_writeToInfoB

Dummy calibration data is written to infoB.

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

*Click on the link above to import the project. The imported project is available in the **Project Explorer** view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.*

Step 2: Build the imported project

*To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** toolbar button, or select the **Project | Build Project** menu item.*

Step 3: Debugger Configuration

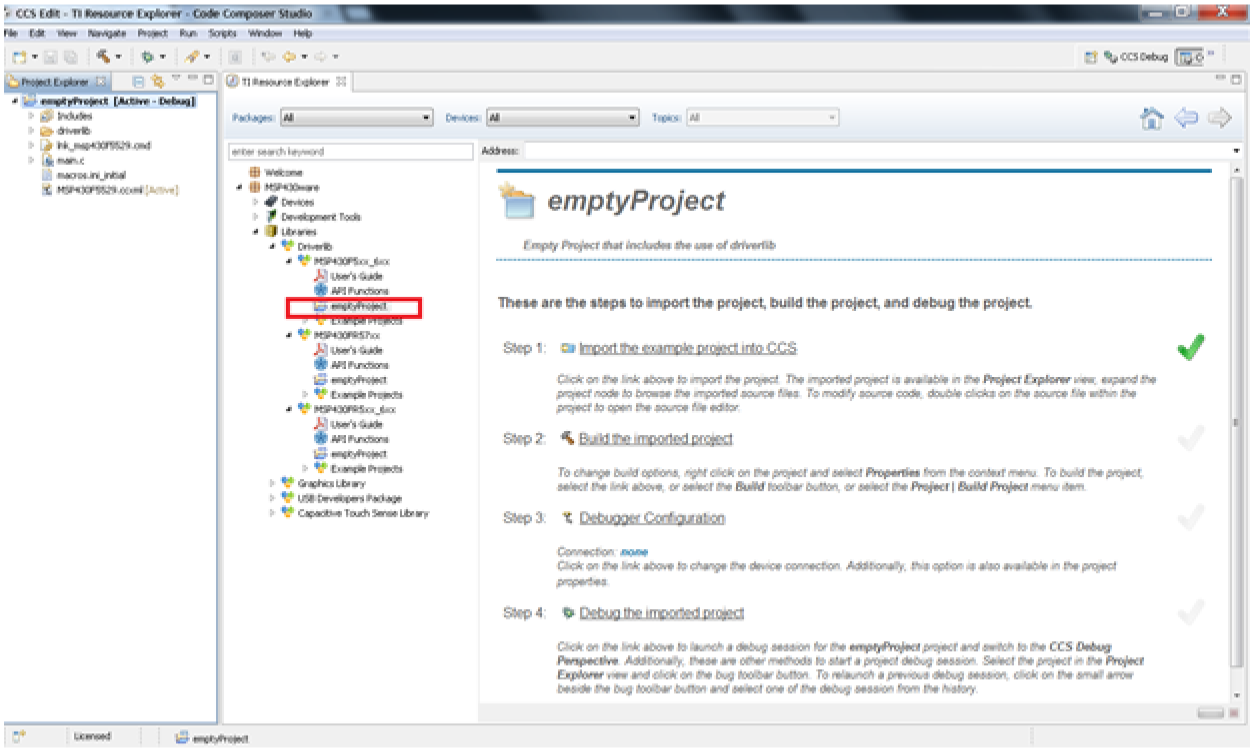
Connection: none
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: Debug the imported project

*Click on the link above to launch a debug session for the **flash_ex1_writeToInfoB** project and switch to the **CCS Debug Perspective**. Additionally, these are other methods to start a project debug session. Select the project in the **Project Explorer** view and click on the bug toolbar button. To relaunch a previous debug session, click on the small arrow beside the bug toolbar button and select one of the debug session from the history.*

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.



emptyProject

Empty Project that includes the use of driverlib

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

Click on the link above to import the project. The imported project is available in the ***Project Explorer*** *view, expand the project node to browse the imported source files. To modify source code, double clicks on the source file within the project to open the source file editor.*

Step 2: Build the imported project

To change build options, right click on the project and select ***Properties*** *from the context menu. To build the project, select the link above, or select the* ***Build*** *toolbar button, or select the* ***Project | Build Project*** *menu item.*

Step 3: Debugger Configuration

Connection: none
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: Debug the imported project

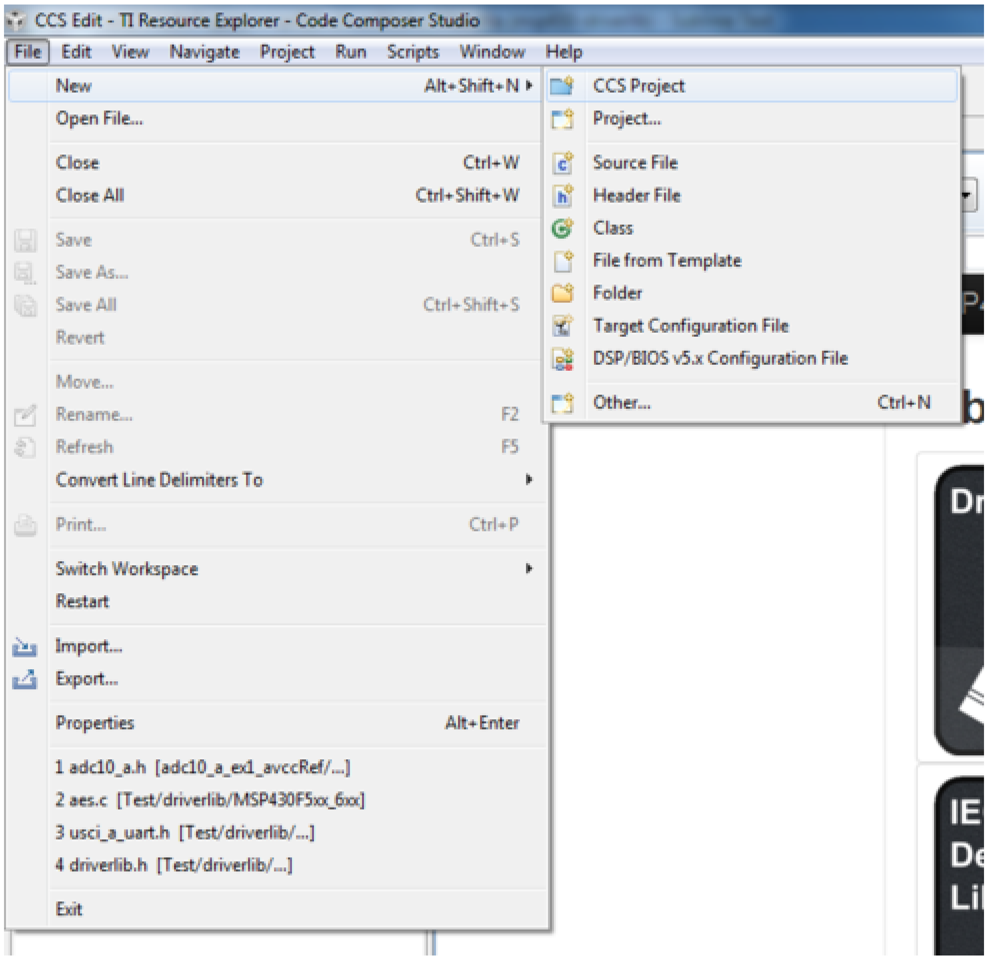
Click on the link above to launch a debug session for the ***emptyProject*** *project and switch to the* ***CCS Debug Perspective****. Additionally, these are other methods to start a project debug session. Select the project in the* ***Project Explorer*** *view and click on the bug toolbar button. To relaunch a previous debug session, click on the small arrow beside the bug toolbar button and select one of the debug session from the history.*

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

3 How to create a new user project that uses Driverlib

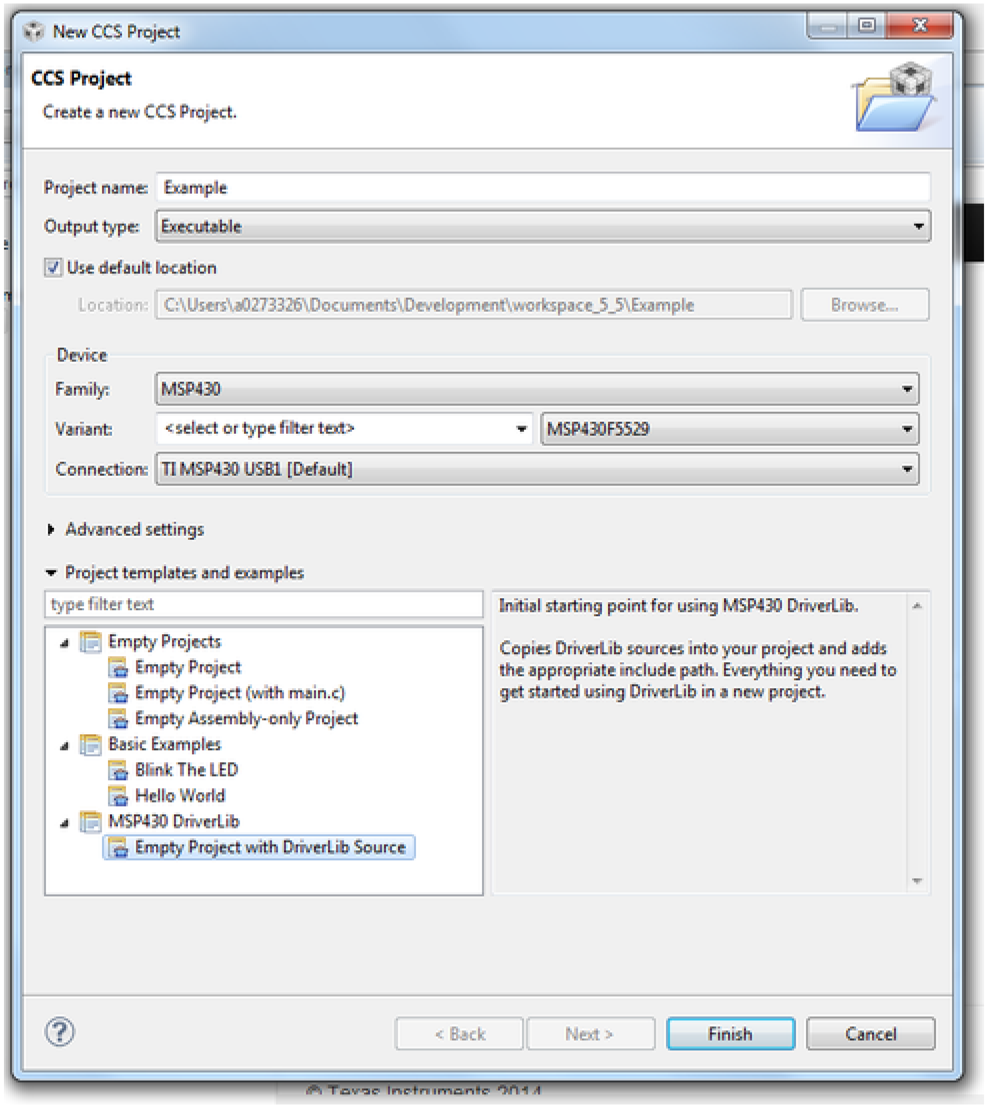
To get started on a new project we recommend using the new project wizard. For driver library to work with the new project wizard CCS must have discovered the driver library RTSC product. For more information refer to the installation steps of the release notes. The new project wizard adds the needed 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.


CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 adc10_a.h [adc10_a_ex1_avccRef/...]
2 aes.c [Test/driverlib/MSP430F5xx_6xx]
3 usci_a_uart.h [Test/driverlib/...]
4 driverlib.h [Test/driverlib/...]
Exit
CCS Project
Project...
Source File
Header File
Class
File from Template
Folder
Target Configuration File
DSP/BIOS v5.x Configuration File
Other... Ctrl+N

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.



Project name: Example
Output type: Executable
[✓] Use default location
Location: C:\Users\a0273326\Documents\Development\workspace_5_5\Example

Device
Family: MSP430
Variant: <select or type filter text> MSP430F5529
Connection: TI MSP430 USB1 [Default]

Initial starting point for using MSP430 DriverLib.

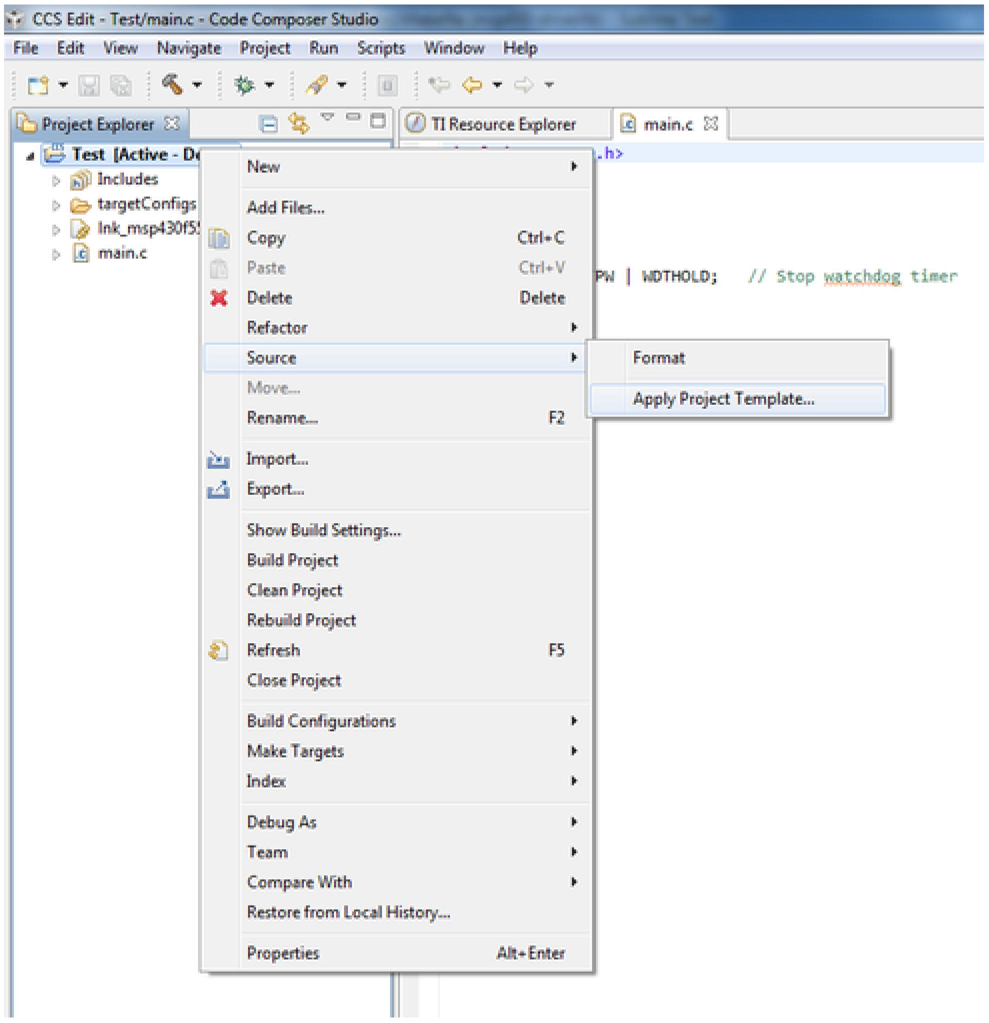
Copies DriverLib sources into your project and adds the appropriate include path. Everything you need to get started using DriverLib in a new project.

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

4 How to include driverlib into your existing project

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.

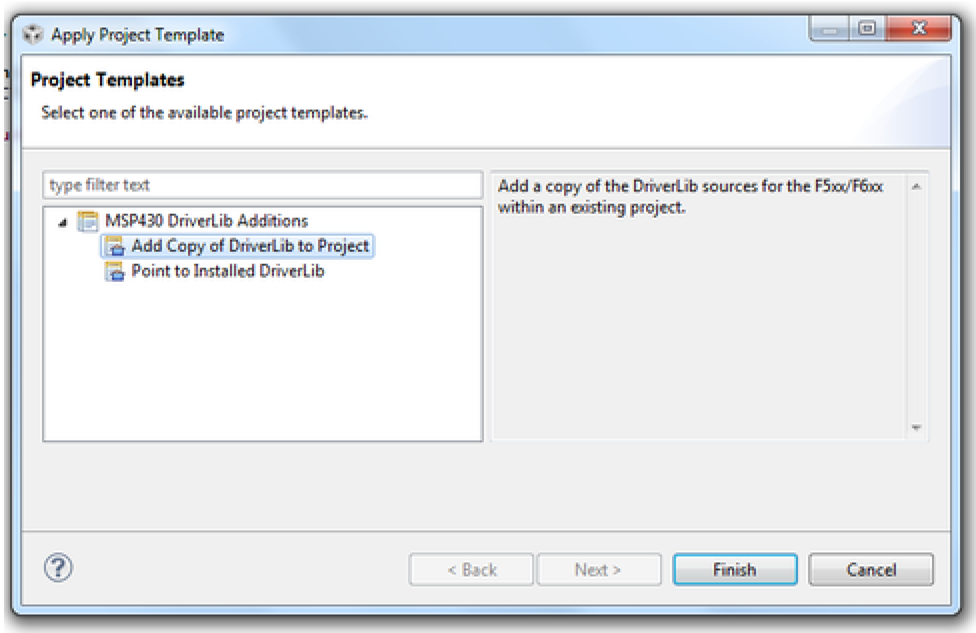
To apply a project template right click on an existing project then go to Source -> Apply Project Template as seen in the screenshot below.



```
PW | WDTHOLD;    // Stop watchdog timer
```

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.


Apply Project Template
Project Templates
Select one of the available project templates.
type filter text
MSP430 DriverLib Additions
Add Copy of DriverLib to Project
Point to Installed DriverLib
< Back
Next >
Finish
Cancel

Add a copy of the DriverLib sources for the F5xx/F6xx within an existing project.

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

5 10-Bit Analog-to-Digital Converter (ADC10_A)

5.1 Introduction

The 10-Bit Analog-to-Digital (ADC10_A) API provides a set of functions for using the MSP430Ware ADC10_A modules. Functions are provided to initialize the ADC10_A modules, setup signal sources and reference voltages, and manage interrupts for the ADC10_A modules.

The ADC10_A module provides the ability to convert analog signals into a digital value in respect to given reference voltages. The ADC10_A can generate digital values from 0 to Vcc with an 8- or 10-bit resolution. It operates in 2 different sampling modes, and 4 different conversion modes. The sampling modes are extended sampling and pulse sampling, in extended sampling the sample/hold signal must stay high for the duration of sampling, while in pulse mode a sampling timer is setup to start on a rising edge of the sample/hold signal and sample for a specified amount of clock cycles. The 4 conversion modes are single-channel single conversion, sequence of channels single-conversion, repeated single channel conversions, and repeated sequence of channels conversions.

The ADC10_A module can generate multiple interrupts. An interrupt can be asserted when a conversion is complete, when a conversion is about to overwrite the converted data in the memory buffer before it has been read out, and/or when a conversion is about to start before the last conversion is complete. The ADC10_A also has a window comparator feature which asserts interrupts when the input signal is above a high threshold, below a low threshold, or between the two at any given moment.

This driver is contained in `adc10_a.c`, with `adc10_a.h` containing the API definitions for use by applications.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	720
TI Compiler 4.2.1	Size	286
TI Compiler 4.2.1	Speed	286
IAR 5.51.6	None	432
IAR 5.51.6	Size	310
IAR 5.51.6	Speed	310
MSPGCC 4.8.0	None	1076
MSPGCC 4.8.0	Size	308
MSPGCC 4.8.0	Speed	310

5.2 API Functions

Functions

- void ADC10_A_clearInterrupt (uint16_t baseAddress, uint8_t interruptFlagMask)
- void ADC10_A_disable (uint16_t baseAddress)
- void ADC10_A_disableConversions (uint16_t baseAddress, bool preempt)
- void ADC10_A_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- void ADC10_A_disableReferenceBurst (uint16_t baseAddress)
- void ADC10_A_disableSamplingTimer (uint16_t baseAddress)
- void ADC10_A_enable (uint16_t baseAddress)
- void ADC10_A_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- void ADC10_A_enableReferenceBurst (uint16_t baseAddress)
- uint8_t ADC10_A_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
- uint32_t ADC10_A_getMemoryAddressForDMA (uint16_t baseAddress)
- int16_t ADC10_A_getResults (uint16_t baseAddress)
- bool ADC10_A_init (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
- uint16_t ADC10_A_isBusy (uint16_t baseAddress)
- void ADC10_A_memoryConfigure (uint16_t baseAddress, uint8_t inputSourceSelect, uint8_t positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect)
- void ADC10_A_setDataReadBackFormat (uint16_t baseAddress, uint16_t readBackFormat)
- void ADC10_A_setReferenceBufferSamplingRate (uint16_t baseAddress, uint16_t samplingRateSelect)
- void ADC10_A_setResolution (uint16_t baseAddress, uint8_t resolutionSelect)
- void ADC10_A_setSampleHoldSignalInversion (uint16_t baseAddress, uint16_t invertedSignal)
- void ADC10_A_setupSamplingTimer (uint16_t baseAddress, uint16_t clockCycleHoldCount, uint16_t multipleSamplesEnabled)
- void ADC10_A_setWindowComp (uint16_t baseAddress, uint16_t highThreshold, uint16_t lowThreshold)
- void ADC10_A_startConversion (uint16_t baseAddress, uint8_t conversionSequenceModeSelect)

5.2.1 Detailed Description

The ADC10_A 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 ADC10_A.

The ADC10_A initialization and conversion functions are

- ADC10_A_init()
- ADC10_A_memoryConfigure()
- ADC10_A_setupSamplingTimer()
- ADC10_A_disableSamplingTimer()
- ADC10_A_setWindowComp()
- ADC10_A_startConversion()
- ADC10_A_disableConversions()
- ADC10_A_getResults()
- ADC10_A_isBusy()

The ADC10_A interrupts are handled by

- ADC10_A_enableInterrupt()
- ADC10_A_disableInterrupt()
- ADC10_A_clearInterrupt()
- ADC10_A_getInterruptStatus()

Auxiliary features of the ADC10_A are handled by

- ADC10_A_setResolution()
- ADC10_A_setSampleHoldSignalInversion()
- ADC10_A_setDataReadBackFormat()
- ADC10_A_enableReferenceBurst()
- ADC10_A_disableReferenceBurst()
- ADC10_A_setReferenceBufferSamplingRate()
- ADC10_A_getMemoryAddressForDMA()
- ADC10_A_enable()
- ADC10_A_disable()

5.2.2 Function Documentation

5.2.2.1 void ADC10_A_clearInterrupt (uint16_t *baseAddress*, uint8_t *interruptFlagMask*)

Clears ADC10_A selected interrupt flags.

The selected ADC10_A interrupt flags are cleared, so that it no longer asserts. The memory buffer interrupt flags are only cleared when the memory buffer is accessed.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

baseAddress is the base address of the ADC10_A module.

interruptFlagMask is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:

- **ADC10_A_TIMEOVERFLOW_INTFLAG** - Interrupts flag when a new conversion is starting before the previous one has finished
- **ADC10_A_OVERFLOW_INTFLAG** - Interrupts flag when a new conversion is about to overwrite the previous one
- **ADC10_A_ABOVETHRESHOLD_INTFLAG** - Interrupts flag when the input signal has gone above the high threshold of the window comparator
- **ADC10_A_BELOWTHRESHOLD_INTFLAG** - Interrupts flag when the input signal has gone below the low threshold of the low window comparator
- **ADC10_A_INSIDEWINDOW_INTFLAG** - Interrupts flag when the input signal is in between the high and low thresholds of the window comparator
- **ADC10_A_COMPLETED_INTFLAG** - Interrupt flag for new conversion data in the memory buffer

Modified bits of **ADC10IFG** register.

Returns:

None

5.2.2.2 void ADC10_A_disable (uint16_t *baseAddress*)

Disables the ADC10_A block.

This will disable operation of the ADC10_A block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC10_A module.

Modified bits are **ADC10ON** of **ADC10CTL0** register.

Returns:
None

5.2.2.3 void ADC10_A_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 immediately if the preempt parameter is set as ADC10_A_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 ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	40
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	86
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	34

Parameters:
baseAddress is the base address of the ADC10_A module.

preempt specifies if the current conversion should be pre-empted before the end of the conversion Valid values are:
- **ADC10_A_COMPLETECONVERSION** - Allows the ADC10_A to end the current conversion before disabling conversions.
- **ADC10_A_PREEMPTCONVERSION** - Stops the ADC10_A immediately, with unpredictable results of the current conversion. Cannot be used with repeated conversion.

Modified bits of **ADC10CTL1** register and bits of **ADC10CTL0** register.

Returns:
None

5.2.2.4 void ADC10_A_disableInterrupt (uint16_t *baseAddress*, uint8_t *interruptMask*)

Disables selected ADC10_A interrupt sources.

Disables the indicated ADC10_A interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the ADC10_A 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:
- **ADC10_A_TIMEOVERFLOW_INT** - Interrupts when a new conversion is starting before the previous one has finished
- **ADC10_A_OVERFLOW_INT** - Interrupts when a new conversion is about to overwrite the previous one
- **ADC10_A_ABOVETHRESHOLD_INT** - Interrupts when the input signal has gone above the high threshold of the window comparator
- **ADC10_A_BELOWTHRESHOLD_INT** - Interrupts when the input signal has gone below the low threshold of the low window comparator
- **ADC10_A_INSIDEWINDOW_INT** - Interrupts when the input signal is in between the high and low thresholds of the window comparator
- **ADC10_A_COMPLETED_INT** - Interrupt for new conversion data in the memory buffer

Modified bits of **ADC10IE** register.

Returns:
None

5.2.2.5 void ADC10_A_disableReferenceBurst (uint16_t *baseAddress*)

Disables the reference buffer's burst ability.

Disables the reference buffer's burst ability, forcing the reference buffer to remain on continuously.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the ADC10_A module.

Returns:
None

5.2.2.6 void ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC10_A module.

Returns:
None

5.2.2.7 void ADC10_A_enable (uint16_t *baseAddress*)

Enables the ADC10_A block.

This will enable operation of the ADC10_A block.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC10_A module.

Modified bits are **ADC10ON** of **ADC10CTL0** register.

Returns:
None

5.2.2.8 void ADC10_A_enableInterrupt (uint16_t *baseAddress*, uint8_t *interruptMask*)

Enables selected ADC10_A interrupt sources.

Enables the indicated ADC10_A 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

baseAddress is the base address of the ADC10_A module.

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:

- **ADC10_A_TIMEOVERFLOW_INT** - Interrupts when a new conversion is starting before the previous one has finished
- **ADC10_A_OVERFLOW_INT** - Interrupts when a new conversion is about to overwrite the previous one
- **ADC10_A_ABOVETHRESHOLD_INT** - Interrupts when the input signal has gone above the high threshold of the window comparator
- **ADC10_A_BELOWTHRESHOLD_INT** - Interrupts when the input signal has gone below the low threshold of the low window comparator
- **ADC10_A_INSIDEWINDOW_INT** - Interrupts when the input signal is in between the high and low thresholds of the window comparator
- **ADC10_A_COMPLETED_INT** - Interrupt for new conversion data in the memory buffer

Modified bits of **ADC10IE** register.

Returns:

None

5.2.2.9 void ADC10_A_enableReferenceBurst (uint16_t *baseAddress*)

Enables the reference buffer's burst ability.

Enables the reference buffer's burst ability, allowing the reference buffer to turn off while the ADC is not converting, and automatically turning on when the ADC needs the generated reference voltage for a conversion.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the ADC10_A module.

Returns:

None

5.2.2.10 uint8_t ADC10_A_getInterruptStatus (uint16_t *baseAddress*, uint8_t *interruptFlagMask*)

Returns the status of the selected memory interrupt flags.

Returns the status of the selected interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the ADC10_A module.

interruptFlagMask is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following:

- **ADC10_A_TIMEOVERFLOW_INTFLAG** - Interrupts flag when a new conversion is starting before the previous one has finished
- **ADC10_A_OVERFLOW_INTFLAG** - Interrupts flag when a new conversion is about to overwrite the previous one
- **ADC10_A_ABOVETHRESHOLD_INTFLAG** - Interrupts flag when the input signal has gone above the high threshold of the window comparator

- **ADC10_A_BELOWTHRESHOLD_INTFLAG** - Interrupts flag when the input signal has gone below the low threshold of the low window comparator
- **ADC10_A_INSIDEWINDOW_INTFLAG** - Interrupts flag when the input signal is in between the high and low thresholds of the window comparator
- **ADC10_A_COMPLETED_INTFLAG** - Interrupt flag for new conversion data in the memory buffer

Returns:
The current interrupt flag status for the corresponding mask.

5.2.2.11 uint32_t ADC10_A_getMemoryAddressForDMA (uint16_t *baseAddress*)

Returns the address of the memory buffer for the DMA 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC10_A module.

Returns:
The memory address of the memory buffer

5.2.2.12 int16_t ADC10_A_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-significant 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the ADC10_A module.

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

5.2.2.13 bool ADC10_A_init (uint16_t *baseAddress*, uint16_t *sampleHoldSignalSourceSelect*, uint8_t *clockSourceSelect*, uint16_t *clockSourceDivider*)

Initializes the ADC10_A 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	108
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	94
IAR 5.51.6	Size	76
IAR 5.51.6	Speed	76
MSPGCC 4.8.0	None	146
MSPGCC 4.8.0	Size	64
MSPGCC 4.8.0	Speed	64

Parameters:
baseAddress is the base address of the ADC10_A 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:

- **ADC10_A_SAMPLEHOLDSOURCE_SC**
- **ADC10_A_SAMPLEHOLDSOURCE_1**
- **ADC10_A_SAMPLEHOLDSOURCE_2**
- **ADC10_A_SAMPLEHOLDSOURCE_3**
  Modified bits are **ADC10SHSx** of **ADC10CTL1** register.

clockSourceSelect selects the clock that will be used by the ADC10_A core and the sampling timer if a sampling pulse mode is enabled. Valid values are:

- **ADC10_A_CLOCKSOURCE_ADC10OSC** [Default] - MODOSC 5 MHz oscillator from the UCS
- **ADC10_A_CLOCKSOURCE_ACLK** - The Auxiliary Clock
- **ADC10_A_CLOCKSOURCE_MCLK** - The Master Clock
- **ADC10_A_CLOCKSOURCE_SMCLK** - The Sub-Master Clock
  Modified bits are **ADC10SSELx** of **ADC10CTL1** register.

clockSourceDivider selects the amount that the clock will be divided. Valid values are:

- **ADC10_A_CLOCKDIVIDER_1** [Default]
- **ADC10_A_CLOCKDIVIDER_2**
- **ADC10_A_CLOCKDIVIDER_3**
- **ADC10_A_CLOCKDIVIDER_4**
- **ADC10_A_CLOCKDIVIDER_5**
- **ADC10_A_CLOCKDIVIDER_6**
- **ADC10_A_CLOCKDIVIDER_7**
- **ADC10_A_CLOCKDIVIDER_8**
- **ADC10_A_CLOCKDIVIDER_12**
- **ADC10_A_CLOCKDIVIDER_16**
- **ADC10_A_CLOCKDIVIDER_20**
- **ADC10_A_CLOCKDIVIDER_24**
- **ADC10_A_CLOCKDIVIDER_28**
- **ADC10_A_CLOCKDIVIDER_32**
- **ADC10_A_CLOCKDIVIDER_64**
- **ADC10_A_CLOCKDIVIDER_128**
- **ADC10_A_CLOCKDIVIDER_192**
- **ADC10_A_CLOCKDIVIDER_256**
- **ADC10_A_CLOCKDIVIDER_320**
- **ADC10_A_CLOCKDIVIDER_384**
- **ADC10_A_CLOCKDIVIDER_448**
- **ADC10_A_CLOCKDIVIDER_512**
  Modified bits are **ADC10DIVx** of **ADC10CTL1** register; bits **ADC10PDIVx** of **ADC10CTL2** register.

Returns:

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

5.2.2.14 uint16_t ADC10_A_isBusy (uint16_t *baseAddress*)

Returns the busy status of the ADC10_A core.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC10_A module.

Returns:
One of the following:

- **ADC10_A_BUSY**
- **ADC10_A_NOTBUSY**
  indicating if there is a conversion currently taking place

5.2.2.15 void ADC10_A_memoryConfigure (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	22
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the ADC10_A module.

inputSourceSelect is the input that will store the converted data into the specified memory buffer. Valid values are:
- **ADC10_A_INPUT_A0** [Default]
- **ADC10_A_INPUT_A1**
- **ADC10_A_INPUT_A2**
- **ADC10_A_INPUT_A3**
- **ADC10_A_INPUT_A4**
- **ADC10_A_INPUT_A5**
- **ADC10_A_INPUT_A6**
- **ADC10_A_INPUT_A7**
- **ADC10_A_INPUT_A8**
- **ADC10_A_INPUT_A9**
- **ADC10_A_INPUT_TEMPSENSOR**
- **ADC10_A_INPUT_BATTERYMONITOR**
- **ADC10_A_INPUT_A12**
- **ADC10_A_INPUT_A13**
- **ADC10_A_INPUT_A14**
- **ADC10_A_INPUT_A15**
  Modified bits are **ADC10INCHx** of **ADC10MCTL0** 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:
- **ADC10_A_VREFPOS_AVCC** [Default]
- **ADC10_A_VREFPOS_EXT**
- **ADC10_A_VREFPOS_INT**
  Modified bits are **ADC10SREF** of **ADC10MCTL0** 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:
- **ADC10_A_VREFNEG_AVSS**
- **ADC10_A_VREFNEG_EXT**
  Modified bits are **ADC10SREF** of **ADC10CTL0** register.

Returns:
None

5.2.2.16 void ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the ADC10_A module.

readBackFormat is the specified format to store the conversions in the memory buffer. Valid values are:

- **ADC10_A_UNSIGNED_BINARY** [Default]
- **ADC10_A_SIGNED_2SCOMPLEMENT**
  Modified bits are **ADC10DF** of **ADC10CTL2** register.

Returns:

None

5.2.2.17 void ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the ADC10_A module.

samplingRateSelect is the specified maximum sampling rate. Valid values are:

- **ADC10_A_MAXSAMPLINGRATE_200KSPS** [Default]

- **ADC10_A_MAXSAMPLINGRATE_50KSPS**
  Modified bits are **ADC10SR** of **ADC10CTL2** register.

Returns:
None

5.2.2.18 void ADC10_A_setResolution (uint16_t *baseAddress*, uint8_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 12-bits.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	22
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:
baseAddress is the base address of the ADC10_A module.
resolutionSelect determines the resolution of the converted data. Valid values are:
- **ADC10_A_RESOLUTION_8BIT**
- **ADC10_A_RESOLUTION_10BIT** [Default]
  Modified bits are **ADC10RES** of **ADC10CTL2** register.

Returns:
None

5.2.2.19 void ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

baseAddress is the base address of the ADC10_A module.

invertedSignal set if the sample/hold signal should be inverted Valid values are:

- **ADC10_A_NONINVERTEDSIGNAL** [Default] - a sample-and-hold of an input signal for conversion will be started on a rising edge of the sample/hold signal.
- **ADC10_A_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 **ADC10ISSH** of **ADC10CTL1** register.

Returns:

None

5.2.2.20 void ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	54
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	26
IAR 5.51.6	Size	20
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

Parameters:

baseAddress is the base address of the ADC10_A module.

clockCycleHoldCount sets the amount of clock cycles to sample-and- hold for the memory buffer. Valid values are:
- **ADC10_A_CYCLEHOLD_4_CYCLES** [Default]
- **ADC10_A_CYCLEHOLD_8_CYCLES**
- **ADC10_A_CYCLEHOLD_16_CYCLES**
- **ADC10_A_CYCLEHOLD_32_CYCLES**
- **ADC10_A_CYCLEHOLD_64_CYCLES**
- **ADC10_A_CYCLEHOLD_96_CYCLES**
- **ADC10_A_CYCLEHOLD_128_CYCLES**
- **ADC10_A_CYCLEHOLD_192_CYCLES**
- **ADC10_A_CYCLEHOLD_256_CYCLES**
- **ADC10_A_CYCLEHOLD_384_CYCLES**
- **ADC10_A_CYCLEHOLD_512_CYCLES**
- **ADC10_A_CYCLEHOLD_768_CYCLES**
- **ADC10_A_CYCLEHOLD_1024_CYCLES**
  Modified bits are **ADC10SHTx** of **ADC10CTL0** register.

multipleSamplesEnabled allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are:
- **ADC10_A_MULTIPLESAMPLESDISABLE** - a timer trigger will be needed to start every ADC conversion.
- **ADC10_A_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 **ADC10MSC** of **ADC10CTL0** register.

Returns:
None

5.2.2.21 void ADC10_A_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 ADC10HIIE, ADC10INIE, ADC10LOIE interrupts to utilize this feature.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	20
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

baseAddress is the base address of the ADC10_A module.

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.

Returns:

None

5.2.2.22 void ADC10_A_startConversion (uint16_t *baseAddress*, uint8_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 ADC10OSC, 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 ADC10_A_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 ADC10_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	40
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	24
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

Parameters:

baseAddress is the base address of the ADC10_A module.

conversionSequenceModeSelect determines the ADC operating mode. Valid values are:

- **ADC10_A_SINGLECHANNEL** [Default] - one-time conversion of a single channel into a single memory buffer
- **ADC10_A_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

- **ADC10_A_REPEATED_SINGLECHANNEL** - repeated conversions of one channel into a single memory buffer
- **ADC10_A_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 **ADC10CONSEQx** of **ADC10CTL1** register.

Returns:
None

5.3 Programming Example

The following example shows how to initialize and use the ADC10_A API to start a single channel, single conversion.

```
    // Initialize ADC10_A with ADC10_A's built-in oscillator
    ADC10_A_init (ADC10_A_BASE,
                            ADC10_A_SAMPLEHOLDSOURCE_SC,
                            ADC10_A_CLOCKSOURCE_ADC10_AOSC,
                            ADC10_A_CLOCKDIVIDEBY_1);

    //Switch ON ADC10_A
ADC10_A_enable(ADC10_A_BASE);

    // Setup sampling timer to sample-and-hold for 16 clock cycles
    ADC10_A_setupSamplingTimer (ADC10_A_BASE,
                                                   ADC10_A_CYCLEHOLD_16_CYCLES,
                                                   FALSE);

    // Configure the Input to the Memory Buffer with the specified Reference Voltages
    ADC10_A_memoryConfigure (ADC10_A_BASE,
                                            ADC10_A_INPUT_A0,
                                            ADC10_A_VREF_AVCC, // Vref+ = AVcc
                                            ADC10_A_VREF_AVSS  // Vref- = AVss
                                            );
    while (1)
    {
            // Start a single conversion, no repeating or sequences.
            ADC10_A_startConversion (ADC10_A_BASE,
                                                    ADC10_A_SINGLECHANNEL);

            // Wait for the Interrupt Flag to assert
            while( !(ADC10_A_getInterruptStatus(ADC10_A_BASE,ADC10_AIFG0)) );

            // Clear the Interrupt Flag and start another conversion
            ADC10_A_clearInterrupt(ADC10_A_BASE,ADC10_AIFG0);
    }
```

6 12-Bit Analog-to-Digital Converter (ADC12_A)

6.1 Introduction

The 12-Bit Analog-to-Digital (ADC12_A) API provides a set of functions for using the MSP430Ware ADC12_A modules. Functions are provided to initialize the ADC12_A modules, setup signal sources and reference voltages for each memory buffer, and manage interrupts for the ADC12_A modules.

The ADC12_A module provides the ability to convert analog signals into a digital value in respect to given reference voltages. The ADC12_A can generate digital values from 0 to Vcc with an 8-, 10- or 12-bit resolution, with 16 different memory buffers to store conversion results. It operates in 2 different sampling modes, and 4 different conversion modes. The sampling modes are extended sampling and pulse sampling, in extended sampling the sample/hold signal must stay high for the duration of sampling, while in pulse mode a sampling timer is setup to start on a rising edge of the sample/hold signal and sample for a specified amount of clock cycles. The 4 conversion modes are single-channel single conversion, sequence of channels single-conversion, repeated single channel conversions, and repeated sequence of channels conversions.

The ADC12_A module can generate multiple interrupts. An interrupt can be asserted for each memory buffer when a conversion is complete, or when a conversion is about to overwrite the converted data in any of the memory buffers before it has been read out, and/or when a conversion is about to start before the last conversion is complete.

This driver is contained in `adc12_a.c`, with `adc12_a.h` containing the API definitions for use by applications.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1002
TI Compiler 4.2.1	Size	454
TI Compiler 4.2.1	Speed	448
IAR 5.51.6	None	606
IAR 5.51.6	Size	492
IAR 5.51.6	Speed	442
MSPGCC 4.8.0	None	1534
MSPGCC 4.8.0	Size	486
MSPGCC 4.8.0	Speed	430

6.2 API Functions

Functions

- void ADC12_A_clearInterrupt (uint16_t baseAddress, uint16_t memoryInterruptFlagMask)
- void ADC12_A_configureMemory (uint16_t baseAddress, ADC12_A_configureMemoryParam *param)
- void ADC12_A_disable (uint16_t baseAddress)
- void ADC12_A_disableConversions (uint16_t baseAddress, bool preempt)
- void ADC12_A_disableInterrupt (uint16_t baseAddress, uint32_t interruptMask)

- void ADC12_A_disableReferenceBurst (uint16_t baseAddress)
- void ADC12_A_disableSamplingTimer (uint16_t baseAddress)
- void ADC12_A_enable (uint16_t baseAddress)
- void ADC12_A_enableInterrupt (uint16_t baseAddress, uint32_t interruptMask)
- void ADC12_A_enableReferenceBurst (uint16_t baseAddress)
- uint8_t ADC12_A_getInterruptStatus (uint16_t baseAddress, uint16_t memoryInterruptFlagMask)
- uint32_t ADC12_A_getMemoryAddressForDMA (uint16_t baseAddress, uint8_t memoryIndex)
- uint16_t ADC12_A_getResults (uint16_t baseAddress, uint8_t memoryBufferIndex)
- bool ADC12_A_init (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
- uint16_t ADC12_A_isBusy (uint16_t baseAddress)
- void ADC12_A_memoryConfigure (uint16_t baseAddress, uint8_t memoryBufferControlIndex, uint8_t inputSourceSelect, uint8_t positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect, uint8_t endOfSequence)
- void ADC12_A_setDataReadBackFormat (uint16_t baseAddress, uint8_t readBackFormat)
- void ADC12_A_setReferenceBufferSamplingRate (uint16_t baseAddress, uint8_t samplingRateSelect)
- void ADC12_A_setResolution (uint16_t baseAddress, uint8_t resolutionSelect)
- void ADC12_A_setSampleHoldSignalInversion (uint16_t baseAddress, uint16_t invertedSignal)
- void ADC12_A_setupSamplingTimer (uint16_t baseAddress, uint16_t clockCycleHoldCountLowMem, uint16_t clockCycleHoldCountHighMem, uint16_t multipleSamplesEnabled)
- void ADC12_A_startConversion (uint16_t baseAddress, uint16_t startingMemoryBufferIndex, uint8_t conversionSequenceModeSelect)

6.2.1 Detailed Description

The ADC12_A 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 ADC12_A.

The ADC12_A initialization and conversion functions are

- ADC12_A_init()
- ADC12_A_memoryConfigure()
- ADC12_A_setupSamplingTimer()
- ADC12_A_disableSamplingTimer()
- ADC12_A_startConversion()
- ADC12_A_disableConversions()
- ADC12_A_readResults()
- ADC12_A_isBusy()

The ADC12_A interrupts are handled by

- ADC12_A_enableInterrupt()
- ADC12_A_disableInterrupt()
- ADC12_A_clearInterrupt()
- ADC12_A_getInterruptStatus()

Auxiliary features of the ADC12_A are handled by

- ADC12_A_setResolution()
- ADC12_A_setSampleHoldSignalInversion()
- ADC12_A_setDataReadBackFormat()
- ADC12_A_enableReferenceBurst()
- ADC12_A_disableReferenceBurst()
- ADC12_A_setReferenceBufferSamplingRate()
- ADC12_A_getMemoryAddressForDMA()
- ADC12_A_enable()
- ADC12_A_disable()

6.2.2 Function Documentation

6.2.2.1 void ADC12_A_clearInterrupt (uint16_t *baseAddress*, uint16_t *memoryInterruptFlagMask*)

Clears ADC12_A selected interrupt flags.

The selected ADC12_A interrupt flags are cleared, so that it no longer asserts. The memory buffer interrupt flags are only cleared when the memory buffer is accessed. Note that the overflow interrupts do not have an interrupt flag to clear; they must be accessed directly from the interrupt vector.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the ADC12_A module.

memoryInterruptFlagMask is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:

- **ADC12_A_IFG0**
- **ADC12_A_IFG1**
- **ADC12_A_IFG2**
- **ADC12_A_IFG3**
- **ADC12_A_IFG4**
- **ADC12_A_IFG5**
- **ADC12_A_IFG6**
- **ADC12_A_IFG7**
- **ADC12_A_IFG8**
- **ADC12_A_IFG9**
- **ADC12_A_IFG10**
- **ADC12_A_IFG11**
- **ADC12_A_IFG12**
- **ADC12_A_IFG13**
- **ADC12_A_IFG14**
- **ADC12_A_IFG15**

Modified bits of **ADC12IFG** register.

Returns:

None

6.2.2.2 void ADC12_A_configureMemory (uint16_t *baseAddress*, ADC12_A_configureMemoryParam $*$ *param*)

Configures the controls of the selected memory buffer.

Maps an input signal conversion into the selected memory buffer, as well as the positive and negative reference voltages for each conversion being stored into this memory buffer. If the internal reference is used for the positive reference voltage, the internal REF module must be used 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	50
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	38
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	88
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

baseAddress is the base address of the ADC12_A module.

param is the pointer to struct for memory configuration.

Returns:

None

6.2.2.3 void ADC12_A_disable (uint16_t *baseAddress*)

Disables the ADC12_A block.

This will disable operation of the ADC12_A block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the ADC12_A module.

Modified bits are **ADC12ON** of **ADC12CTL0** register.

Returns:

None

6.2.2.4 void ADC12_A_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 immediately if the preempt parameter is set as TRUE, 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 ADC12_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	62
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	42
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	92
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	34

Parameters:

baseAddress is the base address of the ADC12_A module.

preempt specifies if the current conversion should be pre-empted before the end of the conversion. Valid values are:

- **ADC12_A_COMPLETECONVERSION** - Allows the ADC12_A to end the current conversion before disabling conversions.
- **ADC12_A_PREEMPTCONVERSION** - Stops the ADC12_A immediately, with unpredictable results of the current conversion.

Modified bits of **ADC12CTL1** register and bits of **ADC12CTL0** register.

Returns:

None

6.2.2.5 void ADC12_A_disableInterrupt (uint16_t *baseAddress*, uint32_t *interruptMask*)

Disables selected ADC12_A interrupt sources.

Disables the indicated ADC12_A interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt, disabled sources have no effect on the processor.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	52
TI Compiler 4.2.1	Speed	52
IAR 5.51.6	None	36
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	138
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the ADC12_A module.

interruptMask Mask value is the logical OR of any of the following:

- **ADC12_A_IE0**
- **ADC12_A_IE1**
- **ADC12_A_IE2**
- **ADC12_A_IE3**
- **ADC12_A_IE4**
- **ADC12_A_IE5**
- **ADC12_A_IE6**
- **ADC12_A_IE7**

- **ADC12_A_IE8**
- **ADC12_A_IE9**
- **ADC12_A_IE10**
- **ADC12_A_IE11**
- **ADC12_A_IE12**
- **ADC12_A_IE13**
- **ADC12_A_IE14**
- **ADC12_A_IE15**
- **ADC12_A_OVERFLOW_IE**
- **ADC12_A_CONVERSION_TIME_OVERFLOW_IE**

Modified bits of **ADC12IE** register and bits of **ADC12CTL0** register.

Returns:
None

6.2.2.6 void ADC12_A_disableReferenceBurst (uint16_t *baseAddress*)

Disables the reference buffer's burst ability.

Disables the reference buffer's burst ability, forcing the reference buffer to remain on continuously.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the ADC12_A module.

Returns:
None

6.2.2.7 void ADC12_A_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.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC12_A module.

Modified bits are **ADC12SHP** of **ADC12CTL0** register.

Returns:
None

6.2.2.8 void ADC12_A_enable (uint16_t *baseAddress*)

Enables the ADC12_A block.

This will enable operation of the ADC12_A block.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC12_A module.

Modified bits are **ADC12ON** of **ADC12CTL0** register.

Returns:
None

6.2.2.9 void ADC12_A_enableInterrupt (uint16_t *baseAddress*, uint32_t *interruptMask*)

Enables selected ADC12_A interrupt sources.

Enables the indicated ADC12_A 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	52
TI Compiler 4.2.1	Speed	52
IAR 5.51.6	None	36
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	136
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the ADC12_A module.

interruptMask Mask value is the logical OR of any of the following:

- **ADC12_A_IE0**
- **ADC12_A_IE1**
- **ADC12_A_IE2**
- **ADC12_A_IE3**
- **ADC12_A_IE4**
- **ADC12_A_IE5**
- **ADC12_A_IE6**
- **ADC12_A_IE7**
- **ADC12_A_IE8**
- **ADC12_A_IE9**
- **ADC12_A_IE10**
- **ADC12_A_IE11**
- **ADC12_A_IE12**
- **ADC12_A_IE13**
- **ADC12_A_IE14**
- **ADC12_A_IE15**
- **ADC12_A_OVERFLOW_IE**
- **ADC12_A_CONVERSION_TIME_OVERFLOW_IE**

Modified bits of **ADC12IE** register and bits of **ADC12CTL0** register.

Returns:

None

6.2.2.10 void ADC12_A_enableReferenceBurst (uint16_t *baseAddress*)

Enables the reference buffer's burst ability.

Enables the reference buffer's burst ability, allowing the reference buffer to turn off while the ADC is not converting, and automatically turning on when the ADC needs the generated reference voltage for a conversion.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the ADC12_A module.

Returns:
None

6.2.2.11 uint8_t ADC12_A_getInterruptStatus (uint16_t *baseAddress*, uint16_t *memoryInterruptFlagMask*)

Returns the status of the selected memory interrupt flags.

Returns the status of the selected memory interrupt flags. Note that the overflow interrupts do not have an interrupt flag to clear; they must be accessed directly from the interrupt vector.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the ADC12_A module.
memoryInterruptFlagMask is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following:
- **ADC12_A_IFG0**
- **ADC12_A_IFG1**
- **ADC12_A_IFG2**
- **ADC12_A_IFG3**
- **ADC12_A_IFG4**
- **ADC12_A_IFG5**
- **ADC12_A_IFG6**
- **ADC12_A_IFG7**
- **ADC12_A_IFG8**
- **ADC12_A_IFG9**
- **ADC12_A_IFG10**
- **ADC12_A_IFG11**
- **ADC12_A_IFG12**
- **ADC12_A_IFG13**
- **ADC12_A_IFG14**
- **ADC12_A_IFG15**

Returns:
The current interrupt flag status for the corresponding mask.

6.2.2.12 uint32_t ADC12_A_getMemoryAddressForDMA (uint16_t *baseAddress*, uint8_t *memoryIndex*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	14
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the ADC12_A module.

memoryIndex is the memory buffer to return the address of. Valid values are:

- **ADC12_A_MEMORY_0** [Default]
- **ADC12_A_MEMORY_1**
- **ADC12_A_MEMORY_2**
- **ADC12_A_MEMORY_3**
- **ADC12_A_MEMORY_4**
- **ADC12_A_MEMORY_5**
- **ADC12_A_MEMORY_6**
- **ADC12_A_MEMORY_7**
- **ADC12_A_MEMORY_8**
- **ADC12_A_MEMORY_9**
- **ADC12_A_MEMORY_10**
- **ADC12_A_MEMORY_11**
- **ADC12_A_MEMORY_12**
- **ADC12_A_MEMORY_13**
- **ADC12_A_MEMORY_14**
- **ADC12_A_MEMORY_15**

Returns:

address of the specified memory buffer

6.2.2.13 uint16_t ADC12_A_getResults (uint16_t *baseAddress*, uint8_t *memoryBufferIndex*)

A Signed Integer of the 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-significant 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	14
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the ADC12_A module.

memoryBufferIndex is the specified Memory Buffer to read. Valid values are:

- **ADC12_A_MEMORY_0** [Default]
- **ADC12_A_MEMORY_1**
- **ADC12_A_MEMORY_2**
- **ADC12_A_MEMORY_3**
- **ADC12_A_MEMORY_4**
- **ADC12_A_MEMORY_5**
- **ADC12_A_MEMORY_6**
- **ADC12_A_MEMORY_7**
- **ADC12_A_MEMORY_8**
- **ADC12_A_MEMORY_9**
- **ADC12_A_MEMORY_10**
- **ADC12_A_MEMORY_11**
- **ADC12_A_MEMORY_12**
- **ADC12_A_MEMORY_13**
- **ADC12_A_MEMORY_14**
- **ADC12_A_MEMORY_15**

Returns:

A signed integer of the contents of the specified memory buffer

6.2.2.14 bool ADC12_A_init (uint16_t *baseAddress*, uint16_t *sampleHoldSignalSourceSelect*, uint8_t *clockSourceSelect*, uint16_t *clockSourceDivider*)

Initializes the ADC12_A 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	58
TI Compiler 4.2.1	Speed	58
IAR 5.51.6	None	100
IAR 5.51.6	Size	80
IAR 5.51.6	Speed	80
MSPGCC 4.8.0	None	166
MSPGCC 4.8.0	Size	70
MSPGCC 4.8.0	Speed	70

Parameters:

baseAddress is the base address of the ADC12_A 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:

- **ADC12_A_SAMPLEHOLDSOURCE_SC** [Default]
- **ADC12_A_SAMPLEHOLDSOURCE_1**
- **ADC12_A_SAMPLEHOLDSOURCE_2**

- **ADC12_A_SAMPLEHOLDSOURCE_3** - This parameter is device specific and sources should be found in the device's datasheet.
  Modified bits are **ADC12SHSx** of **ADC12CTL1** register.

clockSourceSelect selects the clock that will be used by the ADC12_A core, and the sampling timer if a sampling pulse mode is enabled. Valid values are:

- **ADC12_A_CLOCKSOURCE_ADC12OSC** [Default] - MODOSC 5 MHz oscillator from the UCS
- **ADC12_A_CLOCKSOURCE_ACLK** - The Auxiliary Clock
- **ADC12_A_CLOCKSOURCE_MCLK** - The Master Clock
- **ADC12_A_CLOCKSOURCE_SMCLK** - The Sub-Master Clock
  Modified bits are **ADC12SSELx** of **ADC12CTL1** register.

clockSourceDivider selects the amount that the clock will be divided. Valid values are:

- **ADC12_A_CLOCKDIVIDER_1** [Default]
- **ADC12_A_CLOCKDIVIDER_2**
- **ADC12_A_CLOCKDIVIDER_3**
- **ADC12_A_CLOCKDIVIDER_4**
- **ADC12_A_CLOCKDIVIDER_5**
- **ADC12_A_CLOCKDIVIDER_6**
- **ADC12_A_CLOCKDIVIDER_7**
- **ADC12_A_CLOCKDIVIDER_8**
- **ADC12_A_CLOCKDIVIDER_12**
- **ADC12_A_CLOCKDIVIDER_16**
- **ADC12_A_CLOCKDIVIDER_20**
- **ADC12_A_CLOCKDIVIDER_24**
- **ADC12_A_CLOCKDIVIDER_28**
- **ADC12_A_CLOCKDIVIDER_32**
  Modified bits are **ADC12PDIV** of **ADC12CTL2** register; bits **ADC12DIVx** of **ADC12CTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

6.2.2.15 uint16_t ADC12_A_isBusy (uint16_t *baseAddress*)

Returns the busy status of the ADC12_A core.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the ADC12_A module.

Returns:
One of the following:

- **ADC12_A_NOTBUSY**
- **ADC12_A_BUSY**
  indicating if a conversion is taking place

6.2.2.16 void ADC12_A_memoryConfigure (uint16_t *baseAddress*, uint8_t *memoryBufferControlIndex*, uint8_t *inputSourceSelect*, uint8_t *positiveRefVoltageSourceSelect*, uint8_t *negativeRefVoltageSourceSelect*, uint8_t *endOfSequence*)

DEPRECATED - Configures the controls of the selected memory buffer.

Maps an input signal conversion into the selected memory buffer, as well as the positive and negative reference voltages for each conversion being stored into this memory buffer. If the internal reference is used for the positive reference voltage, the internal REF module must be used 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	86
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	72
IAR 5.51.6	Size	68
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	90
MSPGCC 4.8.0	Size	82
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the ADC12_A module.

memoryBufferControlIndex is the selected memory buffer to set the configuration for. Valid values are:
- **ADC12_A_MEMORY_0** [Default]
- **ADC12_A_MEMORY_1**
- **ADC12_A_MEMORY_2**
- **ADC12_A_MEMORY_3**
- **ADC12_A_MEMORY_4**
- **ADC12_A_MEMORY_5**
- **ADC12_A_MEMORY_6**
- **ADC12_A_MEMORY_7**
- **ADC12_A_MEMORY_8**
- **ADC12_A_MEMORY_9**
- **ADC12_A_MEMORY_10**
- **ADC12_A_MEMORY_11**
- **ADC12_A_MEMORY_12**
- **ADC12_A_MEMORY_13**
- **ADC12_A_MEMORY_14**
- **ADC12_A_MEMORY_15**

inputSourceSelect is the input that will store the converted data into the specified memory buffer. Valid values are:
- **ADC12_A_INPUT_A0** [Default]
- **ADC12_A_INPUT_A1**
- **ADC12_A_INPUT_A2**
- **ADC12_A_INPUT_A3**
- **ADC12_A_INPUT_A4**
- **ADC12_A_INPUT_A5**
- **ADC12_A_INPUT_A6**
- **ADC12_A_INPUT_A7**
- **ADC12_A_INPUT_A8**
- **ADC12_A_INPUT_A9**
- **ADC12_A_INPUT_TEMPSENSOR**
- **ADC12_A_INPUT_BATTERYMONITOR**

- **ADC12_A_INPUT_A12**
- **ADC12_A_INPUT_A13**
- **ADC12_A_INPUT_A14**
- **ADC12_A_INPUT_A15**
  Modified bits are **ADC12INCHx** of **ADC12MCTLx** register.

positiveRefVoltageSourceSelect is the reference voltage source to set as the upper limit for the conversion stored in the specified memory. Valid values are:
- **ADC12_A_VREFPOS_AVCC** [Default]
- **ADC12_A_VREFPOS_EXT**
- **ADC12_A_VREFPOS_INT**
  Modified bits are **ADC12SREF** of **ADC12MCTLx** register.

negativeRefVoltageSourceSelect is the reference voltage source to set as the lower limit for the conversion stored in the specified memory. Valid values are:
- **ADC12_A_VREFNEG_AVSS** [Default]
- **ADC12_A_VREFNEG_EXT**
  Modified bits are **ADC12SREF** of **ADC12MCTLx** register.

endOfSequence indicates that the specified memory buffer will be the end of the sequence if a sequenced conversion mode is selected Valid values are:
- **ADC12_A_NOTENDOFSEQUENCE** [Default] - The specified memory buffer will NOT be the end of the sequence OR a sequenced conversion mode is not selected.
- **ADC12_A_ENDOFSEQUENCE** - The specified memory buffer will be the end of the sequence.
  Modified bits are **ADC12EOS** of **ADC12MCTLx** register.

Returns:
None

6.2.2.17 void ADC12_A_setDataReadBackFormat (uint16_t *baseAddress*, uint8_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	66
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the ADC12_A module.

readBackFormat is the specified format to store the conversions in the memory buffer. Valid values are:
- **ADC12_A_UNSIGNED_BINARY** [Default]
- **ADC12_A_SIGNED_2SCOMPLEMENT**
  Modified bits are **ADC12DF** of **ADC12CTL2** register.

Returns:
None

6.2.2.18 void ADC12_A_setReferenceBufferSamplingRate (uint16_t *baseAddress*, uint8_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	66
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the ADC12_A module.

samplingRateSelect is the specified maximum sampling rate. Valid values are:

- **ADC12_A_MAXSAMPLINGRATE_200KSPS** [Default]
- **ADC12_A_MAXSAMPLINGRATE_50KSPS**
  Modified bits are **ADC12SR** of **ADC12CTL2** register.

Returns:

None

6.2.2.19 void ADC12_A_setResolution (uint16_t *baseAddress*, uint8_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 12-bits.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	66
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

baseAddress is the base address of the ADC12_A module.

resolutionSelect determines the resolution of the converted data. Valid values are:

- **ADC12_A_RESOLUTION_8BIT**
- **ADC12_A_RESOLUTION_10BIT**
- **ADC12_A_RESOLUTION_12BIT** [Default]
  Modified bits are **ADC12RESx** of **ADC12CTL2** register.

Returns:

None

6.2.2.20 void ADC12_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

baseAddress is the base address of the ADC12_A module.

invertedSignal set if the sample/hold signal should be inverted Valid values are:

- **ADC12_A_NONINVERTEDSIGNAL** [Default] - a sample-and-hold of an input signal for conversion will be started on a rising edge of the sample/hold signal.
- **ADC12_A_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 **ADC12ISSH** of **ADC12CTL1** register.

Returns:

None

6.2.2.21 void ADC12_A_setupSamplingTimer (uint16_t *baseAddress*, uint16_t *clockCycleHoldCountLowMem*, uint16_t *clockCycleHoldCountHighMem*, 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	36
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

baseAddress is the base address of the ADC12_A module.

clockCycleHoldCountLowMem sets the amount of clock cycles to sample- and-hold for the higher memory buffers 0-7. Valid values are:

- **ADC12_A_CYCLEHOLD_4_CYCLES** [Default]
- **ADC12_A_CYCLEHOLD_8_CYCLES**
- **ADC12_A_CYCLEHOLD_16_CYCLES**
- **ADC12_A_CYCLEHOLD_32_CYCLES**
- **ADC12_A_CYCLEHOLD_64_CYCLES**
- **ADC12_A_CYCLEHOLD_96_CYCLES**
- **ADC12_A_CYCLEHOLD_128_CYCLES**
- **ADC12_A_CYCLEHOLD_192_CYCLES**
- **ADC12_A_CYCLEHOLD_256_CYCLES**
- **ADC12_A_CYCLEHOLD_384_CYCLES**
- **ADC12_A_CYCLEHOLD_512_CYCLES**
- **ADC12_A_CYCLEHOLD_768_CYCLES**
- **ADC12_A_CYCLEHOLD_1024_CYCLES**
  Modified bits are **ADC12SHT0x** of **ADC12CTL0** register.

clockCycleHoldCountHighMem sets the amount of clock cycles to sample-and-hold for the higher memory buffers 8-15. Valid values are:

- **ADC12_A_CYCLEHOLD_4_CYCLES** [Default]
- **ADC12_A_CYCLEHOLD_8_CYCLES**
- **ADC12_A_CYCLEHOLD_16_CYCLES**
- **ADC12_A_CYCLEHOLD_32_CYCLES**
- **ADC12_A_CYCLEHOLD_64_CYCLES**
- **ADC12_A_CYCLEHOLD_96_CYCLES**
- **ADC12_A_CYCLEHOLD_128_CYCLES**
- **ADC12_A_CYCLEHOLD_192_CYCLES**
- **ADC12_A_CYCLEHOLD_256_CYCLES**
- **ADC12_A_CYCLEHOLD_384_CYCLES**
- **ADC12_A_CYCLEHOLD_512_CYCLES**
- **ADC12_A_CYCLEHOLD_768_CYCLES**
- **ADC12_A_CYCLEHOLD_1024_CYCLES**
  Modified bits are **ADC12SHT1x** of **ADC12CTL0** register.

multipleSamplesEnabled allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are:

- **ADC12_A_MULTIPLESAMPLESDISABLE** [Default] - a timer trigger will be needed to start every ADC conversion.
- **ADC12_A_MULTIPLESAMPLESENABLE** - during a sequenced and/or repeated conversion mode, after the first conversion, no sample/hold signal is necessary to start subsequent sample/hold and convert processes.
  Modified bits are **ADC12MSC** of **ADC12CTL0** register.

Returns:

None

6.2.2.22 void ADC12_A_startConversion (uint16_t *baseAddress*, uint16_t *startingMemoryBufferIndex*, uint8_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 ADC12OSC, 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 ADC12_A_setupSamplingTimer() function. Note that after this function is called, the ADC12_A_disableConversions() 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	74
TI Compiler 4.2.1	Size	34
TI Compiler 4.2.1	Speed	34
IAR 5.51.6	None	58
IAR 5.51.6	Size	44
IAR 5.51.6	Speed	44
MSPGCC 4.8.0	None	158
MSPGCC 4.8.0	Size	58
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the ADC12_A module.

startingMemoryBufferIndex is the memory buffer that will hold the first or only conversion. Valid values are:

- **ADC12_A_MEMORY_0** [Default]
- **ADC12_A_MEMORY_1**
- **ADC12_A_MEMORY_2**
- **ADC12_A_MEMORY_3**
- **ADC12_A_MEMORY_4**
- **ADC12_A_MEMORY_5**
- **ADC12_A_MEMORY_6**
- **ADC12_A_MEMORY_7**
- **ADC12_A_MEMORY_8**
- **ADC12_A_MEMORY_9**
- **ADC12_A_MEMORY_10**
- **ADC12_A_MEMORY_11**
- **ADC12_A_MEMORY_12**
- **ADC12_A_MEMORY_13**
- **ADC12_A_MEMORY_14**
- **ADC12_A_MEMORY_15**
  Modified bits are **ADC12STARTADDx** of **ADC12CTL1** register.

conversionSequenceModeSelect determines the ADC operating mode. Valid values are:

- **ADC12_A_SINGLECHANNEL** [Default] - one-time conversion of a single channel into a single memory buffer.
- **ADC12_A_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.
- **ADC12_A_REPEATED_SINGLECHANNEL** - repeated conversions of one channel into a single memory buffer.
- **ADC12_A_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 **ADC12CONSEQx** of **ADC12CTL1** register.

Modified bits of **ADC12CTL1** register and bits of **ADC12CTL0** register.

Returns:

None

6.3 Programming Example

The following example shows how to initialize and use the ADC12 API to start a single channel, single conversion.

```
    // Initialize ADC12 with ADC12’s built-in oscillator
    ADC12_A_init (ADC12_A_BASE,
                            ADC12_A_SAMPLEHOLDSOURCE_SC,
                            ADC12_A_CLOCKSOURCE_ADC12OSC,
                            ADC12_A_CLOCKDIVIDEBY_1);

//Switch ON ADC12
ADC12_A_enable(ADC12_A_BASE);

    // Setup sampling timer to sample-and-hold for 16 clock cycles
    ADC12_A_setupSamplingTimer (ADC12_A_BASE,
                                                  ADC12_A_CYCLEHOLD_64_CYCLES,
                                                  ADC12_A_CYCLEHOLD_4_CYCLES,
                                                  FALSE);

    // Configure the Input to the Memory Buffer with the specified Reference Voltages
    ADC12_A_memoryConfigure (ADC12_A_BASE,
                           ADC12_A_MEMORY_0,
                                           ADC12_A_INPUT_A0,
                                           ADC12_A_VREF_AVCC, // Vref+ = AVcc
                                           ADC12_A_VREF_AVSS, // Vref- = AVss
                                           FALSE
                                           );
    while (1)
    {
            // Start a single conversion, no repeating or sequences.
            ADC12_A_startConversion (ADC12_A_BASE,
                                   ADC12_A_MEMORY_0,
                                                   ADC12_A_SINGLECHANNEL);

            // Wait for the Interrupt Flag to assert
            while( !(ADC12_A_getInterruptStatus(ADC12_A_BASE,ADC12IFG0)) );

            // Clear the Interrupt Flag and start another conversion
            ADC12_A_clearInterrupt(ADC12_A_BASE,ADC12IFG0);
    }
```

7 Advanced Encryption Standard (AES)

7.1 Introduction

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

- Encryption and decryption according to AES FIPS PUB 197 with 128-bit key
- On-the-fly key expansion for encryption and decryption
- Off-line key generation for decryption
- Byte and word access to key, input, and output data
- AES ready interrupt flag The AES256 accelerator module performs encryption and decryption of 128-bit data with 128-/192-/256-bit keys according to the advanced encryption standard (AES) (FIPS PUB 197) in hardware. The AES accelerator features are: AES encryption Ű 128 bit - 168 cycles Ű 192 bit - 204 cycles Ű 256 bit - 234 cycles AES decryption Ű 128 bit - 168 cycles Ű 192 bit - 206 cycles Ű 256 bit - 234 cycles
- On-the-fly key expansion for encryption and decryption
- Offline key generation for decryption
- Shadow register storing the initial key for all key lengths
- Byte and word access to key, input data, and output data
- AES ready interrupt flag

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1452
TI Compiler 4.2.1	Size	622
TI Compiler 4.2.1	Speed	2132
IAR 5.51.6	None	1084
IAR 5.51.6	Size	322
IAR 5.51.6	Speed	592
MSPGCC 4.8.0	None	1832
MSPGCC 4.8.0	Size	702
MSPGCC 4.8.0	Speed	2036

7.2 API Functions

Functions

- void AES_clearErrorFlag (uint16_t baseAddress)
- void AES_clearInterruptFlag (uint16_t baseAddress)
- uint8_t AES_decryptData (uint16_t baseAddress, const uint8_t ∗Data, uint8_t ∗decryptedData)

- uint8_t AES_decryptDataUsingEncryptionKey (uint16_t baseAddress, const uint8_t ∗Data, uint8_t ∗decryptedData)
- void AES_disableInterrupt (uint16_t baseAddress)
- void AES_enableInterrupt (uint16_t baseAddress)
- uint8_t AES_encryptData (uint16_t baseAddress, const uint8_t ∗Data, uint8_t ∗encryptedData)
- uint8_t AES_getDataOut (uint16_t baseAddress, uint8_t ∗OutputData)
- uint32_t AES_getErrorFlagStatus (uint16_t baseAddress)
- uint32_t AES_getInterruptFlagStatus (uint16_t baseAddress)
- uint8_t AES_isBusy (uint16_t baseAddress)
- void AES_reset (uint16_t baseAddress)
- uint8_t AES_setCipherKey (uint16_t baseAddress, const uint8_t ∗CipherKey)
- uint8_t AES_setDecipherKey (uint16_t baseAddress, const uint8_t ∗CipherKey)
- uint8_t AES_startDecryptData (uint16_t baseAddress, const uint8_t ∗Data)
- uint8_t AES_startDecryptDataUsingEncryptionKey (uint16_t baseAddress, const uint8_t ∗Data)
- uint8_t AES_startEncryptData (uint16_t baseAddress, const uint8_t ∗Data, uint8_t ∗encryptedData)
- uint8_t AES_startSetDecipherKey (uint16_t baseAddress, const uint8_t ∗CipherKey)

7.2.1 Detailed Description

The AES module APIs are

- AES_setCipherKey()
- AES_encryptData()
- AES_decryptDataUsingEncryptionKey()
- AES_generateFirstRoundKey()
- AES_decryptData()
- AES_reset()
- AES_startEncryptData()
- AES_startDecryptDataUsingEncryptionKey()
- AES_startDecryptData()
- AES_startGenerateFirstRoundKey()
- AES_getDataOut()

The AES interrupt handler functions

- AES_enableInterrupt()
- AES_disableInterrupt()
- AES_clearInterruptFlag()

7.2.2 Function Documentation

7.2.2.1 void AES_clearErrorFlag (uint16_t *baseAddress*)

Clears the AES error flag.

Clears the AES error flag that results from a key or data being written while the AES module is busy. Modified bit is AESERRFG of AESACTL0 register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	2
MSPGCC 4.8.0	Speed	2

Parameters:
baseAddress is the base address of the AES module.

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

Returns:
None

7.2.2.2 void AES_clearInterruptFlag (uint16_t *baseAddress*)

Clears the AES ready interrupt flag.

This function clears the AES ready interrupt flag. This flag is automatically cleared when AESADOUT is read, or when AESAKEY or AESADIN is written. This function should be used when the flag needs to be reset and it has not been automatically cleared by one of the previous actions.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	2
MSPGCC 4.8.0	Speed	2

Parameters:
baseAddress is the base address of the AES module.

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

Returns:
None

7.2.2.3 uint8_t AES_decryptData (uint16_t *baseAddress*, const uint8_t ∗ *Data*, uint8_t ∗ *decryptedData*)

Decrypts a block of data using the AES module.

This function requires a pre-generated decryption key. A key can be loaded and pre-generated by using function **AES_startSetDecipherKey()** or **AES_setDecipherKey()**. The decryption takes 167 MCLK.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	190
TI Compiler 4.2.1	Size	80
TI Compiler 4.2.1	Speed	306
IAR 5.51.6	None	148
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	56
MSPGCC 4.8.0	None	238
MSPGCC 4.8.0	Size	94
MSPGCC 4.8.0	Speed	290

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

decryptedData is a pointer to an uint8_t array with a length of 16 bytes in that the decrypted data will be written.

Returns:

STATUS_SUCCESS

7.2.2.4 uint8_t AES_decryptDataUsingEncryptionKey (uint16_t *baseAddress*, const uint8_t ∗ *Data*, uint8_t ∗ *decryptedData*)

DEPRECATED Decrypts a block of data using the AES module.

This function can be used to decrypt data by using the same key as used for a previous performed encryption. The decryption takes 214 MCLK.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	194
TI Compiler 4.2.1	Size	82
TI Compiler 4.2.1	Speed	308
IAR 5.51.6	None	154
IAR 5.51.6	Size	62
IAR 5.51.6	Speed	114
MSPGCC 4.8.0	None	252
MSPGCC 4.8.0	Size	96
MSPGCC 4.8.0	Speed	292

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

decryptedData is a pointer to an uint8_t array with a length of 16 bytes in that the decrypted data will be written.

Returns:

STATUS_SUCCESS

7.2.2.5 void AES_disableInterrupt (uint16_t *baseAddress*)

Disables AES ready interrupt.

Disables AES ready interrupt. This interrupt is reset by a PUC, but not reset by AES_reset.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	2
MSPGCC 4.8.0	Speed	2

Parameters:
baseAddress is the base address of the AES module.

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

Returns:
None

7.2.2.6 void AES_enableInterrupt (uint16_t *baseAddress*)

Enables AES ready interrupt.

Enables AES ready interrupt. This interrupt is reset by a PUC, but not reset by AES_reset. Does not clear interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	2
MSPGCC 4.8.0	Speed	2

Parameters:
baseAddress is the base address of the AES module.

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

Returns:
None

7.2.2.7 uint8_t AES_encryptData (uint16_t *baseAddress*, const uint8_t ∗ *Data*, uint8_t ∗ *encryptedData*)

Encrypts a block of data using the AES module.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	190
TI Compiler 4.2.1	Size	80
TI Compiler 4.2.1	Speed	304
IAR 5.51.6	None	148
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	56
MSPGCC 4.8.0	None	238
MSPGCC 4.8.0	Size	94
MSPGCC 4.8.0	Speed	290

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.

encryptedData is a pointer to an uint8_t array with a length of 16 bytes in that the encrypted data will be written.

Returns:

STATUS_SUCCESS

7.2.2.8 uint8_t AES_getDataOut (uint16_t *baseAddress*, uint8_t ∗ *OutputData*)

Reads back the output data from AES module.

This function is meant to use after an encryption or decryption process that was started and finished by initiating an interrupt by use of the **AES_startEncryptData()** or **AES_startDecryptData()** functions.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	146
IAR 5.51.6	None	82
IAR 5.51.6	Size	56
IAR 5.51.6	Speed	58
MSPGCC 4.8.0	None	124
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	130

Parameters:

baseAddress is the base address of the AES module.

OutputData is a pointer to an uint8_t array with a length of 16 bytes in which the output data of the AES module is available. If AES module is busy returns NULL.

Returns:

STATUS_SUCCESS if AES is not busy, STATUS_FAIL if it is busy

7.2.2.9 uint32_t AES_getErrorFlagStatus (uint16_t *baseAddress*)

Gets the AES error flag status.

Checks the AES error flag that results from a key or data being written while the AES module is busy. If the flag is set, it needs to be cleared using AES_clearErrorFlag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the AES module.

Returns:
One of the following:

- **AES_ERROR_OCCURRED**
- **AES_NO_ERROR**
  indicating if AESAKEY or AESADIN were written while an AES operation was in progress

7.2.2.10 uint32_t AES_getInterruptFlagStatus (uint16_t *baseAddress*)

Gets the AES ready interrupt flag status.

This function checks the AES ready interrupt flag. This flag is automatically cleared when AESADOUT is read, or when AESAKEY or AESADIN is written. This function can be used to confirm that this has been done.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the AES module.

Returns:
uint32_t - AES_READY_INTERRUPT or 0x00.

7.2.2.11 uint8_t AES_isBusy (uint16_t *baseAddress*)

Gets the AES module busy status.

Gets the AES module busy status. If a key or data are written while the AES module is busy, an error flag will be thrown.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the AES module.

Returns:
One of the following:

- **AES_BUSY**
- **AES_NOT_BUSY**
  indicating if encryption/decryption/key generation is taking place

7.2.2.12 void AES_reset (uint16_t *baseAddress*)

Resets AES Module immediately.

This function performs a software reset on the AES Module, note that this does not affect the AES ready interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the AES module.

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

Returns:
None

7.2.2.13 uint8_t AES_setCipherKey (uint16_t *baseAddress*, const uint8_t ∗ *CipherKey*)

Loads a 128 bit cipher key to AES module.

This function loads a 128 bit cipher key to AES module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	108
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	172
IAR 5.51.6	None	88
IAR 5.51.6	Size	26
IAR 5.51.6	Speed	74
MSPGCC 4.8.0	None	124
MSPGCC 4.8.0	Size	58
MSPGCC 4.8.0	Speed	170

Parameters:

baseAddress is the base address of the AES module.

CipherKey is a pointer to an uint8_t array with a length of 16 bytes that contains a 128 bit cipher key.

Returns:

STATUS_SUCCESS

7.2.2.14 uint8_t AES_setDecipherKey (uint16_t *baseAddress*, const uint8_t $*$ *CipherKey*)

Sets the decipher key The API.

The API **AES_startSetDecipherKey()** or **AES_setDecipherKey()** must be invoked before invoking **AES_setDecipherKey()**.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	110
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	172
IAR 5.51.6	None	88
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	136
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	172

Parameters:

baseAddress is the base address of the AES module.

CipherKey is a pointer to an uint8_t array with a length of 16 bytes that contains the initial AES key.

Returns:

STATUS_SUCCESS

7.2.2.15 uint8_t AES_startDecryptData (uint16_t *baseAddress*, const uint8_t $*$ *Data*)

Decrypts a block of data using the AES module.

This is the non-blocking equivalent of AES_decryptData(). This function requires a pre-generated decryption key. A key can be loaded and pre- generated by using function **AES_setDecipherKey()** or **AES_startSetDecipherKey()**. The decryption takes 167 MCLK. It is recommended to use interrupt to check for procedure completion then using AES_getDataOut() API to retrieve the decrypted data.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	102
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	170
IAR 5.51.6	None	78
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	124
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	164

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

Returns:

STATUS_SUCCESS

7.2.2.16 uint8_t AES_startDecryptDataUsingEncryptionKey (uint16_t *baseAddress*, const uint8_t ∗ *Data*)

DEPRECATED Starts an decryption process on the AES module.

This is the non-blocking equivalent of AES_decryptDataUsingEncryptionKey(). This function can be used to decrypt data by using the same key as used for a previous performed encryption. The decryption takes 214 MCLK.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	172
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	138
MSPGCC 4.8.0	Size	54
MSPGCC 4.8.0	Speed	166

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

Returns:

STATUS_SUCCESS

7.2.2.17 uint8_t AES_startEncryptData (uint16_t *baseAddress*, const uint8_t ∗ *Data*, uint8_t ∗ *encryptedData*)

Starts an encryption process on the AES module.

This is the non-blocking equivalent of AES_encryptData(). The cipher key that is used for decryption should be loaded in advance by using function **AES_setCipherKey()**. It is recommended to use interrupt to check for procedure completion then using AES_getDataOut() API to retrieve the encrypted data.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	110
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	166
IAR 5.51.6	None	78
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	56
MSPGCC 4.8.0	None	136
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	164

Parameters:

baseAddress is the base address of the AES module.

Data is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.

encryptedData is a pointer to an uint8_t array with a length of 16 bytes in that the encrypted data will be written.

Returns:

STATUS_SUCCESS

7.2.2.18 uint8_t AES_startSetDecipherKey (uint16_t *baseAddress*, const uint8_t ∗ *CipherKey*)

Loads the decipher key.

This is the non-blocking equivalent of AES_setDecipherKey(). The API **AES_startSetDecipherKey()** or **AES_setDecipherKey()** must be invoked before invoking **AES_startSetDecipherKey()**.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	160
IAR 5.51.6	None	78
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	62
MSPGCC 4.8.0	None	120
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	158

Parameters:

baseAddress is the base address of the AES module.

CipherKey is a pointer to an uint8_t array with a length of 16 bytes that contains the initial AES key.

Returns:

STATUS_SUCCESS

7.3 Programming Example

The following example shows some AES operations using the APIs

```
unsigned char Data[16] =              {       0x30, 0x31, 0x32, 0x33,
                                                                    0x34, 0x35, 0x36, 0x37,
                                                                    0x38, 0x39, 0x0A, 0x0B,
                                                                        0x0C, 0x0D, 0x0E, 0x0F  };
unsigned char CipherKey[16] =   {       0xAA, 0xBB, 0x02, 0x03,
                                                                        0x04, 0x05, 0x06, 0x07,
                                                                        0x08, 0x09, 0x0A, 0x0B,
                                                                        0x0C, 0x0D, 0x0E, 0x0F  };
unsigned char DataAES[16];              // Encrypted data
unsigned char DataunAES[16];    // Decrypted data

// Load a cipher key to module
AES_setCipherKey(AES_BASE, CipherKey);

// Encrypt data with preloaded cipher key
AES_encryptData(AES_BASE, Data, DataAES);

// Decrypt data with keys that were generated during encryption - takes 214 MCLK
// This function will generate all round keys needed for decryption first and then
// the encryption process starts
AES_decryptDataUsingEncryptionKey(AES_BASE, DataAES, DataunAES);
```

8 Battery Backup System

8.1 Introduction

The Battery Backup System (BATBCK) API provides a set of functions for using the MSP430Ware BATBCK modules. Functions are provided to handle the backup Battery sub-system, initialize and enable the backup Battery charger, and control access to and from the backup RAM space.

The BATBCK module offers no interrupt, and is used only to control the Battery backup sub-system, Battery charger, and backup RAM space.

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

8.2 API Functions

Functions

- void BAK_BATT_chargerInitAndEnable (uint16_t baseAddress, uint8_t chargerEndVoltage, uint8_t chargeCurrent)
- void BAK_BATT_disable (uint16_t baseAddress)
- void BAK_BATT_disableBackupSupplyToADC (uint16_t baseAddress)
- void BAK_BATT_disableCharger (uint16_t baseAddress)
- void BAK_BATT_enableBackupSupplyToADC (uint16_t baseAddress)
- uint16_t BAK_BATT_getBackupRAMData (uint16_t baseAddress, uint8_t backupRAMSelect)
- void BAK_BATT_manuallySwitchToBackupSupply (uint16_t baseAddress)
- void BAK_BATT_setBackupRAMData (uint16_t baseAddress, uint8_t backupRAMSelect, uint16_t data)
- uint16_t BAK_BATT_unlockBackupSubSystem (uint16_t baseAddress)

8.2.1 Detailed Description

The BATBCK API is divided into three groups: one that handles the Battery backup sub-system, one that controls the charger, and one that controls access to and from the backup RAM space.

The BATBCK sub-system controls are handled by

- BAK_BATT_unlockBackupSubSystem()
- BAK_BATT_enableBackupSupplyToADC()
- BAK_BATT_disableBackupSupplyToADC()
- BAK_BATT_manuallySwitchToBackupSupply()
- BAK_BATT_disable()

The BATBCK charger is controlled by

- BAK_BATT_chargerInitAndEnable()
- BAK_BATT_disableCharger()

The backup RAM space is accessed by

- BAK_BATT_setBackupRAMData()
- BAK_BATT_getBackupRAMData()

8.2.2 Function Documentation

8.2.2.1 void BAK_BATT_chargerInitAndEnable (uint16_t *baseAddress*, uint8_t *chargerEndVoltage*, uint8_t *chargeCurrent*)

Initializes and enables the backup battery charger.

This function initializes the backup battery charger with the selected settings.

Parameters:

baseAddress is the base address of the BAK_BATT module.

chargerEndVoltage is the maximum voltage to charge the backup battery to. Valid values are:

- **BAK_BATT_CHARGERENDVOLTAGE_VCC** - charges backup battery up to Vcc
- **BAK_BATT_CHARGERENDVOLTAGE2_7V** - charges backup battery up to 2.7V OR up to Vcc if Vcc is less than 2.7V.
  Modified bits are **BAKCHVx** of **BAKCHCTL** register.

chargeCurrent is the maximum current to charge the backup battery at. Valid values are:

- **BAK_BATT_CHARGECURRENT_5KOHM**
- **BAK_BATT_CHARGECURRENT_10KOHM**
- **BAK_BATT_CHARGECURRENT_20KOHM**
  Modified bits are **BAKCHCx** of **BAKCHCTL** register.

Returns:

None

8.2.2.2 void BAK_BATT_disable (uint16_t *baseAddress*)

Disables backup battery system.

This function disables the battery backup system from being used. The battery backup system is re-enabled after a power cycle.

Parameters:

baseAddress is the base address of the BAK_BATT module.

Returns:

None

8.2.2.3 void BAK_BATT_disableBackupSupplyToADC (uint16_t *baseAddress*)

Disables the backup supply input to the ADC module.

This function disables the ability to monitor the backup supply voltage from the ADC battery monitor input.

Parameters:

baseAddress is the base address of the BAK_BATT module.

Returns:

None

8.2.2.4 void BAK_BATT_disableCharger (uint16_t *baseAddress*)

Disables and resets backup battery charger settings.

This function clears all backup battery charger settings and disables it. To re-enable the charger, a call to BAK_BATT_chargerInitAndEnable() is required.

Parameters:
baseAddress is the base address of the BAK_BATT module.

Returns:
None

8.2.2.5 void BAK_BATT_enableBackupSupplyToADC (uint16_t *baseAddress*)

Enables the backup supply to be measured by the ADC battery monitor input.

This function enables the backup supply signal to be monitored by the ADC battery supply monitor input, to allow a measurement of the voltage from the backup battery.

Parameters:
baseAddress is the base address of the BAK_BATT module.

Returns:
None

8.2.2.6 uint16_t BAK_BATT_getBackupRAMData (uint16_t *baseAddress*, uint8_t *backupRAMSelect*)

Returns the data from the selected backup RAM space.

This function returns the 16-bit data currently residing in the selected backup RAM space.

Parameters:
baseAddress is the base address of the BAK_BATT module.
backupRAMSelect is the backup RAM space to read out from. Valid values are:
- **BAK_BATT_RAMSELECT_0**
- **BAK_BATT_RAMSELECT_1**
- **BAK_BATT_RAMSELECT_2**
- **BAK_BATT_RAMSELECT_3**

Returns:
Data residing in the selected backup RAM space.

8.2.2.7 void BAK_BATT_manuallySwitchToBackupSupply (uint16_t *baseAddress*)

Manually switches to backup supply.

This function uses software to manually switch to the backup battery supply. Once this bit is set, it will be automatically reset by a POR and the system returns to an automatic switch to backup supply.

Parameters:
baseAddress is the base address of the BAK_BATT module.

Returns:
None

8.2.2.8 void BAK_BATT_setBackupRAMData (uint16_t *baseAddress*, uint8_t *backupRAMSelect*, uint16_t *data*)

Sets data into the selected backup RAM space.

This function sets the given 16-bit data into the selected backup RAM space.

Parameters:
baseAddress is the base address of the BAK_BATT module.
backupRAMSelect is the backup RAM space to set data into. Valid values are:
- **BAK_BATT_RAMSELECT_0**
- **BAK_BATT_RAMSELECT_1**
- **BAK_BATT_RAMSELECT_2**
- **BAK_BATT_RAMSELECT_3**

data is the data to set into the selected backup RAM space.

Returns:
None

8.2.2.9 uint16_t BAK_BATT_unlockBackupSubSystem (uint16_t *baseAddress*)

Unlocks any pending backup input pins and RTC_B interrupts to be serviced.

This function unlocks the ability to view and service any pending backup input pin interrupts, as well as pending RTC_B interrupts. The backup sub- system can only be unlocked when the backup domain has settled, so this function returns the status of the unlock bit after it has been to be verified by user code. Please note, the backup sub-system should only be unlocked after modifying the RTC_B registers.

Parameters:
baseAddress is the base address of the BAK_BATT module.

Returns:
One of the following:

- **BAK_BATT_UNLOCKFAILURE** backup system has not yet settled
- **BAK_BATT_UNLOCKSUCCESS** successfully unlocked
  indicating if the backup system has been successfully unlocked

8.3 Programming Example

9 Comparator (COMP_B)

9.1 Introduction

The Comparator B (COMP_B) API provides a set of functions for using the MSP430Ware COMP_B modules. Functions are provided to initialize the COMP_B modules, setup reference voltages for input, and manage interrupts for the COMP_B modules.

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

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

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	852
TI Compiler 4.2.1	Size	438
TI Compiler 4.2.1	Speed	438
IAR 5.51.6	None	632
IAR 5.51.6	Size	458
IAR 5.51.6	Speed	510
MSPGCC 4.8.0	None	1254
MSPGCC 4.8.0	Size	476
MSPGCC 4.8.0	Speed	792

9.2 API Functions

Functions

- void COMP_B_clearInterrupt (uint16_t baseAddress, uint16_t interruptFlagMask)
- void COMP_B_configureReferenceVoltage (uint16_t baseAddress, COMP_B_configureReferenceVoltageParam ∗param)
- void COMP_B_disable (uint16_t baseAddress)
- void COMP_B_disableInputBuffer (uint16_t baseAddress, uint8_t inputPort)
- void COMP_B_disableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
- void COMP_B_enable (uint16_t baseAddress)
- void COMP_B_enableInputBuffer (uint16_t baseAddress, uint8_t inputPort)
- void COMP_B_enableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
- uint8_t COMP_B_getInterruptStatus (uint16_t baseAddress, uint16_t interruptFlagMask)
- bool COMP_B_init (uint16_t baseAddress, uint8_t positiveTerminalInput, uint8_t negativeTerminalInput, uint16_t powerModeSelect, uint8_t outputFilterEnableAndDelayLevel, uint16_t invertedOutputPolarity)
- bool COMP_B_initialize (uint16_t baseAddress, COMP_B_initializeParam ∗param)
- void COMP_B_interruptSetEdgeDirection (uint16_t baseAddress, uint16_t edgeDirection)
- void COMP_B_interruptToggleEdgeDirection (uint16_t baseAddress)
- void COMP_B_IOSwap (uint16_t baseAddress)
- uint16_t COMP_B_outputValue (uint16_t baseAddress)
- void COMP_B_setReferenceVoltage (uint16_t baseAddress, uint16_t supplyVoltageReferenceBase, uint16_t lowerLimitSupplyVoltageFractionOf32, uint16_t upperLimitSupplyVoltageFractionOf32, uint16_t referenceAccuracy)
- void COMP_B_shortInputs (uint16_t baseAddress)
- void COMP_B_unshortInputs (uint16_t baseAddress)

9.2.1 Detailed Description

The COMP_B 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 COMP_B.

The COMP_B initialization and output functions are

- COMP_B_init()
- COMP_B_setReferenceVoltage()
- COMP_B_enable()
- COMP_B_disable()
- COMP_B_outputValue()

The COMP_B interrupts are handled by

- COMP_B_enableInterrupt()
- COMP_B_disableInterrupt()
- COMP_B_clearInterrupt()

- COMP_B_getInterruptStatus()
- COMP_B_interruptSetEdgeDirection()
- COMP_B_interruptToggleEdgeDirection()

Auxiliary features of the COMP_B are handled by

- COMP_B_enableShortOfInputs()
- COMP_B_disableShortOfInputs()
- COMP_B_disableInputBuffer()
- COMP_B_enableInputBuffer()
- COMP_B_IOSwap()

9.2.2 Function Documentation

9.2.2.1 void COMP_B_clearInterrupt (uint16_t *baseAddress*, uint16_t *interruptFlagMask*)

Clears COMP_B interrupt flags.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the COMP_B module.

interruptFlagMask is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following:
- **COMP_B_OUTPUT_FLAG** - Output interrupt
- **COMP_B_OUTPUTINVERTED_FLAG** - Output interrupt inverted polarity

Modified bits of **CBINT** register.

Returns:

None

9.2.2.2 void COMP_B_configureReferenceVoltage (uint16_t *baseAddress*, COMP_B_configureReferenceVoltageParam ∗ *param*)

Generates a Reference Voltage to the terminal selected during initialization.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	140
TI Compiler 4.2.1	Size	90
TI Compiler 4.2.1	Speed	90
IAR 5.51.6	None	118
IAR 5.51.6	Size	90
IAR 5.51.6	Speed	88
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	92
MSPGCC 4.8.0	Speed	160

Parameters:

baseAddress is the base address of the COMP_B module.

param is the pointer to struct for reference voltage configuration.

Returns:

None

9.2.2.3 void COMP_B_disable (uint16_t *baseAddress*)

Turns off the COMP_B module.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the COMP_B module.

Returns:

None

9.2.2.4 void COMP_B_disableInputBuffer (uint16_t *baseAddress*, uint8_t *inputPort*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	24
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the COMP_B module.

inputPort is the port in which the input buffer will be disabled. Valid values are:

- **COMP_B_INPUT0** [Default]
- **COMP_B_INPUT1**
- **COMP_B_INPUT2**
- **COMP_B_INPUT3**
- **COMP_B_INPUT4**
- **COMP_B_INPUT5**
- **COMP_B_INPUT6**
- **COMP_B_INPUT7**
- **COMP_B_INPUT8**
- **COMP_B_INPUT9**
- **COMP_B_INPUT10**
- **COMP_B_INPUT11**
- **COMP_B_INPUT12**
- **COMP_B_INPUT13**
- **COMP_B_INPUT14**
- **COMP_B_INPUT15**

- **COMP_B_VREF**
  Modified bits are **CBPDx** of **CBCTL3** register.

Returns:
None

9.2.2.5 void COMP_B_disableInterrupt (uint16_t *baseAddress*, uint16_t *interruptMask*)

Disables selected COMP_B interrupt sources.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the COMP_B module.
interruptMask is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following:
- **COMP_B_OUTPUT_INT** - Output interrupt
- **COMP_B_OUTPUTINVERTED_INT** - Output interrupt inverted polarity
  Modified bits of **CBINT** register.

Returns:
None

9.2.2.6 void COMP_B_enable (uint16_t *baseAddress*)

Turns on the COMP_B module.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the COMP_B module.

Returns:

None

9.2.2.7 void COMP_B_enableInputBuffer (uint16_t *baseAddress*, uint8_t *inputPort*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	24
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the COMP_B module.

inputPort is the port in which the input buffer will be enabled. Valid values are:

- **COMP_B_INPUT0** [Default]
- **COMP_B_INPUT1**
- **COMP_B_INPUT2**
- **COMP_B_INPUT3**
- **COMP_B_INPUT4**
- **COMP_B_INPUT5**
- **COMP_B_INPUT6**

- **COMP_B_INPUT7**
- **COMP_B_INPUT8**
- **COMP_B_INPUT9**
- **COMP_B_INPUT10**
- **COMP_B_INPUT11**
- **COMP_B_INPUT12**
- **COMP_B_INPUT13**
- **COMP_B_INPUT14**
- **COMP_B_INPUT15**
- **COMP_B_VREF**
  Modified bits are **CBPDx** of **CBCTL3** register.

Returns:
None

9.2.2.8 void COMP_B_enableInterrupt (uint16_t *baseAddress*, uint16_t *interruptMask*)

Enables selected COMP_B interrupt sources.

Enables the indicated COMP_B 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the COMP_B module.
interruptMask is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following:
- **COMP_B_OUTPUT_INT** - Output interrupt
- **COMP_B_OUTPUTINVERTED_INT** - Output interrupt inverted polarity
  Modified bits of **CBINT** register.

Returns:
None

9.2.2.9 uint8_t COMP_B_getInterruptStatus (uint16_t *baseAddress*, uint16_t *interruptFlagMask*)

Gets the current COMP_B interrupt status.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the COMP_B module.

interruptFlagMask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:

- **COMP_B_OUTPUT_FLAG** - Output interrupt
- **COMP_B_OUTPUTINVERTED_FLAG** - Output interrupt inverted polarity

Returns:

Logical OR of any of the following:

- **COMP_B_OUTPUT_FLAG** Output interrupt
- **COMP_B_OUTPUTINVERTED_FLAG** Output interrupt inverted polarity indicating the status of the masked interrupts

9.2.2.10 bool COMP_B_init (uint16_t *baseAddress*, uint8_t *positiveTerminalInput*, uint8_t *negativeTerminalInput*, uint16_t *powerModeSelect*, uint8_t *outputFilterEnableAndDelayLevel*, uint16_t *invertedOutputPolarity*)

DEPRECATED - Initializes the COMP_B Module.

Upon successful initialization of the COMP_B module, this function will have reset all necessary register bits and set the given options in the registers. To actually use the COMP_B module, the COMP_B_enable() function must be explicitly called before use. If a Reference Voltage is set to a terminal, the Voltage should be set using the COMP_B_setReferenceVoltage() function.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	90
TI Compiler 4.2.1	Size	66
TI Compiler 4.2.1	Speed	66
IAR 5.51.6	None	72
IAR 5.51.6	Size	50
IAR 5.51.6	Speed	62
MSPGCC 4.8.0	None	90
MSPGCC 4.8.0	Size	40
MSPGCC 4.8.0	Speed	168

Parameters:

baseAddress is the base address of the COMP_B module.

positiveTerminalInput selects the input to the positive terminal. Valid values are:

- **COMP_B_INPUT0** [Default]
- **COMP_B_INPUT1**
- **COMP_B_INPUT2**
- **COMP_B_INPUT3**
- **COMP_B_INPUT4**
- **COMP_B_INPUT5**
- **COMP_B_INPUT6**
- **COMP_B_INPUT7**
- **COMP_B_INPUT8**
- **COMP_B_INPUT9**
- **COMP_B_INPUT10**
- **COMP_B_INPUT11**
- **COMP_B_INPUT12**
- **COMP_B_INPUT13**
- **COMP_B_INPUT14**
- **COMP_B_INPUT15**
- **COMP_B_VREF**
  Modified bits are **CBIPEN** and **CBIPSEL** of **CBCTL0** register; bits **CBRSEL** of **CBCTL2** register; bits **CBPDx** of **CBCTL3** register.

negativeTerminalInput selects the input to the negative terminal. Valid values are:

- **COMP_B_INPUT0** [Default]
- **COMP_B_INPUT1**
- **COMP_B_INPUT2**
- **COMP_B_INPUT3**
- **COMP_B_INPUT4**
- **COMP_B_INPUT5**
- **COMP_B_INPUT6**
- **COMP_B_INPUT7**
- **COMP_B_INPUT8**
- **COMP_B_INPUT9**
- **COMP_B_INPUT10**
- **COMP_B_INPUT11**

- **COMP_B_INPUT12**
- **COMP_B_INPUT13**
- **COMP_B_INPUT14**
- **COMP_B_INPUT15**
- **COMP_B_VREF**
  Modified bits are **CBIMEN** and **CBIMSEL** of **CBCTL0** register; bits **CBRSEL** of **CBCTL2** register; bits **CBPDx** of **CBCTL3** register.

powerModeSelect selects the power mode at which the COMP_B module will operate at. Valid values are:

- **COMP_B_POWERMODE_HIGHSPEED** [Default]
- **COMP_B_POWERMODE_NORMALMODE**
- **COMP_B_POWERMODE_ULTRALOWPOWER**
  Modified bits are **CBWRMD** of **CBCTL1** register.

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:

- **COMP_B_FILTEROUTPUT_OFF** [Default]
- **COMP_B_FILTEROUTPUT_DLYLVL1**
- **COMP_B_FILTEROUTPUT_DLYLVL2**
- **COMP_B_FILTEROUTPUT_DLYLVL3**
- **COMP_B_FILTEROUTPUT_DLYLVL4**
  Modified bits are **CBFDLY** and **CBF** of **CBCTL1** register.

invertedOutputPolarity controls if the output will be inverted or not Valid values are:

- **COMP_B_NORMALOUTPUTPOLARITY** [Default]
- **COMP_B_INVERTEDOUTPUTPOLARITY**

Returns:

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

9.2.2.11 bool COMP_B_initialize (uint16_t *baseAddress*, COMP_B_initializeParam ∗ *param*)

Initializes the COMP_B Module.

Upon successful initialization of the COMP_B module, this function will have reset all necessary register bits and set the given options in the registers. To actually use the COMP_B module, the COMP_B_enable() function must be explicitly called before use. If a Reference Voltage is set to a terminal, the Voltage should be set using the COMP_B_setReferenceVoltage() function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	192
TI Compiler 4.2.1	Size	118
TI Compiler 4.2.1	Speed	118
IAR 5.51.6	None	178
IAR 5.51.6	Size	146
IAR 5.51.6	Speed	154
MSPGCC 4.8.0	None	310
MSPGCC 4.8.0	Size	152
MSPGCC 4.8.0	Speed	172

Parameters:

baseAddress is the base address of the COMP_B module.

param is the pointer to struct for initialization.

Returns:

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

9.2.2.12 void COMP_B_interruptSetEdgeDirection (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	44
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	28
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	20

Parameters:

baseAddress is the base address of the COMP_B 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:

- **COMP_B_FALLINGEDGE** [Default] - sets the bit to generate an interrupt when the output of the COMP_B falls from HIGH to LOW if the normal interrupt bit is set(and LOW to HIGH if the inverted interrupt enable bit is set).

- **COMP_B_RISINGEDGE** - sets the bit to generate an interrupt when the output of the COMP_B rises from LOW to HIGH if the normal interrupt bit is set(and HIGH to LOW if the inverted interrupt enable bit is set).
  Modified bits are **CBIES** of **CBCTL1** register.

Returns:
None

9.2.2.13 void COMP_B_interruptToggleEdgeDirection (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.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the COMP_B module.

Returns:
None

9.2.2.14 void COMP_B_IOSwap (uint16_t *baseAddress*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the COMP_B module.

Returns:
None

9.2.2.15 uint16_t COMP_B_outputValue (uint16_t *baseAddress*)

Returns the output value of the COMP_B module.

Returns the output value of the COMP_B module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the COMP_B module.

Returns:
One of the following:

- **COMP_B_LOW**
- **COMP_B_HIGH**
  indicating the output value of the COMP_B module

9.2.2.16 void COMP_B_setReferenceVoltage (uint16_t *baseAddress*, uint16_t *supplyVoltageReferenceBase*, uint16_t *lowerLimitSupplyVoltageFractionOf32*, uint16_t *upperLimitSupplyVoltageFractionOf32*, uint16_t *referenceAccuracy*)

DEPRECATED - Generates a Reference Voltage to the terminal selected during initialization.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	64
IAR 5.51.6	Size	42
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	132

Parameters:

baseAddress is the base address of the COMP_B module.

supplyVoltageReferenceBase decides the source and max amount of Voltage that can be used as a reference. Valid values are:
- **COMP_B_VREFBASE_VCC**
- **COMP_B_VREFBASE1_5V**
- **COMP_B_VREFBASE2_0V**
- **COMP_B_VREFBASE2_5V**
  Modified bits are **CBREFL** of **CBCTL2** register.

lowerLimitSupplyVoltageFractionOf32 is the numerator of the equation to generate the reference voltage for the lower limit reference voltage.
Modified bits are **CBREF0** of **CBCTL2** register.

upperLimitSupplyVoltageFractionOf32 is the numerator of the equation to generate the reference voltage for the upper limit reference voltage.
Modified bits are **CBREF1** of **CBCTL2** register.

referenceAccuracy is the reference accuracy setting of the COMP_B. Clocked is for low power/low accuracy. Valid values are:
- **COMP_B_ACCURACY_STATIC**
- **COMP_B_ACCURACY_CLOCKED**
  Modified bits are **CDREFACC** of **CDCTL2** register.

Returns:

None

9.2.2.17 void COMP_B_shortInputs (uint16_t *baseAddress*)

Shorts the two input pins chosen during initialization.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the COMP_B module.

Returns:
None

9.2.2.18 void COMP_B_unshortInputs (uint16_t *baseAddress*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the COMP_B module.

Returns:
None

9.3 Programming Example

The following example shows how to initialize and use the COMP_B API to turn on an LED when the input to the positive terminal is higher than the input to the negative terminal.

```
// Initialize the Comparator B module
/* Base Address of Comparator B,
  Pin CB0 to Positive(+) Terminal,
  Reference Voltage to Negative(-) Terminal,
  Normal Power Mode,
  Output Filter On with minimal delay,
  Non-Inverted Output Polarity
*/
COMP_B_init(COMP_B_BASE,
          COMP_B_INPUT0,
          COMP_B_VREF,
          COMP_B_POWERMODE_NORMALMODE,
          COMP_B_FILTEROUTPUT_DLYLVL1,
          COMP_B_NORMALOUTPUTPOLARITY
          );

// Set the reference voltage that is being supplied to the (-) terminal
/* Base Address of Comparator B,
 Reference Voltage of 2.0 V,
 Upper Limit of 2.0*(32/32) = 2.0V,
 Lower Limit of 2.0*(32/32) = 2.0V
*/
COMP_B_setReferenceVoltage(COMP_B_BASE,
                         COMP_B_VREFBASE2_5V,
                         32,
                         32
                         );

// Allow power to Comparator module
COMP_B_enable(COMP_B_BASE);

// delay for the reference to settle
__delay_cycles(75);
```

10 Cyclical Redundancy Check (CRC)

10.1 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).

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	146
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	60
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	168
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

10.2 API Functions

Functions

- uint16_t CRC_getData (uint16_t baseAddress)
- uint16_t CRC_getResult (uint16_t baseAddress)
- uint16_t CRC_getResultBitsReversed (uint16_t baseAddress)
- void CRC_set16BitData (uint16_t baseAddress, uint16_t dataIn)
- void CRC_set16BitDataReversed (uint16_t baseAddress, uint16_t dataIn)
- void CRC_set8BitData (uint16_t baseAddress, uint8_t dataIn)
- void CRC_set8BitDataReversed (uint16_t baseAddress, uint8_t dataIn)
- void CRC_setSeed (uint16_t baseAddress, uint16_t seed)

10.2.1 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()

10.2.2 Function Documentation

10.2.2.1 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	4
TI Compiler 4.2.1	Speed	4
IAR 5.51.6	None	4
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	14
MSPGCC 4.8.0	Size	4
MSPGCC 4.8.0	Speed	4

Parameters:
baseAddress is the base address of the CRC module.

Returns:
The value currently in the data register

10.2.2.2 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	16
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the CRC module.

Returns:
The value currently in the data register

10.2.2.3 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the CRC module.

Returns:
The bit-wise reversed format of the Signature Result

10.2.2.4 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

10.2.2.5 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

10.2.2.6 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

10.2.2.7 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

10.2.2.8 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	10
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

10.3 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);
```

11 12-bit Digital-to-Analog Converter (DAC12_A)

11.1 Introduction

The 12-Bit Digital-to-Analog (DAC12_A) API provides a set of functions for using the MSP430Ware DAC12_A modules. Functions are provided to initialize setup the DAC12_A modules, calibrate the output signal, and manage the interrupts for the DAC12_A modules.

The DAC12_A module provides the ability to convert digital values into an analog signal for output to a pin. The DAC12_A can generate signals from 0 to Vcc from an 8- or 12-bit value. There can be one or two DAC12_A modules in a device, and if there are two they can be grouped together to create two analog signals in simultaneously. There are 3 ways to latch data in to the DAC module, and those are by software with the startConversion API function call, as well as by the Timer A output of CCR1 or Timer B output of CCR2.

The calibration API will unlock and start calibration, then wait for the calibration to end before locking it back up, all in one API. There are also functions to read out the calibration data, as well as be able to set it manually.

The DAC12_A module can generate one interrupt for each DAC module. It will generate the interrupt when the data has been latched into the DAC module to be output into an analog signal.

This driver is contained in `dac12_a.c`, with `dac12_a.h` containing the API definitions for use by applications.

11.2 API Functions

Functions

- void DAC12_A_calibrateOutput (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_clearInterrupt (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_disable (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_disableConversions (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_disableGrouping (uint16_t baseAddress)
- void DAC12_A_disableInterrupt (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_enableConversions (uint16_t baseAddress, uint8_t submoduleSelect)
- void DAC12_A_enableGrouping (uint16_t baseAddress)
- void DAC12_A_enableInterrupt (uint16_t baseAddress, uint8_t submoduleSelect)
- uint16_t DAC12_A_getCalibrationData (uint16_t baseAddress, uint8_t submoduleSelect)
- uint32_t DAC12_A_getDataBufferMemoryAddressForDMA (uint16_t baseAddress, uint8_t submoduleSelect)
- uint16_t DAC12_A_getInterruptStatus (uint16_t baseAddress, uint8_t submoduleSelect)
- bool DAC12_A_init (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t outputSelect, uint16_t positiveReferenceVoltage, uint16_t outputVoltageMultiplier, uint8_t amplifierSetting, uint16_t conversionTriggerSelect)
- bool DAC12_A_initialize (uint16_t baseAddress, DAC12_A_initializeParam ∗param)
- void DAC12_A_setAmplifierSetting (uint16_t baseAddress, uint8_t submoduleSelect, uint8_t amplifierSetting)
- void DAC12_A_setCalibrationOffset (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t calibrationOffsetValue)
- void DAC12_A_setData (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t data)
- void DAC12_A_setInputDataFormat (uint16_t baseAddress, uint8_t submoduleSelect, uint8_t inputJustification, uint8_t inputSign)
- void DAC12_A_setResolution (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t resolutionSelect)

11.2.1 Detailed Description

The DAC12_A API is broken into three groups of functions: those that deal with initialization and conversions, those that deal with calibration of the output, and those that handle interrupts.

The DAC12_A initialization and conversion functions are

- DAC12_A_init()
- DAC12_A_setAmplifierSetting()
- DAC12_A_disable()
- DAC12_A_enableGrouping()
- DAC12_A_disableGrouping()
- DAC12_A_enableConversions()
- DAC12_A_setData()
- DAC12_A_disableConversions()
- DAC12_A_setResolution()
- DAC12_A_setInputDataFormat()
- DAC12_A_getDataBufferMemoryAddressForDMA()

Calibration features of the DAC12_A are handled by

- DAC12_A_calibrateOutput()
- DAC12_A_getCalibrationData()
- DAC12_A_setCalibrationOffset()

The DAC12_A interrupts are handled by

- DAC12_A_enableInterrupt()
- DAC12_A_disableInterrupt()
- DAC12_A_getInterruptStatus()
- DAC12_A_clearInterrupt()

11.2.2 Function Documentation

11.2.2.1 void DAC12_A_calibrateOutput (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Calibrates the output offset.

This function disables the calibration lock, starts the calibration, whats for the calibration to complete, and then re-locks the calibration lock. Please note, this function should be called after initializing the dac12 module, and before using it.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Modified bits are **DAC12CALON** of **DAC12_xCTL0** register; bits **DAC12PW** of **DAC12_xCALCTL** register.

Returns:

None

11.2.2.2 void DAC12_A_clearInterrupt (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Clears the DAC12_A module interrupt flag.

The DAC12_A module interrupt flag is cleared, so that it no longer asserts. Note that an interrupt is not thrown when DAC12_A_TRIGGER_AUTO has been set for the parameter conversionTriggerSelect in initialization.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Modified bits are **DAC12IFG** of **DAC12_xCTL0** register.

Returns:

None

11.2.2.3 void DAC12_A_disable (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Clears the amplifier settings to disable the DAC12_A module.

This function clears the amplifier settings for the selected DAC12_A module to disable the DAC12_A module.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Modified bits are **DAC12AMP_7** of **DAC12_xCTL0** register.

Returns:

None

11.2.2.4 void DAC12_A_disableConversions (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Disables triggers to start conversions.

This function is used to disallow triggers to start a conversion. Note that this function does not have any affect if DAC12_A_TRIGGER_AUTO was set for the conversionTriggerSelect parameter during initialization.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Modified bits are **DAC12ENC** of **DAC12_xCTL0** register.

Returns:

None

11.2.2.5 void DAC12_A_disableGrouping (uint16_t *baseAddress*)

Disables grouping of two DAC12_A modules in a dual DAC12_A system.

This function disables grouping of two DAC12_A modules in a dual DAC12_A system.

Parameters:
baseAddress is the base address of the DAC12_A module.

Returns:
None

11.2.2.6 void DAC12_A_disableInterrupt (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Disables the DAC12_A module interrupt source.

Enables the DAC12_A module 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 DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Returns:
None

11.2.2.7 void DAC12_A_enableConversions (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Enables triggers to start conversions.

This function is used to allow triggers to start a conversion. Note that this function does not need to be used if DAC12_A_TRIGGER_AUTO was set for the conversionTriggerSelect parameter during initialization. If DAC grouping is enabled, this has to be called for both DAC's.

Parameters:
baseAddress is the base address of the DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Modified bits are **DAC12ENC** of **DAC12_xCTL0** register.

Returns:
None

11.2.2.8 void DAC12_A_enableGrouping (uint16_t *baseAddress*)

Enables grouping of two DAC12_A modules in a dual DAC12_A system.

This function enables grouping two DAC12_A modules in a dual DAC12_A system. Both DAC12_A modules will work in sync, converting data at the same time. To convert data, the same trigger should be set for both DAC12_A modules during initialization (which should not be DAC12_A_TRIGGER_ENCBYPASS), the enableConversions() function needs to be called with both DAC12_A modules, and data needs to be set for both DAC12_A modules separately.

Parameters:
baseAddress is the base address of the DAC12_A module.

Modified bits are **DAC12GRP** of **DAC12_xCTL0** register.

Returns:
None

11.2.2.9 void DAC12_A_enableInterrupt (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Enables the DAC12_A module interrupt source.

This function to enable the DAC12_A module interrupt, which throws an interrupt when the data buffer is available for new data to be set. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Note that an interrupt is not thrown when DAC12_A_TRIGGER_AUTO has been set for the parameter conversionTriggerSelect in initialization. Does not clear interrupt flags.

Parameters:
baseAddress is the base address of the DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Returns:
None

11.2.2.10 uint16_t DAC12_A_getCalibrationData (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Returns the calibrated offset of the output buffer.

This function returns the calibrated offset of the output buffer. The output buffer offset is used to obtain accurate results from the output pin. This function should only be used while the calibration lock is enabled. Only the lower byte of the word of the register is returned, and the value is between -128 and +127.

Parameters:
baseAddress is the base address of the DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Returns:
The calibrated offset of the output buffer.

11.2.2.11 uint32_t DAC12_A_getDataBufferMemoryAddressForDMA (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Returns the address of the specified DAC12_A data buffer for the DMA module.

Returns the address of the specified memory buffer. This can be used in conjunction with the DMA to obtain the data directly from memory.

Parameters:
baseAddress is the base address of the DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Returns:
The address of the specified memory buffer

11.2.2.12 uint16_t DAC12_A_getInterruptStatus (uint16_t *baseAddress*, uint8_t *submoduleSelect*)

Returns the status of the DAC12_A module interrupt flag.

This function returns the status of the DAC12_A module interrupt flag. Note that an interrupt is not thrown when DAC12_A_TRIGGER_AUTO has been set for the conversionTriggerSelect parameter in initialization.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

Returns:
One of the following:

- **DAC12_A_INT_ACTIVE**
- **DAC12_A_INT_INACTIVE**
  indicating the status for the selected DAC12_A module

11.2.2.13 bool DAC12_A_init (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint16_t *outputSelect*, uint16_t *positiveReferenceVoltage*, uint16_t *outputVoltageMultiplier*, uint8_t *amplifierSetting*, uint16_t *conversionTriggerSelect*)

DEPRECATED - Initializes the DAC12_A module with the specified settings.

This function initializes the DAC12_A module with the specified settings. Upon successful completion of the initialization of this module the control registers and interrupts of this module are all reset, and the specified variables will be set. Please note, that if conversions are enabled with the enableConversions() function, then disableConversions() must be called before re-initializing the DAC12_A module with this function.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

outputSelect selects the output pin that the selected DAC12_A module will output to. Valid values are:
- **DAC12_A_OUTPUT_1** [Default]
- **DAC12_A_OUTPUT_2**
  Modified bits are **DAC12OPS** of **DAC12_xCTL0** register.

positiveReferenceVoltage is the upper limit voltage that the data can be converted in to. Valid values are:
- **DAC12_A_VREF_AVCC** [Default]
- **DAC12_A_VREF_INT**
- **DAC12_A_VREF_EXT**
  Modified bits are **DAC12SREFx** of **DAC12_xCTL0** register.

outputVoltageMultiplier is the multiplier of the Vout voltage. Valid values are:
- **DAC12_A_VREFx1** [Default]
- **DAC12_A_VREFx2**
- **DAC12_A_VREFx3**
  Modified bits are **DAC12IR** of **DAC12_xCTL0** register; bits **DAC12OG** of **DAC12_xCTL1** register.

amplifierSetting is the setting of the settling speed and current of the Vref+ and the Vout buffer. Valid values are:
- **DAC12_A_AMP_OFF_PINOUTHIGHZ** [Default] - Initialize the DAC12_A Module with settings, but do not turn it on.

- **DAC12_A_AMP_OFF_PINOUTLOW** - Initialize the DAC12_A Module with settings, and allow it to take control of the selected output pin to pull it low (Note: this takes control away port mapping module).
- **DAC12_A_AMP_LOWIN_LOWOUT** - Select a slow settling speed and current for Vref+ input buffer and for Vout output buffer.
- **DAC12_A_AMP_LOWIN_MEDOUT** - Select a slow settling speed and current for Vref+ input buffer and a medium settling speed and current for Vout output buffer.
- **DAC12_A_AMP_LOWIN_HIGHOUT** - Select a slow settling speed and current for Vref+ input buffer and a high settling speed and current for Vout output buffer.
- **DAC12_A_AMP_MEDIN_MEDOUT** - Select a medium settling speed and current for Vref+ input buffer and for Vout output buffer.
- **DAC12_A_AMP_MEDIN_HIGHOUT** - Select a medium settling speed and current for Vref+ input buffer and a high settling speed and current for Vout output buffer.
- **DAC12_A_AMP_HIGHIN_HIGHOUT** - Select a high settling speed and current for Vref+ input buffer and for Vout output buffer.
  Modified bits are **DAC12AMPx** of **DAC12_xCTL0** register.

conversionTriggerSelect selects the trigger that will start a conversion. Valid values are:

- **DAC12_A_TRIGGER_ENCBYPASS** [Default] - Automatically converts data as soon as it is written into the data buffer. (Note: Do not use this selection if grouping DAC's).
- **DAC12_A_TRIGGER_ENC** - Requires a call to enableConversions() to allow a conversion, but starts a conversion as soon as data is written to the data buffer (Note: with DAC12_A module's grouped, data has to be set in BOTH DAC12_A data buffers to start a conversion).
- **DAC12_A_TRIGGER_TA** - Requires a call to enableConversions() to allow a conversion, and a rising edge of Timer_A's Out1 (TA1) to start a conversion.
- **DAC12_A_TRIGGER_TB** - Requires a call to enableConversions() to allow a conversion, and a rising edge of Timer_B's Out2 (TB2) to start a conversion.
  Modified bits are **DAC12LSELx** of **DAC12_xCTL0** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

11.2.2.14 bool DAC12_A_initialize (uint16_t *baseAddress*, DAC12_A_initializeParam ∗ *param*)

Initializes the DAC12_A module with the specified settings.

This function initializes the DAC12_A module with the specified settings. Upon successful completion of the initialization of this module the control registers and interrupts of this module are all reset, and the specified variables will be set. Please note, that if conversions are enabled with the enableConversions() function, then disableConversions() must be called before re-initializing the DAC12_A module with this function.

Parameters:
baseAddress is the base address of the DAC12_A module.
param is the pointer to struct for initialization.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

11.2.2.15 void DAC12_A_setAmplifierSetting (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint8_t *amplifierSetting*)

Sets the amplifier settings for the Vref+ and Vout buffers.

This function sets the amplifier settings of the DAC12_A module for the Vref+ and Vout buffers without re-initializing the DAC12_A module. This can be used to disable the control of the pin by the DAC12_A module.

Parameters:
baseAddress is the base address of the DAC12_A module.
submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

amplifierSetting is the setting of the settling speed and current of the Vref+ and the Vout buffer. Valid values are:

- **DAC12_A_AMP_OFF_PINOUTHIGHZ** [Default] - Initialize the DAC12_A Module with settings, but do not turn it on.
- **DAC12_A_AMP_OFF_PINOUTLOW** - Initialize the DAC12_A Module with settings, and allow it to take control of the selected output pin to pull it low (Note: this takes control away port mapping module).
- **DAC12_A_AMP_LOWIN_LOWOUT** - Select a slow settling speed and current for Vref+ input buffer and for Vout output buffer.
- **DAC12_A_AMP_LOWIN_MEDOUT** - Select a slow settling speed and current for Vref+ input buffer and a medium settling speed and current for Vout output buffer.
- **DAC12_A_AMP_LOWIN_HIGHOUT** - Select a slow settling speed and current for Vref+ input buffer and a high settling speed and current for Vout output buffer.
- **DAC12_A_AMP_MEDIN_MEDOUT** - Select a medium settling speed and current for Vref+ input buffer and for Vout output buffer.
- **DAC12_A_AMP_MEDIN_HIGHOUT** - Select a medium settling speed and current for Vref+ input buffer and a high settling speed and current for Vout output buffer.
- **DAC12_A_AMP_HIGHIN_HIGHOUT** - Select a high settling speed and current for Vref+ input buffer and for Vout output buffer.

Returns:
None

11.2.2.16 void DAC12_A_setCalibrationOffset (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint16_t *calibrationOffsetValue*)

Returns the calibrated offset of the output buffer.

This function is used to manually set the calibration offset value. The calibration is automatically unlocked and re-locked to be able to allow for the offset value to be set.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:

- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

calibrationOffsetValue calibration offset value

Modified bits are **DAC12LOCK** of **DAC12_xCALDAT** register; bits **DAC12PW** of **DAC12_xCTL0** register; bits **DAC12PW** of **DAC12_xCALCTL** register.

Returns:
None

11.2.2.17 void DAC12_A_setData (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint16_t *data*)

Sets the given data into the buffer to be converted.

This function is used to set the given data into the data buffer of the DAC12_A module. The data given should be in the format set (12-bit Unsigned, Right-justified by default). Note if DAC12_A_TRIGGER_AUTO was set for the conversionTriggerSelect during initialization then using this function will set the data and automatically trigger a conversion. If any other trigger was set during initialization, then the DAC12_A_enableConversions() function needs to be called before a conversion can be started. If grouping DAC's and DAC12_A_TRIGGER_ENC was set during initialization, then both data buffers must be set before a conversion will be started.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

data is the data to be set into the DAC12_A data buffer to be converted.
Modified bits are **DAC12_DATA** of **DAC12_xDAT** register.

Modified bits of **DAC12_xDAT** register.

Returns:
None

11.2.2.18 void DAC12_A_setInputDataFormat (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint8_t *inputJustification*, uint8_t *inputSign*)

Sets the input data format for the DAC12_A module.

This function sets the input format for the binary data to be converted.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

inputJustification is the justification of the data to be converted. Valid values are:
- **DAC12_A_JUSTIFICATION_RIGHT** [Default]
- **DAC12_A_JUSTIFICATION_LEFT**
  Modified bits are **DAC12DFJ** of **DAC12_xCTL1** register.

inputSign is the sign of the data to be converted. Valid values are:
- **DAC12_A_UNSIGNED_BINARY** [Default]
- **DAC12_A_SIGNED_2SCOMPLEMENT**
  Modified bits are **DAC12DF** of **DAC12_xCTL0** register.

Returns:
None

11.2.2.19 void DAC12_A_setResolution (uint16_t *baseAddress*, uint8_t *submoduleSelect*, uint16_t *resolutionSelect*)

Sets the resolution to be used by the DAC12_A module.

This function sets the resolution of the data to be converted.

Parameters:

baseAddress is the base address of the DAC12_A module.

submoduleSelect decides which DAC12_A sub-module to configure. Valid values are:
- **DAC12_A_SUBMODULE_0**
- **DAC12_A_SUBMODULE_1**

resolutionSelect is the resolution to use for conversions. Valid values are:
- **DAC12_A_RESOLUTION_8BIT**
- **DAC12_A_RESOLUTION_12BIT** [Default]
  Modified bits are **DAC12RES** of **DAC12_xCTL0** register.

Modified bits are **DAC12ENC** and **DAC12RES** of **DAC12_xCTL0** register.

Returns:
None

11.3 Programming Example

The following example shows how to initialize and use the DAC12_A API to output a 1.5V analog signal.

```
DAC12_A_init (DAC12_A_BASE,
            DAC12_A_SUBMODULE_0,            // Initialize DAC12_A_0
            DAC12_A_OUTPUT_1,               // Choose P6.6 as output
            DAC12_A_VREF_AVCC,              // Use AVcc as Vref+
            DAC12_A_VREFx1,                 // Multiply Vout by 1
            DAC12_A_AMP_MEDIN_MEDOUT,       // Use medium settling speed/current
            DAC12_A_TRIGGER_ENCBYPASS       // Auto trigger as soon as data is set
            );

// Calibrate output buffer for DAC12_A_0
DAC12_A_calibrateOutput(DAC12_A_BASE,
                      DAC12_A_SUBMODULE_0);

DAC12_A_setData(DAC12_A_BASE,
              DAC12_A_SUBMODULE_0,          // Set 0x7FF (~1.5V)
              0x7FF                        // into data buffer for DAC12_A_0
              );
```

12 Direct Memory Access (DMA)

12.1 Introduction

The Direct Memory Access (DMA) API provides a set of functions for using the MSP430Ware DMA modules. Functions are provided to initialize and setup each DMA channel with the source and destination addresses, manage the interrupts for each channel, and set bits that affect all DMA channels.

The DMA module provides the ability to move data from one address in the device to another, and that includes other peripheral addresses to RAM or vice-versa, all without the actual use of the CPU. Please be advised, that the DMA module does halt the CPU for 2 cycles while transferring, but does not have to edit any registers or anything. The DMA can transfer by bytes or words at a time, and will automatically increment or decrement the source or destination address if desired. There are also 6 different modes to transfer by, including single-transfer, block-transfer, and burst-block-transfer, as well as repeated versions of those three different kinds which allows transfers to be repeated without having re-enable transfers.

The DMA settings that affect all DMA channels include prioritization, from a fixed priority to dynamic round-robin priority. Another setting that can be changed is when transfers occur, the CPU may be in a read-modify-write operation which can be disastrous to time sensitive material, so this can be disabled. And Non-Maskable-Interrupts can indeed be maskable to the DMA module if not enabled.

The DMA module can generate one interrupt per channel. The interrupt is only asserted when the specified amount of transfers has been completed. With single-transfer, this occurs when that many single transfers have occurred, while with block or burst-block transfers, once the block is completely transferred the interrupt is asserted.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	612
TI Compiler 4.2.1	Size	370
TI Compiler 4.2.1	Speed	376
IAR 5.51.6	None	466
IAR 5.51.6	Size	362
IAR 5.51.6	Speed	382
MSPGCC 4.8.0	None	980
MSPGCC 4.8.0	Size	406
MSPGCC 4.8.0	Speed	430

12.2 API Functions

Functions

- void DMA_clearInterrupt (uint8_t channelSelect)
- void DMA_clearNMIAbort (uint8_t channelSelect)
- void DMA_disableInterrupt (uint8_t channelSelect)
- void DMA_disableNMIAbort (void)
- void DMA_disableRoundRobinPriority (void)
- void DMA_disableTransferDuringReadModifyWrite (void)

- void DMA_disableTransfers (uint8_t channelSelect)
- void DMA_enableInterrupt (uint8_t channelSelect)
- void DMA_enableNMIAbort (void)
- void DMA_enableRoundRobinPriority (void)
- void DMA_enableTransferDuringReadModifyWrite (void)
- void DMA_enableTransfers (uint8_t channelSelect)
- uint16_t DMA_getInterruptStatus (uint8_t channelSelect)
- bool DMA_init (uint8_t channelSelect, uint16_t transferModeSelect, uint16_t transferSize, uint8_t triggerSourceSelect, uint8_t transferUnitSelect, uint8_t triggerTypeSelect)
- bool DMA_initialize (DMA_initializeParam ∗param)
- uint16_t DMA_NMIAbortStatus (uint8_t channelSelect)
- void DMA_setDstAddress (uint8_t channelSelect, uint32_t dstAddress, uint16_t directionSelect)
- void DMA_setSrcAddress (uint8_t channelSelect, uint32_t srcAddress, uint16_t directionSelect)
- void DMA_setTransferSize (uint8_t channelSelect, uint16_t transferSize)
- void DMA_startTransfer (uint8_t channelSelect)

12.2.1 Detailed Description

The DMA API is broken into three groups of functions: those that deal with initialization and transfers, those that handle interrupts, and those that affect all DMA channels.

The DMA initialization and transfer functions are: DMA_init() DMA_setSrcAddress() DMA_setDstAddress() DMA_enableTransfers() DMA_disableTransfers() DMA_startTransfer() DMA_setTransferSize()

The DMA interrupts are handled by: DMA_enableInterrupt() DMA_disableInterrupt() DMA_getInterruptStatus() DMA_clearInterrupt() DMA_NMIAbortStatus() DMA_clearNMIAbort()

Features of the DMA that affect all channels are handled by: DMA_disableTransferDuringReadModifyWrite() DMA_enableTransferDuringReadModifyWrite() DMA_enableRoundRobinPriority() DMA_disableRoundRobinPriority() DMA_enableNMIAbort() DMA_disableNMIAbort()

12.2.2 Function Documentation

12.2.2.1 void DMA_clearInterrupt (uint8_t *channelSelect*)

Clears the interrupt flag for the selected channel.

This function clears the DMA interrupt flag is cleared, so that it no longer asserts.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

channelSelect is the specified channel to clear the interrupt flag for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**

- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.2.2.2 void DMA_clearNMIAbort (uint8_t *channelSelect*)

Clears the status of the NMIAbort to proceed with transfers for the selected channel.

This function clears the status of the NMI Abort flag for the selected channel to allow for transfers on the channel to continue.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
channelSelect is the specified channel to clear the NMI Abort flag for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.2.2.3 void DMA_disableInterrupt (uint8_t *channelSelect*)

Disables the DMA interrupt for the selected channel.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

channelSelect is the specified channel to disable the interrupt for. Valid values are:

- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:

None

12.2.2.4 void DMA_disableNMIAbort (void)

Disables any NMI from interrupting a DMA transfer.

This function disables NMI's from interrupting any DMA transfer currently in progress.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:

None

12.2.2.5 void DMA_disableRoundRobinPriority (void)

Disables Round Robin prioritization.

This function disables Round Robin Prioritization, enabling static prioritization of the DMA channels. In static prioritization, the DMA channels are prioritized with the lowest DMA channel index having the highest priority (i.e. DMA Channel 0 has the highest priority).

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

12.2.2.6 void DMA_disableTransferDuringReadModifyWrite (void)

Disables the DMA from stopping the CPU during a Read-Modify-Write Operation to start a transfer.

This function allows the CPU to finish any read-modify-write operations it may be in the middle of before transfers of and DMA channel stop the CPU.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

12.2.2.7 void DMA_disableTransfers (uint8_t *channelSelect*)

Disables transfers from being triggered.

This function disables transfer from being triggered for the selected channel. This function should be called before any re-initialization of the selected DMA channel.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	14
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
channelSelect is the specified channel to disable transfers for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.2.2.8 void DMA_enableInterrupt (uint8_t *channelSelect*)

Enables the DMA interrupt for the selected channel.

Enables the DMA 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
channelSelect is the specified channel to enable the interrupt for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.2.2.9 void DMA_enableNMIAbort (void)

Enables a NMI to interrupt a DMA transfer.

This function allow NMI's to interrupting any DMA transfer currently in progress and stops any future transfers to begin before the NMI is done processing.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

12.2.2.10 void DMA_enableRoundRobinPriority (void)

Enables Round Robin prioritization.

This function enables Round Robin Prioritization of DMA channels. In the case of Round Robin Prioritization, the last DMA channel to have transferred data then has the last priority, which comes into play when multiple DMA channels are ready to transfer at the same time.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

12.2.2.11 void DMA_enableTransferDuringReadModifyWrite (void)

Enables the DMA to stop the CPU during a Read-Modify-Write Operation to start a transfer.

This function allows the DMA to stop the CPU in the middle of a read- modify-write operation to transfer data.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

12.2.2.12 void DMA_enableTransfers (uint8_t *channelSelect*)

Enables transfers to be triggered.

This function enables transfers upon appropriate trigger of the selected trigger source for the selected channel.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	14
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
channelSelect is the specified channel to enable transfer for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.2.2.13 uint16_t DMA_getInterruptStatus (uint8_t *channelSelect*)

Returns the status of the interrupt flag for the selected channel.

Returns the status of the interrupt flag for the selected channel.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
channelSelect is the specified channel to return the interrupt flag status from. Valid values are:

- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
One of the following:

- **DMA_INT_INACTIVE**
- **DMA_INT_ACTIVE**
indicating the status of the current interrupt flag

12.2.2.14 bool DMA_init (uint8_t *channelSelect*, uint16_t *transferModeSelect*, uint16_t *transferSize*, uint8_t *triggerSourceSelect*, uint8_t *transferUnitSelect*, uint8_t *triggerTypeSelect*)

DEPRECATED - Initializes the specified DMA channel.

This function initializes the specified DMA channel. Upon successful completion of initialization of the selected channel the control registers will be cleared and the given variables will be set. Please note, if transfers have been enabled with the enableTransfers() function, then a call to disableTransfers() is necessary before re-initialization. Also note, that the trigger sources are device dependent and can be found in the device family data sheet. The amount of DMA channels available are also device specific.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	84
TI Compiler 4.2.1	Size	74
TI Compiler 4.2.1	Speed	74
IAR 5.51.6	None	74
IAR 5.51.6	Size	70
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	92

Parameters:
channelSelect is the specified channel to initialize. Valid values are:

- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

transferModeSelect is the transfer mode of the selected channel. Valid values are:

- **DMA_TRANSFER_SINGLE** [Default] - Single transfer, transfers disabled after transferAmount of transfers.
- **DMA_TRANSFER_BLOCK** - Multiple transfers of transferAmount, transfers disabled once finished.
- **DMA_TRANSFER_BURSTBLOCK** - Multiple transfers of transferAmount interleaved with CPU activity, transfers disabled once finished.
- **DMA_TRANSFER_REPEATED_SINGLE** - Repeated single transfer by trigger.

- **DMA_TRANSFER_REPEATED_BLOCK** - Multiple transfers of transferAmount by trigger.
- **DMA_TRANSFER_REPEATED_BURSTBLOCK** - Multiple transfers of transferAmount by trigger interleaved with CPU activity.
  Modified bits are **DMADT** of **DMAxCTL** register.

transferSize is the amount of transfers to complete in a block transfer mode, as well as how many transfers to complete before the interrupt flag is set. Valid value is between 1-65535, if 0, no transfers will occur.
Modified bits are **DMAxSZ** of **DMAxSZ** register.

triggerSourceSelect is the source that will trigger the start of each transfer, note that the sources are device specific. Valid values are:

- **DMA_TRIGGERSOURCE_0** [Default]
- **DMA_TRIGGERSOURCE_1**
- **DMA_TRIGGERSOURCE_2**
- **DMA_TRIGGERSOURCE_3**
- **DMA_TRIGGERSOURCE_4**
- **DMA_TRIGGERSOURCE_5**
- **DMA_TRIGGERSOURCE_6**
- **DMA_TRIGGERSOURCE_7**
- **DMA_TRIGGERSOURCE_8**
- **DMA_TRIGGERSOURCE_9**
- **DMA_TRIGGERSOURCE_10**
- **DMA_TRIGGERSOURCE_11**
- **DMA_TRIGGERSOURCE_12**
- **DMA_TRIGGERSOURCE_13**
- **DMA_TRIGGERSOURCE_14**
- **DMA_TRIGGERSOURCE_15**
- **DMA_TRIGGERSOURCE_16**
- **DMA_TRIGGERSOURCE_17**
- **DMA_TRIGGERSOURCE_18**
- **DMA_TRIGGERSOURCE_19**
- **DMA_TRIGGERSOURCE_20**
- **DMA_TRIGGERSOURCE_21**
- **DMA_TRIGGERSOURCE_22**
- **DMA_TRIGGERSOURCE_23**
- **DMA_TRIGGERSOURCE_24**
- **DMA_TRIGGERSOURCE_25**
- **DMA_TRIGGERSOURCE_26**
- **DMA_TRIGGERSOURCE_27**
- **DMA_TRIGGERSOURCE_28**
- **DMA_TRIGGERSOURCE_29**
- **DMA_TRIGGERSOURCE_30**
- **DMA_TRIGGERSOURCE_31**
  Modified bits are **DMAxTSEL** of **DMACTLx** register.

transferUnitSelect is the specified size of transfers. Valid values are:

- **DMA_SIZE_SRCWORD_DSTWORD** [Default]
- **DMA_SIZE_SRCBYTE_DSTWORD**
- **DMA_SIZE_SRCWORD_DSTBYTE**
- **DMA_SIZE_SRCBYTE_DSTBYTE**
  Modified bits are **DMASRCBYTE** and **DMADSTBYTE** of **DMAxCTL** register.

triggerTypeSelect is the type of trigger that the trigger signal needs to be to start a transfer. Valid values are:

- **DMA_TRIGGER_RISINGEDGE** [Default]
- **DMA_TRIGGER_HIGH** - A trigger would be a high signal from the trigger source, to be held high through the length of the transfer(s).
  Modified bits are **DMALEVEL** of **DMAxCTL** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

12.2.2.15 bool DMA_initialize (DMA_initializeParam ∗ *param*)

Initializes the specified DMA channel.

This function initializes the specified DMA channel. Upon successful completion of initialization of the selected channel the control registers will be cleared and the given variables will be set. Please note, if transfers have been enabled with the enableTransfers() function, then a call to disableTransfers() is necessary before re-initialization. Also note, that the trigger sources are device dependent and can be found in the device family data sheet. The amount of DMA channels available are also device specific.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	154
TI Compiler 4.2.1	Size	94
TI Compiler 4.2.1	Speed	100
IAR 5.51.6	None	144
IAR 5.51.6	Size	94
IAR 5.51.6	Speed	92
MSPGCC 4.8.0	None	244
MSPGCC 4.8.0	Size	94
MSPGCC 4.8.0	Speed	102

Parameters:
param is the pointer to struct for initialization.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

12.2.2.16 uint16_t DMA_NMIAbortStatus (uint8_t *channelSelect*)

Returns the status of the NMIAbort for the selected channel.

This function returns the status of the NMI Abort flag for the selected channel. If this flag has been set, it is because a transfer on this channel was aborted due to a interrupt from an NMI.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
channelSelect is the specified channel to return the status of the NMI Abort flag for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**

- **DMA_CHANNEL_7**

Returns:
One of the following:

- **DMA_NOTABORTED**
- **DMA_ABORTED**
indicating the status of the NMIAbort for the selected channel

12.2.2.17 void DMA_setDstAddress (uint8_t *channelSelect*, uint32_t *dstAddress*, uint16_t *directionSelect*)

Sets the destination address and the direction that the destination address will move after a transfer.

This function sets the destination address and the direction that the destination address will move after a transfer is complete. It may be incremented, decremented, or unchanged.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	70
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	46
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	116
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

Parameters:
channelSelect is the specified channel to set the destination address direction for. Valid values are:
- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

dstAddress is the address of where the data will be transferred to.
Modified bits are **DMAxDA** of **DMAxDA** register.

directionSelect is the specified direction of the destination address after a transfer. Valid values are:
- **DMA_DIRECTION_UNCHANGED**
- **DMA_DIRECTION_DECREMENT**
- **DMA_DIRECTION_INCREMENT**
Modified bits are **DMADSTINCR** of **DMAxCTL** register.

Returns:
None

12.2.2.18 void DMA_setSrcAddress (uint8_t *channelSelect*, uint32_t *srcAddress*, uint16_t *directionSelect*)

Sets source address and the direction that the source address will move after a transfer.

This function sets the source address and the direction that the source address will move after a transfer is complete. It may be incremented, decremented or unchanged.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	66
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	42
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	44
MSPGCC 4.8.0	Speed	44

Parameters:

channelSelect is the specified channel to set source address direction for. Valid values are:

- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

srcAddress is the address of where the data will be transferred from.
Modified bits are **DMAxSA** of **DMAxSA** register.

directionSelect is the specified direction of the source address after a transfer. Valid values are:

- **DMA_DIRECTION_UNCHANGED**
- **DMA_DIRECTION_DECREMENT**
- **DMA_DIRECTION_INCREMENT**
  Modified bits are **DMASRCINCR** of **DMAxCTL** register.

Returns:

None

12.2.2.19 void DMA_setTransferSize (uint8_t *channelSelect*, uint16_t *transferSize*)

Sets the specified amount of transfers for the selected DMA channel.

This function sets the specified amount of transfers for the selected DMA channel without having to reinitialize the DMA channel.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

channelSelect is the specified channel to set source address direction for. Valid values are:

- **DMA_CHANNEL_0**

- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

transferSize is the amount of transfers to complete in a block transfer mode, as well as how many transfers to complete before the interrupt flag is set. Valid value is between 1-65535, if 0, no transfers will occur. Modified bits are **DMAxSZ** of **DMAxSZ** register.

Returns:
None

12.2.2.20 void DMA_startTransfer (uint8_t *channelSelect*)

Starts a transfer if using the default trigger source selected in initialization.

This functions triggers a transfer of data from source to destination if the trigger source chosen from initialization is the DMA_TRIGGERSOURCE_0. Please note, this function needs to be called for each (repeated-)single transfer, and when transferAmount of transfers have been complete in (repeated-)block transfers.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
channelSelect is the specified channel to start transfers for. Valid values are:

- **DMA_CHANNEL_0**
- **DMA_CHANNEL_1**
- **DMA_CHANNEL_2**
- **DMA_CHANNEL_3**
- **DMA_CHANNEL_4**
- **DMA_CHANNEL_5**
- **DMA_CHANNEL_6**
- **DMA_CHANNEL_7**

Returns:
None

12.3 Programming Example

The following example shows how to initialize and use the DMA API to transfer words from one spot in RAM to another.

```
 // Initialize and Setup DMA Channel 0
 /*
Base Address of the DMA Module
Configure DMA channel 0
Configure channel for repeated block transfers
DMA interrupt flag will be set after every 16 transfers
Use DMA_startTransfer() function to trigger transfers
Transfer Word-to-Word
Trigger upon Rising Edge of Trigger Source Signal
  */
 DMA_init(DMA_BASE,
          DMA_CHANNEL_0,
          DMA_TRANSFER_REPEATED_BLOCK,
          16,
          DMA_TRIGGERSOURCE_0,
          DMA_SIZE_SRCWORD_DSTWORD,
          DMA_TRIGGER_RISINGEDGE);
 /*
Base Address of the DMA Module
Configure DMA channel 0
Use 0x1C00 as source
Increment source address after every transfer
  */
 DMA_setSrcAddress(DMA_BASE,
                   DMA_CHANNEL_0,
                   0x1C00,
                   DMA_DIRECTION_INCREMENT);
 /*
Base Address of the DMA Module
Configure DMA channel 0
Use 0x1C20 as destination
Increment destination address after every transfer
  */
 DMA_setDstAddress(DMA_BASE,
                   DMA_CHANNEL_0,
                   0x1C20,
                   DMA_DIRECTION_INCREMENT);

 // Enable transfers on DMA channel 0
 DMA_enableTransfers(DMA_BASE,
                     DMA_CHANNEL_0);

 while(1)
 {
   // Start block transfer on DMA channel 0
   DMA_startTransfer(DMA_BASE,
                     DMA_CHANNEL_0);
 }
```

13 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)

13.1 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.

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

13.2 API Functions

Functions

- void EUSCI_A_UART_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void EUSCI_A_UART_disable (uint16_t baseAddress)
- void EUSCI_A_UART_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void EUSCI_A_UART_enable (uint16_t baseAddress)
- void EUSCI_A_UART_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t EUSCI_A_UART_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t EUSCI_A_UART_getReceiveBufferAddress (uint16_t baseAddress)
- uint32_t EUSCI_A_UART_getTransmitBufferAddress (uint16_t baseAddress)
- bool EUSCI_A_UART_init (uint16_t baseAddress, EUSCI_A_UART_initParam ∗param)
- bool EUSCI_A_UART_initAdvance (uint16_t baseAddress, uint8_t selectClockSource, uint16_t clockPrescalar, uint8_t firstModReg, uint8_t secondModReg, uint8_t parity, uint16_t msborLsbFirst, uint16_t numberofStopBits, uint16_t uartMode, uint8_t overSampling)
- uint8_t EUSCI_A_UART_queryStatusFlags (uint16_t baseAddress, uint8_t mask)
- uint8_t EUSCI_A_UART_receiveData (uint16_t baseAddress)
- void EUSCI_A_UART_resetDormant (uint16_t baseAddress)
- void EUSCI_A_UART_selectDeglitchTime (uint16_t baseAddress, uint16_t deglitchTime)
- void EUSCI_A_UART_setDormant (uint16_t baseAddress)
- void EUSCI_A_UART_transmitAddress (uint16_t baseAddress, uint8_t transmitAddress)
- void EUSCI_A_UART_transmitBreak (uint16_t baseAddress)
- void EUSCI_A_UART_transmitData (uint16_t baseAddress, uint8_t transmitData)

13.2.1 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()

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_clearInterruptFlag()
- EUSCI_A_UART_queryStatusFlags()

DMA related

- EUSCI_A_UART_getReceiveBufferAddressForDMA()
- EUSCI_A_UART_getTransmitBufferAddressForDMA()

13.2.2 Function Documentation

13.2.2.1 void EUSCI_A_UART_clearInterruptFlag (uint16_t *baseAddress*, uint8_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

13.2.2.2 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

13.2.2.3 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

13.2.2.4 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

13.2.2.5 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.

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

13.2.2.6 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

13.2.2.7 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

13.2.2.8 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

13.2.2.9 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

13.2.2.10 bool EUSCI_A_UART_initAdvance (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint16_t *clockPrescalar*, uint8_t *firstModReg*, uint8_t *secondModReg*, uint8_t *parity*, uint16_t *msborLsbFirst*, uint16_t *numberofStopBits*, uint16_t *uartMode*, uint8_t *overSampling*)

DEPRECATED - 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.
selectClockSource selects Clock source. Valid values are:
- **EUSCI_A_UART_CLOCKSOURCE_SMCLK**
- **EUSCI_A_UART_CLOCKSOURCE_ACLK**

clockPrescalar is the value to be written into UCBRx bits
firstModReg is First modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device Users Guide. This value is written into UCBRFx bits of UCAxMCTLW.

secondModReg is Second modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device Users Guide. This value is written into UCBRSx bits of UCAxMCTLW.

parity is the desired parity. Valid values are:
- **EUSCI_A_UART_NO_PARITY** [Default]
- **EUSCI_A_UART_ODD_PARITY**
- **EUSCI_A_UART_EVEN_PARITY**

msborLsbFirst controls direction of receive and transmit shift register. Valid values are:
- **EUSCI_A_UART_MSB_FIRST**
- **EUSCI_A_UART_LSB_FIRST** [Default]

numberofStopBits indicates one/two STOP bits Valid values are:
- **EUSCI_A_UART_ONE_STOP_BIT** [Default]
- **EUSCI_A_UART_TWO_STOP_BITS**

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**

overSampling indicates low frequency or oversampling baud generation Valid values are:
- **EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION**
- **EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

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

13.2.2.11 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

13.2.2.12 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.

13.2.2.13 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

13.2.2.14 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

13.2.2.15 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

13.2.2.16 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

13.2.2.17 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

13.2.2.18 void EUSCI_A_UART_transmitData (uint16_t *baseAddress*, uint8_t *transmitData*)

Transmits a byte from the UART Module.

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

13.3 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
if ( STATUS_FAIL == EUSCI_A_UART_init(EUSCI_A0_BASE,
            EUSCI_A_UART_CLOCKSOURCE_ACLK,
            CLOCK_VALUE,
            32768,
            EUSCI_A_UART_NO_PARITY,
            EUSCI_A_UART_LSB_FIRST,
            EUSCI_A_UART_ONE_STOP_BIT,
            EUSCI_A_UART_MODE,
            EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION )){
       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);
```

14 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)

14.1 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.

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

14.2 Functions

Functions

- void EUSCI_A_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
- void EUSCI_A_SPI_changeMasterClock (uint16_t baseAddress, EUSCI_A_SPI_changeMasterClockParam ∗param)
- void EUSCI_A_SPI_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void EUSCI_A_SPI_disable (uint16_t baseAddress)
- void EUSCI_A_SPI_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void EUSCI_A_SPI_enable (uint16_t baseAddress)
- void EUSCI_A_SPI_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t EUSCI_A_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t EUSCI_A_SPI_getReceiveBufferAddress (uint16_t baseAddress)
- uint32_t EUSCI_A_SPI_getTransmitBufferAddress (uint16_t baseAddress)
- void EUSCI_A_SPI_initMaster (uint16_t baseAddress, EUSCI_A_SPI_initMasterParam ∗param)
- void EUSCI_A_SPI_initSlave (uint16_t baseAddress, EUSCI_A_SPI_initSlaveParam ∗param)
- uint16_t EUSCI_A_SPI_isBusy (uint16_t baseAddress)
- void EUSCI_A_SPI_masterChangeClock (uint16_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- void EUSCI_A_SPI_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t clockSourceFrequency, uint32_t desiredSpiClock, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- uint8_t EUSCI_A_SPI_receiveData (uint16_t baseAddress)
- void EUSCI_A_SPI_select4PinFunctionality (uint16_t baseAddress, uint8_t select4PinFunctionality)
- void EUSCI_A_SPI_slaveInit (uint16_t baseAddress, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- void EUSCI_A_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)

14.2.1 Detailed Description

To use the module as a master, the user must call EUSCI_A_SPI_masterInit() 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_slaveInit() 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_masterInit()
- EUSCI_A_SPI_slaveInit()
- 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_clearInterruptFlag()

DMA related

- EUSCI_A_SPI_getReceiveBufferAddressForDMA()
- EUSCI_A_SPI_getTransmitBufferAddressForDMA()

14.2.2 Function Documentation

14.2.2.1 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

14.2.2.2 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

14.2.2.3 void EUSCI_A_SPI_clearInterruptFlag (uint16_t *baseAddress*, uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Parameters:

baseAddress is the base address of the EUSCI_A_SPI module.

mask is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following:

- **EUSCI_A_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns:

None

14.2.2.4 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

14.2.2.5 void EUSCI_A_SPI_disableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Disables individual SPI interrupt sources.

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

Parameters:

baseAddress is the base address of the EUSCI_A_SPI module.

mask is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following:

- **EUSCI_A_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIE** register.

Returns:
None

14.2.2.6 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

14.2.2.7 void EUSCI_A_SPI_enableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Enables individual SPI interrupt sources.

Enables the indicated SPI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters:
baseAddress is the base address of the EUSCI_A_SPI module.
mask is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following:
- **EUSCI_A_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_A_SPI_RECEIVE_INTERRUPT**

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

Returns:
None

14.2.2.8 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

14.2.2.9 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

14.2.2.10 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

14.2.2.11 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

14.2.2.12 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

14.2.2.13 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

14.2.2.14 void EUSCI_A_SPI_masterChangeClock (uint16_t *baseAddress*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*)

DEPRECATED - Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.

Parameters:

baseAddress is the base address of the EUSCI_A_SPI module.

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

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

Returns:

None

14.2.2.15 void EUSCI_A_SPI_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*, uint16_t *msbFirst*, uint16_t *clockPhase*, uint16_t *clockPolarity*, uint16_t *spiMode*)

DEPRECATED - 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.

selectClockSource selects Clock source. Valid values are:
- **EUSCI_A_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_A_SPI_CLOCKSOURCE_SMCLK**

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

msbFirst controls the direction of the receive and transmit shift register. Valid values are:
- **EUSCI_A_SPI_MSB_FIRST**
- **EUSCI_A_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:
- **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity is clock polarity select Valid values are:
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

spiMode is SPI mode select Valid values are:
- **EUSCI_A_SPI_3PIN**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_LOW**

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

Returns:
STATUS_SUCCESS

14.2.2.16 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.

14.2.2.17 void EUSCI_A_SPI_select4PinFunctionality (uint16_t *baseAddress*, uint8_t *select4PinFunctionality*)

Selects 4Pin Functionality.

This function should be invoked only in 4-wire mode. Invoking this function has no effect in 3-wire mode.

Parameters:
baseAddress is the base address of the EUSCI_A_SPI module.
select4PinFunctionality selects 4 pin functionality Valid values are:
- **EUSCI_A_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS**
- **EUSCI_A_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

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

Returns:
None

14.2.2.18 void EUSCI_A_SPI_slaveInit (uint16_t *baseAddress*, uint16_t *msbFirst*, uint16_t *clockPhase*, uint16_t *clockPolarity*, uint16_t *spiMode*)

DEPRECATED - 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.
msbFirst controls the direction of the receive and transmit shift register. Valid values are:
- **EUSCI_A_SPI_MSB_FIRST**
- **EUSCI_A_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:
- **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity is clock polarity select Valid values are:
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

spiMode is SPI mode select Valid values are:
- **EUSCI_A_SPI_3PIN**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_LOW**

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

Returns:
STATUS_SUCCESS

14.2.2.19 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

14.3 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
returnValue = EUSCI_A_SPI_slaveInit(EUSCI_A0_BASE,
        EUSCI_A_SPI_MSB_FIRST,
        EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,
        EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH
        );

if (STATUS_FAIL == returnValue){
        return;
    }

//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
        );
```

15 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)

15.1 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.

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

15.2 Functions

Functions

- void EUSCI_B_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint16_t clockPhase, uint16_t clockPolarity)
- void EUSCI_B_SPI_changeMasterClock (uint16_t baseAddress, EUSCI_B_SPI_changeMasterClockParam ∗param)
- void EUSCI_B_SPI_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void EUSCI_B_SPI_disable (uint16_t baseAddress)
- void EUSCI_B_SPI_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void EUSCI_B_SPI_enable (uint16_t baseAddress)
- void EUSCI_B_SPI_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t EUSCI_B_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t EUSCI_B_SPI_getReceiveBufferAddress (uint16_t baseAddress)
- uint32_t EUSCI_B_SPI_getTransmitBufferAddress (uint16_t baseAddress)
- void EUSCI_B_SPI_initMaster (uint16_t baseAddress, EUSCI_B_SPI_initMasterParam ∗param)
- void EUSCI_B_SPI_initSlave (uint16_t baseAddress, EUSCI_B_SPI_initSlaveParam ∗param)
- uint16_t EUSCI_B_SPI_isBusy (uint16_t baseAddress)
- void EUSCI_B_SPI_masterChangeClock (uint16_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- void EUSCI_B_SPI_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t clockSourceFrequency, uint32_t desiredSpiClock, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- uint8_t EUSCI_B_SPI_receiveData (uint16_t baseAddress)
- void EUSCI_B_SPI_select4PinFunctionality (uint16_t baseAddress, uint8_t select4PinFunctionality)
- void EUSCI_B_SPI_slaveInit (uint16_t baseAddress, uint16_t msbFirst, uint16_t clockPhase, uint16_t clockPolarity, uint16_t spiMode)
- void EUSCI_B_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)

15.2.1 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_clearInterruptFlag()

DMA related

- EUSCI_B_SPI_getReceiveBufferAddressForDMA()
- EUSCI_B_SPI_getTransmitBufferAddressForDMA()

15.2.2 Function Documentation

15.2.2.1 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_ON_NEXT** [Default]
- **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity is clock polarity select Valid values are:

- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

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

Returns:

None

15.2.2.2 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

15.2.2.3 void EUSCI_B_SPI_clearInterruptFlag (uint16_t *baseAddress*, uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Parameters:
baseAddress is the base address of the EUSCI_B_SPI module.
mask is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following:

- **EUSCI_B_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns:
None

15.2.2.4 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

15.2.2.5 void EUSCI_B_SPI_disableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Disables individual SPI interrupt sources.

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

Parameters:
baseAddress is the base address of the EUSCI_B_SPI module.
mask is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following:

- **EUSCI_B_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIE** register.

Returns:
None

15.2.2.6 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

15.2.2.7 void EUSCI_B_SPI_enableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Enables individual SPI interrupt sources.

Enables the indicated SPI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters:
baseAddress is the base address of the EUSCI_B_SPI module.
mask is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following:
- **EUSCI_B_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_B_SPI_RECEIVE_INTERRUPT**

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

Returns:
None

15.2.2.8 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

15.2.2.9 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

15.2.2.10 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

15.2.2.11 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

15.2.2.12 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

15.2.2.13 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

15.2.2.14 void EUSCI_B_SPI_masterChangeClock (uint16_t *baseAddress*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*)

DEPRECATED - Initializes the SPI Master clock. At the end of this function call, SPI module is left enabled.

Parameters:

baseAddress is the base address of the EUSCI_B_SPI module.

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

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

Returns:

None

15.2.2.15 void EUSCI_B_SPI_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*, uint16_t *msbFirst*, uint16_t *clockPhase*, uint16_t *clockPolarity*, uint16_t *spiMode*)

DEPRECATED - 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.

selectClockSource selects Clock source. Valid values are:
- **EUSCI_B_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_B_SPI_CLOCKSOURCE_SMCLK**

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

msbFirst controls the direction of the receive and transmit shift register. Valid values are:
- **EUSCI_B_SPI_MSB_FIRST**
- **EUSCI_B_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:
- **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity is clock polarity select Valid values are:
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

spiMode is SPI mode select Valid values are:
- **EUSCI_B_SPI_3PIN**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_LOW**

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

Returns:
STATUS_SUCCESS

15.2.2.16 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.

15.2.2.17 void EUSCI_B_SPI_select4PinFunctionality (uint16_t *baseAddress*, uint8_t *select4PinFunctionality*)

Selects 4Pin Functionality.

This function should be invoked only in 4-wire mode. Invoking this function has no effect in 3-wire mode.

Parameters:
baseAddress is the base address of the EUSCI_B_SPI module.
select4PinFunctionality selects 4 pin functionality Valid values are:
- **EUSCI_B_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS**
- **EUSCI_B_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

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

Returns:
None

15.2.2.18 void EUSCI_B_SPI_slaveInit (uint16_t *baseAddress*, uint16_t *msbFirst*, uint16_t *clockPhase*, uint16_t *clockPolarity*, uint16_t *spiMode*)

DEPRECATED - 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.
msbFirst controls the direction of the receive and transmit shift register. Valid values are:
- **EUSCI_B_SPI_MSB_FIRST**
- **EUSCI_B_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:
- **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity is clock polarity select Valid values are:
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

spiMode is SPI mode select Valid values are:
- **EUSCI_B_SPI_3PIN**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_LOW**

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

Returns:
STATUS_SUCCESS

15.2.2.19 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

15.3 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
returnValue = EUSCI_B_SPI_slaveInit(EUSCI_B0_BASE,
        EUSCI_B_SPI_MSB_FIRST,
        EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,
        EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH
        );

if (STATUS_FAIL == returnValue){
        return;
    }

//Enable SPI Module
EUSCI_B_SPI_enable(EUSCI_A0_BASE);

//Enable Receive interrupt
EUSCI_B_SPI_enableInterrupt(EUSCI_A0_BASE,
                EUSCI_B_SPI_RECEIVE_INTERRUPT
        );
```

16 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)

16.1 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.

16.1.1 Master Operations

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

- **EUSCI_B_I2C_masterInit**
- **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_masterInit(). 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_masterMultiByteSendStart()
- EUSCI_B_I2C_masterMultiByteSendNext()
- EUSCI_B_I2C_masterMultiByteSendStop()

Master Single Byte Reception

- EUSCI_B_I2C_masterReceiveSingleByte()

Master Multiple Byte Reception

- EUSCI_B_I2C_masterMultiByteReceiveStart()
- EUSCI_B_I2C_masterMultiByteReceiveNext()
- EUSCI_B_I2C_masterMultiByteReceiveFinish()
- EUSCI_B_I2C_masterMultiByteReceiveStop()

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

16.1.2 Slave Operations

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

- **EUSCI_B_I2C_slaveInit()**
- **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_slaveInit 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_slaveDataPut()

Slave Reception API

- EUSCI_B_I2C_slaveDataGet()

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

This driver is contained in `eusci_b_i2c.c`, with `eusci_b_i2c.h` containing the API definitions for use by applications.

16.2 API Functions

Functions

- void EUSCI_B_I2C_clearInterruptFlag (uint16_t baseAddress, uint16_t mask)
- void EUSCI_B_I2C_disable (uint16_t baseAddress)
- void EUSCI_B_I2C_disableInterrupt (uint16_t baseAddress, uint16_t mask)
- void EUSCI_B_I2C_disableMultiMasterMode (uint16_t baseAddress)
- void EUSCI_B_I2C_enable (uint16_t baseAddress)
- void EUSCI_B_I2C_enableInterrupt (uint16_t baseAddress, uint16_t mask)
- void EUSCI_B_I2C_enableMultiMasterMode (uint16_t baseAddress)
- uint16_t EUSCI_B_I2C_getInterruptStatus (uint16_t baseAddress, uint16_t mask)
- uint8_t EUSCI_B_I2C_getMode (uint16_t baseAddress)
- uint32_t EUSCI_B_I2C_getReceiveBufferAddress (uint16_t baseAddress)
- uint32_t EUSCI_B_I2C_getTransmitBufferAddress (uint16_t baseAddress)
- void EUSCI_B_I2C_initMaster (uint16_t baseAddress, EUSCI_B_I2C_initMasterParam ∗param)

- void EUSCI_B_I2C_initSlave (uint16_t baseAddress, EUSCI_B_I2C_initSlaveParam ∗param)
- uint16_t EUSCI_B_I2C_isBusBusy (uint16_t baseAddress)
- void EUSCI_B_I2C_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t i2cClk, uint32_t dataRate, uint8_t byteCounterThreshold, uint8_t autoSTOPGeneration)
- uint16_t EUSCI_B_I2C_masterIsStartSent (uint16_t baseAddress)
- uint16_t EUSCI_B_I2C_masterIsStopSent (uint16_t baseAddress)
- uint8_t EUSCI_B_I2C_masterMultiByteReceiveFinish (uint16_t baseAddress)
- bool EUSCI_B_I2C_masterMultiByteReceiveFinishWithTimeout (uint16_t baseAddress, uint8_t ∗txData, uint32_t timeout)
- uint8_t EUSCI_B_I2C_masterMultiByteReceiveNext (uint16_t baseAddress)
- void EUSCI_B_I2C_masterMultiByteReceiveStop (uint16_t baseAddress)
- void EUSCI_B_I2C_masterMultiByteSendFinish (uint16_t baseAddress, uint8_t txData)
- bool EUSCI_B_I2C_masterMultiByteSendFinishWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void EUSCI_B_I2C_masterMultiByteSendNext (uint16_t baseAddress, uint8_t txData)
- bool EUSCI_B_I2C_masterMultiByteSendNextWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void EUSCI_B_I2C_masterMultiByteSendStart (uint16_t baseAddress, uint8_t txData)
- bool EUSCI_B_I2C_masterMultiByteSendStartWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void EUSCI_B_I2C_masterMultiByteSendStop (uint16_t baseAddress)
- bool EUSCI_B_I2C_masterMultiByteSendStopWithTimeout (uint16_t baseAddress, uint32_t timeout)
- uint8_t EUSCI_B_I2C_masterReceiveSingleByte (uint16_t baseAddress)
- void EUSCI_B_I2C_masterReceiveStart (uint16_t baseAddress)
- void EUSCI_B_I2C_masterSendSingleByte (uint16_t baseAddress, uint8_t txData)
- bool EUSCI_B_I2C_masterSendSingleByteWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void EUSCI_B_I2C_masterSendStart (uint16_t baseAddress)
- uint8_t EUSCI_B_I2C_masterSingleReceive (uint16_t baseAddress)
- void EUSCI_B_I2C_setMode (uint16_t baseAddress, uint8_t mode)
- void EUSCI_B_I2C_setSlaveAddress (uint16_t baseAddress, uint8_t slaveAddress)
- uint8_t EUSCI_B_I2C_slaveDataGet (uint16_t baseAddress)
- void EUSCI_B_I2C_slaveDataPut (uint16_t baseAddress, uint8_t transmitData)
- void EUSCI_B_I2C_slaveInit (uint16_t baseAddress, uint8_t slaveAddress, uint8_t slaveAddressOffset, uint32_t slaveOwnAddressEnable)

16.2.1 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_clearInterruptFlag
- EUSCI_B_I2C_getInterruptStatus

Status and initialization functions for the I2C modules are

- EUSCI_B_I2C_masterInit
- EUSCI_B_I2C_enable
- EUSCI_B_I2C_disable
- EUSCI_B_I2C_isBusBusy
- EUSCI_B_I2C_isBusy
- EUSCI_B_I2C_slaveInit
- 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_slaveDataPut
- EUSCI_B_I2C_slaveDataGet

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

- EUSCI_B_I2C_masterSendSingleByte
- EUSCI_B_I2C_masterSendStart
- EUSCI_B_I2C_masterMultiByteSendStart
- EUSCI_B_I2C_masterMultiByteSendNext
- EUSCI_B_I2C_masterMultiByteSendFinish
- EUSCI_B_I2C_masterMultiByteSendStop
- EUSCI_B_I2C_masterMultiByteReceiveNext
- EUSCI_B_I2C_masterMultiByteReceiveFinish
- EUSCI_B_I2C_masterMultiByteReceiveStop
- EUSCI_B_I2C_masterReceiveStart
- EUSCI_B_I2C_masterSingleReceive
- EUSCI_B_I2C_getReceiveBufferAddressForDMA
- EUSCI_B_I2C_getTransmitBufferAddressForDMA

DMA related

- EUSCI_B_I2C_getReceiveBufferAddressForDMA
- EUSCI_B_I2C_getTransmitBufferAddressForDMA

16.2.2 Function Documentation

16.2.2.1 void EUSCI_B_I2C_clearInterruptFlag (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

16.2.2.2 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

16.2.2.3 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

16.2.2.4 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

16.2.2.5 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

16.2.2.6 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

16.2.2.7 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

16.2.2.8 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

16.2.2.9 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:
None Return one of the following:

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

16.2.2.10 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

16.2.2.11 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

16.2.2.12 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

16.2.2.13 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

16.2.2.14 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

16.2.2.15 void EUSCI_B_I2C_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *i2cClk*, uint32_t *dataRate*, uint8_t *byteCounterThreshold*, uint8_t *autoSTOPGeneration*)

DEPRECATED - 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.
selectClockSource is the clocksource. Valid values are:
- **EUSCI_B_I2C_CLOCKSOURCE_ACLK**
- **EUSCI_B_I2C_CLOCKSOURCE_SMCLK**

i2cClk is the rate of the clock supplied to the I2C module (the frequency in Hz of the clock source specified in selectClockSource).
dataRate setup for selecting data transfer rate. Valid values are:
- **EUSCI_B_I2C_SET_DATA_RATE_400KBPS**
- **EUSCI_B_I2C_SET_DATA_RATE_100KBPS**

byteCounterThreshold sets threshold for automatic STOP or UCSTPIFG
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**

Returns:
None

16.2.2.16 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

16.2.2.17 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

16.2.2.18 uint8_t EUSCI_B_I2C_masterMultiByteReceiveFinish (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.

16.2.2.19 bool EUSCI_B_I2C_masterMultiByteReceiveFinishWithTimeout (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

16.2.2.20 uint8_t EUSCI_B_I2C_masterMultiByteReceiveNext (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.

16.2.2.21 void EUSCI_B_I2C_masterMultiByteReceiveStop (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

16.2.2.22 void EUSCI_B_I2C_masterMultiByteSendFinish (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

16.2.2.23 bool EUSCI_B_I2C_masterMultiByteSendFinishWithTimeout (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.

16.2.2.24 void EUSCI_B_I2C_masterMultiByteSendNext (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

16.2.2.25 bool EUSCI_B_I2C_masterMultiByteSendNextWithTimeout (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.

16.2.2.26 void EUSCI_B_I2C_masterMultiByteSendStart (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

16.2.2.27 bool EUSCI_B_I2C_masterMultiByteSendStartWithTimeout (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.

16.2.2.28 void EUSCI_B_I2C_masterMultiByteSendStop (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

16.2.2.29 bool EUSCI_B_I2C_masterMultiByteSendStopWithTimeout (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.

16.2.2.30 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.

16.2.2.31 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

16.2.2.32 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

16.2.2.33 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.

16.2.2.34 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

16.2.2.35 uint8_t EUSCI_B_I2C_masterSingleReceive (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.

16.2.2.36 void EUSCI_B_I2C_setMode (uint16_t *baseAddress*, uint8_t *mode*)

Sets the mode of the I2C device.

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

16.2.2.37 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

16.2.2.38 uint8_t EUSCI_B_I2C_slaveDataGet (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.

16.2.2.39 void EUSCI_B_I2C_slaveDataPut (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

16.2.2.40 void EUSCI_B_I2C_slaveInit (uint16_t *baseAddress*, uint8_t *slaveAddress*, uint8_t *slaveAddressOffset*, uint32_t *slaveOwnAddressEnable*)

DEPRECATED - 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.

slaveAddress 7-bit slave address

slaveAddressOffset Own address Offset referred to- ’x’ value of UCBxI2COAx. Valid values are:
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET0**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET1**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET2**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET3**

slaveOwnAddressEnable selects if the specified address is enabled or disabled. Valid values are:
- **EUSCI_B_I2C_OWN_ADDRESS_DISABLE**
- **EUSCI_B_I2C_OWN_ADDRESS_ENABLE**

Returns:
None

16.3 Programming Example

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

```
//Initialize Slave
 EUSCI_B_I2C_slaveInit(EUSCI_B_B0_BASE,
             0x48,
             EUSCI_B_I2C_OWN_ADDRESS_OFFSET0,
             EUSCI_B_I2C_OWN_ADDRESS_ENABLE);

 EUSCI_B_I2C_enable(EUSCI_B0_BASE);

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

17 Flash Memory Controller

17.1 Introduction

The flash memory is byte, word, and long-word addressable and programmable. The flash memory module has an integrated controller that controls programming and erase operations. The flash main memory is partitioned into 512-byte segments. Single bits, bytes, or words can be written to flash memory, but a segment is the smallest size of the flash memory that can be erased. The flash memory is partitioned into main and information memory sections. There is no difference in the operation of the main and information memory sections. Code and data can be located in either section. The difference between the sections is the segment size. There are four information memory segments, A through D. Each information memory segment contains 128 bytes and can be erased individually. The bootstrap loader (BSL) memory consists of four segments, A through D. Each BSL memory segment contains 512 bytes and can be erased individually. The main memory segment size is 512 byte. See the device-specific data sheet for the start and end addresses of each bank, when available, and for the complete memory map of a device. This library provides the API for flash segment erase, flash writes and flash operation status check.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	670
TI Compiler 4.2.1	Size	406
TI Compiler 4.2.1	Speed	408
IAR 5.51.6	None	420
IAR 5.51.6	Size	210
IAR 5.51.6	Speed	280
MSPGCC 4.8.0	None	948
MSPGCC 4.8.0	Size	466
MSPGCC 4.8.0	Speed	436

17.2 API Functions

Functions

- void FLASH_bankErase (uint8_t ∗flash_ptr)
- bool FLASH_eraseCheck (uint8_t ∗flash_ptr, uint16_t numberOfBytes)
- void FLASH_lockInfoA (void)
- void FLASH_massErase (uint8_t ∗flash_ptr)
- void FLASH_memoryFill32 (uint32_t value, uint32_t ∗flash_ptr, uint16_t count)
- void FLASH_segmentErase (uint8_t ∗flash_ptr)
- uint8_t FLASH_status (uint8_t mask)
- void FLASH_unlockInfoA (void)
- void FLASH_write16 (uint16_t ∗data_ptr, uint16_t ∗flash_ptr, uint16_t count)
- void FLASH_write32 (uint32_t ∗data_ptr, uint32_t ∗flash_ptr, uint16_t count)
- void FLASH_write8 (uint8_t ∗data_ptr, uint8_t ∗flash_ptr, uint16_t count)

17.2.1 Detailed Description

FLASH_segmentErase helps erase a single segment of the flash memory. A pointer to the flash segment being erased is passed on to this function.

FLASH_eraseCheck helps check if a specific number of bytes in flash are currently erased. A pointer to the starting location of the erase check and the number of bytes to be checked is passed into this function.

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

FLASH_write8 facilitates writing into the flash memory in byte format. FLASH_write16 facilitates writing into the flash memory in word format. FLASH_write32 facilitates writing into the flash memory in long format, pass by reference. FLASH_memoryFill32 facilitates writing into the flash memory in long format, pass by value. FLASH_status checks if the flash is currently busy erasing or programming. FLASH_lockInfoA locks segment A of information memory. FLASH_unlockInfoA unlocks segment A of information memory.

The Flash API is broken into 4 groups of functions: those that deal with flash erase, those that write into flash, those that give status of flash, and those that lock/unlock segment A of information memory.

The flash erase operations are managed by

- FLASH_segmentErase()
- FLASH_eraseCheck()
- FLASH_bankErase()

Flash writes are managed by

- FLASH_write8()
- FLASH_write16()
- FLASH_write32()
- FLASH_memoryFill32()

The status is given by

- FLASH_status()
- FLASH_eraseCheck()

The segment A of information memory lock/unlock operations are managed by

- FLASH_lockInfoA()
- FLASH_unlockInfoA()

17.2.2 Function Documentation

17.2.2.1 void FLASH_bankErase (uint8_t ∗ *flash_ptr*)

Erase a single bank of the flash memory.

This function erases a single bank of the flash memory. This API will erase the entire flash if device contains only one flash bank.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	50
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	42
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

Parameters:
flash_ptr is a pointer into the bank to be erased

Returns:
None

17.2.2.2 bool FLASH_eraseCheck (uint8_t ∗ *flash_ptr*, uint16_t *numberOfBytes*)

Erase check of the flash memory.

This function checks bytes in flash memory to make sure that they are in an erased state (are set to 0xFF).

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	62
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	30
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	36

Parameters:
flash_ptr is the pointer to the starting location of the erase check
numberOfBytes is the number of bytes to be checked

Returns:
STATUS_SUCCESS or STATUS_FAIL

17.2.2.3 void FLASH_lockInfoA (void)

Locks the information flash memory segment A.

This function is typically called after an erase or write operation on the information flash segment is performed by any of the other API functions in order to re-lock the information flash segment.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	40
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	34
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	32

Returns:
None

17.2.2.4 void FLASH_massErase (uint8_t ∗ *flash_ptr*)

Erase all flash memory.

This function erases all the flash memory banks. For devices like MSP430i204x, this API erases main memory and information flash memory if the FLASH_unlockInfo API was previously executed (otherwise the information flash is not erased). Also note that erasing information flash memory in the MSP430i204x impacts the TLV calibration constants located at the information memory.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	50
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	42
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

Parameters:
flash_ptr is a pointer into the bank to be erased

Returns:
None

17.2.2.5 void FLASH_memoryFill32 (uint32_t *value*, uint32_t ∗ *flash_ptr*, uint16_t *count*)

Write data into the flash memory in 32-bit word format, pass by value.

This function writes a 32-bit data value into flash memory, count times. Assumes the flash memory is already erased and unlocked. FLASH_segmentErase can be used to erase a segment.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	90
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	50
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	120
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	52

Parameters:
value value to fill memory with
flash_ptr is the pointer into which to write the data
count number of times to write the value

Returns:
None

17.2.2.6 void FLASH_segmentErase (uint8_t ∗ *flash_ptr*)

Erase a single segment of the flash memory.

For devices like MSP430i204x, if the specified segment is the information flash segment, the FLASH_unlockInfo API must be called prior to calling this API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	44
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	36
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

Parameters:
flash_ptr is the pointer into the flash segment to be erased

Returns:
None

17.2.2.7 uint8_t FLASH_status (uint8_t *mask*)

Check FLASH status to see if it is currently busy erasing or programming.

This function checks the status register to determine if the flash memory is ready for writing.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
mask FLASH status to read Mask value is the logical OR of any of the following:
- **FLASH_READY_FOR_NEXT_WRITE**
- **FLASH_ACCESS_VIOLATION_INTERRUPT_FLAG**
- **FLASH_PASSWORD_WRITTEN_INCORRECTLY**
- **FLASH_BUSY**

Returns:
Logical OR of any of the following:

- **FLASH_READY_FOR_NEXT_WRITE**
- **FLASH_ACCESS_VIOLATION_INTERRUPT_FLAG**
- **FLASH_PASSWORD_WRITTEN_INCORRECTLY**
- **FLASH_BUSY**
  indicating the status of the FLASH

17.2.2.8 void FLASH_unlockInfoA (void)

Unlocks the information flash memory segment A.

This function must be called before an erase or write operation on the information flash segment is performed by any of the other API functions.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Returns:
None

17.2.2.9 void FLASH_write16 (uint16_t ∗ *data_ptr*, uint16_t ∗ *flash_ptr*, uint16_t *count*)

Write data into the flash memory in 16-bit word format, pass by reference.

This function writes a 16-bit word array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FLASH_segmentErase can be used to erase a segment.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	48
IAR 5.51.6	Size	24
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	126
MSPGCC 4.8.0	Size	62
MSPGCC 4.8.0	Speed	50

Parameters:
data_ptr is the pointer to the data to be written
flash_ptr is the pointer into which to write the data
count number of times to write the value

Returns:
None

17.2.2.10 void FLASH_write32 (uint32_t ∗ *data_ptr*, uint32_t ∗ *flash_ptr*, uint16_t *count*)

Write data into the flash memory in 32-bit word format, pass by reference.

This function writes a 32-bit array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FLASH_segmentErase can be used to erase a segment.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	54
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	134
MSPGCC 4.8.0	Size	68
MSPGCC 4.8.0	Speed	56

Parameters:

data_ptr is the pointer to the data to be written

flash_ptr is the pointer into which to write the data

count number of times to write the value

Returns:

None

17.2.2.11 void FLASH_write8 (uint8_t ∗ *data_ptr*, uint8_t ∗ *flash_ptr*, uint16_t *count*)

Write data into the flash memory in byte format, pass by reference.

This function writes a byte array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FLASH_segmentErase can be used to erase a segment.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	48
IAR 5.51.6	Size	24
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	126
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	50

Parameters:

data_ptr is the pointer to the data to be written

flash_ptr is the pointer into which to write the data

count number of times to write the value

Returns:

None

17.3 Programming Example

The following example shows some flash operations using the APIs

```
do{
    FLASH_segmentErase(FLASH_BASE,
                       (unsigned char *)INFOD_START
```

```
                  );
status = FLASH_eraseCheck(FLASH_BASE,
                          (unsigned char *)INFOD_START,
                          128
                          );
}while(status == STATUS_FAIL);

//Flash write
FLASH_write32(FLASH_BASE,
              calibration_data,
              (unsigned long *)(INFOD_START),1);
```

18 GPIO

18.1 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, as well as, configurable drive strength, full or reduced. 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().

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1134
TI Compiler 4.2.1	Size	432
TI Compiler 4.2.1	Speed	432
IAR 5.51.6	None	738
IAR 5.51.6	Size	236
IAR 5.51.6	Speed	566
MSPGCC 4.8.0	None	1556
MSPGCC 4.8.0	Size	524
MSPGCC 4.8.0	Speed	510

18.2 API Functions

Functions

- void GPIO_clearInterruptFlag (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_disableInterrupt (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_enableInterrupt (uint8_t selectedPort, uint16_t selectedPins)
- uint8_t GPIO_getInputPinValue (uint8_t selectedPort, uint16_t selectedPins)
- uint16_t GPIO_getInterruptStatus (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_interruptEdgeSelect (uint8_t selectedPort, uint16_t selectedPins, uint8_t edgeSelect)
- void GPIO_setAsInputPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setAsInputPinWithPullDownResistor (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setAsInputPinWithPullUpResistor (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setAsOutputPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setAsPeripheralModuleFunctionInputPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setAsPeripheralModuleFunctionOutputPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setDriveStrength (uint8_t selectedPort, uint16_t selectedPins, uint8_t driveStrength)
- void GPIO_setOutputHighOnPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_setOutputLowOnPin (uint8_t selectedPort, uint16_t selectedPins)
- void GPIO_toggleOutputOnPin (uint8_t selectedPort, uint16_t selectedPins)

18.2.1 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_setDriveStrength()
- GPIO_setAsPeripheralModuleFunctionOutputPin()
- GPIO_setAsPeripheralModuleFunctionInputPin()

The GPIO interrupts are handled with

- GPIO_enableInterrupt()
- GPIO_disableInterrupt()
- GPIO_clearInterruptFlag()
- GPIO_getInterruptStatus()

- GPIO_interruptEdgeSelect()

The GPIO pin state is accessed with

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

18.2.2 Function Documentation

18.2.2.1 void GPIO_clearInterruptFlag (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. Note that only Port 1, 2, A have this capability.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	36
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	80
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

selectedPort is the selected port. Valid values are:
- **GPIO_PORT_P1**
- **GPIO_PORT_P2**
- **GPIO_PORT_PA**

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

Modified bits of **PxIFG** register.

Returns:

None

18.2.2.2 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. Note that only Port 1, 2, A have this capability.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	36
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	80
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

selectedPort is the selected port. Valid values are:
- **GPIO_PORT_P1**
- **GPIO_PORT_P2**
- **GPIO_PORT_PA**

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

Modified bits of **PxIE** register.

Returns:

None

18.2.2.3 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. Note that only Port 1, 2, A have this capability.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	36
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

selectedPort is the selected port. Valid values are:

- **GPIO_PORT_P1**
- **GPIO_PORT_P2**
- **GPIO_PORT_PA**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

Modified bits of **PxIE** register.

Returns:

None

18.2.2.4 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	66
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	46
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	78
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

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**

selectedPins is the specified pin in the selected port. Valid values are:

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

Returns:

One of the following:

- **GPIO_INPUT_PIN_HIGH**
- **GPIO_INPUT_PIN_LOW**
  indicating the status of the pin

18.2.2.5 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. Note that only Port 1, 2, A have this capability.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	56
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	36
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	64
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:

selectedPort is the selected port. Valid values are:

- **GPIO_PORT_P1**
- **GPIO_PORT_P2**
- **GPIO_PORT_PA**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

Returns:

Logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**
- **GPIO_PIN6**
- **GPIO_PIN7**
- **GPIO_PIN8**
- **GPIO_PIN9**
- **GPIO_PIN10**
- **GPIO_PIN11**
- **GPIO_PIN12**
- **GPIO_PIN13**
- **GPIO_PIN14**
- **GPIO_PIN15**
  indicating the interrupt status of the selected pins [Default: 0]

18.2.2.6 void GPIO_interruptEdgeSelect (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	56
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	118
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	40

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**

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

- **GPIO_PIN15**

edgeSelect specifies what transition sets the interrupt flag Valid values are:
- **GPIO_HIGH_TO_LOW_TRANSITION**
- **GPIO_LOW_TO_HIGH_TRANSITION**

Modified bits of **PxIES** register.

Returns:
None

18.2.2.7 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	86
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	54
IAR 5.51.6	Size	34
IAR 5.51.6	Speed	50
MSPGCC 4.8.0	None	132
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:
- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**

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

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

Returns:
None

18.2.2.8 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	62
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	60
MSPGCC 4.8.0	None	152
MSPGCC 4.8.0	Size	42
MSPGCC 4.8.0	Speed	42

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**

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

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

Returns:
None

18.2.2.9 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	62
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	56
MSPGCC 4.8.0	None	148
MSPGCC 4.8.0	Size	42
MSPGCC 4.8.0	Speed	42

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

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

Returns:
None

18.2.2.10 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	72
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	44
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	102
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

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

Returns:
None

18.2.2.11 void GPIO_setAsPeripheralModuleFunctionInputPin (uint8_t *selectedPort*, uint16_t *selectedPins*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	72
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	44
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	100
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

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

Returns:

None

18.2.2.12 void GPIO_setAsPeripheralModuleFunctionOutputPin (uint8_t *selectedPort*, uint16_t *selectedPins*)

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

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	72
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	44
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

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

Returns:

None

18.2.2.13 void GPIO_setDriveStrength (uint8_t *selectedPort*, uint16_t *selectedPins*, uint8_t *driveStrength*)

This function sets the drive strength for the selected port pin.

his function sets the drive strength for the selected port pin. Acceptable values for driveStrength are GPIO_REDUCED_OUTPUT_DRIVE_STRENGTH and GPIO_FULL_OUTPUT_DRIVE_STRENGTH.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	52
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	110
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	40

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**

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**

driveStrength specifies the drive strength of the pin Valid values are:

- **GPIO_REDUCED_OUTPUT_DRIVE_STRENGTH**
- **GPIO_FULL_OUTPUT_DRIVE_STRENGTH**

Modified bits of **PxDS** register.

Returns:
None

18.2.2.14 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	34
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**

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

Modified bits of **PxOUT** register.

Returns:
None

18.2.2.15 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	34
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

Modified bits of **PxOUT** register.

Returns:
None

18.2.2.16 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	34
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

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**

selectedPins is the specified pin in the selected port. Mask value is the logical OR of any of the following:

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

Modified bits of **PxOUT** register.

Returns:
None

18.3 Programming Example

The following example shows how to use the GPIO API.

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

// Set P1.4 to input direction
GPIO_setAsInputPin(GPIO_PORT_P1,
                    GPIO_PIN4
                    );

while (1)
{
  // Test P1.4
  if(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(
                                    GPIO_PORT_P1,
                                    GPIO_PIN4
                                    ))
  {
    // if P1.4 set, set P1.0
    GPIO_setOutputHighOnPin(
                    GPIO_PORT_P1,
                    GPIO_PIN0
                    );
  }
  else
  {
    // else reset
    GPIO_setOutputLowOnPin(
                    GPIO_PORT_P1,
                    GPIO_PIN0
                    );
  }
}
```

19 LDO-PWR

19.1 Introduction

The features of the LDO-PWR module include:

- Integrated 3.3-V LDO regulator with sufficient output to power the entire MSP430? microcontroller and system circuitry from 5-V external supply
- Current-limiting capability on 3.3-V LDO output with detection flag and interrupt generation
- LDO input voltage detection flag and interrupt generation

The LDO-PWR power system incorporates an integrated 3.3-V LDO regulator that allows the entire MSP430 microcontroller to be powered from nominal 5-V LDOI when it is made available from the system. Alternatively, the power system can supply power only to other components within the system, or it can be unused altogether.

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

19.2 API Functions

Functions

- void LDOPWR_clearInterruptStatus (uint16_t baseAddress, uint16_t mask)
- void LDOPWR_disable (uint16_t baseAddress)
- void LDOPWR_disableInterrupt (uint16_t baseAddress, uint16_t mask)
- void LDOPWR_disableOverloadAutoOff (uint16_t baseAddress)
- void LDOPWR_disablePort_U_inputs (uint16_t baseAddress)
- void LDOPWR_disablePort_U_outputs (uint16_t baseAddress)
- void LDOPWR_enable (uint16_t baseAddress)
- void LDOPWR_enableInterrupt (uint16_t baseAddress, uint16_t mask)
- void LDOPWR_enableOverloadAutoOff (uint16_t baseAddress)
- void LDOPWR_enablePort_U_inputs (uint16_t baseAddress)
- void LDOPWR_enablePort_U_outputs (uint16_t baseAddress)
- uint8_t LDOPWR_getInterruptStatus (uint16_t baseAddress, uint16_t mask)
- uint8_t LDOPWR_getOverloadAutoOffStatus (uint16_t baseAddress)
- uint8_t LDOPWR_getPort_U0_inputData (uint16_t baseAddress)
- uint8_t LDOPWR_getPort_U0_outputData (uint16_t baseAddress)
- uint8_t LDOPWR_getPort_U1_inputData (uint16_t baseAddress)
- uint8_t LDOPWR_getPort_U1_outputData (uint16_t baseAddress)
- uint8_t LDOPWR_isLDOInputValid (uint16_t baseAddress)
- void LDOPWR_lockConfiguration (uint16_t baseAddress)
- void LDOPWR_setPort_U0_outputData (uint16_t baseAddress, uint8_t value)
- void LDOPWR_setPort_U1_outputData (uint16_t baseAddress, uint8_t value)
- void LDOPWR_togglePort_U0_outputData (uint16_t baseAddress)
- void LDOPWR_togglePort_U1_outputData (uint16_t baseAddress)
- void LDOPWR_unLockConfiguration (uint16_t baseAddress)

19.2.1 Detailed Description

The LDOPWR configuration is handled by

- LDOPWR_unLockConfiguration()
- LDOPWR_lockConfiguration()
- LDOPWR_enablePort_U_inputs()
- LDOPWR_disablePort_U_inputs()
- LDOPWR_enablePort_U_outputs()
- LDOPWR_disablePort_U_outputs()
- LDOPWR_enable()
- LDOPWR_disable()
- LDOPWR_enableOverloadAutoOff()
- LDOPWR_disableOverloadAutoOff()

Handling the read/write of output data is handled by

- LDOPWR_getPort_U1_inputData()
- LDOPWR_getPort_U0_inputData()
- LDOPWR_getPort_U1_outputData()
- LDOPWR_getPort_U0_outputData()
- LDOPWR_getOverloadAutoOffStatus()
- LDOPWR_setPort_U0_outputData()
- LDOPWR_togglePort_U1_outputData()
- LDOPWR_togglePort_U0_outputData()
- LDOPWR_setPort_U1_outputData()

The interrupt and status operations are handled by

- LDOPWR_enableInterrupt()
- LDOPWR_disableInterrupt()
- LDOPWR_getInterruptStatus()
- LDOPWR_clearInterruptStatus()
- LDOPWR_isLDOInputValid()
- LDOPWR_getOverloadAutoOffStatus()

19.2.2 Function Documentation

19.2.2.1 void LDOPWR_clearInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Clears the interrupt status of LDO-PWR module interrupts.

Parameters:

baseAddress is the base address of the LDOPWR module.

mask mask of interrupts to clear the status of Mask value is the logical OR of any of the following:

- **LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT**
- **LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT**
- **LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT**

Modified bits of **LDOPWRCTL** register.

Returns:

None

19.2.2.2 void LDOPWR_disable (uint16_t *baseAddress*)

Disables LDO-PWR module.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.3 void LDOPWR_disableInterrupt (uint16_t *baseAddress*, uint16_t *mask*)

Disables LDO-PWR module interrupts.

Parameters:
baseAddress is the base address of the LDOPWR module.
mask mask of interrupts to disable Mask value is the logical OR of any of the following:
- **LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT**
- **LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT**
- **LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT**

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.4 void LDOPWR_disableOverloadAutoOff (uint16_t *baseAddress*)

Disables the LDO overload auto-off.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.5 void LDOPWR_disablePort_U_inputs (uint16_t *baseAddress*)

Disables Port U inputs.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.6 void LDOPWR_disablePort_U_outputs (uint16_t *baseAddress*)

Disables Port U inputs.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.7 void LDOPWR_enable (uint16_t *baseAddress*)

Enables LDO-PWR module.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.8 void LDOPWR_enableInterrupt (uint16_t *baseAddress*, uint16_t *mask*)

Enables LDO-PWR module interrupts.

Does not clear interrupt flags.

Parameters:
baseAddress is the base address of the LDOPWR module.
mask mask of interrupts to enable Mask value is the logical OR of any of the following:
- **LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT**
- **LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT**
- **LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT**

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.9 void LDOPWR_enableOverloadAutoOff (uint16_t *baseAddress*)

Enables the LDO overload auto-off.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOPWRCTL** register.

Returns:
None

19.2.2.10 void LDOPWR_enablePort_U_inputs (uint16_t *baseAddress*)

Enables Port U inputs.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.11 void LDOPWR_enablePort_U_outputs (uint16_t *baseAddress*)

Enables Port U outputs.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.12 uint8_t LDOPWR_getInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Returns the interrupt status of LDO-PWR module interrupts.

Parameters:
baseAddress is the base address of the LDOPWR module.
mask mask of interrupts to get the status of Mask value is the logical OR of any of the following:
- **LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT**
- **LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT**
- **LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT**

Returns:
Logical OR of any of the following:

- **LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT**
- **LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT**
- **LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT**
indicating the status of the masked interrupts

19.2.2.13 uint8_t LDOPWR_getOverloadAutoOffStatus (uint16_t *baseAddress*)

Returns if the LDOI overload auto-off is enabled or disabled.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_AUTOOFF_ENABLED**
- **LDOPWR_AUTOOFF_DISABLED**

19.2.2.14 uint8_t LDOPWR_getPort_U0_inputData (uint16_t *baseAddress*)

Returns PU.0 input data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

19.2.2.15 uint8_t LDOPWR_getPort_U0_outputData (uint16_t *baseAddress*)

Returns PU.0 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

19.2.2.16 uint8_t LDOPWR_getPort_U1_inputData (uint16_t *baseAddress*)

Returns PU.1 input data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

19.2.2.17 uint8_t LDOPWR_getPort_U1_outputData (uint16_t *baseAddress*)

Returns PU.1 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

19.2.2.18 uint8_t LDOPWR_isLDOInputValid (uint16_t *baseAddress*)

Returns if the the LDOI is valid and within bounds.

Parameters:
baseAddress is the base address of the LDOPWR module.

Returns:
One of the following:

- **LDOPWR_LDO_INPUT_VALID**
- **LDOPWR_LDO_INPUT_INVALID**

19.2.2.19 void LDOPWR_lockConfiguration (uint16_t *baseAddress*)

Locks the configuration registers and disables write access.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOKEYPID** register.

Returns:
None

19.2.2.20 void LDOPWR_setPort_U0_outputData (uint16_t *baseAddress*, uint8_t *value*)

Sets PU.0 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.
value Valid values are:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.21 void LDOPWR_setPort_U1_outputData (uint16_t *baseAddress*, uint8_t *value*)

Sets PU.1 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.
value Valid values are:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.22 void LDOPWR_togglePort_U0_outputData (uint16_t *baseAddress*)

Toggles PU.0 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.23 void LDOPWR_togglePort_U1_outputData (uint16_t *baseAddress*)

Toggles PU.1 output data.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **PUCTL** register.

Returns:
None

19.2.2.24 void LDOPWR_unLockConfiguration (uint16_t *baseAddress*)

Unlocks the configuration registers and enables write access.

Parameters:
baseAddress is the base address of the LDOPWR module.

Modified bits of **LDOKEYPID** register.

Returns:
None

19.3 Programming Example

The following example shows how to use the LDO-PWR API.

```
{
  // Enable access to config registers
  LDOPWR_unLockConfiguration(LDOPWR_BASE);

  // Configure PU.0 as output pins
  LDOPWR_enablePort_U_outputs(LDOPWR_BASE);

  //Set PU.1 = high
  LDOPWR_setPort_U1_outputData(LDOPWR_BASE,
                              LDOPWR_PORTU_PIN_HIGH
                                                                    );

  //Set PU.0 = low
  LDOPWR_setPort_U0_outputData(LDOPWR_BASE,
                              LDOPWR_PORTU_PIN_LOW
                              );
  // Enable LDO overload indication interrupt
  LDOPWR_enableInterrupt(LDOPWR_BASE,
                         LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                         );

  // Disable access to config registers
  LDOPWR_lockConfiguration(LDOPWR_BASE);

  // continuous loop
  while(1)
  {
    // Delay
    for(i=50000;i>0;i--);

    // Enable access to config registers
    LDOPWR_unLockConfiguration(LDOPWR_BASE);

    // XOR PU.0/1
    LDOPWR_togglePort_U1_outputData(LDOPWR_BASE);
    LDOPWR_togglePort_U0_outputData(LDOPWR_BASE);

    // Disable access to config registers
    LDOPWR_lockConfiguration(LDOPWR_BASE);
  }
}


//******************************************************************************
//
// This is the LDO_PWR_VECTOR interrupt vector service routine.
//
//******************************************************************************
__interrupt void LDOInterruptHandler(void)
{
  if(LDOPWR_getInterruptStatus(LDOPWR_BASE,
                               LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                               ))

  {
    // Enable access to config registers
    LDOPWR_unLockConfiguration(LDOPWR_BASE);

   // Software clear IFG
    LDOPWR_clearInterruptStatus(LDOPWR_BASE,
                                LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT
                                );

   // Disable access to config registers
   LDOPWR_lockConfiguration(LDOPWR_BASE);

   // Over load indication; take necessary steps in application firmware
   while(1);

  }
}
```

20 32-Bit Hardware Multiplier (MPY32)

20.1 Introduction

The 32-Bit Hardware Multiplier (MPY32) API provides a set of functions for using the MSP430Ware MPY32 modules. Functions are provided to setup the MPY32 modules, set the operand registers, and obtain the results.

The MPY32 Modules does not generate any interrupts.

This driver is contained in `mpy32.c`, with `mpy32.h` containing the API definitions for use by applications.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	578
TI Compiler 4.2.1	Size	294
TI Compiler 4.2.1	Speed	294
IAR 5.51.6	None	404
IAR 5.51.6	Size	250
IAR 5.51.6	Speed	274
MSPGCC 4.8.0	None	1702
MSPGCC 4.8.0	Size	644
MSPGCC 4.8.0	Speed	642

20.2 API Functions

Functions

- void MPY32_clearCarryBitValue (void)
- void MPY32_disableFractionalMode (void)
- void MPY32_disableSaturationMode (void)
- void MPY32_enableFractionalMode (void)
- void MPY32_enableSaturationMode (void)
- uint16_t MPY32_getCarryBitValue (void)
- uint8_t MPY32_getFractionalMode (void)
- uint64_t MPY32_getResult (void)
- uint16_t MPY32_getResult16Bit (void)
- uint32_t MPY32_getResult24Bit (void)
- uint32_t MPY32_getResult32Bit (void)
- uint64 MPY32_getResult64Bit (void)
- uint8_t MPY32_getResult8Bit (void)
- uint8_t MPY32_getSaturationMode (void)
- uint16_t MPY32_getSumExtension (void)
- void MPY32_preloadResult (uint64_t result)
- void MPY32_setOperandOne16Bit (uint8_t multiplicationType, uint16_t operand)

- void MPY32_setOperandOne24Bit (uint8_t multiplicationType, uint32_t operand)
- void MPY32_setOperandOne32Bit (uint8_t multiplicationType, uint32_t operand)
- void MPY32_setOperandOne8Bit (uint8_t multiplicationType, uint8_t operand)
- void MPY32_setOperandTwo16Bit (uint16_t operand)
- void MPY32_setOperandTwo24Bit (uint32_t operand)
- void MPY32_setOperandTwo32Bit (uint32_t operand)
- void MPY32_setOperandTwo8Bit (uint8_t operand)
- void MPY32_setWriteDelay (uint16_t writeDelaySelect)

20.2.1 Detailed Description

The MPY32 API is broken into three groups of functions: those that control the settings, those that set the operand registers, and those that return the results, sum extension, and carry bit value.

The settings are handled by

- MPY32_setWriteDelay()
- MPY32_enableSaturationMode()
- MPY32_disableSaturationMode()
- MPY32_enableFractionalMode()
- MPY32_disableFractionalMode()
- MPY32_preloadResult()

The operand registers are set by

- MPY32_setOperandOne8Bit()
- MPY32_setOperandOne16Bit()
- MPY32_setOperandOne24Bit()
- MPY32_setOperandOne32Bit()
- MPY32_setOperandTwo8Bit()
- MPY32_setOperandTwo16Bit()
- MPY32_setOperandTwo24Bit()
- MPY32_setOperandTwo32Bit()

The results can be returned by

- MPY32_getResult()
- MPY32_getSumExtension()
- MPY32_getCarryBitValue()
- MPY32_getSaturationMode()
- MPY32_getFractionalMode()

20.2.2 Function Documentation

20.2.2.1 void MPY32_clearCarryBitValue (void)

Clears the Carry Bit of the last multiplication operation.

This function clears the Carry Bit of the MPY module

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
The value of the MPY32 module Carry Bit 0x0 or 0x1.

20.2.2.2 void MPY32_disableFractionalMode (void)

Disables Fraction Mode.

This function disables fraction mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

20.2.2.3 void MPY32_disableSaturationMode (void)

Disables Saturation Mode.

This function disables saturation mode, which allows the raw result of the MPY result registers to be returned.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

20.2.2.4 void MPY32_enableFractionalMode (void)

Enables Fraction Mode.

This function enables fraction mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

20.2.2.5 void MPY32_enableSaturationMode (void)

Enables Saturation Mode.

This function enables saturation mode. When this is enabled, the result read out from the MPY result registers is converted to the most-positive number in the case of an overflow, or the most-negative number in the case of an underflow. Please note, that the raw value in the registers does not reflect the result returned, and if the saturation mode is disabled, then the raw value of the registers will be returned instead.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

20.2.2.6 uint16_t MPY32_getCarryBitValue (void)

Returns the Carry Bit of the last multiplication operation.

This function returns the Carry Bit of the MPY module, which either gives the sign after a signed operation or shows a carry after a multiply- and- accumulate operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	10
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
The value of the MPY32 module Carry Bit 0x0 or 0x1.

20.2.2.7 uint8_t MPY32_getFractionalMode (void)

Gets the Fractional Mode.

This function gets the current fractional mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	16
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
Gets the fractional mode Return one of the following:

- **MPY32_FRACTIONAL_MODE_DISABLED**
- **MPY32_FRACTIONAL_MODE_ENABLED**
  Gets the Fractional Mode

20.2.2.8 uint64_t MPY32_getResult (void)

Returns an 64-bit result of the last multiplication operation.

This function returns all 64 bits of the result registers

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	102
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	694
MSPGCC 4.8.0	Size	242
MSPGCC 4.8.0	Speed	240

Returns:
The 64-bit result is returned as a uint64_t type

20.2.2.9 uint16_t MPY32_getResult16Bit (void)

Deprecated - Returns an 16-bit result of the last multiplication operation.

This function returns the 16 least significant bits of the result registers. This can improve efficiency if the operation has no more than a 16-bit result.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	8
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
The 16-bit result of the last multiplication operation.

20.2.2.10 uint32_t MPY32_getResult24Bit (void)

Deprecated - Returns an 24-bit result of the last multiplication operation.

This function returns the 24 least significant bits of the result registers. This can improve efficiency if the operation has no more than an 24-bit result.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	22
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Returns:
The 24-bit result of the last multiplication operation.

20.2.2.11 uint32_t MPY32_getResult32Bit (void)

Deprecated - Returns an 32-bit result of the last multiplication operation.

This function returns a 32-bit result of the last multiplication operation, which is the maximum amount of bits of a 16 x 16 operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	22
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Returns:
The 32-bit result of the last multiplication operation.

20.2.2.12 uint64 MPY32_getResult64Bit (void)

Deprecated - Returns an 64-bit result of the last multiplication operation.

This function returns all 64 bits of the result registers. The way this is passed is with 4 integers contained within a uint16 struct.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	52
IAR 5.51.6	Size	52
IAR 5.51.6	Speed	52
MSPGCC 4.8.0	None	82
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	32

Returns:
The 64-bit result separated into 4 uint16_ts in a uint16 struct

20.2.2.13 uint8_t MPY32_getResult8Bit (void)

Deprecated - Returns an 8-bit result of the last multiplication operation.

This function returns the 8 least significant bits of the result registers. This can improve efficiency if the operation has no more than an 8-bit result.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	10
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
The 8-bit result of the last multiplication operation.

20.2.2.14 uint8_t MPY32_getSaturationMode (void)

Gets the Saturation Mode.

This function gets the current saturation mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	16
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
Gets the Saturation Mode Return one of the following:

- **MPY32_SATURATION_MODE_DISABLED**
- **MPY32_SATURATION_MODE_ENABLED**
  Gets the Saturation Mode

20.2.2.15 uint16_t MPY32_getSumExtension (void)

Returns the Sum Extension of the last multiplication operation.

This function returns the Sum Extension of the MPY module, which either gives the sign after a signed operation or shows a carry after a multiply- and-accumulate operation. The Sum Extension acts as a check for overflows or underflows.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	8
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
The value of the MPY32 module Sum Extension.

20.2.2.16 void MPY32_preloadResult (uint64_t *result*)

Preloads the result register.

This function Preloads the result register

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	52
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	18
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	198
MSPGCC 4.8.0	Size	78
MSPGCC 4.8.0	Speed	78

Returns:
None

20.2.2.17 void MPY32_setOperandOne16Bit (uint8_t *multiplicationType*, uint16_t *operand*)

Sets an 16-bit value into operand 1.

This function sets the first operand for multiplication and determines what type of operation should be performed. Once the second operand is set, then the operation will begin.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
multiplicationType is the type of multiplication to perform once the second operand is set. Valid values are:
- **MPY32_MULTIPLY_UNSIGNED**
- **MPY32_MULTIPLY_SIGNED**
- **MPY32_MULTIPLYACCUMULATE_UNSIGNED**
- **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand is the 16-bit value to load into the 1st operand.

Returns:
None

20.2.2.18 void MPY32_setOperandOne24Bit (uint8_t *multiplicationType*, uint32_t *operand*)

Sets an 24-bit value into operand 1.

This function sets the first operand for multiplication and determines what type of operation should be performed. Once the second operand is set, then the operation will begin.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	24
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	38
MSPGCC 4.8.0	Speed	38

Parameters:

multiplicationType is the type of multiplication to perform once the second operand is set. Valid values are:
- **MPY32_MULTIPLY_UNSIGNED**
- **MPY32_MULTIPLY_SIGNED**
- **MPY32_MULTIPLYACCUMULATE_UNSIGNED**
- **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand is the 24-bit value to load into the 1st operand.

Returns:

None

20.2.2.19 void MPY32_setOperandOne32Bit (uint8_t *multiplicationType*, uint32_t *operand*)

Sets an 32-bit value into operand 1.

This function sets the first operand for multiplication and determines what type of operation should be performed. Once the second operand is set, then the operation will begin.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	44
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	24
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	82
MSPGCC 4.8.0	Size	38
MSPGCC 4.8.0	Speed	38

Parameters:

multiplicationType is the type of multiplication to perform once the second operand is set. Valid values are:
- **MPY32_MULTIPLY_UNSIGNED**
- **MPY32_MULTIPLY_SIGNED**
- **MPY32_MULTIPLYACCUMULATE_UNSIGNED**
- **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand is the 32-bit value to load into the 1st operand.

Returns:
None

20.2.2.20 void MPY32_setOperandOne8Bit (uint8_t *multiplicationType*, uint8_t *operand*)

Sets an 8-bit value into operand 1.

This function sets the first operand for multiplication and determines what type of operation should be performed. Once the second operand is set, then the operation will begin.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
multiplicationType is the type of multiplication to perform once the second operand is set. Valid values are:
- **MPY32_MULTIPLY_UNSIGNED**
- **MPY32_MULTIPLY_SIGNED**
- **MPY32_MULTIPLYACCUMULATE_UNSIGNED**
- **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand is the 8-bit value to load into the 1st operand.

Returns:
None

20.2.2.21 void MPY32_setOperandTwo16Bit (uint16_t *operand*)

Sets an 16-bit value into operand 2, which starts the multiplication.

This function sets the second operand of the multiplication operation and starts the operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
operand is the 16-bit value to load into the 2nd operand.

Returns:
None

20.2.2.22 void MPY32_setOperandTwo24Bit (uint32_t *operand*)

Sets an 24-bit value into operand 2, which starts the multiplication.

This function sets the second operand of the multiplication operation and starts the operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:
operand is the 24-bit value to load into the 2nd operand.

Returns:
None

20.2.2.23 void MPY32_setOperandTwo32Bit (uint32_t *operand*)

Sets an 32-bit value into operand 2, which starts the multiplication.

This function sets the second operand of the multiplication operation and starts the operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:
operand is the 32-bit value to load into the 2nd operand.

Returns:
None

20.2.2.24 void MPY32_setOperandTwo8Bit (uint8_t *operand*)

Sets an 8-bit value into operand 2, which starts the multiplication.

This function sets the second operand of the multiplication operation and starts the operation.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
operand is the 8-bit value to load into the 2nd operand.

Returns:
None

20.2.2.25 void MPY32_setWriteDelay (uint16_t *writeDelaySelect*)

Sets the write delay setting for the MPY32 module.

This function sets up a write delay to the MPY module's registers, which holds any writes to the registers until all calculations are complete. There are two different settings, one which waits for 32-bit results to be ready, and one which waits for 64-bit results to be ready. This prevents unpredicatble results if registers are changed before the results are ready.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:
writeDelaySelect delays the write to any MPY32 register until the selected bit size of result has been written. Valid values are:
- **MPY32_WRITEDELAY_OFF** [Default] - writes are not delayed
- **MPY32_WRITEDELAY_32BIT** - writes are delayed until a 32-bit result is available in the result registers
- **MPY32_WRITEDELAY_64BIT** - writes are delayed until a 64-bit result is available in the result registers
Modified bits are **MPYDLY32** and **MPYDLYWRTEN** of **MPY32CTL0** register.

Returns:
None

20.3 Programming Example

The following example shows how to initialize and use the MPY32 API to calculate a 16-bit by 16-bit unsigned multiplication operation.

```
WDT_hold(WDT_A_BASE);   // Stop WDT

// Set a 16-bit Operand into the specific Operand 1 register to specify
 // unsigned multiplication
MPY32_setOperandOne16Bit(MPY32_MULTIPLY_UNSIGNED,
                         0x1234);
// Set Operand 2 to begin the multiplication operation
MPY32_setOperandTwo16Bit(0x5678);

__bis_SR_register(LPM4_bits);             // Enter LPM4
__no_operation();                         // BREAKPOINT HERE to verify the
                                           // correct result in the registers
```

21 Port Mapping Controller

21.1 Introduction

The port mapping controller allows the flexible and re-configurable mapping of digital functions to port pins. The port mapping controller features are:

- Configuration protected by write access key.
- Default mapping provided for each port pin (device-dependent, the device pinout in the device-specific data sheet).
- Mapping can be reconfigured during runtime.
- Each output signal can be mapped to several output pins.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	188
TI Compiler 4.2.1	Size	108
TI Compiler 4.2.1	Speed	112
IAR 5.51.6	None	140
IAR 5.51.6	Size	124
IAR 5.51.6	Speed	180
MSPGCC 4.8.0	None	252
MSPGCC 4.8.0	Size	106
MSPGCC 4.8.0	Speed	284

21.2 API Functions

Functions

- void PMAP_configurePorts (uint16_t baseAddress, const uint8_t ∗portMapping, uint8_t ∗PxMAPy, uint8_t numberOfPorts, uint8_t portMapReconfigure)
- void PMAP_initPorts (uint16_t baseAddress, PMAP_initPortsParam ∗param)

21.2.1 Detailed Description

The MSP430ware API that configures Port Mapping is PMAP_configurePorts()

It needs the following data to configure port mapping. portMapping - pointer to init Data PxMAPy - pointer start of first Port Mapper to initialize numberOfPorts - number of Ports to initialize portMapReconfigure - to enable/disable reconfiguration

21.2.2 Function Documentation

21.2.2.1 void PMAP_configurePorts (uint16_t *baseAddress*, const uint8_t ∗ *portMapping*, uint8_t ∗ *PxMAPy*, uint8_t *numberOfPorts*, uint8_t *portMapReconfigure*)

DEPRECATED - This function configures the MSP430 Port Mapper.

This function port maps a set of pins to a new set.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	76
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	74
MSPGCC 4.8.0	Size	34
MSPGCC 4.8.0	Speed	210

Parameters:

baseAddress is the base address of the PMAP control module.
portMapping is the pointer to init Data
PxMAPy is the pointer start of first PMAP to initialize
numberOfPorts is the number of Ports to initialize
portMapReconfigure is used to enable/disable reconfiguration Valid values are:

- **PMAP_ENABLE_RECONFIGURATION**
- **PMAP_DISABLE_RECONFIGURATION** [Default]

Modified bits of **PMAPKETID** register and bits of **PMAPCTL** register.

Returns:

None

21.2.2.2 void PMAP_initPorts (uint16_t *baseAddress*, PMAP_initPortsParam ∗ *param*)

This function configures the MSP430 Port Mapper.

This function port maps a set of pins to a new set.

Modified bits of **PMAPKETID** register and bits of **PMAPCTL** register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	112
TI Compiler 4.2.1	Size	72
TI Compiler 4.2.1	Speed	76
IAR 5.51.6	None	76
IAR 5.51.6	Size	70
IAR 5.51.6	Speed	126
MSPGCC 4.8.0	None	178
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	74

Returns:

None

21.3 Programming Example

The following example shows some Port Mapping Controller operations using the APIs

```
        const unsigned char port_mapping[] = {
                //Port P4:
                PM_TB0CCR0A,
                PM_TB0CCR1A,
                PM_TB0CCR2A,
                PM_TB0CCR3A,
                PM_TB0CCR4A,
                PM_TB0CCR5A,
                PM_TB0CCR6A,
                PM_NONE
        };

        //CONFIGURE PORTS- pass the port_mapping array, start @ P4MAP01, initialize
    //a single port, do not allow run-time reconfiguration of port mapping

        PMAP_configurePorts(P4MAP_BASE,
        (const unsigned char  *)port_mapping,
        (unsigned char  *)&P4MAP01,
        1,
        PMAP_DISABLE_RECONFIGURATION
        );
```

22 Power Management Module (PMM)

22.1 Introduction

The PMM manages the following internal circuitry:

- An integrated low-dropout voltage regulator (LDO) that produces a secondary core voltage (VCORE) from the primary voltage that is applied to the device (DVCC)
- Supply voltage supervisors (SVS) and supply voltage monitors (SVM) for the primary voltage (DVCC) and the secondary voltage (VCORE). The SVS and SVM include programmable threshold levels and power-fail indicators. Therefore, the PMM plays a crucial role in defining the maximum performance, valid voltage conditions, and current consumption for an application running on an MSP430x5xx or MSP430x6xx device. The secondary voltage that is generated by the integrated LDO, VCORE, is programmable to one of four core voltage levels, shown as 0, 1, 2, and 3. Each increase in VCORE allows the CPU to operate at a higher maximum frequency. The values of these frequencies are specified in the device-specific data sheet. This feature allows the user the flexibility to trade power consumption in active and low-power modes for different degrees of maximum performance and minimum supply voltage.

NOTE: To align with the nomenclature in the MSP430x5xx/MSP430x6xx Family User?s Guide, the primary voltage domain (DVCC) is referred to as the high-side voltage (SvsH/SVMH) and the secondary voltage domain (VCORE) is referred to as the low-side voltage (SvsL/SvmL).

Moving between the different VCORE voltages requires a specific sequence of events and can be done only one level at a time; for example, to change from level 0 to level 3, the application code must step through level 1 and level 2.

VCORE increase: 1. SvmL monitor level is incremented. 2. VCORE level is incremented. 3. The SvmL Level Reached Interrupt Flag (SVSMLVLRIFG) in the PMMIFG register is polled. When asserted, SVSMLVLRIFG indicates that the VCORE voltage has reached its next level. 4. SvsL is increased. SvsL is changed last, because if SVSL were incremented prior to VCORE, it would potentially cause a reset.

VCORE decrease: 5. Decrement SvmL and SVSL levels. 6. Decrement VCORE. The PMM_setVCore() function appropriately handles an increase or decrease of the core voltage. NOTE: The procedure recommended above provides a workaround for the erratum FLASH37. See the device-specific erratasheet to determine if a device is affected by FLASH37. The workaround is also highlighted in the source code for the PMM library

Recommended SVS and SVM Settings The SVS and SVM on both the high side and the low side are enabled in normal performance mode following a brown-out reset condition. The device is held in reset until the SVS and SVM verify that the external and core voltages meet the minimum requirements of the default core voltage, which is level zero. The SVS and SVM remain enabled unless disabled by the firmware. The low-side SVS and SVM are useful for verifying the startup conditions and for verifying any modification to the core voltage. However, in their default mode, they prevent the CPU from executing code on wake-up from low-power modes 2, 3, and 4 for a full 150 ?s, not 5 ?s. This is because, in their default states, the SVSL and SvmL are powered down in the low-power mode of the PMM and need time for their comparators to wake and stabilize before they can verify the voltage condition and release the CPU for execution. Note that the high-side SVS and SVM do not influence the wake time from low-power modes. If the wake-up from low-power modes needs to be shortened to 5 ?s, the SVSL and SvmL should be disabled after the initialization of the core voltage at the beginning of the application. Disabling SVSL and SvmL prevents them from gating the CPU on wake-up from LPM2, LPM3, and LPM4. The application is still protected on the high side with SvsH and SVMH. The PMM_setVCore() function automatically enables and disables the SVS and SVM as necessary if a non-zero core voltage level is required. If the application does not require a change in the core voltage (that is, when the target MCLK is less than 8 MHz), the PMM_disableSVSLSvmL() and PMM_enableSvsHReset() macros can be used to disable the low-side SVS and SVM circuitry and enable only the high-side SVS POR reset, respectively.

Setting SVS/SVM Threshold Levels The voltage thresholds for the SVS and SVM modules are programmable. On the high side, there are two bit fields that control these threshold levels ? the SvsHRVL and SVSMHRRL. The SvsHRVL field defines the voltage threshold at which the SvsH triggers a reset (also known as the SvsH ON voltage level). The SVSMHRRL field defines the voltage threshold at which the SvsH releases the device from a reset (also known as SvsH OFF voltage level). The MSP430x5xx/MSP430x6xx Family User?s Guide (SLAU208) [1] recommends the settings shown in Table 1 when setting these bits. The PMM_setVCore() function follows these recommendations and ensures that the SVS levels match the core voltage levels that are used.

Advanced SVS Controls and Trade-offs In addition to the default SVS settings that are provided with the PMM_setVCore() function, the SVS/SVM modules can be optimized for wake-up speed, response time (propagation delay), and current consumption, as needed. The following controls can be optimized for the SVS/SVM modules:

- Protection in low power modes - LPM2, LPM3, and LPM4
- Wake-up time from LPM2, LPM3, and LPM4
- Response time to react to an SVS event Selecting the LPM option, wake-up time, and response time that is best suited for the application is left to the user. A few typical examples illustrate the trade-offs: Case A: The most robust protection that stays on in LPMs and has the fastest response and wake-up time consumes the most power. Case B: With SVS high side active only in AM, no protection in LPMs, slow wake-up, and slow response time has SVS protection with the least current consumption. Case C: An optimized case is described - turn off the low-side monitor and supervisor, thereby saving power while keeping response time fast on the high side to help with timing critical applications. The user can call the PMM_setVCore() function, which configures SVS/SVM high side and low side with the recommended or default configurations, or can call the APIs provided to control the parameters as the application demands.

Any writes to the SVSMLCTL and SVSMHCTL registers require a delay time for these registers to settle before the new settings take effect. This delay time is dependent on whether the SVS and SVM modules are configured for normal or full performance. See device-specific data sheet for exact delay times.

22.2 API Functions

Functions

- void PMM_clearPMMIFGS (void)
- void PMM_disableSvmH (void)
- void PMM_disableSvmHInterrupt (void)
- void PMM_disableSvmL (void)
- void PMM_disableSvmLInterrupt (void)
- void PMM_disableSvsH (void)
- void PMM_disableSvsHReset (void)
- void PMM_disableSvsHSvmH (void)
- void PMM_disableSvsL (void)
- void PMM_disableSvsLReset (void)
- void PMM_disableSvsLSvmL (void)
- void PMM_enableSvmH (void)
- void PMM_enableSvmHInterrupt (void)
- void PMM_enableSvmL (void)
- void PMM_enableSvmLInterrupt (void)
- void PMM_enableSvsH (void)
- void PMM_enableSvsHReset (void)
- void PMM_enableSvsHSvmH (void)
- void PMM_enableSvsL (void)
- void PMM_enableSvsLReset (void)
- void PMM_enableSvsLSvmL (void)
- uint16_t PMM_getInterruptStatus (uint16_t mask)
- bool PMM_setVCore (uint8_t level)
- uint16_t PMM_setVCoreDown (uint8_t level)
- uint16_t PMM_setVCoreUp (uint8_t level)
- void PMM_SvsHDisabledInLPMFullPerf (void)
- void PMM_SvsHDisabledInLPMNormPerf (void)
- void PMM_SvsHEnabledInLPMFullPerf (void)
- void PMM_SvsHEnabledInLPMNormPerf (void)
- void PMM_SvsHOptimizedInLPMFullPerf (void)
- void PMM_SvsLDisabledInLPMFastWake (void)

- void PMM_SvsLDisabledInLPMSlowWake (void)
- void PMM_SvsLEnabledInLPMFastWake (void)
- void PMM_SvsLEnabledInLPMSlowWake (void)
- void PMM_SvsLOptimizedInLPMFastWake (void)

22.2.1 Detailed Description

PMM_enableSvsL() / **PMM_disableSvsL()** Enables or disables the low-side SVS circuitry

PMM_enableSvmL() / **PMM_disableSvmL()** Enables or disables the low-side SVM circuitry

PMM_enableSvsH() / **PMM_disableSvsH()** Enables or disables the high-side SVS circuitry

PMM_enableSVMH() / **PMM_disableSVMH()** Enables or disables the high-side SVM circuitry

PMM_enableSvsLSvmL() / **PMM_disableSvsLSvmL()** Enables or disables the low-side SVS and SVM circuitry

PMM_enableSvsHSvmH() / **PMM_disableSvsHSvmH()** Enables or disables the high-side SVS and SVM circuitry

PMM_enableSvsLReset() / **PMM_disableSvsLReset()** Enables or disables the POR signal generation when a low-voltage event is registered by the low-side SVS

PMM_enableSvmLInterrupt() / **PMM_disableSvmLInterrupt()** Enables or disables the interrupt generation when a low-voltage event is registered by the low-side SVM

PMM_enableSvsHReset() / **PMM_disableSvsHReset()** Enables or disables the POR signal generation when a low-voltage event is registered by the high-side SVS

PMM_enableSVMHInterrupt() / **PMM_disableSVMHInterrupt()** Enables or disables the interrupt generation when a low-voltage event is registered by the high-side SVM

PMM_clearPMMIFGS() Clear all interrupt flags for the PMM

PMM_SvsLEnabledInLPMFastWake() Enables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

PMM_SvsLEnabledInLPMSlowWake() Enables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_SvsLDisabledInLPMFastWake() Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

PMM_SvsLDisabledInLPMSlowWake() Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_SvsHEnabledInLPMNormPerf() Enables supervisor high side in LPM with tpd = 20 ?s(1)

PMM_SvsHEnabledInLPMFullPerf() Enables supervisor high side in LPM with tpd = 2.5 ?s(1)

PMM_SvsHDisabledInLPMNormPerf() Disables supervisor high side in LPM with tpd = 20 ?s(1)

PMM_SvsHDisabledInLPMFullPerf() Disables supervisor high side in LPM with tpd = 2.5 ?s(1)

PMM_SvsLOptimizedInLPMFastWake() Optimized to provide twake-up-fast from LPM2, LPM3, and LPM4 with least power

PMM_SvsHOptimizedinLPMFullPerf() Optimized to provide tpd = 2.5 ?s(1) in LPM with least power

PMM_getInterruptStatus() Returns interrupt status of the PMM module

PMM_setVCore() Sets the appropriate VCORE level. Calls the PMM_setVCoreUp() or PMM_setVCoreDown() function the required number of times depending on the current VCORE level, because the levels must be stepped through individually. A status indicator equal to STATUS_SUCCESS or STATUS_FAIL that indicates a valid or invalid VCORE transition, respectively. An invalid VCORE transition exists if DVCC is less than the minimum required voltage for the target VCORE voltage.

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

22.2.2 Function Documentation

22.2.2.1 void PMM_clearPMMIFGS (void)

Clear all interrupt flags for the PMM.

Modified bits of **PMMCTL0** register and bits of **PMMIFG** register.

Returns:
None

22.2.2.2 void PMM_disableSvmH (void)

Disables the high-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.3 void PMM_disableSvmHInterrupt (void)

Disables the interrupt generation when a low-voltage event is registered by the high-side SVM.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.4 void PMM_disableSvmL (void)

Disables the low-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.5 void PMM_disableSvmLInterrupt (void)

Disables the interrupt generation when a low-voltage event is registered by the low-side SVM.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.6 void PMM_disableSvsH (void)

Disables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.7 void PMM_disableSvsHReset (void)

Disables the POR signal generation when a low-voltage event is registered by the high-side SVS.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.8 void PMM_disableSvsHSvmH (void)

Disables the high-side SVS and SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.9 void PMM_disableSvsL (void)

Disables the low-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.10 void PMM_disableSvsLReset (void)

Disables the POR signal generation when a low-voltage event is registered by the low-side SVS.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.11 void PMM_disableSvsLSvmL (void)

Disables the low-side SVS and SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.12 void PMM_enableSvmH (void)

Enables the high-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.13 void PMM_enableSvmHInterrupt (void)

Enables the interrupt generation when a low-voltage event is registered by the high-side SVM.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.14 void PMM_enableSvmL (void)

Enables the low-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.15 void PMM_enableSvmLInterrupt (void)

Enables the interrupt generation when a low-voltage event is registered by the low-side SVM.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.16 void PMM_enableSvsH (void)

Enables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.17 void PMM_enableSvsHReset (void)

Enables the POR signal generation when a low-voltage event is registered by the high-side SVS.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.18 void PMM_enableSvsHSvmH (void)

Enables the high-side SVS and SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.19 void PMM_enableSvsL (void)

Enables the low-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.20 void PMM_enableSvsLReset (void)

Enables the POR signal generation when a low-voltage event is registered by the low-side SVS.

Modified bits of **PMMCTL0** register and bits of **PMMIE** register.

Returns:
None

22.2.2.21 void PMM_enableSvsLSvmL (void)

Enables the low-side SVS and SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.22 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_SVSMLDLYIFG**
- **PMM_SVMLIFG**
- **PMM_SVMLVLRIFG**
- **PMM_SVSMHDLYIFG**
- **PMM_SVMHIFG**
- **PMM_SVMHVLRIFG**
- **PMM_PMMBORIFG**
- **PMM_PMMRSTIFG**
- **PMM_PMMPORIFG**
- **PMM_SVSHIFG**
- **PMM_SVSLIFG**
- **PMM_PMMLPM5IFG**

Returns:
Logical OR of any of the following:

- **PMM_SVSMLDLYIFG**
- **PMM_SVMLIFG**
- **PMM_SVMLVLRIFG**
- **PMM_SVSMHDLYIFG**
- **PMM_SVMHIFG**
- **PMM_SVMHVLRIFG**
- **PMM_PMMBORIFG**

- **PMM_PMMRSTIFG**
- **PMM_PMMPORIFG**
- **PMM_SVSHIFG**
- **PMM_SVSLIFG**
- **PMM_PMMLPM5IFG**
  indicating the status of the masked interrupts

22.2.2.23 bool PMM_setVCore (uint8_t *level*)

Set Vcore to expected level.

Parameters:
level level to which Vcore needs to be decreased/increased Valid values are:
- **PMM_CORE_LEVEL_0** [Default]
- **PMM_CORE_LEVEL_1**
- **PMM_CORE_LEVEL_2**
- **PMM_CORE_LEVEL_3**

Modified bits of **PMMCTL0** register, bits of **PMMIFG** register, bits of **PMMRIE** register, bits of **SVSMHCTL** register and bits of **SVSMLCTL** register.

Returns:
STATUS_SUCCESS or STATUS_FAIL

22.2.2.24 uint16_t PMM_setVCoreDown (uint8_t *level*)

Decrease Vcore by one level.

Parameters:
level level to which Vcore needs to be decreased Valid values are:
- **PMM_CORE_LEVEL_0** [Default]
- **PMM_CORE_LEVEL_1**
- **PMM_CORE_LEVEL_2**
- **PMM_CORE_LEVEL_3**

Modified bits of **PMMCTL0** register, bits of **PMMIFG** register, bits of **PMMRIE** register, bits of **SVSMHCTL** register and bits of **SVSMLCTL** register.

Returns:
STATUS_SUCCESS

22.2.2.25 uint16_t PMM_setVCoreUp (uint8_t *level*)

Increase Vcore by one level.

Parameters:
level level to which Vcore needs to be increased Valid values are:
- **PMM_CORE_LEVEL_0** [Default]
- **PMM_CORE_LEVEL_1**
- **PMM_CORE_LEVEL_2**
- **PMM_CORE_LEVEL_3**

Modified bits of **PMMCTL0** register, bits of **PMMIFG** register, bits of **PMMRIE** register, bits of **SVSMHCTL** register and bits of **SVSMLCTL** register.

Returns:
STATUS_SUCCESS or STATUS_FAIL

22.2.2.26 void PMM_SvsHDisabledInLPMFullPerf (void)

Disables supervisor high side in LPM with tpd = 2.5 ?s(1).

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.27 void PMM_SvsHDisabledInLPMNormPerf (void)

Disables supervisor high side in LPM with tpd = 20 ?s(1).

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.28 void PMM_SvsHEnabledInLPMFullPerf (void)

Enables supervisor high side in LPM with tpd = 2.5 ?s(1).

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.29 void PMM_SvsHEnabledInLPMNormPerf (void)

Enables supervisor high side in LPM with tpd = 20 ?s(1).

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns:
None

22.2.2.30 void PMM_SvsHOptimizedInLPMFullPerf (void)

Optimized to provide tpd = 2.5 ?s(1) in LPM with least power.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.31 void PMM_SvsLDisabledInLPMFastWake (void)

Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.32 void PMM_SvsLDisabledInLPMSlowWake (void)

Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.33 void PMM_SvsLEnabledInLPMFastWake (void)

Enables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.34 void PMM_SvsLEnabledInLPMSlowWake (void)

Enables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.2.2.35 void PMM_SvsLOptimizedInLPMFastWake (void)

Optimized to provide twake-up-fast from LPM2, LPM3, and LPM4 with least power.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns:
None

22.3 Programming Example

The following example shows some pmm operations using the APIs

```
//Use the line below to bring the level back to 0
    status = PMM_setVCore(PMM_BASE,
        PMMCOREV_0
        );

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

    //continuous loop
    while (1)
    {
        //Toggle P1.0
        GPIO_toggleOutputOnPin(
            GPIO_PORT_P1,
            GPIO_PIN0
            );
        //Delay
        __delay_cycles(20000);
    }
```

23 RAM Controller

23.1 Introduction

The RAMCTL provides access to the different power modes of the RAM. The RAMCTL allows the ability to reduce the leakage current while the CPU is off. The RAM can also be switched off. In retention mode, the RAM content is saved while the RAM content is lost in off mode. The RAM is partitioned in sectors, typically of 4KB (sector) size. See the device-specific data sheet for actual block allocation and size. Each sector is controlled by the RAM controller RAM Sector Off control bit (RCRSyOFF) of the RAMCTL Control 0 register (RCCTL0). The RCCTL0 register is protected with a key. Only if the correct key is written during a word write, the RCCTL0 register content can be modified. Byte write accesses or write accesses with a wrong key are ignored.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	18
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

23.2 API Functions

Functions

- uint8_t RAM_getSectorState (uint8_t sector)
- void RAM_setSectorOff (uint8_t sector)

23.2.1 Detailed Description

The MSP430ware API that configure the RAM controller are:

RAM_setSectorOff() - Set specified RAM sector off RAM_getSectorState() - Get RAM sector ON/OFF status

23.2.2 Function Documentation

23.2.2.1 uint8_t RAM_getSectorState (uint8_t *sector*)

Get RAM sector ON/OFF status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
sector is specified sector Mask value is the logical OR of any of the following:
- **RAM_SECTOR0**
- **RAM_SECTOR1**
- **RAM_SECTOR2**
- **RAM_SECTOR3**
- **RAM_SECTOR4**
- **RAM_SECTOR5**
- **RAM_SECTOR6**
- **RAM_SECTOR7**

Modified bits of **RCCTL0** register.

Returns:
Logical OR of any of the following:

- **RAM_SECTOR0**
- **RAM_SECTOR1**
- **RAM_SECTOR2**
- **RAM_SECTOR3**
- **RAM_SECTOR4**
- **RAM_SECTOR5**
- **RAM_SECTOR6**
- **RAM_SECTOR7**
indicating the status of the masked sectors

23.2.2.2 void RAM_setSectorOff (uint8_t *sector*)

Set specified RAM sector off.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

sector is specified sector to be set off. Mask value is the logical OR of any of the following:

- **RAM_SECTOR0**
- **RAM_SECTOR1**
- **RAM_SECTOR2**
- **RAM_SECTOR3**
- **RAM_SECTOR4**
- **RAM_SECTOR5**
- **RAM_SECTOR6**
- **RAM_SECTOR7**

Modified bits of **RCCTL0** register.

Returns:

None

23.3 Programming Example

The following example shows some RAM Controller operations using the APIs

```
    //Start timer
    Timer_startUpMode(   TIMER_B0_BASE,
        TIMER_CLOCKSOURCE_ACLK,
        TIMER_CLOCKSOURCE_DIVIDER_1,
        25000,
        TIMER_TAIE_INTERRUPT_DISABLE,
        TIMER_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_DO_CLEAR
        );

    //RAM controller sector off
    RAM_setSectorOff(RAM_BASE,
        RAMCONTROL_SECTOR2
        ) ;


    //Enter LPM0, enable interrupts
    __bis_SR_register(LPM3_bits + GIE);

    //For debugger
    __no_operation();
}

//******************************************************************************
//
//This is the Timer B0 interrupt vector service routine.
//
```

```
//*****************************************************************************
#pragma vector=TIMERB0_VECTOR
__interrupt void TIMERB0_ISR (void)
{
    returnValue = RAM_getSectorState(RAM_BASE,
        RAM_SECTOR0 +
        RAM_SECTOR1 +
        RAM_SECTOR2 +
        RAM_SECTOR3);

}
```

24 Internal Reference (REF)

24.1 Introduction

The Internal Reference (REF) API provides a set of functions for using the MSP430Ware REF modules. Functions are provided to setup and enable use of the Reference voltage, enable or disable the internal temperature sensor, and view the status of the inner workings of the REF module.

The reference module (REF) is responsible for generation of all critical reference voltages that can be used by various analog peripherals in a given device. These include, but are not necessarily limited to, the ADC10_A, ADC12_A, DAC12_A, LCD_B, and COMP_B modules dependent upon the particular device. The heart of the reference system is the bandgap from which all other references are derived by unity or non-inverting gain stages. The REFGEN sub-system consists of the bandgap, the bandgap bias, and the non-inverting buffer stage which generates the three primary voltage reference available in the system, namely 1.5 V, 2.0 V, and 2.5 V. In addition, when enabled, a buffered bandgap voltage is also available.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	214
TI Compiler 4.2.1	Size	100
TI Compiler 4.2.1	Speed	100
IAR 5.51.6	None	112
IAR 5.51.6	Size	78
IAR 5.51.6	Speed	78
MSPGCC 4.8.0	None	314
MSPGCC 4.8.0	Size	86
MSPGCC 4.8.0	Speed	86

24.2 API Functions

Functions

- void REF_disableReferenceVoltage (uint16_t baseAddress)
- void REF_disableReferenceVoltageOutput (uint16_t baseAddress)
- void REF_disableTempSensor (uint16_t baseAddress)
- void REF_enableReferenceVoltage (uint16_t baseAddress)
- void REF_enableReferenceVoltageOutput (uint16_t baseAddress)
- void REF_enableTempSensor (uint16_t baseAddress)
- uint16_t REF_getBandgapMode (uint16_t baseAddress)
- bool REF_isBandgapActive (uint16_t baseAddress)
- bool REF_isRefGenActive (uint16_t baseAddress)
- uint16_t REF_isRefGenBusy (uint16_t baseAddress)
- void REF_setReferenceVoltage (uint16_t baseAddress, uint8_t referenceVoltageSelect)

24.2.1 Detailed Description

The DMA API is broken into three groups of functions: those that deal with the reference voltage, those that handle the internal temperature sensor, and those that return the status of the REF module.

The reference voltage of the REF module is handled by

- REF_setReferenceVoltage()
- REF_enableReferenceVoltageOutput()
- REF_disableReferenceVoltageOutput()
- REF_enableReferenceVoltage()
- REF_disableReferenceVoltage()

The internal temperature sensor is handled by

- REF_disableTempSensor()
- REF_enableTempSensor()

The status of the REF module is handled by

- REF_getBandgapMode()
- REF_isBandgapActive()
- REF_isRefGenBusy()
- REF_isRefGen()

24.2.2 Function Documentation

24.2.2.1 void REF_disableReferenceVoltage (uint16_t *baseAddress*)

Disables the reference voltage.

This function is used to disable the generated reference voltage. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Modified bits are **REFON** of **REFCTL0** register.

Returns:
None

24.2.2.2 void REF_disableReferenceVoltageOutput (uint16_t *baseAddress*)

Disables the reference voltage as an output to a pin.

This function is used to disables the reference voltage being generated to be given to an output pin. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Returns:
None

24.2.2.3 void REF_disableTempSensor (uint16_t *baseAddress*)

Disables the internal temperature sensor to save power consumption.

This function is used to turn off the internal temperature sensor to save on power consumption. The temperature sensor is enabled by default. Please note, that giving ADC12 module control over the REF module, the state of the temperature sensor is dependent on the controls of the ADC12 module. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Modified bits are **RFETCOFF** of **REFCTL0** register.

Returns:
None

24.2.2.4 void REF_enableReferenceVoltage (uint16_t *baseAddress*)

Enables the reference voltage to be used by peripherals.

This function is used to enable the generated reference voltage to be used other peripherals or by an output pin, if enabled. Please note, that giving ADC12 module control over the REF module, the state of the reference voltage is dependent on the controls of the ADC12 module. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Modified bits are **REFON** of **REFCTL0** register.

Returns:
None

24.2.2.5 void REF_enableReferenceVoltageOutput (uint16_t *baseAddress*)

Outputs the reference voltage to an output pin.

This function is used to output the reference voltage being generated to an output pin. Please note, the output pin is device specific. Please note, that giving ADC12 module control over the REF module, the state of the reference voltage as an output to a pin is dependent on the controls of the ADC12 module. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Modified bits are **REFOUT** of **REFCTL0** register.

Returns:
None

24.2.2.6 void REF_enableTempSensor (uint16_t *baseAddress*)

Enables the internal temperature sensor.

This function is used to turn on the internal temperature sensor to use by other peripherals. The temperature sensor is enabled by default. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the REF module.

Modified bits are **REFTCOFF** of **REFCTL0** register.

Returns:
None

24.2.2.7 uint16_t REF_getBandgapMode (uint16_t *baseAddress*)

Returns the bandgap mode of the REF module.

This function is used to return the bandgap mode of the REF module, requested by the peripherals using the bandgap. If a peripheral requests static mode, then the bandgap mode will be static for all modules, whereas if all of the peripherals using the bandgap request sample mode, then that will be the mode returned. Sample mode allows the bandgap to be active only when necessary to save on power consumption, static mode requires the bandgap to be active until no peripherals are using it anymore.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the REF module.

Returns:
One of the following:

- **REF_STATICMODE** if the bandgap is operating in static mode
- **REF_SAMPLEMODE** if the bandgap is operating in sample mode
indicating the REF bandgap mode

24.2.2.8 bool REF_isBandgapActive (uint16_t *baseAddress*)

Returns the active status of the bandgap in the REF module.

This function is used to return the active status of the bandgap in the REF module. If the bandgap is in use by a peripheral, then the status will be seen as active.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	18
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
baseAddress is the base address of the REF module.

Returns:
One of the following:

- **REF_INACTIVE**
- **REF_ACTIVE**
  indicating the bandgap active status of the REF module

24.2.2.9 bool REF_isRefGenActive (uint16_t *baseAddress*)

Returns the active status of the reference generator in the REF module.

This function is used to return the active status of the reference generator in the REF module. If the ref. generator is on and ready to use, then the status will be seen as active.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	18
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the REF module.

Returns:
One of the following:

- **REF_INACTIVE**
- **REF_ACTIVE**
  indicating the REF generator status

24.2.2.10 uint16_t REF_isRefGenBusy (uint16_t *baseAddress*)

Returns the busy status of the reference generator in the REF module.

This function is used to return the busy status of the reference generator in the REF module. If the ref. generator is in use by a peripheral, then the status will be seen as busy.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the REF module.

Returns:
One of the following:

- **REF_NOTBUSY** if the reference generator is not being used
- **REF_BUSY** if the reference generator is being used, disallowing any changes to be made to the REF module controls
  indicating the REF generator busy status

24.2.2.11 void REF_setReferenceVoltage (uint16_t *baseAddress*, uint8_t *referenceVoltageSelect*)

Sets the reference voltage for the voltage generator.

This function sets the reference voltage generated by the voltage generator to be used by other peripherals. This reference voltage will only be valid while the REF module is in control. Please note, if the REF_isRefGenBusy() returns REF_BUSY, this function will have no effect.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
baseAddress is the base address of the REF module.
referenceVoltageSelect is the desired voltage to generate for a reference voltage. Valid values are:

- **REF_VREF1_5V** [Default]
- **REF_VREF2_0V**

- **REF_VREF2_5V**
  Modified bits are **REFVSEL** of **REFCTL0** register.

Returns:
None

24.3 Programming Example

The following example shows how to initialize and use the REF API with the ADC12_A module to use as a positive reference to the analog signal input.

```
 // By default, REFMSTR=1 => REFCTL is used to configure the internal reference

 // If ref generator busy, WAIT
 while(REF_refGenBusyStatus(REF_BASE));
 // Select internal ref = 2.5V
 REF_setReferenceVoltage(REF_BASE,
                         REF_VREF2_5V);
 // Internal Reference ON
 REF_enableReferenceVoltage(REF_BASE);

 __delay_cycles(75);                       // Delay (~75us) for Ref to settle

 // Initialize the ADC12_A Module
 /*
Base address of ADC12_A Module
Use internal ADC12_A bit as sample/hold signal to start conversion
USE MODOSC 5MHZ Digital Oscillator as clock source
Use default clock divider of 1
  */
 ADC12_A_init(ADC12_A_BASE,
           ADC12_A_SAMPLEHOLDSOURCE_SC,
           ADC12_A_CLOCKSOURCE_ADC12OSC,
           ADC12_A_CLOCKDIVIDEBY_1);
 /*
Base address of ADC12 Module
For memory buffers 0-7 sample/hold for 64 clock cycles
For memory buffers 8-15 sample/hold for 4 clock cycles (default)
Disable Multiple Sampling
  */
 ADC12_A_setupSamplingTimer(ADC12_A_BASE,
                         ADC12_A_CYCLEHOLD_64_CYCLES,
                         ADC12_A_CYCLEHOLD_4_CYCLES,
                         ADC12_A_MULTIPLESAMPLESENABLE);

 // Configure Memory Buffer
 /*
Base address of the ADC12 Module
Configure memory buffer 0
Map input A0 to memory buffer 0
Vref+ = Vref+ (INT)
Vref- = AVss
  */
 ADC12_A_memoryConfigure(ADC12_A_BASE,
                       ADC12_A_MEMORY_0,
                       ADC12_A_INPUT_A0,
                       ADC12_A_VREFPOS_INT,
                       ADC12_A_VREFNEG_AVSS,
                       ADC12_A_NOTENDOFSEQUENCE);

 while (1)
 {
   // Enable/Start sampling and conversion
   /*
Base address of ADC12 Module
Start the conversion into memory buffer 0
Use the single-channel, single-conversion mode
    */
   ADC12_A_startConversion(ADC12_A_BASE,
                          ADC12_A_MEMORY_0,
                          ADC12_A_SINGLECHANNEL);

   // Poll for interrupt on memory buffer 0
   while(!ADC12_A_interruptStatus(ADC12_A_BASE, ADC12IFG0));

   __no_operation();                       // SET BREAKPOINT HERE
 }
```

25 Real-Time Clock (RTC_A)

25.1 Introduction

The Real Time Clock (RTC_A) API provides a set of functions for using the MSP430Ware RTC_A modules. Functions are provided to calibrate the clock, initialize the RTC_A modules in calendar mode/counter mode and setup conditions for, and enable, interrupts for the RTC_A modules. If an RTC_A module is used, then counter mode may also be initialized, as well as prescale counters.

The RTC_A module provides the ability to keep track of the current time and date in calendar mode, or can be setup as a 32-bit counter (RTC_A Only).

The RTC_A module generates multiple interrupts. There are 2 interrupts that can be defined in calendar mode, and 1 interrupt in counter mode for counter overflow, as well as an interrupt for each prescaler.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1320
TI Compiler 4.2.1	Size	672
TI Compiler 4.2.1	Speed	668
IAR 5.51.6	None	1006
IAR 5.51.6	Size	830
IAR 5.51.6	Speed	864
MSPGCC 4.8.0	None	2242
MSPGCC 4.8.0	Size	798
MSPGCC 4.8.0	Speed	802

25.2 API Functions

Functions

- void RTC_A_calendarInit (uint16_t baseAddress, Calendar CalendarTime, uint16_t formatSelect)
- void RTC_A_clearInterrupt (uint16_t baseAddress, uint8_t interruptFlagMask)
- void RTC_A_configureCalendarAlarm (uint16_t baseAddress, RTC_A_configureCalendarAlarmParam ∗param)
- void RTC_A_definePrescaleEvent (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
- void RTC_A_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- void RTC_A_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- Calendar RTC_A_getCalendarTime (uint16_t baseAddress)
- uint32_t RTC_A_getCounterValue (uint16_t baseAddress)
- uint8_t RTC_A_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
- uint8_t RTC_A_getPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect)
- void RTC_A_holdClock (uint16_t baseAddress)
- void RTC_A_holdCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect)

- void RTC_A_initCalendar (uint16_t baseAddress, Calendar ∗CalendarTime, uint16_t formatSelect)
- void RTC_A_initCounter (uint16_t baseAddress, uint16_t clockSelect, uint16_t counterSizeSelect)
- void RTC_A_initCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect, uint16_t prescaleClockSelect, uint16_t prescaleDivider)
- void RTC_A_setCalendarAlarm (uint16_t baseAddress, uint8_t minutesAlarm, uint8_t hoursAlarm, uint8_t dayOfWeekAlarm, uint8_t dayOfMonthAlarm)
- void RTC_A_setCalendarEvent (uint16_t baseAddress, uint16_t eventSelect)
- void RTC_A_setCalibrationData (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
- void RTC_A_setCalibrationFrequency (uint16_t baseAddress, uint16_t frequencySelect)
- void RTC_A_setCounterValue (uint16_t baseAddress, uint32_t counterValue)
- void RTC_A_setPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
- void RTC_A_startClock (uint16_t baseAddress)
- void RTC_A_startCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect)

25.2.1 Detailed Description

The RTC_A API is broken into 5 groups of functions: clock settings, calender mode, counter mode, prescale counter, and interrupt condition setup/enable functions and data conversion.

The RTC_A clock settings are handled by

- RTC_A_startClock()
- RTC_A_holdClock()
- RTC_A_setCalibrationFrequency()
- RTC_A_setCalibrationData()

The RTC_A calender mode is initialized and setup by

- RTC_A_calenderInit()
- RTC_A_getCalenderTime()

The RTC_A counter mode is initialized and setup by

- RTC_A_counterInit()
- RTC_A_getCounterValue()
- RTC_A_setCounterValue()
- RTC_A_initCounterPrescale()
- RTC_A_holdCounterPrescale()
- RTC_A_startCounterPrescale()

The RTC_A prescale counter is handled by

- RTC_A_getPrescaleValue()
- RTC_A_setPrescaleValue()

The RTC_A interrupts are handled by

- RTC_A_configureCalendarAlarm()
- RTC_A_setCalenderEvent()
- RTC_A_definePrescaleEvent()
- RTC_A_enableInterrupt()
- RTC_A_disableInterrupt()
- RTC_A_getInterruptStatus()
- RTC_A_clearInterrupt()

25.2.2 Function Documentation

25.2.2.1 void RTC_A_calendarInit (uint16_t *baseAddress*, Calendar *CalendarTime*, uint16_t *formatSelect*)

Deprecated - Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	126
TI Compiler 4.2.1	Size	84
TI Compiler 4.2.1	Speed	84
IAR 5.51.6	None	108
IAR 5.51.6	Size	90
IAR 5.51.6	Speed	104
MSPGCC 4.8.0	None	218
MSPGCC 4.8.0	Size	126
MSPGCC 4.8.0	Speed	126

Parameters:

baseAddress is the base address of the RTC_A module.

CalendarTime is the structure containing the values for the Calendar to be initialized to. Valid values should be of type Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.

formatSelect is the format for the Calendar registers to use. Valid values are:

- **RTC_A_FORMAT_BINARY** [Default]
- **RTC_A_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:

None

25.2.2.2 void RTC_A_clearInterrupt (uint16_t *baseAddress*, uint8_t *interruptFlagMask*)

Clears selected RTC interrupt flags.

This function clears the RTC interrupt flag is cleared, so that it no longer asserts.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	70
TI Compiler 4.2.1	Size	40
TI Compiler 4.2.1	Speed	40
IAR 5.51.6	None	48
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	154
MSPGCC 4.8.0	Size	40
MSPGCC 4.8.0	Speed	40

Parameters:

baseAddress is the base address of the RTC_A module.

interruptFlagMask is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:
- **RTC_A_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_A_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_A_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_A_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_A_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.

Returns:
None

25.2.2.3 void RTC_A_configureCalendarAlarm (uint16_t *baseAddress*, RTC_A_configureCalendarAlarmParam ∗ *param*)

Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_A_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	48
IAR 5.51.6	None	72
IAR 5.51.6	Size	68
IAR 5.51.6	Speed	68
MSPGCC 4.8.0	None	116
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

Parameters:
baseAddress is the base address of the RTC_A module.
param is the pointer to struct for calendar alarm configuration.

Returns:
None

25.2.2.4 void RTC_A_definePrescaleEvent (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleEventDivider*)

Sets up an interrupt condition for the selected Prescaler.

This function sets the condition for an interrupt to assert based on the individual prescalers.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	28
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

Parameters:

baseAddress is the base address of the RTC_A module.

prescaleSelect is the prescaler to define an interrupt for. Valid values are:

- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

prescaleEventDivider is a divider to specify when an interrupt can occur based on the clock source of the selected prescaler. (Does not affect timer of the selected prescaler). Valid values are:

- **RTC_A_PSEVENTDIVIDER_2** [Default]
- **RTC_A_PSEVENTDIVIDER_4**
- **RTC_A_PSEVENTDIVIDER_8**
- **RTC_A_PSEVENTDIVIDER_16**
- **RTC_A_PSEVENTDIVIDER_32**
- **RTC_A_PSEVENTDIVIDER_64**
- **RTC_A_PSEVENTDIVIDER_128**
- **RTC_A_PSEVENTDIVIDER_256**
  Modified bits are **RTxIP** of **RTCPSxCTL** register.

Returns:

None

25.2.2.5 void RTC_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	66
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	44
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	148
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

Parameters:

baseAddress is the base address of the RTC_A module.

interruptMask is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following:

- **RTC_A_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_A_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.

- **RTC_A_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_A_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_A_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.

Returns:
None

25.2.2.6 void RTC_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	66
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	44
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	136
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

Parameters:
baseAddress is the base address of the RTC_A module.
interruptMask is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following:

- **RTC_A_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_A_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_A_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_A_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_A_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.

Returns:
None

25.2.2.7 Calendar RTC_A_getCalendarTime (uint16_t *baseAddress*)

Returns the Calendar Time stored in the Calendar registers of the RTC.

This function returns the current Calendar time in the form of a Calendar structure. The RTCRDY polling is used in this function to prevent reading invalid time.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	68
TI Compiler 4.2.1	Speed	68
IAR 5.51.6	None	104
IAR 5.51.6	Size	104
IAR 5.51.6	Speed	104
MSPGCC 4.8.0	None	148
MSPGCC 4.8.0	Size	68
MSPGCC 4.8.0	Speed	68

Parameters:
baseAddress is the base address of the RTC_A module.

Returns:
A Calendar structure containing the current time.

25.2.2.8 uint32_t RTC_A_getCounterValue (uint16_t *baseAddress*)

Returns the value of the Counter register.

This function returns the value of the counter register for the RTC_A module. It will return the 32-bit value no matter the size set during initialization. The RTC should be held before trying to use this function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	68
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	62
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	122
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	52

Parameters:
baseAddress is the base address of the RTC_A module.

Returns:
The raw value of the full 32-bit Counter Register.

25.2.2.9 uint8_t RTC_A_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	86
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	44
IAR 5.51.6	None	58
IAR 5.51.6	Size	48
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	146
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	54

Parameters:
baseAddress is the base address of the RTC_A module.
interruptFlagMask is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following:

- **RTC_A_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_A_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_A_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_A_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_A_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.

Returns:
Logical OR of any of the following:

- **RTC_A_TIME_EVENT_INTERRUPT** asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_A_CLOCK_ALARM_INTERRUPT** asserts when alarm condition in Calendar mode is met.
- **RTC_A_CLOCK_READ_READY_INTERRUPT** asserts when Calendar registers are settled.
- **RTC_A_PRESCALE_TIMER0_INTERRUPT** asserts when Prescaler 0 event condition is met.
- **RTC_A_PRESCALE_TIMER1_INTERRUPT** asserts when Prescaler 1 event condition is met.
indicating the status of the masked interrupts

25.2.2.10 uint8_t RTC_A_getPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Returns the selected prescaler value.

This function returns the value of the selected prescale counter register. Note that the counter value should be held by calling RTC_A_holdClock() before calling this API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	56
TI Compiler 4.2.1	Size	34
TI Compiler 4.2.1	Speed	34
IAR 5.51.6	None	50
IAR 5.51.6	Size	44
IAR 5.51.6	Speed	44
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	38

Parameters:
baseAddress is the base address of the RTC_A module.
prescaleSelect is the prescaler to obtain the value of. Valid values are:
- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

Returns:
The value of the specified prescaler count register

25.2.2.11 void RTC_A_holdClock (uint16_t *baseAddress*)

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the RTC_A module.

Returns:

None

25.2.2.12 void RTC_A_holdCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Holds the selected Prescaler.

This function holds the prescale counter from continuing. This will only work in counter mode, in Calendar mode, the RTC_A_holdClock() must be used. In counter mode, if using both prescalers in conjunction with the main RTC counter, then stopping RT0PS will stop RT1PS, but stopping RT1PS will not stop RT0PS.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	16
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the RTC_A module.

prescaleSelect is the prescaler to hold. Valid values are:

- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

Returns:

None

25.2.2.13 void RTC_A_initCalendar (uint16_t *baseAddress*, Calendar $*$ *CalendarTime*, uint16_t *formatSelect*)

Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	130
TI Compiler 4.2.1	Size	58
TI Compiler 4.2.1	Speed	58
IAR 5.51.6	None	106
IAR 5.51.6	Size	88
IAR 5.51.6	Speed	102
MSPGCC 4.8.0	None	220
MSPGCC 4.8.0	Size	58
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the RTC_A module.

CalendarTime is the pointer to the structure containing the values for the Calendar to be initialized to. Valid values should be of type pointer to Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.

formatSelect is the format for the Calendar registers to use. Valid values are:

- **RTC_A_FORMAT_BINARY** [Default]
- **RTC_A_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:

None

25.2.2.14 void RTC_A_initCounter (uint16_t *baseAddress*, uint16_t *clockSelect*, uint16_t *counterSizeSelect*)

Initializes the settings to operate the RTC in Counter mode.

This function initializes the Counter mode of the RTC_A. Setting the clock source and counter size will allow an interrupt from the RTCTEVIFG once an overflow to the counter register occurs.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	64
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	36
IAR 5.51.6	Size	34
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	114
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the RTC_A module.

clockSelect is the selected clock for the counter mode to use. Valid values are:

- **RTC_A_CLOCKSELECT_ACLK** [Default]
- **RTC_A_CLOCKSELECT_SMCLK**
- **RTC_A_CLOCKSELECT_RT1PS** - use Prescaler 1 as source to RTC
  Modified bits are **RTCSSEL** of **RTCCTL1** register.

counterSizeSelect is the size of the counter. Valid values are:

- **RTC_A_COUNTERSIZE_8BIT** [Default]
- **RTC_A_COUNTERSIZE_16BIT**
- **RTC_A_COUNTERSIZE_24BIT**

- **RTC_A_COUNTERSIZE_32BIT**
  Modified bits are **RTCTEV** of **RTCCTL1** register.

Returns:
None

25.2.2.15 void RTC_A_initCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint16_t *prescaleClockSelect*, uint16_t *prescaleDivider*)

Initializes the Prescaler for Counter mode.

This function initializes the selected prescaler for the counter mode in the RTC_A module. If the RTC is initialized in Calendar mode, then these are automatically initialized. The Prescalers can be used to divide a clock source additionally before it gets to the main RTC clock.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	22
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:
baseAddress is the base address of the RTC_A module.
prescaleSelect is the prescaler to initialize. Valid values are:
- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

prescaleClockSelect is the clock to drive the selected prescaler. Valid values are:
- **RTC_A_PSCLOCKSELECT_ACLK**
- **RTC_A_PSCLOCKSELECT_SMCLK**
- **RTC_A_PSCLOCKSELECT_RT0PS** - use Prescaler 0 as source to Prescaler 1 (May only be used if prescaleSelect is RTC_A_PRESCALE_1)
  Modified bits are **RTxSSEL** of **RTCPSxCTL** register.

prescaleDivider is the divider for the selected clock source. Valid values are:
- **RTC_A_PSDIVIDER_2** [Default]
- **RTC_A_PSDIVIDER_4**
- **RTC_A_PSDIVIDER_8**
- **RTC_A_PSDIVIDER_16**
- **RTC_A_PSDIVIDER_32**
- **RTC_A_PSDIVIDER_64**
- **RTC_A_PSDIVIDER_128**
- **RTC_A_PSDIVIDER_256**
  Modified bits are **RTxPSDIV** of **RTCPSxCTL** register.

Returns:
None

25.2.2.16 void RTC_A_setCalendarAlarm (uint16_t *baseAddress*, uint8_t *minutesAlarm*, uint8_t *hoursAlarm*, uint8_t *dayOfWeekAlarm*, uint8_t *dayOfMonthAlarm*)

DEPRECATED - Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_A_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	60
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	38

Parameters:

baseAddress is the base address of the RTC_A module.

minutesAlarm is the alarm condition for the minutes. Valid values are:
- **RTC_A_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-59

hoursAlarm is the alarm condition for the hours. Valid values are:
- **RTC_A_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-24

dayOfWeekAlarm is the alarm condition for the day of week. Valid values are:
- **RTC_A_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-6

dayOfMonthAlarm is the alarm condition for the day of the month. Valid values are:
- **RTC_A_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-31

Returns:

None

25.2.2.17 void RTC_A_setCalendarEvent (uint16_t *baseAddress*, uint16_t *eventSelect*)

Sets a single specified Calendar interrupt condition.

This function sets a specified event to assert the RTCTEVIFG interrupt. This interrupt is independent from the Calendar alarm interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the RTC_A module.

eventSelect is the condition selected. Valid values are:

- **RTC_A_CALENDAREVENT_MINUTECHANGE** - assert interrupt on every minute
- **RTC_A_CALENDAREVENT_HOURCHANGE** - assert interrupt on every hour
- **RTC_A_CALENDAREVENT_NOON** - assert interrupt when hour is 12
- **RTC_A_CALENDAREVENT_MIDNIGHT** - assert interrupt when hour is 0
  Modified bits are **RTCTEV** of **RTCCTL** register.

Returns:

None

25.2.2.18 void RTC_A_setCalibrationData (uint16_t *baseAddress*, uint8_t *offsetDirection*, uint8_t *offsetValue*)

Sets the specified calibration for the RTC.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +4-ppm or -2-ppm, and the offsetValue should be from 1-63 and is multiplied by the direction setting (i.e. +4-ppm $*$ 8 (offsetValue) = +32-ppm). Please note, when measuring the frequency after setting the calibration, you will only see a change on the 1Hz frequency.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	18
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the RTC_A module.

offsetDirection is the direction that the calibration offset will go. Valid values are:

- **RTC_A_CALIBRATION_DOWN2PPM** - calibrate at steps of -2
- **RTC_A_CALIBRATION_UP4PPM** - calibrate at steps of +4
  Modified bits are **RTCCALS** of **RTCCTL2** register.

offsetValue is the value that the offset will be a factor of; a valid value is any integer from 1-63.
Modified bits are **RTCCAL** of **RTCCTL2** register.

Returns:

None

25.2.2.19 void RTC_A_setCalibrationFrequency (uint16_t *baseAddress*, uint16_t *frequencySelect*)

Allows and Sets the frequency output to RTCCLK pin for calibration measurement.

This function sets a frequency to measure at the RTCCLK output pin. After testing the set frequency, the calibration could be set accordingly.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

baseAddress is the base address of the RTC_A module.

frequencySelect is the frequency output to RTCCLK. Valid values are:

- **RTC_A_CALIBRATIONFREQ_OFF** [Default] - turn off calibration output
- **RTC_A_CALIBRATIONFREQ_512HZ** - output signal at 512Hz for calibration
- **RTC_A_CALIBRATIONFREQ_256HZ** - output signal at 256Hz for calibration
- **RTC_A_CALIBRATIONFREQ_1HZ** - output signal at 1Hz for calibration
Modified bits are **RTCCALF** of **RTCCTL3** register.

Returns:

None

25.2.2.20 void RTC_A_setCounterValue (uint16_t *baseAddress*, uint32_t *counterValue*)

Sets the value of the Counter register.

This function sets the counter register of the RTC_A module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	22
IAR 5.51.6	Size	20
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	32

Parameters:

baseAddress is the base address of the RTC_A module.

counterValue is the value to set the Counter register to; a valid value may be any 32-bit integer.

Returns:

None

25.2.2.21 void RTC_A_setPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleCounterValue*)

Sets the selected prescaler value.

This function sets the prescale counter value. Before setting the prescale counter, it should be held by calling RTC_A_holdClock().

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	44
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	32
IAR 5.51.6	Size	24
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the RTC_A module.

prescaleSelect is the prescaler to set the value for. Valid values are:

- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

prescaleCounterValue is the specified value to set the prescaler to. Valid values are any integer between 0-255 Modified bits are **RTxPS** of **RTxPS** register.

Returns:

None

25.2.2.22 void RTC_A_startClock (uint16_t *baseAddress*)

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the RTC_A module.

Returns:

None

25.2.2.23 void RTC_A_startCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Starts the selected Prescaler.

This function starts the selected prescale counter. This function will only work if the RTC is in counter mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	16
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the RTC_A module.

prescaleSelect is the prescaler to start. Valid values are:

- **RTC_A_PRESCALE_0**
- **RTC_A_PRESCALE_1**

Returns:

None

25.3 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 calendar struct
    Calendar currentTime;
    currentTime.Seconds    = 0x00;
    currentTime.Minutes    = 0x26;
    currentTime.Hours      = 0x13;
    currentTime.DayOfWeek  = 0x03;
    currentTime.DayOfMonth = 0x20;
    currentTime.Month      = 0x07;
    currentTime.Year       = 0x2011;

    //Initialize alarm struct
    RTC_A_configureCalendarAlarmParam alarmParam;
    alarmParam.minutesAlarm = 0x00;
    alarmParam.hoursAlarm = 0x17;
    alarmParam.dayOfWeekAlarm = RTC_A_ALARMCONDITION_OFF;
    alarmParam.dayOfMonthAlarm = 0x05;

    //Initialize Calendar Mode of RTC_A
    /*
 Base Address of the RTC_A
 Pass in current time, initialized above
 Use BCD as Calendar Register Format
     */
    RTC_A_initCalendar(RTC_A_BASE,
        &currentTime,
        RTC_A_FORMAT_BCD);

    //Setup Calendar Alarm for 5:00pm on the 5th day of the month.
    //Note: Does not specify day of the week.
    RTC_C_configureCalendarAlarm(RTC_A_BASE, &alarmParam);

    //Specify an interrupt to assert every minute
    RTC_A_setCalendarEvent(RTC_A_BASE,
        RTC_A_CALENDAREVENT_MINUTECHANGE);
```

```
//Enable interrupt for RTC_A Ready Status, which asserts when the RTC_A
//Calendar registers are ready to read.
//Also, enable interrupts for the Calendar alarm and Calendar event.
RTC_A_enableInterrupt(RTC_A_BASE,
    RTC_A_CLOCK_READ_READY_INTERRUPT +
    RTC_A_TIME_EVENT_INTERRUPT +
    RTC_A_CLOCK_ALARM_INTERRUPT);

//Start RTC_A Clock
RTC_A_startClock(RTC_A_BASE);

//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

26 Real-Time Clock (RTC_B)

26.1 Introduction

The Real Time Clock (RTC_B) API provides a set of functions for using the MSP430Ware RTC_B modules. Functions are provided to calibrate the clock, initialize the RTC modules in calendar mode, and setup conditions for, and enable, interrupts for the RTC modules. If an RTC_B module is used, then prescale counters are also initialized.

The RTC_B module provides the ability to keep track of the current time and date in calendar mode.

The RTC_B module generates multiple interrupts. There are 2 interrupts that can be defined in calendar mode, and 1 interrupt for user-configured event, as well as an interrupt for each prescaler.

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

26.2 API Functions

Functions

- void RTC_B_calendarInit (uint16_t baseAddress, Calendar CalendarTime, uint16_t formatSelect)
- void RTC_B_clearInterrupt (uint16_t baseAddress, uint8_t interruptFlagMask)
- void RTC_B_configureCalendarAlarm (uint16_t baseAddress, RTC_B_configureCalendarAlarmParam ∗param)
- uint16_t RTC_B_convertBCDToBinary (uint16_t baseAddress, uint16_t valueToConvert)
- uint16_t RTC_B_convertBinaryToBCD (uint16_t baseAddress, uint16_t valueToConvert)
- void RTC_B_definePrescaleEvent (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
- void RTC_B_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- void RTC_B_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- Calendar RTC_B_getCalendarTime (uint16_t baseAddress)
- uint8_t RTC_B_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
- uint8_t RTC_B_getPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect)
- void RTC_B_holdClock (uint16_t baseAddress)
- void RTC_B_initCalendar (uint16_t baseAddress, Calendar ∗CalendarTime, uint16_t formatSelect)
- void RTC_B_setCalendarAlarm (uint16_t baseAddress, uint8_t minutesAlarm, uint8_t hoursAlarm, uint8_t dayOfWeekAlarm, uint8_t dayOfMonthAlarm)
- void RTC_B_setCalendarEvent (uint16_t baseAddress, uint16_t eventSelect)
- void RTC_B_setCalibrationData (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
- void RTC_B_setCalibrationFrequency (uint16_t baseAddress, uint16_t frequencySelect)
- void RTC_B_setPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
- void RTC_B_startClock (uint16_t baseAddress)

26.2.1 Detailed Description

The RTC_B API is broken into 5 groups of functions: clock settings, calender mode, prescale counter, interrupt condition setup/enable functions and data conversion.

The RTC_B clock settings are handled by

- RTC_B_startClock()
- RTC_B_holdClock()

- RTC_B_setCalibrationFrequency()
- RTC_B_setCalibrationData()

The RTC_B calender mode is initialized and handled by

- RTC_B_initCalendar()
- RTC_B_configureCalendarAlarm()
- RTC_B_getCalendarTime()

The RTC_B prescale counter is handled by

- RTC_B_getPrescaleValue()
- RTC_B_setPrescaleValue()

The RTC_B interrupts are handled by

- RTC_B_definePrescaleEvent()
- RTC_B_setCalendarEvent()
- RTC_B_enableInterrupt()
- RTC_B_disableInterrupt()
- RTC_B_getInterruptStatus()
- RTC_B_clearInterrupt()

The RTC_B conversions are handled by

- RTC_B_convertBCDToBinary()
- RTC_B_convertBinaryToBCD()

26.2.2 Function Documentation

26.2.2.1 void RTC_B_calendarInit (uint16_t *baseAddress*, Calendar *CalendarTime*, uint16_t *formatSelect*)

Deprecated - Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Parameters:

baseAddress is the base address of the RTC_B module.

CalendarTime is the structure containing the values for the Calendar to be initialized to. Valid values should be of type Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.

formatSelect is the format for the Calendar registers to use. Valid values are:

- **RTC_B_FORMAT_BINARY** [Default]
- **RTC_B_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:

None

26.2.2.2 void RTC_B_clearInterrupt (uint16_t *baseAddress*, uint8_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_B module.

interruptFlagMask is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:

None

26.2.2.3 void RTC_B_configureCalendarAlarm (uint16_t *baseAddress*, RTC_B_configureCalendarAlarmParam ∗ *param*)

Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_B_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Parameters:

baseAddress is the base address of the RTC_B module.

param is the pointer to struct for calendar alarm configuration.

Returns:

None

26.2.2.4 uint16_t RTC_B_convertBCDToBinary (uint16_t *baseAddress*, uint16_t *valueToConvert*)

Convert the given BCD value to binary format.

This function converts BCD values to binary format. This API uses the hardware registers to perform the conversion rather than a software method.

Parameters:

baseAddress is the base address of the RTC_B module.

valueToConvert is the raw value in BCD format to convert to Binary.
Modified bits are **BCD2BIN** of **BCD2BIN** register.

Returns:

The binary version of the input parameter

26.2.2.5 uint16_t RTC_B_convertBinaryToBCD (uint16_t *baseAddress*, uint16_t *valueToConvert*)

Convert the given binary value to BCD format.

This function converts binary values to BCD format. This API uses the hardware registers to perform the conversion rather than a software method.

Parameters:

baseAddress is the base address of the RTC_B module.

valueToConvert is the raw value in Binary format to convert to BCD.
Modified bits are **BIN2BCD** of **BIN2BCD** register.

Returns:

The BCD version of the valueToConvert parameter

26.2.2.6 void RTC_B_definePrescaleEvent (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleEventDivider*)

Sets up an interrupt condition for the selected Prescaler.

This function sets the condition for an interrupt to assert based on the individual prescalers.

Parameters:

baseAddress is the base address of the RTC_B module.

prescaleSelect is the prescaler to define an interrupt for. Valid values are:

- **RTC_B_PRESCALE_0**
- **RTC_B_PRESCALE_1**

prescaleEventDivider is a divider to specify when an interrupt can occur based on the clock source of the selected prescaler. (Does not affect timer of the selected prescaler). Valid values are:

- **RTC_B_PSEVENTDIVIDER_2** [Default]
- **RTC_B_PSEVENTDIVIDER_4**
- **RTC_B_PSEVENTDIVIDER_8**
- **RTC_B_PSEVENTDIVIDER_16**
- **RTC_B_PSEVENTDIVIDER_32**
- **RTC_B_PSEVENTDIVIDER_64**
- **RTC_B_PSEVENTDIVIDER_128**
- **RTC_B_PSEVENTDIVIDER_256**
Modified bits are **RTxIP** of **RTCPSxCTL** register.

Returns:

None

26.2.2.7 void RTC_B_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_B module.

interruptMask is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.

- **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:
None

26.2.2.8 void RTC_B_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_B module.
interruptMask is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:
None

26.2.2.9 Calendar RTC_B_getCalendarTime (uint16_t *baseAddress*)

Returns the Calendar Time stored in the Calendar registers of the RTC.

This function returns the current Calendar time in the form of a Calendar structure. The RTCRDY polling is used in this function to prevent reading invalid time.

Parameters:
baseAddress is the base address of the RTC_B module.

Returns:
A Calendar structure containing the current time.

26.2.2.10 uint8_t RTC_B_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_B module.
interruptFlagMask is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.

- **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:
Logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_B_CLOCK_ALARM_INTERRUPT** asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** asserts when Calendar registers are settled.
- **RTC_B_PRESCALE_TIMER0_INTERRUPT** asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** asserts if there is a problem with the 32kHz oscillator, while the RTC is running.
indicating the status of the masked interrupts

26.2.2.11 uint8_t RTC_B_getPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Returns the selected prescaler value.

This function returns the value of the selected prescale counter register. Note that the counter value should be held by calling RTC_B_holdClock() before calling this API.

Parameters:
baseAddress is the base address of the RTC_B module.
prescaleSelect is the prescaler to obtain the value of. Valid values are:

- **RTC_B_PRESCALE_0**
- **RTC_B_PRESCALE_1**

Returns:
The value of the specified prescaler count register

26.2.2.12 void RTC_B_holdClock (uint16_t *baseAddress*)

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Parameters:
baseAddress is the base address of the RTC_B module.

Returns:
None

26.2.2.13 void RTC_B_initCalendar (uint16_t *baseAddress*, Calendar $*$ *CalendarTime*, uint16_t *formatSelect*)

Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Parameters:
baseAddress is the base address of the RTC_B module.

CalendarTime is the pointer to the structure containing the values for the Calendar to be initialized to. Valid values should be of type pointer to Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.

formatSelect is the format for the Calendar registers to use. Valid values are:

- **RTC_B_FORMAT_BINARY** [Default]
- **RTC_B_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:
None

26.2.2.14 void RTC_B_setCalendarAlarm (uint16_t *baseAddress*, uint8_t *minutesAlarm*, uint8_t *hoursAlarm*, uint8_t *dayOfWeekAlarm*, uint8_t *dayOfMonthAlarm*)

DEPRECATED - Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_B_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Parameters:

baseAddress is the base address of the RTC_B module.

minutesAlarm is the alarm condition for the minutes. Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-59

hoursAlarm is the alarm condition for the hours. Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-24

dayOfWeekAlarm is the alarm condition for the day of week. Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-6

dayOfMonthAlarm is the alarm condition for the day of the month. Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-31

Returns:
None

26.2.2.15 void RTC_B_setCalendarEvent (uint16_t *baseAddress*, uint16_t *eventSelect*)

Sets a single specified Calendar interrupt condition.

This function sets a specified event to assert the RTCTEVIFG interrupt. This interrupt is independent from the Calendar alarm interrupt.

Parameters:

baseAddress is the base address of the RTC_B module.

eventSelect is the condition selected. Valid values are:

- **RTC_B_CALENDAREVENT_MINUTECHANGE** - assert interrupt on every minute
- **RTC_B_CALENDAREVENT_HOURCHANGE** - assert interrupt on every hour
- **RTC_B_CALENDAREVENT_NOON** - assert interrupt when hour is 12
- **RTC_B_CALENDAREVENT_MIDNIGHT** - assert interrupt when hour is 0
  Modified bits are **RTCTEV** of **RTCCTL** register.

Returns:
None

26.2.2.16 void RTC_B_setCalibrationData (uint16_t *baseAddress*, uint8_t *offsetDirection*, uint8_t *offsetValue*)

Sets the specified calibration for the RTC.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +4-ppm or -2-ppm, and the offsetValue should be from 1-63 and is multiplied by the direction setting (i.e. +4-ppm $*$ 8 (offsetValue) = +32-ppm). Please note, when measuring the frequency after setting the calibration, you will only see a change on the 1Hz frequency.

Parameters:

baseAddress is the base address of the RTC_B module.

offsetDirection is the direction that the calibration offset will go. Valid values are:

- **RTC_B_CALIBRATION_DOWN2PPM** - calibrate at steps of -2
- **RTC_B_CALIBRATION_UP4PPM** - calibrate at steps of +4
  Modified bits are **RTCCALS** of **RTCCTL2** register.

offsetValue is the value that the offset will be a factor of; a valid value is any integer from 1-63.
Modified bits are **RTCCAL** of **RTCCTL2** register.

Returns:

None

26.2.2.17 void RTC_B_setCalibrationFrequency (uint16_t *baseAddress*, uint16_t *frequencySelect*)

Allows and Sets the frequency output to RTCCLK pin for calibration measurement.

This function sets a frequency to measure at the RTCCLK output pin. After testing the set frequency, the calibration could be set accordingly.

Parameters:

baseAddress is the base address of the RTC_B module.

frequencySelect is the frequency output to RTCCLK. Valid values are:

- **RTC_B_CALIBRATIONFREQ_OFF** [Default] - turn off calibration output
- **RTC_B_CALIBRATIONFREQ_512HZ** - output signal at 512Hz for calibration
- **RTC_B_CALIBRATIONFREQ_256HZ** - output signal at 256Hz for calibration
- **RTC_B_CALIBRATIONFREQ_1HZ** - output signal at 1Hz for calibration
  Modified bits are **RTCCALF** of **RTCCTL3** register.

Returns:

None

26.2.2.18 void RTC_B_setPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleCounterValue*)

Sets the selected prescaler value.

This function sets the prescale counter value. Before setting the prescale counter, it should be held by calling RTC_B_holdClock().

Parameters:

baseAddress is the base address of the RTC_B module.

prescaleSelect is the prescaler to set the value for. Valid values are:

- **RTC_B_PRESCALE_0**
- **RTC_B_PRESCALE_1**

prescaleCounterValue is the specified value to set the prescaler to. Valid values are any integer between 0-255
Modified bits are **RTxPS** of **RTxPS** register.

Returns:

None

26.2.2.19 void RTC_B_startClock (uint16_t *baseAddress*)

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Parameters:
baseAddress is the base address of the RTC_B module.

Returns:
None

26.3 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 calendar struct
    Calendar currentTime;
    currentTime.Seconds    = 0x00;
    currentTime.Minutes    = 0x26;
    currentTime.Hours      = 0x13;
    currentTime.DayOfWeek  = 0x03;
    currentTime.DayOfMonth = 0x20;
    currentTime.Month      = 0x07;
    currentTime.Year       = 0x2011;

    //Initialize alarm struct
    RTC_B_configureCalendarAlarmParam alarmParam;
    alarmParam.minutesAlarm = 0x00;
    alarmParam.hoursAlarm = 0x17;
    alarmParam.dayOfWeekAlarm = RTC_B_ALARMCONDITION_OFF;
    alarmParam.dayOfMonthAlarm = 0x05;

    //Initialize Calendar Mode of RTC_B
    /*
 Base Address of the RTC_B
 Pass in current time, initialized above
 Use BCD as Calendar Register Format
     */
    RTC_B_initCalendar(RTC_B_BASE,
        &currentTime,
        RTC_B_FORMAT_BCD);

    //Setup Calendar Alarm for 5:00pm on the 5th day of the month.
    //Note: Does not specify day of the week.
    RTC_B_setCalendarAlarm(RTC_B_BASE, &alarmParam);

    //Specify an interrupt to assert every minute
    RTC_B_setCalendarEvent(RTC_B_BASE,
        RTC_B_CALENDAREVENT_MINUTECHANGE);

    //Enable interrupt for RTC_B Ready Status, which asserts when the RTC_B
    //Calendar registers are ready to read.
    //Also, enable interrupts for the Calendar alarm and Calendar event.
    RTC_B_enableInterrupt(RTC_B_BASE,
        RTC_B_CLOCK_READ_READY_INTERRUPT +
        RTC_B_TIME_EVENT_INTERRUPT +
        RTC_B_CLOCK_ALARM_INTERRUPT);

    //Start RTC_B Clock
    RTC_B_startClock(RTC_B_BASE);
```

```
//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

27 Real-Time Clock (RTC_C)

27.1 Introduction

The Real Time Clock (RTC_C) API provides a set of functions for using the MSP430Ware RTC_C modules. Functions are provided to calibrate the clock, initialize the RTC_C modules in Calendar mode, and setup conditions for, and enable, interrupts for the RTC_C modules.

The RTC_C module provides the ability to keep track of the current time and date in calendar mode. The counter mode (device-dependent) provides a 32-bit counter.

The RTC_C module generates multiple interrupts. There are 2 interrupts that can be defined in calendar mode, and 1 interrupt in counter mode for counter overflow, as well as an interrupt for each prescaler.

If the device header file defines the baseaddress as RTC_C_BASE, pass in RTC_C_BASE as the baseaddress parameter.If the device header file defines the baseaddress as RTC_CE_BASE, pass in RTC_CE_BASE as the baseaddress parameter.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1670
TI Compiler 4.2.1	Size	944
TI Compiler 4.2.1	Speed	940
IAR 5.51.6	None	1390
IAR 5.51.6	Size	1050
IAR 5.51.6	Speed	1118
MSPGCC 4.8.0	None	2764
MSPGCC 4.8.0	Size	1090
MSPGCC 4.8.0	Speed	1174

27.2 API Functions

Functions

- void RTC_C_calendarInit (uint16_t baseAddress, Calendar CalendarTime, uint16_t formatSelect)
- void RTC_C_clearInterrupt (uint16_t baseAddress, uint8_t interruptFlagMask)
- void RTC_C_configureCalendarAlarm (uint16_t baseAddress, RTC_C_configureCalendarAlarmParam ∗param)
- uint16_t RTC_C_convertBCDToBinary (uint16_t baseAddress, uint16_t valueToConvert)
- uint16_t RTC_C_convertBinaryToBCD (uint16_t baseAddress, uint16_t valueToConvert)
- void RTC_C_definePrescaleEvent (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
- void RTC_C_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- void RTC_C_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
- Calendar RTC_C_getCalendarTime (uint16_t baseAddress)
- uint32_t RTC_C_getCounterValue (uint16_t baseAddress)

- uint8_t RTC_C_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
- uint8_t RTC_C_getPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect)
- void RTC_C_holdClock (uint16_t baseAddress)
- void RTC_C_holdCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect)
- void RTC_C_initCalendar (uint16_t baseAddress, Calendar ∗CalendarTime, uint16_t formatSelect)
- void RTC_C_initCounter (uint16_t baseAddress, uint16_t clockSelect, uint16_t counterSizeSelect)
- void RTC_C_initCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect, uint16_t prescaleClockSelect, uint16_t prescaleDivider)
- void RTC_C_setCalendarAlarm (uint16_t baseAddress, uint8_t minutesAlarm, uint8_t hoursAlarm, uint8_t dayOfWeekAlarm, uint8_t dayOfMonthAlarm)
- void RTC_C_setCalendarEvent (uint16_t baseAddress, uint16_t eventSelect)
- void RTC_C_setCalibrationData (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
- void RTC_C_setCalibrationFrequency (uint16_t baseAddress, uint16_t frequencySelect)
- void RTC_C_setCounterValue (uint16_t baseAddress, uint32_t counterValue)
- void RTC_C_setPrescaleValue (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
- bool RTC_C_setTemperatureCompensation (uint16_t baseAddress, uint16_t offsetDirection, uint8_t offsetValue)
- void RTC_C_startClock (uint16_t baseAddress)
- void RTC_C_startCounterPrescale (uint16_t baseAddress, uint8_t prescaleSelect)

27.2.1 Detailed Description

The RTC_C API is broken into 6 groups of functions: clock settings, calender mode, counter mode, prescale counter, interrupt condition setup/enable functions and data conversion.

The RTC_C clock settings are handled by

- RTC_C_startClock()
- RTC_C_holdClock()
- RTC_C_setCalibrationFrequency()
- RTC_C_setCalibrationData()
- RTC_C_setTemperatureCompensation()

The RTC_C calender mode is initialized and setup by

- RTC_C_initCalendar()
- RTC_C_getCalenderTime()

The RTC_C counter mode is initialized and handled by

- RTC_C_initCounter()
- RTC_C_setCounterValue()
- RTC_C_getCounterValue()
- RTC_C_initCounterPrescale()
- RTC_C_holdCounterPrescale()
- RTC_C_startCounterPrescale()

The RTC_C prescale counter is handled by

- RTC_C_getPrescaleValue()
- RTC_C_setPrescaleValue()

The RTC_C interrupts are handled by

- RTC_C_configureCalendarAlarm()
- RTC_C_setCalenderEvent()
- RTC_C_definePrescaleEvent()

- RTC_C_enableInterrupt()
- RTC_C_disableInterrupt()
- RTC_C_getInterruptStatus()
- RTC_C_clearInterrupt()

The RTC_C data conversion is handled by

- RTC_C_convertBCDToBinary()
- RTC_C_convertBinaryToBCD()

27.2.2 Function Documentation

27.2.2.1 void RTC_C_calendarInit (uint16_t *baseAddress*, Calendar *CalendarTime*, uint16_t *formatSelect*)

Deprecated - Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	142
TI Compiler 4.2.1	Size	102
TI Compiler 4.2.1	Speed	102
IAR 5.51.6	None	132
IAR 5.51.6	Size	122
IAR 5.51.6	Speed	122
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	144
MSPGCC 4.8.0	Speed	144

Parameters:

baseAddress is the base address of the RTC_C module.

CalendarTime is the structure containing the values for the Calendar to be initialized to. Valid values should be of type Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.

formatSelect is the format for the Calendar registers to use. Valid values are:

- **RTC_C_FORMAT_BINARY** [Default]
- **RTC_C_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:

None

27.2.2.2 void RTC_C_clearInterrupt (uint16_t *baseAddress*, uint8_t *interruptFlagMask*)

Clears selected RTC interrupt flags.

This function clears the RTC interrupt flag is cleared, so that it no longer asserts.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	48
IAR 5.51.6	None	62
IAR 5.51.6	Size	50
IAR 5.51.6	Speed	56
MSPGCC 4.8.0	None	166
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	50

Parameters:

baseAddress is the base address of the RTC_C module.

interruptFlagMask is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:

- **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:

None

27.2.2.3 void RTC_C_configureCalendarAlarm (uint16_t *baseAddress*, RTC_C_configureCalendarAlarmParam $*$ *param*)

Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_C_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	48
IAR 5.51.6	None	72
IAR 5.51.6	Size	68
IAR 5.51.6	Speed	68
MSPGCC 4.8.0	None	116
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

Parameters:

baseAddress is the base address of the RTC_C module.

param is the pointer to struct for calendar alarm configuration.

Returns:

None

27.2.2.4 uint16_t RTC_C_convertBCDToBinary (uint16_t *baseAddress*, uint16_t *valueToConvert*)

Convert the given BCD value to binary format.

This function converts BCD values to binary format. This API uses the hardware registers to perform the conversion rather than a software method.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the RTC_C module.

valueToConvert is the raw value in BCD format to convert to Binary.
Modified bits are **BCD2BIN** of **BCD2BIN** register.

Returns:

The binary version of the input parameter

27.2.2.5 uint16_t RTC_C_convertBinaryToBCD (uint16_t *baseAddress*, uint16_t *valueToConvert*)

Convert the given binary value to BCD format.

This function converts binary values to BCD format. This API uses the hardware registers to perform the conversion rather than a software method.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the RTC_C module.

valueToConvert is the raw value in Binary format to convert to BCD.
Modified bits are **BIN2BCD** of **BIN2BCD** register.

Returns:

The BCD version of the valueToConvert parameter

27.2.2.6 void RTC_C_definePrescaleEvent (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleEventDivider*)

Sets up an interrupt condition for the selected Prescaler.

This function sets the condition for an interrupt to assert based on the individual prescalers.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	28
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

Parameters:

baseAddress is the base address of the RTC_C module.

prescaleSelect is the prescaler to define an interrupt for. Valid values are:

- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

prescaleEventDivider is a divider to specify when an interrupt can occur based on the clock source of the selected prescaler. (Does not affect timer of the selected prescaler). Valid values are:

- **RTC_C_PSEVENTDIVIDER_2** [Default]
- **RTC_C_PSEVENTDIVIDER_4**
- **RTC_C_PSEVENTDIVIDER_8**
- **RTC_C_PSEVENTDIVIDER_16**
- **RTC_C_PSEVENTDIVIDER_32**
- **RTC_C_PSEVENTDIVIDER_64**
- **RTC_C_PSEVENTDIVIDER_128**
- **RTC_C_PSEVENTDIVIDER_256**
  Modified bits are **RTxIP** of **RTCPSxCTL** register.

Returns:

None

27.2.2.7 void RTC_C_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	62
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	52
MSPGCC 4.8.0	None	170
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	50

Parameters:

baseAddress is the base address of the RTC_C module.

interruptMask is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following:

- **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:

None

27.2.2.8 void RTC_C_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	62
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	52
MSPGCC 4.8.0	None	158
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	50

Parameters:

baseAddress is the base address of the RTC_C module.

interruptMask is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following:

- **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:

None

27.2.2.9 Calendar RTC_C_getCalendarTime (uint16_t *baseAddress*)

Returns the Calendar Time stored in the Calendar registers of the RTC.

This function returns the current Calendar time in the form of a Calendar structure. The RTCRDY polling is used in this function to prevent reading invalid time.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	68
TI Compiler 4.2.1	Speed	68
IAR 5.51.6	None	108
IAR 5.51.6	Size	108
IAR 5.51.6	Speed	108
MSPGCC 4.8.0	None	150
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	72

Parameters:
baseAddress is the base address of the RTC_C module.

Returns:
A Calendar structure containing the current time.

27.2.2.10 uint32_t RTC_C_getCounterValue (uint16_t *baseAddress*)

Returns the value of the Counter register.

This function returns the value of the counter register for the RTC_C module. It will return the 32-bit value no matter the size set during initialization. The RTC should be held before trying to use this function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	68
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	62
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	122
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	52

Parameters:
baseAddress is the base address of the RTC_C module.

Returns:
The raw value of the full 32-bit Counter Register.

27.2.2.11 uint8_t RTC_C_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	86
TI Compiler 4.2.1	Size	40
TI Compiler 4.2.1	Speed	40
IAR 5.51.6	None	54
IAR 5.51.6	Size	44
IAR 5.51.6	Speed	44
MSPGCC 4.8.0	None	142
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	54

Parameters:

baseAddress is the base address of the RTC_C module.

interruptFlagMask is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following:

- **RTC_C_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns:

Logical OR of any of the following:

- **RTC_C_TIME_EVENT_INTERRUPT** asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_C_CLOCK_ALARM_INTERRUPT** asserts when alarm condition in Calendar mode is met.
- **RTC_C_CLOCK_READ_READY_INTERRUPT** asserts when Calendar registers are settled.
- **RTC_C_PRESCALE_TIMER0_INTERRUPT** asserts when Prescaler 0 event condition is met.
- **RTC_C_PRESCALE_TIMER1_INTERRUPT** asserts when Prescaler 1 event condition is met.
- **RTC_C_OSCILLATOR_FAULT_INTERRUPT** asserts if there is a problem with the 32kHz oscillator, while the RTC is running.
  indicating the status of the masked interrupts

27.2.2.12 uint8_t RTC_C_getPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Returns the selected prescaler value.

This function returns the value of the selected prescale counter register. Note that the counter value should be held by calling RTC_C_holdClock() before calling this API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	40
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	30
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	28

Parameters:
baseAddress is the base address of the RTC_C module.
prescaleSelect is the prescaler to obtain the value of. Valid values are:
- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

Returns:
The value of the specified prescaler count register

27.2.2.13 void RTC_C_holdClock (uint16_t *baseAddress*)

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	22

Parameters:
baseAddress is the base address of the RTC_C module.

Returns:
None

27.2.2.14 void RTC_C_holdCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Holds the selected Prescaler.

This function holds the prescale counter from continuing. This will only work in counter mode, in Calendar mode, the RTC_C_holdClock() must be used. In counter mode, if using both prescalers in conjunction with the main RTC counter, then stopping RT0PS will stop RT1PS, but stopping RT1PS will not stop RT0PS.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	16
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:
baseAddress is the base address of the RTC_C module.

prescaleSelect is the prescaler to hold. Valid values are:
- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

Returns:
None

27.2.2.15 void RTC_C_initCalendar (uint16_t *baseAddress*, Calendar ∗ *CalendarTime*, uint16_t *formatSelect*)

Initializes the settings to operate the RTC in calendar mode.

This function initializes the Calendar mode of the RTC module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	146
TI Compiler 4.2.1	Size	76
TI Compiler 4.2.1	Speed	76
IAR 5.51.6	None	130
IAR 5.51.6	Size	120
IAR 5.51.6	Speed	120
MSPGCC 4.8.0	None	250
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	76

Parameters:
baseAddress is the base address of the RTC_C module.
CalendarTime is the pointer to the structure containing the values for the Calendar to be initialized to. Valid values should be of type pointer to Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-24 **DayOfWeek** between 0-6 **DayOfMonth** between 0-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.
formatSelect is the format for the Calendar registers to use. Valid values are:
- **RTC_C_FORMAT_BINARY** [Default]
- **RTC_C_FORMAT_BCD**
  Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns:
None

27.2.2.16 void RTC_C_initCounter (uint16_t *baseAddress*, uint16_t *clockSelect*, uint16_t *counterSizeSelect*)

Initializes the settings to operate the RTC in Counter mode.

This function initializes the Counter mode of the RTC_C. Setting the clock source and counter size will allow an interrupt from the RTCTEVIFG once an overflow to the counter register occurs.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	68
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	48
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	122
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:

baseAddress is the base address of the RTC_C module.

clockSelect is the selected clock for the counter mode to use. Valid values are:

- **RTC_C_CLOCKSELECT_32KHZ_OSC**
- **RTC_C_CLOCKSELECT_RT1PS**
  Modified bits are **RTCSSEL** of **RTCCTL1** register.

counterSizeSelect is the size of the counter. Valid values are:

- **RTC_C_COUNTERSIZE_8BIT** [Default]
- **RTC_C_COUNTERSIZE_16BIT**
- **RTC_C_COUNTERSIZE_24BIT**
- **RTC_C_COUNTERSIZE_32BIT**
  Modified bits are **RTCTEV** of **RTCCTL1** register.

Returns:

None

27.2.2.17 void RTC_C_initCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint16_t *prescaleClockSelect*, uint16_t *prescaleDivider*)

Initializes the Prescaler for Counter mode.

This function initializes the selected prescaler for the counter mode in the RTC_C module. If the RTC is initialized in Calendar mode, then these are automatically initialized. The Prescalers can be used to divide a clock source additionally before it gets to the main RTC clock.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	22
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	42
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

baseAddress is the base address of the RTC_C module.

prescaleSelect is the prescaler to initialize. Valid values are:

- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

prescaleClockSelect is the clock to drive the selected prescaler. Valid values are:

- **RTC_C_PSCLOCKSELECT_ACLK**

- **RTC_C_PSCLOCKSELECT_SMCLK**
- **RTC_C_PSCLOCKSELECT_RT0PS** - use Prescaler 0 as source to Prescaler 1 (May only be used if prescaleSelect is RTC_C_PRESCALE_1)
  Modified bits are **RTxSSEL** of **RTCPSxCTL** register.

prescaleDivider is the divider for the selected clock source. Valid values are:

- **RTC_C_PSDIVIDER_2** [Default]
- **RTC_C_PSDIVIDER_4**
- **RTC_C_PSDIVIDER_8**
- **RTC_C_PSDIVIDER_16**
- **RTC_C_PSDIVIDER_32**
- **RTC_C_PSDIVIDER_64**
- **RTC_C_PSDIVIDER_128**
- **RTC_C_PSDIVIDER_256**
  Modified bits are **RTxPSDIV** of **RTCPSxCTL** register.

Returns:
None

27.2.2.18 void RTC_C_setCalendarAlarm (uint16_t *baseAddress*, uint8_t *minutesAlarm*, uint8_t *hoursAlarm*, uint8_t *dayOfWeekAlarm*, uint8_t *dayOfMonthAlarm*)

DEPRECATED - Sets and Enables the desired Calendar Alarm settings.

This function sets a Calendar interrupt condition to assert the RTCAIFG interrupt flag. The condition is a logical and of all of the parameters. For example if the minutes and hours alarm is set, then the interrupt will only assert when the minutes AND the hours change to the specified setting. Use the RTC_C_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	60
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	38

Parameters:

baseAddress is the base address of the RTC_C module.

minutesAlarm is the alarm condition for the minutes. Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-59

hoursAlarm is the alarm condition for the hours. Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-24

dayOfWeekAlarm is the alarm condition for the day of week. Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-6

dayOfMonthAlarm is the alarm condition for the day of the month. Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]
- **An** integer between 0-31

Returns:
None

27.2.2.19 void RTC_C_setCalendarEvent (uint16_t *baseAddress*, uint16_t *eventSelect*)

Sets a single specified Calendar interrupt condition.

This function sets a specified event to assert the RTCTEVIFG interrupt. This interrupt is independent from the Calendar alarm interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	38
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	92
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:

baseAddress is the base address of the RTC_C module.

eventSelect is the condition selected. Valid values are:

- **RTC_C_CALENDAREVENT_MINUTECHANGE** - assert interrupt on every minute
- **RTC_C_CALENDAREVENT_HOURCHANGE** - assert interrupt on every hour
- **RTC_C_CALENDAREVENT_NOON** - assert interrupt when hour is 12
- **RTC_C_CALENDAREVENT_MIDNIGHT** - assert interrupt when hour is 0
  Modified bits are **RTCTEV** of **RTCCTL** register.

Returns:

None

27.2.2.20 void RTC_C_setCalibrationData (uint16_t *baseAddress*, uint8_t *offsetDirection*, uint8_t *offsetValue*)

Sets the specified calibration for the RTC.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +4-ppm or -2-ppm, and the offsetValue should be from 1-63 and is multiplied by the direction setting (i.e. +4-ppm $*$ 8 (offsetValue) = +32-ppm).

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	38
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the RTC_C module.

offsetDirection is the direction that the calibration offset will go. Valid values are:

- **RTC_C_CALIBRATION_DOWN1PPM** - calibrate at steps of -1

- **RTC_C_CALIBRATION_UP1PPM** - calibrate at steps of +1
  Modified bits are **RTC0CALS** of **RTC0CAL** register.

offsetValue is the value that the offset will be a factor of; a valid value is any integer from 1-240.
Modified bits are **RTC0CALx** of **RTC0CAL** register.

Returns:
None

27.2.2.21 void RTC_C_setCalibrationFrequency (uint16_t *baseAddress*, uint16_t *frequencySelect*)

Allows and Sets the frequency output to RTCCLK pin for calibration measurement.

This function sets a frequency to measure at the RTCCLK output pin. After testing the set frequency, the calibration could be set accordingly.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	38
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	78
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:
baseAddress is the base address of the RTC_C module.
frequencySelect is the frequency output to RTCCLK. Valid values are:

- **RTC_C_CALIBRATIONFREQ_OFF** [Default] - turn off calibration output
- **RTC_C_CALIBRATIONFREQ_512HZ** - output signal at 512Hz for calibration
- **RTC_C_CALIBRATIONFREQ_256HZ** - output signal at 256Hz for calibration
- **RTC_C_CALIBRATIONFREQ_1HZ** - output signal at 1Hz for calibration
  Modified bits are **RTCCALF** of **RTCCTL3** register.

Returns:
None

27.2.2.22 void RTC_C_setCounterValue (uint16_t *baseAddress*, uint32_t *counterValue*)

Sets the value of the Counter register.

This function sets the counter register of the RTC_C module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	136
TI Compiler 4.2.1	Size	82
TI Compiler 4.2.1	Speed	84
IAR 5.51.6	None	106
IAR 5.51.6	Size	82
IAR 5.51.6	Speed	82
MSPGCC 4.8.0	None	192
MSPGCC 4.8.0	Size	106
MSPGCC 4.8.0	Speed	178

Parameters:

baseAddress is the base address of the RTC_C module.

counterValue is the value to set the Counter register to; a valid value may be any 32-bit integer.

Returns:

None

27.2.2.23 void RTC_C_setPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*, uint8_t *prescaleCounterValue*)

Sets the selected Prescaler value.

This function sets the prescale counter value. Before setting the prescale counter, it should be held by calling RTC_C_holdClock().

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	50
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	80
MSPGCC 4.8.0	Size	44
MSPGCC 4.8.0	Speed	52

Parameters:

baseAddress is the base address of the RTC_C module.

prescaleSelect is the prescaler to set the value for. Valid values are:

- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

prescaleCounterValue is the specified value to set the prescaler to. Valid values are any integer between 0-255 Modified bits are **RTxPS** of **RTxPS** register.

Returns:

None

27.2.2.24 bool RTC_C_setTemperatureCompensation (uint16_t *baseAddress*, uint16_t *offsetDirection*, uint8_t *offsetValue*)

Sets the specified temperature compensation for the RTC.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +1-ppm or -1-ppm, and the offsetValue should be from 1-240 and is multiplied by the direction setting (i.e. +1-ppm $*$ 8 (offsetValue) = +8-ppm).

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	34
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	56
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	90
MSPGCC 4.8.0	Size	34
MSPGCC 4.8.0	Speed	34

Parameters:

baseAddress is the base address of the RTC_C module.

offsetDirection is the direction that the calibration offset wil go Valid values are:

- **RTC_C_COMPENSATION_DOWN1PPM**
- **RTC_C_COMPENSATION_UP1PPM**
  Modified bits are **RTCTCMPS** of **RTCTCMP** register.

offsetValue is the value that the offset will be a factor of; a valid value is any integer from 1-240. Modified bits are **RTCTCMPx** of **RTCTCMP** register.

Returns:

STATUS_SUCCESS or STATUS_FAILURE of setting the temperature compensation

27.2.2.25 void RTC_C_startClock (uint16_t *baseAddress*)

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	22
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	22

Parameters:

baseAddress is the base address of the RTC_C module.

Returns:

None

27.2.2.26 void RTC_C_startCounterPrescale (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Starts the selected Prescaler.

This function starts the selected prescale counter. This function will only work if the RTC is in counter mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	16
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the RTC_C module.

prescaleSelect is the prescaler to start. Valid values are:

- **RTC_C_PRESCALE_0**
- **RTC_C_PRESCALE_1**

Returns:

None

27.3 Programming Example

The following example shows how to initialize and use the RTC_C API to setup Calender Mode with the current time and various interrupts.

```
    //Initialize calendar struct
    Calendar currentTime;
    currentTime.Seconds    = 0x00;
    currentTime.Minutes    = 0x26;
    currentTime.Hours      = 0x13;
    currentTime.DayOfWeek  = 0x03;
    currentTime.DayOfMonth = 0x20;
    currentTime.Month      = 0x07;
    currentTime.Year       = 0x2011;

    //Initialize alarm struct
    RTC_C_configureCalendarAlarmParam alarmParam;
    alarmParam.minutesAlarm = 0x00;
    alarmParam.hoursAlarm = 0x17;
    alarmParam.dayOfWeekAlarm = RTC_C_ALARMCONDITION_OFF;
    alarmParam.dayOfMonthAlarm = 0x05;

    //Initialize Calendar Mode of RTC_C
    /*
 Base Address of the RTC_C_A
 Pass in current time, initialized above
 Use BCD as Calendar Register Format
     */
    RTC_C_initCalendar(RTC_C_BASE,
        &currentTime,
        RTC_C_FORMAT_BCD);

    //Setup Calendar Alarm for 5:00pm on the 5th day of the month.
    //Note: Does not specify day of the week.
    RTC_C_setCalendarAlarm(RTC_C_BASE, &alarmParam);

    //Specify an interrupt to assert every minute
    RTC_C_setCalendarEvent(RTC_C_BASE,
        RTC_C_CALENDAREVENT_MINUTECHANGE);
```

```
//Enable interrupt for RTC_C Ready Status, which asserts when the RTC_C
//Calendar registers are ready to read.
//Also, enable interrupts for the Calendar alarm and Calendar event.
RTC_C_enableInterrupt(RTC_C_BASE,
    RTC_C_CLOCK_READ_READY_INTERRUPT +
    RTC_C_TIME_EVENT_INTERRUPT +
    RTC_C_CLOCK_ALARM_INTERRUPT);

//Start RTC_C Clock
RTC_C_startClock(RTC_C_BASE);

//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

28 24-Bit Sigma Delta Converter (SD24_B)

28.1 Introduction

The SD24_B module consists of up to eight independent sigma-delta analog-to-digital converters. The converters are based on second-order oversampling sigma-delta modulators and digital decimation filters. The decimation filters are comb type filters with selectable oversampling ratios of up to 1024. Additional filtering can be done in software.

A sigma-delta analog-to-digital converter basically consists of two parts: the analog part

- called modulator - and the digital part - a decimation filter. The modulator of the SD24_B provides a bit stream of zeros and ones to the digital decimation filter. The digital filter averages the bitstream from the modulator over a given number of bits (specified by the oversampling rate) and provides samples at a reduced rate for further processing to the CPU.

As commonly known averaging can be used to increase the signal-to-noise performance of a conversion. With a conventional ADC each factor-of-4 oversampling can improve the SNR by about 6 dB or 1 bit. To achieve a 16-bit resolution out of a simple 1-bit ADC would require an impractical oversampling rate of 415 = 1.073.741.824. To overcome this limitation the sigma-delta modulator implements a technique called noise-shaping - due to an implemented feedback-loop and integrators the quantization noise is pushed to higher frequencies and thus much lower oversampling rates are sufficient to achieve high resolutions.

This driver is contained in `sd24_b.c`, with `sd24_b.h` containing the API definitions for use by applications.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1392
TI Compiler 4.2.1	Size	716
TI Compiler 4.2.1	Speed	698
IAR 5.51.6	None	964
IAR 5.51.6	Size	666
IAR 5.51.6	Speed	714
MSPGCC 4.8.0	None	1982
MSPGCC 4.8.0	Size	808
MSPGCC 4.8.0	Speed	840

28.2 API Functions

Functions

- void SD24_B_clearInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
- void SD24_B_configureConverter (uint16_t baseAddress, uint8_t converter, uint8_t alignment, uint8_t startSelect, uint8_t conversionMode)
- void SD24_B_configureConverterAdvanced (uint16_t baseAddress, uint8_t converter, uint8_t alignment, uint8_t startSelect, uint8_t conversionMode, uint8_t dataFormat, uint8_t sampleDelay, uint16_t oversampleRatio, uint8_t gain)

- void SD24_B_configureDMATrigger (uint16_t baseAddress, uint16_t interruptFlag)
- void SD24_B_disableInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
- void SD24_B_enableInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
- uint16_t SD24_B_getHighWordResults (uint16_t baseAddress, uint8_t converter)
- uint16_t SD24_B_getInterruptStatus (uint16_t baseAddress, uint8_t converter, uint16_t mask)
- uint32_t SD24_B_getResults (uint16_t baseAddress, uint8_t converter)
- void SD24_B_init (uint16_t baseAddress, uint16_t clockSourceSelect, uint16_t clockPreDivider, uint16_t clockDivider, uint16_t referenceSelect)
- void SD24_B_initConverter (uint16_t baseAddress, SD24_B_initConverterParam ∗param)
- void SD24_B_initConverterAdvanced (uint16_t baseAddress, SD24_B_initConverterAdvancedParam ∗param)
- void SD24_B_initialize (uint16_t baseAddress, SD24_B_initializeParam ∗param)
- void SD24_B_setConverterDataFormat (uint16_t baseAddress, uint8_t converter, uint8_t dataFormat)
- void SD24_B_setGain (uint16_t baseAddress, uint8_t converter, uint8_t gain)
- void SD24_B_setInterruptDelay (uint16_t baseAddress, uint8_t converter, uint8_t sampleDelay)
- void SD24_B_setOversampling (uint16_t baseAddress, uint8_t converter, uint16_t oversampleRatio)
- void SD24_B_startConverterConversion (uint16_t baseAddress, uint8_t converter)
- void SD24_B_startGroupConversion (uint16_t baseAddress, uint8_t group)
- void SD24_B_stopConverterConversion (uint16_t baseAddress, uint8_t converter)
- void SD24_B_stopGroupConversion (uint16_t baseAddress, uint8_t group)

28.2.1 Detailed Description

The SD24_B 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 SD24_B.

The SD24_B initialization and conversion functions are

- SD24_B_init()
- SD24_B_configureConverter()
- SD24_B_configureConverterAdvanced()
- SD24_B_startGroupConversion()
- SD24_B_stopGroupConversion()
- SD24_B_stopConverterConversion()
- SD24_B_startConverterConversion()
- SD24_B_configureDMATrigger()
- SD24_B_getResults()
- SD24_B_getHighWordResults()

The SD24_B interrupts are handled by

- SD24_B_enableInterrupt()
- SD24_B_disableInterrupt()
- SD24_B_clearInterrupt()
- SD24_B_getInterruptStatus()

Auxiliary features of the SD24_B are handled by

- SD24_B_setConverterDataFormat()
- SD24_B_setInterruptDelay()
- SD24_B_setOversampling()
- SD24_B_setGain()

28.2.2 Function Documentation

28.2.2.1 void SD24_B_clearInterrupt (uint16_t *baseAddress*, uint8_t *converter*, uint16_t *mask*)

Clears interrupts for the SD24_B Module.

This function clears interrupt flags for the SD24_B module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	28
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	64
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the SD24_B module.

converter is the selected converter. Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

mask is the bit mask of the converter interrupt sources to clear. Mask value is the logical OR of any of the following:

- **SD24_B_CONVERTER_INTERRUPT**
- **SD24_B_CONVERTER_OVERFLOW_INTERRUPT**
  Modified bits are **SD24OVIFGx** of **SD24BIFG** register.

Returns:

None

28.2.2.2 void SD24_B_configureConverter (uint16_t *baseAddress*, uint8_t *converter*, uint8_t *alignment*, uint8_t *startSelect*, uint8_t *conversionMode*)

DEPRECATED - Configure SD24_B converter.

This function initializes a converter of the SD24_B module. Upon completion the converter will be ready for a conversion and can be started with the SD24_B_startGroupConversion() or SD24_B_startConverterConversion() depending on the startSelect parameter. Additional configuration such as data format can be configured in SD24_B_setConverterDataFormat().

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	32
TI Compiler 4.2.1	Speed	32
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	40

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be configured. Check check datasheet for available converters on device. Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

alignment selects how the data will be aligned in result Valid values are:

- **SD24_B_ALIGN_RIGHT** [Default]
- **SD24_B_ALIGN_LEFT**
  Modified bits are **SD24ALGN** of **SD24BCCTLx** register.

startSelect selects what will trigger the start of the converter Valid values are:

- **SD24_B_CONVERSION_SELECT_SD24SC** [Default]
- **SD24_B_CONVERSION_SELECT_EXT1**
- **SD24_B_CONVERSION_SELECT_EXT2**
- **SD24_B_CONVERSION_SELECT_EXT3**
- **SD24_B_CONVERSION_SELECT_GROUP0**
- **SD24_B_CONVERSION_SELECT_GROUP1**
- **SD24_B_CONVERSION_SELECT_GROUP2**
- **SD24_B_CONVERSION_SELECT_GROUP3**
  Modified bits are **SD24SCSx** of **SD24BCCTLx** register.

conversionMode determines whether the converter will do continuous samples or a single sample Valid values are:

- **SD24_B_CONTINUOUS_MODE** [Default]
- **SD24_B_SINGLE_MODE**
  Modified bits are **SD24SNGL** of **SD24BCCTLx** register.

Returns:

None

28.2.2.3 void SD24_B_configureConverterAdvanced (uint16_t *baseAddress*, uint8_t *converter*, uint8_t *alignment*, uint8_t *startSelect*, uint8_t *conversionMode*, uint8_t *dataFormat*, uint8_t *sampleDelay*, uint16_t *oversampleRatio*, uint8_t *gain*)

DEPRECATED - Configure SD24_B converter - Advanced Configure.

This function initializes a converter of the SD24_B module. Upon completion the converter will be ready for a conversion and can be started with the SD24_B_startGroupConversion() or SD24_B_startConverterConversion() depending on the startSelect parameter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	104
TI Compiler 4.2.1	Size	106
TI Compiler 4.2.1	Speed	106
IAR 5.51.6	None	98
IAR 5.51.6	Size	92
IAR 5.51.6	Speed	92
MSPGCC 4.8.0	None	108
MSPGCC 4.8.0	Size	96
MSPGCC 4.8.0	Speed	100

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be configured. Check check datasheet for available converters on device. Valid values are:
- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

alignment selects how the data will be aligned in result Valid values are:
- **SD24_B_ALIGN_RIGHT** [Default]
- **SD24_B_ALIGN_LEFT**
  Modified bits are **SD24ALGN** of **SD24BCCTLx** register.

startSelect selects what will trigger the start of the converter Valid values are:
- **SD24_B_CONVERSION_SELECT_SD24SC** [Default]
- **SD24_B_CONVERSION_SELECT_EXT1**
- **SD24_B_CONVERSION_SELECT_EXT2**
- **SD24_B_CONVERSION_SELECT_EXT3**
- **SD24_B_CONVERSION_SELECT_GROUP0**
- **SD24_B_CONVERSION_SELECT_GROUP1**
- **SD24_B_CONVERSION_SELECT_GROUP2**
- **SD24_B_CONVERSION_SELECT_GROUP3**
  Modified bits are **SD24SCSx** of **SD24BCCTLx** register.

conversionMode determines whether the converter will do continuous samples or a single sample Valid values are:
- **SD24_B_CONTINUOUS_MODE** [Default]
- **SD24_B_SINGLE_MODE**
  Modified bits are **SD24SNGL** of **SD24BCCTLx** register.

dataFormat selects how the data format of the results Valid values are:
- **SD24_B_DATA_FORMAT_BINARY** [Default]
- **SD24_B_DATA_FORMAT_2COMPLEMENT**
  Modified bits are **SD24DFx** of **SD24BCCTLx** register.

sampleDelay selects the delay for the interrupt Valid values are:
- **SD24_B_FOURTH_SAMPLE_INTERRUPT** [Default]
- **SD24_B_THIRD_SAMPLE_INTERRUPT**
- **SD24_B_SECOND_SAMPLE_INTERRUPT**
- **SD24_B_FIRST_SAMPLE_INTERRUPT**
  Modified bits are **SD24INTDLYx** of **SD24INCTLx** register.

oversampleRatio selects oversampling ratio for the converter Valid values are:
- **SD24_B_OVERSAMPLE_32**
- **SD24_B_OVERSAMPLE_64**
- **SD24_B_OVERSAMPLE_128**
- **SD24_B_OVERSAMPLE_256**
- **SD24_B_OVERSAMPLE_512**

- **SD24_B_OVERSAMPLE_1024**
  Modified bits are **SD24OSRx** of **SD24BOSRx** register.

gain selects the gain for the converter Valid values are:
- **SD24_B_GAIN_1** [Default]
- **SD24_B_GAIN_2**
- **SD24_B_GAIN_4**
- **SD24_B_GAIN_8**
- **SD24_B_GAIN_16**
- **SD24_B_GAIN_32**
- **SD24_B_GAIN_64**
- **SD24_B_GAIN_128**
  Modified bits are **SD24GAINx** of **SD24BINCTLx** register.

Returns:
None

28.2.2.4 void SD24_B_configureDMATrigger (uint16_t *baseAddress*, uint16_t *interruptFlag*)

Configures the converter that triggers a DMA transfer.

This function chooses which interrupt will trigger a DMA transfer.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:
baseAddress is the base address of the SD24_B module.
interruptFlag selects the converter interrupt that triggers a DMA transfer. Valid values are:
- **SD24_B_DMA_TRIGGER_IFG0**
- **SD24_B_DMA_TRIGGER_IFG1**
- **SD24_B_DMA_TRIGGER_IFG2**
- **SD24_B_DMA_TRIGGER_IFG3**
- **SD24_B_DMA_TRIGGER_IFG4**
- **SD24_B_DMA_TRIGGER_IFG5**
- **SD24_B_DMA_TRIGGER_IFG6**
- **SD24_B_DMA_TRIGGER_IFG7**
- **SD24_B_DMA_TRIGGER_TRGIFG**
  Modified bits are **SD24DMAx** of **SD24BCTL1** register.

Returns:
None

28.2.2.5 void SD24_B_disableInterrupt (uint16_t *baseAddress*, uint8_t *converter*, uint16_t *mask*)

Disables interrupts for the SD24_B Module.

This function disables interrupts for the SD24_B module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	28
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	64
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the SD24_B module.

converter is the selected converter. Valid values are:
- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

mask is the bit mask of the converter interrupt sources to be disabled. Mask value is the logical OR of any of the following:
- **SD24_B_CONVERTER_INTERRUPT**
- **SD24_B_CONVERTER_OVERFLOW_INTERRUPT**
  Modified bits are **SD24OVIEx** of **SD24BIE** register.

Modified bits of **SD24BIE** register.

Returns:

None

28.2.2.6 void SD24_B_enableInterrupt (uint16_t *baseAddress*, uint8_t *converter*, uint16_t *mask*)

Enables interrupts for the SD24_B Module.

This function enables interrupts for the SD24_B module. Does not clear interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	28
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	62
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the SD24_B module.

converter is the selected converter. Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

mask is the bit mask of the converter interrupt sources to be enabled. Mask value is the logical OR of any of the following:

- **SD24_B_CONVERTER_INTERRUPT**
- **SD24_B_CONVERTER_OVERFLOW_INTERRUPT**
  Modified bits are **SD24OVIEx** of **SD24BIE** register.

Returns:

None

28.2.2.7 uint16_t SD24_B_getHighWordResults (uint16_t *baseAddress*, uint8_t *converter*)

Returns the high word results for a converter.

This function gets the results from the SD24MEMHx register and returns it.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	18
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter who's results will be returned Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**

- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

Returns:
Result of conversion

28.2.2.8 uint16_t SD24_B_getInterruptStatus (uint16_t *baseAddress*, uint8_t *converter*, uint16_t *mask*)

Returns the interrupt status for the SD24_B Module.

This function returns interrupt flag statuses for the SD24_B module.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	38
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	20
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	22

Parameters:
baseAddress is the base address of the SD24_B module.
converter is the selected converter. Valid values are:
- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

mask is the bit mask of the converter interrupt sources to return. Mask value is the logical OR of any of the following:
- **SD24_B_CONVERTER_INTERRUPT**
- **SD24_B_CONVERTER_OVERFLOW_INTERRUPT**

Returns:
Logical OR of any of the following:

- **SD24_B_CONVERTER_INTERRUPT**
- **SD24_B_CONVERTER_OVERFLOW_INTERRUPT**
  indicating the status of the masked interrupts

28.2.2.9 uint32_t SD24_B_getResults (uint16_t *baseAddress*, uint8_t *converter*)

Returns the results for a converter.

This function gets the results from the SD24BMEMLx and SD24MEMHx registers and concatenates them to form a long. The actual result is a maximum 24 bits.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	62
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	30
IAR 5.51.6	Size	20
IAR 5.51.6	Speed	20
MSPGCC 4.8.0	None	110
MSPGCC 4.8.0	Size	42
MSPGCC 4.8.0	Speed	42

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter who's results will be returned Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

Returns:

Result of conversion

28.2.2.10 void SD24_B_init (uint16_t *baseAddress*, uint16_t *clockSourceSelect*, uint16_t *clockPreDivider*, uint16_t *clockDivider*, uint16_t *referenceSelect*)

DEPRECATED - Initializes the SD24_B Module.

This function initializes the SD24_B module sigma-delta analog-to-digital conversions. Specifically the function sets up the clock source for the SD24_B core to use for conversions. Upon completion of the initialization the SD24_B interrupt registers will be reset and the given parameters will be set. The converter configuration settings are independent of this function. The values you choose for the clock divider and predivider are used to determine the effective clock frequency. The formula used is: f_sd24 = f_clk /(divider $*$ predivider)

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	60

Parameters:

baseAddress is the base address of the SD24_B module.

clockSourceSelect selects the clock that will be used as the SD24_B core Valid values are:

- **SD24_B_CLOCKSOURCE_MCLK** [Default]
- **SD24_B_CLOCKSOURCE_SMCLK**

- **SD24_B_CLOCKSOURCE_ACLK**
- **SD24_B_CLOCKSOURCE_SD24CLK**
  Modified bits are **SD24SSEL** of **SD24BCTL0** register.

clockPreDivider selects the amount that the clock will be predivided Valid values are:

- **SD24_B_PRECLOCKDIVIDER_1** [Default]
- **SD24_B_PRECLOCKDIVIDER_2**
- **SD24_B_PRECLOCKDIVIDER_4**
- **SD24_B_PRECLOCKDIVIDER_8**
- **SD24_B_PRECLOCKDIVIDER_16**
- **SD24_B_PRECLOCKDIVIDER_32**
- **SD24_B_PRECLOCKDIVIDER_64**
- **SD24_B_PRECLOCKDIVIDER_128**
  Modified bits are **SD24PDIVx** of **SD24BCTL0** register.

clockDivider selects the amount that the clock will be divided. Valid values are:

- **SD24_B_CLOCKDIVIDER_1** [Default]
- **SD24_B_CLOCKDIVIDER_2**
- **SD24_B_CLOCKDIVIDER_3**
- **SD24_B_CLOCKDIVIDER_4**
- **SD24_B_CLOCKDIVIDER_5**
- **SD24_B_CLOCKDIVIDER_6**
- **SD24_B_CLOCKDIVIDER_7**
- **SD24_B_CLOCKDIVIDER_8**
- **SD24_B_CLOCKDIVIDER_9**
- **SD24_B_CLOCKDIVIDER_10**
- **SD24_B_CLOCKDIVIDER_11**
- **SD24_B_CLOCKDIVIDER_12**
- **SD24_B_CLOCKDIVIDER_13**
- **SD24_B_CLOCKDIVIDER_14**
- **SD24_B_CLOCKDIVIDER_15**
- **SD24_B_CLOCKDIVIDER_16**
- **SD24_B_CLOCKDIVIDER_17**
- **SD24_B_CLOCKDIVIDER_18**
- **SD24_B_CLOCKDIVIDER_19**
- **SD24_B_CLOCKDIVIDER_20**
- **SD24_B_CLOCKDIVIDER_21**
- **SD24_B_CLOCKDIVIDER_22**
- **SD24_B_CLOCKDIVIDER_23**
- **SD24_B_CLOCKDIVIDER_24**
- **SD24_B_CLOCKDIVIDER_25**
- **SD24_B_CLOCKDIVIDER_26**
- **SD24_B_CLOCKDIVIDER_27**
- **SD24_B_CLOCKDIVIDER_28**
- **SD24_B_CLOCKDIVIDER_29**
- **SD24_B_CLOCKDIVIDER_30**
- **SD24_B_CLOCKDIVIDER_31**
- **SD24_B_CLOCKDIVIDER_32**
  Modified bits are **SD24DIVx** of **SD24BCTL0** register.

referenceSelect selects the reference source for the SD24_B core Valid values are:

- **SD24_B_REF_EXTERNAL** [Default]
- **SD24_B_REF_INTERNAL**
  Modified bits are **SD24REFS** of **SD24BCTL0** register.

Returns:
None

28.2.2.11 void SD24_B_initConverter (uint16_t *baseAddress*, SD24_B_initConverterParam ∗ *param*)

Configure SD24_B converter.

This function initializes a converter of the SD24_B module. Upon completion the converter will be ready for a conversion and can be started with the SD24_B_startGroupConversion() or SD24_B_startConverterConversion() depending on the startSelect parameter. Additional configuration such as data format can be configured in SD24_B_setConverterDataFormat().

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	74
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	58
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	44
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

Parameters:

baseAddress is the base address of the SD24_B module.

param is the pointer to struct for converter configuration.

Returns:

None

28.2.2.12 void SD24_B_initConverterAdvanced (uint16_t *baseAddress*, SD24_B_initConverterAdvancedParam ∗ *param*)

Configure SD24_B converter - Advanced Configure.

This function initializes a converter of the SD24_B module. Upon completion the converter will be ready for a conversion and can be started with the SD24_B_startGroupConversion() or SD24_B_startConverterConversion() depending on the startSelect parameter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	168
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	96
IAR 5.51.6	None	134
IAR 5.51.6	Size	108
IAR 5.51.6	Speed	106
MSPGCC 4.8.0	None	254
MSPGCC 4.8.0	Size	108
MSPGCC 4.8.0	Speed	108

Parameters:

baseAddress is the base address of the SD24_B module.

param is the pointer to struct for converter advanced configuration.

Returns:

None

28.2.2.13 void SD24_B_initialize (uint16_t *baseAddress*, SD24_B_initializeParam ∗ *param*)

Initializes the SD24_B Module.

This function initializes the SD24_B module sigma-delta analog-to-digital conversions. Specifically the function sets up the clock source for the SD24_B core to use for conversions. Upon completion of the initialization the SD24_B interrupt registers will be reset and the given parameters will be set. The converter configuration settings are independent of this function. The values you choose for the clock divider and predivider are used to determine the effective clock frequency. The formula used is: f_sd24 = f_clk /(divider ∗ predivider)

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	56
TI Compiler 4.2.1	Speed	56
IAR 5.51.6	None	86
IAR 5.51.6	Size	74
IAR 5.51.6	Speed	74
MSPGCC 4.8.0	None	166
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	60

Parameters:
baseAddress is the base address of the SD24_B module.
param is the pointer to struct for initialization.

Returns:
None

28.2.2.14 void SD24_B_setConverterDataFormat (uint16_t *baseAddress*, uint8_t *converter*, uint8_t *dataFormat*)

Set SD24_B converter data format.

This function sets the converter format so that the resulting data can be viewed in either binary or 2's complement.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	28
IAR 5.51.6	Size	20
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:
baseAddress is the base address of the SD24_B module.
converter selects the converter that will be configured. Check check datasheet for available converters on device. Valid values are:
- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**

- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

dataFormat selects how the data format of the results Valid values are:

- **SD24_B_DATA_FORMAT_BINARY** [Default]
- **SD24_B_DATA_FORMAT_2COMPLEMENT**
  Modified bits are **SD24DFx** of **SD24BCCTLx** register.

Returns:
None

28.2.2.15 void SD24_B_setGain (uint16_t *baseAddress*, uint8_t *converter*, uint8_t *gain*)

Configures the gain for the converter.

This function configures the gain for a single converter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	30
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	80
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be configured Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

gain selects the gain for the converter Valid values are:

- **SD24_B_GAIN_1** [Default]
- **SD24_B_GAIN_2**
- **SD24_B_GAIN_4**
- **SD24_B_GAIN_8**
- **SD24_B_GAIN_16**
- **SD24_B_GAIN_32**
- **SD24_B_GAIN_64**
- **SD24_B_GAIN_128**
  Modified bits are **SD24GAINx** of **SD24BINCTLx** register.

Returns:
None

28.2.2.16 void SD24_B_setInterruptDelay (uint16_t *baseAddress*, uint8_t *converter*, uint8_t *sampleDelay*)

Configures the delay for an interrupt to trigger.

This function configures the delay for the first interrupt service request for the corresponding converter. This feature delays the interrupt request for a completed conversion by up to four conversion cycles allowing the digital filter to settle prior to generating an interrupt request.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	30
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	80
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be stopped Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

sampleDelay selects the delay for the interrupt Valid values are:

- **SD24_B_FOURTH_SAMPLE_INTERRUPT** [Default]
- **SD24_B_THIRD_SAMPLE_INTERRUPT**
- **SD24_B_SECOND_SAMPLE_INTERRUPT**
- **SD24_B_FIRST_SAMPLE_INTERRUPT**
  Modified bits are **SD24INTDLYx** of **SD24INCTLx** register.

Returns:

None

28.2.2.17 void SD24_B_setOversampling (uint16_t *baseAddress*, uint8_t *converter*, uint16_t *oversampleRatio*)

Configures the oversampling ratio for a converter.

This function configures the oversampling ratio for a given converter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	54
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	28
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	74
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be configured Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

oversampleRatio selects oversampling ratio for the converter Valid values are:

- **SD24_B_OVERSAMPLE_32**
- **SD24_B_OVERSAMPLE_64**
- **SD24_B_OVERSAMPLE_128**
- **SD24_B_OVERSAMPLE_256**
- **SD24_B_OVERSAMPLE_512**
- **SD24_B_OVERSAMPLE_1024**
  Modified bits are **SD24OSRx** of **SD24BOSRx** register.

Returns:

None

28.2.2.18 void SD24_B_startConverterConversion (uint16_t *baseAddress*, uint8_t *converter*)

Start Conversion for Converter.

This function starts a single converter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	52
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	26
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be started Valid values are:
- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**
  Modified bits are **SD24SC** of **SD24BCCTLx** register.

Returns:
None

28.2.2.19 void SD24_B_startGroupConversion (uint16_t *baseAddress*, uint8_t *group*)

Start Conversion Group.

This function starts all the converters that are associated with a group. To set a converter to a group use the SD24_B_configureConverter() function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	60
IAR 5.51.6	Size	40
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	144
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	42

Parameters:
baseAddress is the base address of the SD24_B module.
group selects the group that will be started Valid values are:
- **SD24_B_GROUP0**
- **SD24_B_GROUP1**
- **SD24_B_GROUP2**
- **SD24_B_GROUP3**
  Modified bits are **SD24DGRPxSC** of **SD24BCTL1** register.

Returns:
None

28.2.2.20 void SD24_B_stopConverterConversion (uint16_t *baseAddress*, uint8_t *converter*)

Stop Conversion for Converter.

This function stops a single converter.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	52
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	22
IAR 5.51.6	None	26
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the SD24_B module.

converter selects the converter that will be stopped Valid values are:

- **SD24_B_CONVERTER_0**
- **SD24_B_CONVERTER_1**
- **SD24_B_CONVERTER_2**
- **SD24_B_CONVERTER_3**
- **SD24_B_CONVERTER_4**
- **SD24_B_CONVERTER_5**
- **SD24_B_CONVERTER_6**
- **SD24_B_CONVERTER_7**

Modified bits are **SD24SC** of **SD24BCCTLx** register.

Returns:

None

28.2.2.21 void SD24_B_stopGroupConversion (uint16_t *baseAddress*, uint8_t *group*)

Stop Conversion Group.

This function stops all the converters that are associated with a group. To set a converter to a group use the SD24_B_configureConverter() function.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	42
TI Compiler 4.2.1	Speed	42
IAR 5.51.6	None	60
IAR 5.51.6	Size	40
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	144
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	42

Parameters:

baseAddress is the base address of the SD24_B module.

group selects the group that will be stopped Valid values are:

- **SD24_B_GROUP0**
- **SD24_B_GROUP1**
- **SD24_B_GROUP2**
- **SD24_B_GROUP3**

Modified bits are **SD24DGRPxSC** of **SD24BCTL1** register.

Returns:

None

28.3 Programming Example

The following example shows how to initialize and use the SD24_B API to start a single channel, single conversion.

```
     unsigned long results;

     SD24_B_init(SD24_BASE,
            SD24_B_CLOCKSOURCE_SMCLK,
            SD24_B_PRECLOCKDIVIDER_1,
            SD24_B_CLOCKDIVIDER_1,
            SD24_B_REF_INTERNAL);                     // Select internal REF
                                         // Select SMCLK as SD24_B clock source

     SD24_B_configureConverter(SD24_BASE,
                    SD24_B_CONVERTER_2,
                    SD24_B_ALIGN_RIGHT,
                    SD24_B_CONVERSION_SELECT_SD24SC,
                    SD24_B_SINGLE_MODE);

__delay_cycles(0x3600);                  // Delay for 1.5V REF startup

while (1)
{
    SD24_B_startConverterConversion(SD24_BASE,
            SD24_B_CONVERTER_2);                                                     // Set bi

    // Poll interrupt flag for channel 2
    while( SD24_B_getInterruptStatus(SD24_BASE,
                    SD24_B_CONVERTER_2
                    SD24_CONVERTER_INTERRUPT) == 0 );

    results = SD24_B_getResults(SD24_BASE,
            SD24_B_CONVERTER_2);                                 // Save CH2 results (clears IFG)

    __no_operation();                    // SET BREAKPOINT HERE
}
```

29 SFR Module

29.1 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.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	58
TI Compiler 4.2.1	Speed	60
IAR 5.51.6	None	56
IAR 5.51.6	Size	42
IAR 5.51.6	Speed	42
MSPGCC 4.8.0	None	316
MSPGCC 4.8.0	Size	68
MSPGCC 4.8.0	Speed	68

29.2 API Functions

Functions

- void SFR_clearInterrupt (uint8_t interruptFlagMask)
- void SFR_disableInterrupt (uint8_t interruptMask)
- void SFR_enableInterrupt (uint8_t interruptMask)
- uint8_t SFR_getInterruptStatus (uint8_t interruptFlagMask)
- void SFR_setNMIEdge (uint16_t edgeDirection)
- void SFR_setResetNMIPinFunction (uint8_t resetPinFunction)
- void SFR_setResetPinPullResistor (uint16_t pullResistorSetup)

29.2.1 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()

29.2.2 Function Documentation

29.2.2.1 void SFR_clearInterrupt (uint8_t *interruptFlagMask*)

Clears the selected SFR interrupt flags.

This function clears the status of the selected SFR interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

interruptFlagMask is the bit mask of interrupt flags that should be cleared Mask value is the logical OR of any of the following:
- **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt enable
- **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt enable
- **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt enable, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt enable
- **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt enable
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt enable
- **SFR_FLASH_CONTROLLER_ACCESS_VIOLATION_INTERRUPT** - Flash controller access violation interrupt enable

Returns:

None

29.2.2.2 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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 enable
- **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt enable
- **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt enable, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt enable
- **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt enable
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt enable
- **SFR_FLASH_CONTROLLER_ACCESS_VIOLATION_INTERRUPT** - Flash controller access violation interrupt enable

Returns:

None

29.2.2.3 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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 enable
- **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt enable
- **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt enable, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt enable
- **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt enable
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt enable
- **SFR_FLASH_CONTROLLER_ACCESS_VIOLATION_INTERRUPT** - Flash controller access violation interrupt enable

Returns:

None

29.2.2.4 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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 enable
- **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt enable
- **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt enable, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt enable
- **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt enable
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt enable
- **SFR_FLASH_CONTROLLER_ACCESS_VIOLATION_INTERRUPT** - Flash controller access violation interrupt enable

Returns:

Logical OR of any of the following:

- **SFR_JTAG_OUTBOX_INTERRUPT** JTAG outbox interrupt enable
- **SFR_JTAG_INBOX_INTERRUPT** JTAG inbox interrupt enable
- **SFR_NMI_PIN_INTERRUPT** NMI pin interrupt enable, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** Vacant memory access interrupt enable
- **SFR_OSCILLATOR_FAULT_INTERRUPT** Oscillator fault interrupt enable
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** Watchdog interval timer interrupt enable
- **SFR_FLASH_CONTROLLER_ACCESS_VIOLATION_INTERRUPT** Flash controller access violation interrupt enable
  indicating the status of the masked interrupts

29.2.2.5 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

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

29.2.2.6 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().

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

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

29.2.2.7 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

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** of **SFRRPCR** register.

Returns:

None

29.3 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));
```

30 SYS Module

30.1 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, and control the JTAG Mailbox.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	268
TI Compiler 4.2.1	Size	160
TI Compiler 4.2.1	Speed	162
IAR 5.51.6	None	178
IAR 5.51.6	Size	150
IAR 5.51.6	Speed	150
MSPGCC 4.8.0	None	618
MSPGCC 4.8.0	Size	196
MSPGCC 4.8.0	Speed	196

30.2 API Functions

Functions

- void SYS_clearJTAGMailboxFlagStatus (uint8_t mailboxFlagMask)
- void SYS_disableBSLMemory (void)
- void SYS_disableBSLProtect (void)
- void SYS_disableRAMBasedInterruptVectors (void)
- void SYS_enableBSLMemory (void)
- void SYS_enableBSLProtect (void)
- void SYS_enableDedicatedJTAGPins (void)
- void SYS_enablePMMAccessProtect (void)
- void SYS_enableRAMBasedInterruptVectors (void)
- uint8_t SYS_getBSLEntryIndication (void)
- uint16_t SYS_getJTAGInboxMessage16Bit (uint8_t inboxSelect)
- uint32_t SYS_getJTAGInboxMessage32Bit (void)
- uint8_t SYS_getJTAGMailboxFlagStatus (uint8_t mailboxFlagMask)
- void SYS_JTAGMailboxInit (uint8_t mailboxSizeSelect, uint8_t autoClearInboxFlagSelect)
- void SYS_setBSLSize (uint8_t BSLSizeSelect)
- void SYS_setJTAGOutgoingMessage16Bit (uint8_t outboxSelect, uint16_t outgoingMessage)
- void SYS_setJTAGOutgoingMessage32Bit (uint32_t outgoingMessage)
- void SYS_setRAMAssignedToBSL (uint8_t BSLRAMAssignment)

30.2.1 Detailed Description

The SYS API is broken into 3 groups: the various SYS controls, the BSL controls, and the JTAG mailbox controls.

The various SYS controls are handled by

- SYS_enableDedicatedJTAGPins()
- SYS_getBSLEntryIndication()
- SYS_enablePMMAccessProtect()
- SYS_enableRAMBasedInterruptVectors()
- SYS_disableRAMBasedInterruptVectors()

The BSL controls are handled by

- SYS_enableBSLProtect()
- SYS_disableBSLProtect()
- SYS_disableBSLMemory()
- SYS_enableBSLMemory()
- SYS_setRAMAssignedToBSL()
- SYS_setBSLSize()

The JTAG Mailbox controls are handled by

- SYS_JTAGMailboxInit()
- SYS_getJTAGMailboxFlagStatus()
- SYS_getJTAGInboxMessage16Bit()
- SYS_getJTAGInboxMessage32Bit()
- SYS_setJTAGOutgoingMessage16Bit()
- SYS_setJTAGOutgoingMessage32Bit()
- SYS_clearJTAGMailboxFlagStatus()

30.2.2 Function Documentation

30.2.2.1 void SYS_clearJTAGMailboxFlagStatus (uint8_t *mailboxFlagMask*)

Clears the status of the selected JTAG Mailbox flags.

This function clears the selected JTAG Mailbox flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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:

- **SYS_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0
- **SYS_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1
- **SYS_JTAGINBOX_FLAG0** - flag for JTAG inbox 0
- **SYS_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns:
None

30.2.2.2 void SYS_disableBSLMemory (void)

Disables BSL memory.

This function disables BSL memory, which makes BSL memory act like vacant memory.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

30.2.2.3 void SYS_disableBSLProtect (void)

Disables BSL memory protection.

This function disables protection on the BSL memory.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

30.2.2.4 void SYS_disableRAMBasedInterruptVectors (void)

Disables RAM-based Interrupt Vectors.

This function disables the interrupt vectors from being generated at the top of the RAM.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

30.2.2.5 void SYS_enableBSLMemory (void)

Enables BSL memory.

This function enables BSL memory, which allows BSL memory to be addressed

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

30.2.2.6 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

30.2.2.7 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

30.2.2.8 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

30.2.2.9 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

30.2.2.10 uint8_t SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	16
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Returns:
One of the following:

- **SYS_BSLENTRY_INDICATED**
- **SYS_BSLENTRY_NOTINDICATED**
  indicating if a BSL entry sequence was detected

30.2.2.11 uint16_t SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

inboxSelect is the chosen JTAG inbox that the contents of should be returned Valid values are:
- **SYS_JTAGINBOX_0** - return contents of JTAG inbox 0
- **SYS_JTAGINBOX_1** - return contents of JTAG inbox 1

Returns:

The contents of the selected JTAG inbox in a 16 bit format.

30.2.2.12 uint32_t SYS_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_JTAGMailboxInit() function. If the auto clear settings for the Inbox flags were set, then using this function will automatically clear both JTAG inbox flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	22
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Returns:

The contents of both JTAG messages in a 32 bit format.

30.2.2.13 uint8_t SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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:

- **SYS_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0
- **SYS_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1
- **SYS_JTAGINBOX_FLAG0** - flag for JTAG inbox 0
- **SYS_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns:

A bit mask of the status of the selected mailbox flags.

30.2.2.14 void SYS_JTAGMailboxInit (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	16
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	74
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:

mailboxSizeSelect is the size of the JTAG Mailboxes, whether 16- or 32-bits. Valid values are:

- **SYS_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)
- **SYS_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.

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:

- **SYS_JTAGINBOX0AUTO_JTAGINBOX1AUTO** [Default] - both JTAG inbox flags will be reset automatically when the corresponding inbox is read from.

- **SYS_JTAGINBOX0AUTO_JTAGINBOX1SW** - only JTAG inbox 0 flag is reset automatically, while JTAG inbox 1 is reset with the
- **SYS_JTAGINBOX0SW_JTAGINBOX1AUTO** - only JTAG inbox 1 flag is reset automatically, while JTAG inbox 0 is reset with the
- **SYS_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

30.2.2.15 void SYS_setBSLSize (uint8_t *BSLSizeSelect*)

Sets the size of the BSL in Flash.

This function sets the size of the BSL in Flash memory.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

Parameters:
BSLSizeSelect is the amount of segments the BSL should take. Valid values are:
- **SYS_BSLSIZE_SEG3**
- **SYS_BSLSIZE_SEGS23**
- **SYS_BSLSIZE_SEGS123**
- **SYS_BSLSIZE_SEGS1234** [Default]
Modified bits are **SYSBSLSIZE** of **SYSBSLC** register.

Returns:
None

30.2.2.16 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

outboxSelect is the chosen JTAG outbox that the message should be set it. Valid values are:

- **SYS_JTAGOUTBOX_0** - set the contents of JTAG outbox 0
- **SYS_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

30.2.2.17 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	54
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:

outgoingMessage is the message to send to the JTAG.
Modified bits are **MSGHI** and **MSGLO** of **SYSJMBOx** register.

Returns:

None

30.2.2.18 void SYS_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

Parameters:

BSLRAMAssignment is the selection of if the BSL should be placed in RAM or not. Valid values are:

- **SYS_BSLRAMASSIGN_NORAM** [Default]
- **SYS_BSLRAMASSIGN_LOWEST16BYTES**
  Modified bits are **SYSBSLR** of **SYSBSLC** register.

Returns:

None

30.3 Programming Example

The following example shows how to initialize and use the SYS API

```
SYS_enableBSLProtect(SYS_BASE);
```

31 TEC

31.1 Introduction

Timer Event Control (TEC) module is the interface between Timer modules and the external events. This chapter describes the TEC Module.

TEC is a module that connects different Timer modules to each other and routes the external signals to the Timer modules. TEC contains the control registers to configure the routing between the Timer modules, and it also has the enable register bits and the interrupt enable and interrupt flags for external event inputs. TEC features include:

- Enabling of internal and external clear signals
- Routing of internal signals (between Timer_D instances) and external clear signals
- Support of external fault input signals
- Interrupt vector generation of external fault and clear signals.
- Generating feedback signals to the Timer capture/compare channels to affect the timer outputs

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

31.2 API Functions

Functions

- void TEC_clearExternalClearStatus (uint16_t baseAddress)
- void TEC_clearExternalFaultStatus (uint16_t baseAddress, uint8_t mask)
- void TEC_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void TEC_configureExternalFaultInput (uint16_t baseAddress, uint8_t selectedExternalFault, uint16_t signalType, uint8_t signalHold, uint8_t polarityBit)
- void TEC_disableAuxiliaryClearSignal (uint16_t baseAddress)
- void TEC_disableExternalClearInput (uint16_t baseAddress)
- void TEC_disableExternalFaultInput (uint16_t baseAddress, uint8_t channelEventBlock)
- void TEC_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void TEC_enableAuxiliaryClearSignal (uint16_t baseAddress)
- void TEC_enableExternalClearInput (uint16_t baseAddress)
- void TEC_enableExternalFaultInput (uint16_t baseAddress, uint8_t channelEventBlock)
- void TEC_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t TEC_getExternalClearStatus (uint16_t baseAddress)
- uint8_t TEC_getExternalFaultStatus (uint16_t baseAddress, uint8_t mask)
- uint8_t TEC_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- void TEC_initExternalClearInput (uint16_t baseAddress, uint8_t signalType, uint8_t signalHold, uint8_t polarityBit)
- void TEC_initExternalFaultInput (uint16_t baseAddress, TEC_initExternalFaultInputParam ∗param)

31.2.1 Detailed Description

The tec configuration is handled by

- TEC_configureExternalClearInput()

- TEC_configureExternalFaultInput()
- TEC_enableExternalFaultInput()
- TEC_disableExternalFaultInput()
- TEC_enableExternalClearInput()
- TEC_disableExternalClearInput()
- TEC_enableAuxiliaryClearSignal()
- TEC_disableAuxiliaryClearSignal()

The interrupt and status operations are handled by

- TEC_enableExternalFaultInput()
- TEC_disableExternalFaultInput()
- TEC_clearInterruptFlag()
- TEC_getInterruptStatus()
- TEC_enableInterrupt()
- TEC_disableInterrupt()
- TEC_getExternalFaultStatus()
- TEC_clearExternalFaultStatus()
- TEC_getExternalClearStatus()
- TEC_clearExternalClearStatus()

31.2.2 Function Documentation

31.2.2.1 void TEC_clearExternalClearStatus (uint16_t *baseAddress*)

Clears the Timer Event Control External Clear Status.

Parameters:
baseAddress is the base address of the TEC module.

Modified bits of **TECxINT** register.

Returns:
None

31.2.2.2 void TEC_clearExternalFaultStatus (uint16_t *baseAddress*, uint8_t *mask*)

Clears the Timer Event Control External Fault Status.

Parameters:
baseAddress is the base address of the TEC module.
mask is the masked status flag be cleared Mask value is the logical OR of any of the following:

- **TEC_CE0**
- **TEC_CE1**
- **TEC_CE2**
- **TEC_CE3** - (available on TEC5 TEC7)
- **TEC_CE4** - (available on TEC5 TEC7)
- **TEC_CE5** - (only available on TEC7)
- **TEC_CE6** - (only available on TEC7)

Modified bits of **TECxINT** register.

Returns:
None

31.2.2.3 void TEC_clearInterruptFlag (uint16_t *baseAddress*, uint8_t *mask*)

Clears the Timer Event Control Interrupt flag.

Parameters:
baseAddress is the base address of the TEC module.
mask is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following:
- **TEC_EXTERNAL_FAULT_INTERRUPT** - External fault interrupt flag
- **TEC_EXTERNAL_CLEAR_INTERRUPT** - External clear interrupt flag
- **TEC_AUXILIARY_CLEAR_INTERRUPT** - Auxiliary clear interrupt flag

Modified bits of **TECxINT** register.

Returns:
None

31.2.2.4 void TEC_configureExternalFaultInput (uint16_t *baseAddress*, uint8_t *selectedExternalFault*, uint16_t *signalType*, uint8_t *signalHold*, uint8_t *polarityBit*)

DEPRECATED - Configures the Timer Event Control External Fault Input.

Parameters:
baseAddress is the base address of the TEC module.
selectedExternalFault is the selected external fault Valid values are:
- **TEC_EXTERNAL_FAULT_0**
- **TEC_EXTERNAL_FAULT_1**
- **TEC_EXTERNAL_FAULT_2**
- **TEC_EXTERNAL_FAULT_3**
- **TEC_EXTERNAL_FAULT_4**
- **TEC_EXTERNAL_FAULT_5**
- **TEC_EXTERNAL_FAULT_6**

signalType is the selected signal type Valid values are:
- **TEC_EXTERNAL_FAULT_SIGNALTYPE_EDGE_SENSITIVE** [Default]
- **TEC_EXTERNAL_FAULT_SIGNALTYPE_LEVEL_SENSITIVE**

signalHold is the selected signal hold Valid values are:
- **TEC_EXTERNAL_FAULT_SIGNAL_NOT_HELD** [Default]
- **TEC_EXTERNAL_FAULT_SIGNAL_HELD**

polarityBit is the selected signal type Valid values are:
- **TEC_EXTERNAL_FAULT_POLARITY_FALLING_EDGE_OR_LOW_LEVEL** [Default]
- **TEC_EXTERNAL_FAULT_POLARITY_RISING_EDGE_OR_HIGH_LEVEL**

Modified bits of **TECxCTL2** register.

Returns:
None

31.2.2.5 void TEC_disableAuxiliaryClearSignal (uint16_t *baseAddress*)

Disable the Timer Event Control Auxiliary Clear Signal.

Parameters:
baseAddress is the base address of the TEC module.

Modified bits of **TECxCTL2** register.

Returns:
None

31.2.2.6 void TEC_disableExternalClearInput (uint16_t *baseAddress*)

Disable the Timer Event Control External Clear Input.

Parameters:
baseAddress is the base address of the TEC module.

Modified bits of **TECxCTL2** register.

Returns:
None

31.2.2.7 void TEC_disableExternalFaultInput (uint16_t *baseAddress*, uint8_t *channelEventBlock*)

Disable the Timer Event Control External fault input.

Parameters:
baseAddress is the base address of the TEC module.
channelEventBlock selects the channel event block Valid values are:
- **TEC_CE0**
- **TEC_CE1**
- **TEC_CE2**
- **TEC_CE3** - (available on TEC5 TEC7)
- **TEC_CE4** - (available on TEC5 TEC7)
- **TEC_CE5** - (only available on TEC7)
- **TEC_CE6** - (only available on TEC7)

Modified bits of **TECxCTL0** register.

Returns:
None

31.2.2.8 void TEC_disableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Disables individual Timer Event Control interrupt sources.

Disables the indicated Timer Event Control 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 TEC module.
mask is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following:
- **TEC_EXTERNAL_FAULT_INTERRUPT** - External fault interrupt flag
- **TEC_EXTERNAL_CLEAR_INTERRUPT** - External clear interrupt flag
- **TEC_AUXILIARY_CLEAR_INTERRUPT** - Auxiliary clear interrupt flag

Modified bits of **TECxINT** register.

Returns:
None

31.2.2.9 void TEC_enableAuxiliaryClearSignal (uint16_t *baseAddress*)

Enable the Timer Event Control Auxiliary Clear Signal.

Parameters:
baseAddress is the base address of the TEC module.

Modified bits of **TECxCTL2** register.

Returns:
None

31.2.2.10 void TEC_enableExternalClearInput (uint16_t *baseAddress*)

Enable the Timer Event Control External Clear Input.

Parameters:
baseAddress is the base address of the TEC module.

Modified bits of **TECxCTL2** register.

Returns:
None

31.2.2.11 void TEC_enableExternalFaultInput (uint16_t *baseAddress*, uint8_t *channelEventBlock*)

Enable the Timer Event Control External fault input.

Parameters:
baseAddress is the base address of the TEC module.
channelEventBlock selects the channel event block Valid values are:

- **TEC_CE0**
- **TEC_CE1**
- **TEC_CE2**
- **TEC_CE3** - (available on TEC5 TEC7)
- **TEC_CE4** - (available on TEC5 TEC7)
- **TEC_CE5** - (only available on TEC7)
- **TEC_CE6** - (only available on TEC7)

Modified bits of **TECxCTL0** register.

Returns:
None

31.2.2.12 void TEC_enableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Enables individual Timer Event Control interrupt sources.

Enables the indicated Timer Event Control 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 TEC module.
mask is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following:

- **TEC_EXTERNAL_FAULT_INTERRUPT** - External fault interrupt flag
- **TEC_EXTERNAL_CLEAR_INTERRUPT** - External clear interrupt flag
- **TEC_AUXILIARY_CLEAR_INTERRUPT** - Auxiliary clear interrupt flag

Modified bits of **TECxINT** register.

Returns:
None

31.2.2.13 uint8_t TEC_getExternalClearStatus (uint16_t *baseAddress*)

Gets the current Timer Event Control External Clear Status.

Parameters:
baseAddress is the base address of the TEC module.

Returns:
One of the following:

- **TEC_EXTERNAL_CLEAR_DETECTED**
- **TEC_EXTERNAL_CLEAR_NOT_DETECTED**
indicating the status of the external clear

31.2.2.14 uint8_t TEC_getExternalFaultStatus (uint16_t *baseAddress*, uint8_t *mask*)

Gets the current Timer Event Control External Fault Status.

This returns the Timer Event Control fault status for the module.

Parameters:
baseAddress is the base address of the TEC module.
mask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:
- **TEC_CE0**
- **TEC_CE1**
- **TEC_CE2**
- **TEC_CE3** - (available on TEC5 TEC7)
- **TEC_CE4** - (available on TEC5 TEC7)
- **TEC_CE5** - (only available on TEC7)
- **TEC_CE6** - (only available on TEC7)

Returns:
Logical OR of any of the following:

- **TEC_CE0**
- **TEC_CE1**
- **TEC_CE2**
- **TEC_CE3** (available on TEC5 TEC7)
- **TEC_CE4** (available on TEC5 TEC7)
- **TEC_CE5** (only available on TEC7)
- **TEC_CE6** (only available on TEC7)
indicating the external fault status of the masked channel event blocks

31.2.2.15 uint8_t TEC_getInterruptStatus (uint16_t *baseAddress*, uint8_t *mask*)

Gets the current Timer Event Control interrupt status.

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

Parameters:

baseAddress is the base address of the TEC module.

mask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:

- **TEC_EXTERNAL_FAULT_INTERRUPT** - External fault interrupt flag
- **TEC_EXTERNAL_CLEAR_INTERRUPT** - External clear interrupt flag
- **TEC_AUXILIARY_CLEAR_INTERRUPT** - Auxiliary clear interrupt flag

Returns:

Logical OR of any of the following:

- **TEC_EXTERNAL_FAULT_INTERRUPT** External fault interrupt flag
- **TEC_EXTERNAL_CLEAR_INTERRUPT** External clear interrupt flag
- **TEC_AUXILIARY_CLEAR_INTERRUPT** Auxiliary clear interrupt flag indicating the status of the masked interrupts

31.2.2.16 void TEC_initExternalClearInput (uint16_t *baseAddress*, uint8_t *signalType*, uint8_t *signalHold*, uint8_t *polarityBit*)

Configures the Timer Event Control External Clear Input.

Parameters:

baseAddress is the base address of the TEC module.

signalType is the selected signal type Valid values are:

- **TEC_EXTERNAL_CLEAR_SIGNALTYPE_EDGE_SENSITIVE** [Default]
- **TEC_EXTERNAL_CLEAR_SIGNALTYPE_LEVEL_SENSITIVE**

signalHold is the selected signal hold Valid values are:

- **TEC_EXTERNAL_CLEAR_SIGNAL_NOT_HELD** [Default]
- **TEC_EXTERNAL_CLEAR_SIGNAL_HELD**

polarityBit is the selected signal type Valid values are:

- **TEC_EXTERNAL_CLEAR_POLARITY_FALLING_EDGE_OR_LOW_LEVEL** [Default]
- **TEC_EXTERNAL_CLEAR_POLARITY_RISING_EDGE_OR_HIGH_LEVEL**

Modified bits of **TECxCTL2** register.

Returns:

None

31.2.2.17 void TEC_initExternalFaultInput (uint16_t *baseAddress*, TEC_initExternalFaultInputParam $*$ *param*)

Configures the Timer Event Control External Fault Input.

Parameters:

baseAddress is the base address of the TEC module.

param is the pointer to struct for external fault input initialization.

Modified bits of **TECxCTL2** register.

Returns:

None

31.3 Programming Example

The following example shows how to use the TEC API.

```
{
  TIMER_D_startCounter(TIMER_D1_BASE,
        TIMERD_UP_MODE);

  // Configure TD1 TEC External Clear
  // Need to physically connect P2.0/TD0.2 to P2.7/TEC1CLR
  GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P2,
    GPIO_PIN7
    );

  // High Level trigger, ext clear enable
  TEC_configureExternalClearInput(TEC1_BASE,
                                                                  TEC_EXTERNAL_CLEAR_SIGNALTYPE_LEVEL_SENSI
                                                                  TEC_EXTERNAL_CLEAR_SIGNAL_NOT_HELD,
                                                                  TEC_EXTERNAL_CLEAR_POLARITY_RISING_EDGE_(
                                                                  );
  TEC_enableExternalClearInput(TEC1_BASE);

}
```

32 16-Bit Timer_A (TIMER_A)

32.1 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.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1992
TI Compiler 4.2.1	Size	1072
TI Compiler 4.2.1	Speed	1074
IAR 5.51.6	None	1500
IAR 5.51.6	Size	660
IAR 5.51.6	Speed	1094
MSPGCC 4.8.0	None	2716
MSPGCC 4.8.0	Size	1312
MSPGCC 4.8.0	Speed	1380

32.2 API Functions

Functions

- void TIMER_A_clear (uint16_t baseAddress)
- void TIMER_A_clearCaptureCompareInterruptFlag (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_A_clearTimerInterruptFlag (uint16_t baseAddress)
- void TIMER_A_configureContinuousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TAIE, uint16_t timerClear)
- void TIMER_A_configureUpDownMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TAIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_A_configureUpMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TAIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_A_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_A_disableInterrupt (uint16_t baseAddress)
- void TIMER_A_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_A_enableInterrupt (uint16_t baseAddress)
- void TIMER_A_generatePWM (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t compareRegister, uint16_t compareOutputMode, uint16_t dutyCycle)
- uint16_t TIMER_A_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint32_t TIMER_A_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
- uint16_t TIMER_A_getCounterValue (uint16_t baseAddress)
- uint32_t TIMER_A_getInterruptStatus (uint16_t baseAddress)
- uint8_t TIMER_A_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint8_t TIMER_A_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
- void TIMER_A_initCapture (uint16_t baseAddress, uint16_t captureRegister, uint16_t captureMode, uint16_t captureInputSelect, uint16_t synchronizeCaptureSource, uint16_t captureInterruptEnable, uint16_t captureOutputMode)
- void TIMER_A_initCaptureMode (uint16_t baseAddress, TIMER_A_initCaptureModeParam ∗param)
- void TIMER_A_initCompare (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareInterruptEnable, uint16_t compareOutputMode, uint16_t compareValue)
- void TIMER_A_initCompareMode (uint16_t baseAddress, TIMER_A_initCompareModeParam ∗param)
- void TIMER_A_initContinuousMode (uint16_t baseAddress, TIMER_A_initContinuousModeParam ∗param)
- void TIMER_A_initUpDownMode (uint16_t baseAddress, TIMER_A_initUpDownModeParam ∗param)
- void TIMER_A_initUpMode (uint16_t baseAddress, TIMER_A_initUpModeParam ∗param)
- void TIMER_A_outputPWM (uint16_t baseAddress, TIMER_A_outputPWMParam ∗param)
- void TIMER_A_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
- void TIMER_A_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
- void TIMER_A_startContinousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TAIE, uint16_t timerClear)

- void TIMER_A_startContinuousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TAIE, uint16_t timerClear)
- void TIMER_A_startCounter (uint16_t baseAddress, uint16_t timerMode)
- void TIMER_A_startUpDownMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TAIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_A_startUpMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TAIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_A_stop (uint16_t baseAddress)

32.2.1 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_configureContinuousMode()
- TIMER_A_configureUpMode()
- TIMER_A_configureUpDownMode()
- TIMER_A_startContinuousMode()
- TIMER_A_startUpMode()
- TIMER_A_startUpDownMode()
- TIMER_A_initCapture()
- TIMER_A_initCompare()
- TIMER_A_clear()
- TIMER_A_stop()

TIMER_A outputs are handled by

- TIMER_A_getSynchronizedCaptureCompareInput()
- TIMER_A_getOutputForOutputModeOutBitValue()
- TIMER_A_setOutputForOutputModeOutBitValue()
- TIMER_A_generatePWM()
- 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_clearCaptureCompareInterruptFlag()
- TIMER_A_clearTimerInterruptFlag()

32.2.2 Function Documentation

32.2.2.1 void TIMER_A_clear (uint16_t *baseAddress*)

Reset/Clear the timer clock divider, count direction, count.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.2 void TIMER_A_clearCaptureCompareInterruptFlag (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits are **CCIFG** of **TAxCCTLn** register.

Returns:
None

32.2.2.3 void TIMER_A_clearTimerInterruptFlag (uint16_t *baseAddress*)

Clears the Timer TAIFG interrupt flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the TIMER_A module.

Modified bits are **TAIFG** of **TAxCTL** register.

Returns:
None

32.2.2.4 void TIMER_A_configureContinuousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *timerClear*)

DEPRECATED - Configures TIMER_A in continuous mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_A_startCounter API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	68
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	52

Parameters:
baseAddress is the base address of the TIMER_A module.
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**

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**

timerInterruptEnable_TAIE is to enable or disable TIMER_A interrupt Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

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]

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.5 void TIMER_A_configureUpDownMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Configures TIMER_A in up down mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_A_startCounter API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	88

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerPeriod is the specified TIMER_A period

timerInterruptEnable_TAIE is to enable or disable TIMER_A interrupt Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns:

None

32.2.2.6 void TIMER_A_configureUpMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Configures TIMER_A in up mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_A_startCounter API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	88

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerPeriod is the specified TIMER_A period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

timerInterruptEnable_TAIE is to enable or disable TIMER_A interrupt Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns:

None

32.2.2.7 void TIMER_A_disableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Disable capture compare interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TAxCCTLn** register.

Returns:
None

32.2.2.8 void TIMER_A_disableInterrupt (uint16_t *baseAddress*)

Disable timer interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.9 void TIMER_A_enableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Enable capture compare interrupt.

Does not clear interrupt flags

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TAxCCTLn** register.

Returns:

None

32.2.2.10 void TIMER_A_enableInterrupt (uint16_t *baseAddress*)

Enable timer interrupt.

Does not clear interrupt flags

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.11 void TIMER_A_generatePWM (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *compareRegister*, uint16_t *compareOutputMode*, uint16_t *dutyCycle*)

DEPRECATED - Generate a PWM with timer running in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	80
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	68
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	76

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerPeriod selects the desired timer period

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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**

dutyCycle specifies the dutycycle for the generated waveform

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register, bits of **TAxCCR0** register and bits of **TAxCCTLn** register.

Returns:
None

32.2.2.12 uint16_t TIMER_A_getCaptureCompareCount (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get current capturecompare count.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Returns:
Current count as an uint16_t

32.2.2.13 uint32_t TIMER_A_getCaptureCompareInterruptStatus (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *mask*)

Return capture compare interrupt status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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

32.2.2.14 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	58
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	102
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	48

Parameters:
baseAddress is the base address of the TIMER_A module.

Returns:
Majority vote of timer count value

32.2.2.15 uint32_t TIMER_A_getInterruptStatus (uint16_t *baseAddress*)

Get timer interrupt status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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

32.2.2.16 uint8_t TIMER_A_getOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get output bit for output mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	20
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Returns:
One of the following:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH**
- **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

32.2.2.17 uint8_t TIMER_A_getSynchronizedCaptureCompareInput (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *synchronized*)

Get synchronized capturecompare input.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

synchronized Valid values are:
- **TIMER_A_READ_SYNCHRONIZED_CAPTURECOMPAREINPUT**
- **TIMER_A_READ_CAPTURE_COMPARE_INPUT**

Returns:
One of the following:

- **TIMER_A_CAPTURECOMPARE_INPUT_HIGH**
- **TIMER_A_CAPTURECOMPARE_INPUT_LOW**

32.2.2.18 void TIMER_A_initCapture (uint16_t *baseAddress*, uint16_t *captureRegister*, uint16_t *captureMode*, uint16_t *captureInputSelect*, uint16_t *synchronizeCaptureSource*, uint16_t *captureInterruptEnable*, uint16_t *captureOutputMode*)

DEPRECATED - Initializes Capture Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	80
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	36

Parameters:

baseAddress is the base address of the TIMER_A module.

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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**

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**

synchronizeCaptureSource decides if capture source should be synchronized with timer clock Valid values are:

- **TIMER_A_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_A_CAPTURE_SYNCHRONOUS**

captureInterruptEnable is to enable or disable timer captureComapre interrupt. Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

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**

Modified bits of **TAxCCTLn** register.

Returns:

None

32.2.2.19 void TIMER_A_initCaptureMode (uint16_t *baseAddress*, TIMER_A_initCaptureModeParam ∗ *param*)

Initializes Capture Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	70
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	50
IAR 5.51.6	Size	48
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	128
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

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

32.2.2.20 void TIMER_A_initCompare (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareInterruptEnable*, uint16_t *compareOutputMode*, uint16_t *compareValue*)

DEPRECATED - Initializes Compare Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	28

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**

- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

compareInterruptEnable is to enable or disable timer captureComapre interrupt. Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

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**

compareValue is the count to be compared with in compare mode

Modified bits of **TAxCCRn** register and bits of **TAxCCTLn** register.

Returns:
None

32.2.2.21 void TIMER_A_initCompareMode (uint16_t *baseAddress*, TIMER_A_initCompareModeParam $*$ *param*)

Initializes Compare Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	72
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	52
IAR 5.51.6	Size	50
IAR 5.51.6	Speed	50
MSPGCC 4.8.0	None	126
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

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

32.2.2.22 void TIMER_A_initContinuousMode (uint16_t *baseAddress*, TIMER_A_initContinuousModeParam $*$ *param*)

Configures TIMER_A in continuous mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	70
TI Compiler 4.2.1	Speed	70
IAR 5.51.6	None	78
IAR 5.51.6	Size	70
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	170
MSPGCC 4.8.0	Size	82
MSPGCC 4.8.0	Speed	82

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

32.2.2.23 void TIMER_A_initUpDownMode (uint16_t *baseAddress*, TIMER_A_initUpDownModeParam ∗ *param*)

Configures TIMER_A in up down mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	152
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	98
IAR 5.51.6	None	118
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	72
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	116
MSPGCC 4.8.0	Speed	118

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

32.2.2.24 void TIMER_A_initUpMode (uint16_t *baseAddress*, TIMER_A_initUpModeParam ∗ *param*)

Configures TIMER_A in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	152
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	98
IAR 5.51.6	None	120
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	74
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	116
MSPGCC 4.8.0	Speed	118

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

32.2.2.25 void TIMER_A_outputPWM (uint16_t *baseAddress*, TIMER_A_outputPWMParam ∗ *param*)

Generate a PWM with timer running in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	136
TI Compiler 4.2.1	Size	86
TI Compiler 4.2.1	Speed	86
IAR 5.51.6	None	108
IAR 5.51.6	Size	100
IAR 5.51.6	Speed	100
MSPGCC 4.8.0	None	228
MSPGCC 4.8.0	Size	100
MSPGCC 4.8.0	Speed	100

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

32.2.2.26 void TIMER_A_setCompareValue (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareValue*)

Sets the value of the capture-compare register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

compareValue is the count to be compared with in compare mode

Modified bits of **TAxCCRn** register.

Returns:

None

32.2.2.27 void TIMER_A_setOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint8_t *outputModeOutBitValue*)

Set output bit for output mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	14
MSPGCC 4.8.0	None	74
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

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**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**

- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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

32.2.2.28 void TIMER_A_startContinousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *timerClear*)

DEPRECATED - Spelling Error Fixed. Starts timer in continuous mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	52
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	34
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerInterruptEnable_TAIE is to enable or disable timer interrupt Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:

- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.29 void TIMER_A_startContinuousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *timerClear*)

DEPRECATED - Starts timer in continuous mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	68
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	64
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerInterruptEnable_TAIE is to enable or disable timer interrupt Valid values are:
- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:
- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Modified bits of **TAxCTL** register.

Returns:
None

32.2.2.30 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_configureContinuousMode, TIMER_A_configureUpMode or TIMER_A_configureUpDownMode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

32.2.2.31 void TIMER_A_startUpDownMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Replaced by TIMER_A_configureUpMode and TIMER_A_startCounter API. Starts timer in up down mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	94

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerPeriod is the specified timer period

timerInterruptEnable_TAIE is to enable or disable timer interrupt Valid values are:
- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

captureCompareInterruptEnable_CCR0_CCIE is to enable or disable timer CCR0 captureComapre interrupt. Valid values are:
- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**

- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:
- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns:
None

32.2.2.32 void TIMER_A_startUpMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TAIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Replaced by TIMER_A_configureUpMode and TIMER_A_startCounter API. Starts timer in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	62
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	94

Parameters:

baseAddress is the base address of the TIMER_A module.

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**

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**

timerPeriod is the specified timer period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

timerInterruptEnable_TAIE is to enable or disable timer interrupt Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

captureCompareInterruptEnable_CCR0_CCIE is to enable or disable timer CCR0 captureComapre interrupt. Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:

- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns:
None

32.2.2.33 void TIMER_A_stop (uint16_t *baseAddress*)

Stops the timer.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns:
None

32.3 Programming Example

The following example shows some TIMER_A operations using the APIs

```
{       //Start TIMER_A
    TIMER_A_configureUpDownMode( TIMER_A1_BASE,
        TIMER_A_CLOCKSOURCE_SMCLK,
        TIMER_A_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_A_TAIE_INTERRUPT_DISABLE,
        TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE,
        TIMER_A_DO_CLEAR
        );

    TIMER_A_startCounter( TIMER_A1_BASE,
                TIMER_A_UPDOWN_MODE
                );

    //Initialize compare registers to generate PWM1
    TIMER_A_initCompare(TIMER_A1_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_1,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_A_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE1
        );
    //Initialize compare registers to generate PWM2
    TIMER_A_initCompare(TIMER_A1_BASE,
        TIMER_A_CAPTURECOMPARE_REGISTER_2,
        TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TIMER_A_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE2
        );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

    //For debugger
    __no_operation();
}
```

33 16-Bit Timer_B (TIMER_B)

33.1 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.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	2094
TI Compiler 4.2.1	Size	1110
TI Compiler 4.2.1	Speed	1112
IAR 5.51.6	None	1544
IAR 5.51.6	Size	674
IAR 5.51.6	Speed	1110
MSPGCC 4.8.0	None	2884
MSPGCC 4.8.0	Size	1350
MSPGCC 4.8.0	Speed	1418

33.2 API Functions

Functions

- void TIMER_B_clear (uint16_t baseAddress)
- void TIMER_B_clearCaptureCompareInterruptFlag (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_B_clearTimerInterruptFlag (uint16_t baseAddress)
- void TIMER_B_configureContinuousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TBIE, uint16_t timerClear)
- void TIMER_B_configureUpDownMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TBIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_B_configureUpMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TBIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_B_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_B_disableInterrupt (uint16_t baseAddress)
- void TIMER_B_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_B_enableInterrupt (uint16_t baseAddress)
- void TIMER_B_generatePWM (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t compareRegister, uint16_t compareOutputMode, uint16_t dutyCycle)
- uint16_t TIMER_B_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint32_t TIMER_B_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
- uint16_t TIMER_B_getCounterValue (uint16_t baseAddress)
- uint32_t TIMER_B_getInterruptStatus (uint16_t baseAddress)
- uint8_t TIMER_B_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint8_t TIMER_B_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
- void TIMER_B_initCapture (uint16_t baseAddress, uint16_t captureRegister, uint16_t captureMode, uint16_t captureInputSelect, uint16_t synchronizeCaptureSource, uint16_t captureInterruptEnable, uint16_t captureOutputMode)
- void TIMER_B_initCaptureMode (uint16_t baseAddress, TIMER_B_initCaptureModeParam ∗param)
- void TIMER_B_initCompare (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareInterruptEnable, uint16_t compareOutputMode, uint16_t compareValue)
- void TIMER_B_initCompareLatchLoadEvent (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
- void TIMER_B_initCompareMode (uint16_t baseAddress, TIMER_B_initCompareModeParam ∗param)
- void TIMER_B_initContinuousMode (uint16_t baseAddress, TIMER_B_initContinuousModeParam ∗param)
- void TIMER_B_initUpDownMode (uint16_t baseAddress, TIMER_B_initUpDownModeParam ∗param)
- void TIMER_B_initUpMode (uint16_t baseAddress, TIMER_B_initUpModeParam ∗param)
- void TIMER_B_outputPWM (uint16_t baseAddress, TIMER_B_outputPWMParam ∗param)
- void TIMER_B_selectCounterLength (uint16_t baseAddress, uint16_t counterLength)
- void TIMER_B_selectLatchingGroup (uint16_t baseAddress, uint16_t groupLatch)
- void TIMER_B_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)

- void TIMER_B_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
- void TIMER_B_startContinousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TBIE, uint16_t timerClear)
- void TIMER_B_startContinuousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerInterruptEnable_TBIE, uint16_t timerClear)
- void TIMER_B_startCounter (uint16_t baseAddress, uint16_t timerMode)
- void TIMER_B_startUpDownMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TBIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_B_startUpMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t timerPeriod, uint16_t timerInterruptEnable_TBIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_B_stop (uint16_t baseAddress)

33.2.1 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_configureContinuousMode()
- TIMER_B_configureUpMode()
- TIMER_B_configureUpDownMode()
- TIMER_B_startContinuousMode()
- TIMER_B_startUpMode()
- TIMER_B_startUpDownMode()
- 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_clearCaptureCompareInterruptFlag()
- TIMER_B_clearTimerInterruptFlag()

33.2.2 Function Documentation

33.2.2.1 void TIMER_B_clear (uint16_t *baseAddress*)

Reset/Clear the TIMER_B clock divider, count direction, count.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns:
None

33.2.2.2 void TIMER_B_clearCaptureCompareInterruptFlag (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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

33.2.2.3 void TIMER_B_clearTimerInterruptFlag (uint16_t *baseAddress*)

Clears the TIMER_B TBIFG interrupt flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the TIMER_B module.

Modified bits are **TBIFG** of **TBxCTL** register.

Returns:
None

33.2.2.4 void TIMER_B_configureContinuousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *timerClear*)

DEPRECATED - 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	68
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	52

Parameters:
baseAddress is the base address of the TIMER_B module.
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**

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**

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

Modified bits of **TBxCTL** register.

Returns:
None

33.2.2.5 void TIMER_B_configureUpDownMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	88

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerPeriod is the specified TIMER_B period

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:
- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns:

None

33.2.2.6 void TIMER_B_configureUpMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	88

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerPeriod is the specified TIMER_B period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:
- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns:

None

33.2.2.7 void TIMER_B_disableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Disable capture compare interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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

33.2.2.8 void TIMER_B_disableInterrupt (uint16_t *baseAddress*)

Disable TIMER_B interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns:

None

33.2.2.9 void TIMER_B_enableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Enable capture compare interrupt.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	36
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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

33.2.2.10 void TIMER_B_enableInterrupt (uint16_t *baseAddress*)

Enable TIMER_B interrupt.

Enables TIMER_B interrupt. Does not clear interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns:
None

33.2.2.11 void TIMER_B_generatePWM (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *compareRegister*, uint16_t *compareOutputMode*, uint16_t *dutyCycle*)

DEPRECATED - Generate a PWM with TIMER_B running in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	80
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	68
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	76

Parameters:
baseAddress is the base address of the TIMER_B module.
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**

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**

timerPeriod selects the desired TIMER_B period
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**

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**

dutyCycle specifies the dutycycle for the generated waveform

Modified bits of **TBxCCTLn** register, bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns:
None

33.2.2.12 uint16_t TIMER_B_getCaptureCompareCount (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get current capturecompare count.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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

33.2.2.13 uint32_t TIMER_B_getCaptureCompareInterruptStatus (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *mask*)

Return capture compare interrupt status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

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**

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

33.2.2.14 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	58
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	38
MSPGCC 4.8.0	None	102
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	48

Parameters:
baseAddress is the base address of the Timer module.

Returns:
Majority vote of timer count value

33.2.2.15 uint32_t TIMER_B_getInterruptStatus (uint16_t *baseAddress*)

Get TIMER_B interrupt status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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

33.2.2.16 uint8_t TIMER_B_getOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get output bit for output mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	20
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

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**

33.2.2.17 uint8_t TIMER_B_getSynchronizedCaptureCompareInput (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *synchronized*)

Get synchronized capturecompare input.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	16
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

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_CAPTURECOMPAREINPUT**
- **TIMER_B_READ_CAPTURE_COMPARE_INPUT**

Returns:
One of the following:

- **TIMER_B_CAPTURECOMPARE_INPUT_HIGH**
- **TIMER_B_CAPTURECOMPARE_INPUT_LOW**

33.2.2.18 void TIMER_B_initCapture (uint16_t *baseAddress*, uint16_t *captureRegister*, uint16_t *captureMode*, uint16_t *captureInputSelect*, uint16_t *synchronizeCaptureSource*, uint16_t *captureInterruptEnable*, uint16_t *captureOutputMode*)

DEPRECATED - Initializes Capture Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	88
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	80
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	96
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	36

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

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**

synchronizeCaptureSource decides if capture source should be synchronized with TIMER_B clock Valid values are:

- **TIMER_B_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_B_CAPTURE_SYNCHRONOUS**

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**

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**

Modified bits of **TBxCCTLn** register.

Returns:
None

33.2.2.19 void TIMER_B_initCaptureMode (uint16_t *baseAddress*, TIMER_B_initCaptureModeParam ∗ *param*)

Initializes Capture Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	70
TI Compiler 4.2.1	Size	48
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	50
IAR 5.51.6	Size	48
IAR 5.51.6	Speed	48
MSPGCC 4.8.0	None	128
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

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

33.2.2.20 void TIMER_B_initCompare (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareInterruptEnable*, uint16_t *compareOutputMode*, uint16_t *compareValue*)

DEPRECATED - Initializes Compare Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	64
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	28

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**

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**

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**

compareValue is the count to be compared with in compare mode

Modified bits of **TBxCCTLn** register and bits of **TBxCCRn** register.

Returns:
None

33.2.2.21 void TIMER_B_initCompareLatchLoadEvent (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareLatchLoadEvent*)

Selects Compare Latch Load Event.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	72
MSPGCC 4.8.0	Size	14
MSPGCC 4.8.0	Speed	14

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**

compareLatchLoadEvent selects the latch load event Valid values are:
- **TIMER_B_LATCH_ON_WRITE_TO_TBxCCRn_COMPARE_REGISTER** [Default]
- **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UP_OR_CONT_MODE**
- **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UPDOWN_MODE**
- **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_CURRENT_COMPARE_LATCH_VALUE**

Modified bits are **CLLD** of **TBxCCTLn** register.

Returns:
None

33.2.2.22 void TIMER_B_initCompareMode (uint16_t *baseAddress*, TIMER_B_initCompareModeParam ∗ *param*)

Initializes Compare Mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	72
TI Compiler 4.2.1	Size	46
TI Compiler 4.2.1	Speed	46
IAR 5.51.6	None	52
IAR 5.51.6	Size	50
IAR 5.51.6	Speed	50
MSPGCC 4.8.0	None	126
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

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

33.2.2.23 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	70
TI Compiler 4.2.1	Speed	70
IAR 5.51.6	None	78
IAR 5.51.6	Size	70
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	170
MSPGCC 4.8.0	Size	82
MSPGCC 4.8.0	Speed	82

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

33.2.2.24 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	152
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	98
IAR 5.51.6	None	118
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	72
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	116
MSPGCC 4.8.0	Speed	118

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

33.2.2.25 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	152
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	98
IAR 5.51.6	None	120
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	74
MSPGCC 4.8.0	None	248
MSPGCC 4.8.0	Size	116
MSPGCC 4.8.0	Speed	118

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

33.2.2.26 void TIMER_B_outputPWM (uint16_t *baseAddress*, TIMER_B_outputPWMParam ∗ *param*)

Generate a PWM with TIMER_B running in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	136
TI Compiler 4.2.1	Size	86
TI Compiler 4.2.1	Speed	86
IAR 5.51.6	None	108
IAR 5.51.6	Size	100
IAR 5.51.6	Speed	100
MSPGCC 4.8.0	None	228
MSPGCC 4.8.0	Size	100
MSPGCC 4.8.0	Speed	100

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

33.2.2.27 void TIMER_B_selectCounterLength (uint16_t *baseAddress*, uint16_t *counterLength*)

Selects TIMER_B counter length.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
baseAddress is the base address of the TIMER_B module.

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

33.2.2.28 void TIMER_B_selectLatchingGroup (uint16_t *baseAddress*, uint16_t *groupLatch*)

Selects TIMER_B Latching Group.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

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

33.2.2.29 void TIMER_B_setCompareValue (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareValue*)

Sets the value of the capture-compare register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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

33.2.2.30 void TIMER_B_setOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint8_t *outputModeOutBitValue*)

Set output bit for output mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	14
TI Compiler 4.2.1	Speed	14
IAR 5.51.6	None	20
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	74
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	16

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

33.2.2.31 void TIMER_B_startContinousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *timerClear*)

DEPRECATED - Spelling Error Fixed. Starts TIMER_B in continuous mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	52
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	34
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	52
MSPGCC 4.8.0	Size	16
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

Modified bits of **TBxCTL** register.

Returns:
None

33.2.2.32 void TIMER_B_startContinuousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *timerClear*)

DEPRECATED - Replaced by TIMER_B_configureContinuousMode and TIMER_B_startCounter API. Starts TIMER_B in continuous mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	38
TI Compiler 4.2.1	Speed	38
IAR 5.51.6	None	68
IAR 5.51.6	Size	14
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	64
MSPGCC 4.8.0	Speed	58

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:
- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

Modified bits of **TBxCTL** register.

Returns:
None

33.2.2.33 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_configureContinuousMode, TIMER_B_configureUpMode or TIMER_B_configureUpDownMode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the TIMER_B module.

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

33.2.2.34 void TIMER_B_startUpDownMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Replaced by TIMER_B_configureUpDownMode and TIMER_B_startCounter API. Starts TIMER_B in up down mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	94

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerPeriod is the specified TIMER_B period

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:
- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns:
None

33.2.2.35 void TIMER_B_startUpMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TBIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Replaced by TIMER_B_configureUpMode and TIMER_B_startCounter API. Starts TIMER_B in up mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	54
IAR 5.51.6	None	84
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	62
MSPGCC 4.8.0	None	104
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	94

Parameters:

baseAddress is the base address of the TIMER_B module.

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**

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**

timerPeriod is the specified TIMER_B period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

timerInterruptEnable_TBIE is to enable or disable TIMER_B interrupt Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

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]

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]

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns:
None

33.2.2.36 void TIMER_B_stop (uint16_t *baseAddress*)

Stops the TIMER_B.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns:
None

33.3 Programming Example

The following example shows some TIMER_B operations using the APIs

```
{       //Start TIMER_B
        TIMER_B_configureUpMode(   TIMER_B0_BASE,
          TIMER_B_CLOCKSOURCE_SMCLK,
          TIMER_B_CLOCKSOURCE_DIVIDER_1,
          511,
          TIMER_B_TBIE_INTERRUPT_DISABLE,
          TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE,
          TIMER_B_DO_CLEAR
          );

        TIMER_B_startCounter(   TIMER_B0_BASE,
                TIMER_B_UP_MODE
                );

        //Initialize compare mode to generate PWM1
        TIMER_B_initCompare(TIMER_B0_BASE,
          TIMER_B_CAPTURECOMPARE_REGISTER_1,
          TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE,
          TIMER_B_OUTPUTMODE_RESET_SET,
          383
          );

        //Initialize compare mode to generate PWM2
        TIMER_B_initCompare(TIMER_B0_BASE,
          TIMER_B_CAPTURECOMPARE_REGISTER_2,
          TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE,
          TIMER_B_OUTPUTMODE_RESET_SET,
          128
          );
}
```

34 TIMER_D

34.1 Introduction

Timer_D is a 16-bit timer/counter with multiple capture/compare registers. Timer_D can support multiple capture/compares, interval timing, and PWM outputs both in general and high resolution modes. Timer_D also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions, from each of the capture/compare registers.

This peripheral API handles Timer D hardware peripheral.

TIMER_D features include:

- Asynchronous 16-bit timer/counter with four operating modes and four selectable lengths
- Selectable and configurable clock source
- Configurable capture/compare registers
- Controlling rising and falling PWM edges by combining two neighbor TDCCR registers in one compare channel output
- Configurable outputs with PWM capability
- High-resolution mode with a fine clock frequency up to 16 times the timer input clock frequency
- Double-buffered compare registers with synchronized loading
- Interrupt vector register for fast decoding of all Timer_D interrupts

Differences From Timer_B Timer_D is identical to Timer_B with the following exceptions:

- Timer_D supports high-resolution mode.
- Timer_D supports the combination of two adjacent TDCCRx registers in one capture/compare channel.
- Timer_D supports the dual capture event mode.
- Timer_D supports external fault input, external clear input, and signal. See the TEC chapter for detailed information.
- Timer_D can synchronize with a second timer instance when available. See the TEC chapter for detailed information.

TIMER_D can operate in 3 modes

- Continuous Mode
- Up Mode
- Down Mode

TIMER_D Interrupts may be generated on counter overflow conditions and during capture compare events.

The TIMER_D may also be used to generate PWM outputs. PWM outputs can be generated by initializing the compare mode with TIMER_D_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_D_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_D_generatePWM() or a combination of TIMER_D_initCompare() and timer start APIs

The TimerD API provides a set of functions for dealing with the TimerD 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.

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

34.2 API Functions

Functions

- void TIMER_D_clear (uint16_t baseAddress)
- void TIMER_D_clearCaptureCompareInterruptFlag (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_D_clearHighResInterruptStatus (uint16_t baseAddress, uint16_t mask)
- void TIMER_D_clearTimerInterruptFlag (uint16_t baseAddress)
- void TIMER_D_combineTDCCRToGeneratePWM (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint16_t timerPeriod, uint16_t combineCCRRegistersCombination, uint16_t compareOutputMode, uint16_t dutyCycle1, uint16_t dutyCycle2)
- void TIMER_D_combineTDCCRToOutputPWM (uint16_t baseAddress, TIMER_D_combineTDCCRToOutputPWMParam ∗param)
- void TIMER_D_configureContinuousMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint16_t timerInterruptEnable_TDIE, uint16_t timerClear)
- void TIMER_D_configureHighResGeneratorInRegulatedMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint8_t highResClockMultiplyFactor, uint8_t highResClockDivider)
- void TIMER_D_configureUpDownMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint16_t timerPeriod, uint16_t timerInterruptEnable_TDIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_D_configureUpMode (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint16_t timerPeriod, uint16_t timerInterruptEnable_TDIE, uint16_t captureCompareInterruptEnable_CCR0_CCIE, uint16_t timerClear)
- void TIMER_D_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_D_disableHighResClockEnhancedAccuracy (uint16_t baseAddress)
- void TIMER_D_disableHighResFastWakeup (uint16_t baseAddress)
- void TIMER_D_DisableHighResGeneratorForceON (uint16_t baseAddress)
- void TIMER_D_disableHighResInterrupt (uint16_t baseAddress, uint16_t mask)
- void TIMER_D_disableTimerInterrupt (uint16_t baseAddress)
- void TIMER_D_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
- void TIMER_D_enableHighResClockEnhancedAccuracy (uint16_t baseAddress)
- void TIMER_D_enableHighResFastWakeup (uint16_t baseAddress)
- void TIMER_D_EnableHighResGeneratorForceON (uint16_t baseAddress)
- void TIMER_D_enableHighResInterrupt (uint16_t baseAddress, uint16_t mask)
- void TIMER_D_enableTimerInterrupt (uint16_t baseAddress)
- void TIMER_D_generatePWM (uint16_t baseAddress, uint16_t clockSource, uint16_t clockSourceDivider, uint16_t clockingMode, uint16_t timerPeriod, uint16_t compareRegister, uint16_t compareOutputMode, uint16_t dutyCycle)
- uint16_t TIMER_D_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint8_t TIMER_D_getCaptureCompareInputSignal (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint32_t TIMER_D_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
- uint16_t TIMER_D_getCaptureCompareLatchCount (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint16_t TIMER_D_getCounterValue (uint16_t baseAddress)
- uint16_t TIMER_D_getHighResInterruptStatus (uint16_t baseAddress, uint16_t mask)
- uint8_t TIMER_D_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
- uint8_t TIMER_D_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
- uint32_t TIMER_D_getTimerInterruptStatus (uint16_t baseAddress)
- void TIMER_D_initCapture (uint16_t baseAddress, uint16_t captureRegister, uint16_t captureMode, uint16_t captureInputSelect, uint16_t synchronizeCaptureSource, uint16_t captureInterruptEnable, uint16_t captureOutputMode, uint8_t channelCaptureMode)
- void TIMER_D_initCaptureMode (uint16_t baseAddress, TIMER_D_initCaptureModeParam ∗param)
- void TIMER_D_initCompare (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareInterruptEnable, uint16_t compareOutputMode, uint16_t compareValue)
- void TIMER_D_initCompareLatchLoadEvent (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
- void TIMER_D_initCompareMode (uint16_t baseAddress, TIMER_D_initCompareModeParam ∗param)

- void TIMER_D_initContinuousMode (uint16_t baseAddress, TIMER_D_initContinuousModeParam ∗param)
- uint8_t TIMER_D_initHighResGeneratorInFreeRunningMode (uint16_t baseAddress, uint8_t desiredHighResFrequency)
- void TIMER_D_initHighResGeneratorInRegulatedMode (uint16_t baseAddress, TIMER_D_initHighResGeneratorInRegulatedModeParam ∗param)
- void TIMER_D_initUpDownMode (uint16_t baseAddress, TIMER_D_initUpDownModeParam ∗param)
- void TIMER_D_initUpMode (uint16_t baseAddress, TIMER_D_initUpModeParam ∗param)
- void TIMER_D_outputPWM (uint16_t baseAddress, TIMER_D_outputPWMParam ∗param)
- void TIMER_D_selectCounterLength (uint16_t baseAddress, uint16_t counterLength)
- void TIMER_D_selectHighResClockRange (uint16_t baseAddress, uint16_t highResClockRange)
- void TIMER_D_selectHighResCoarseClockRange (uint16_t baseAddress, uint16_t highResCoarseClockRange)
- void TIMER_D_selectLatchingGroup (uint16_t baseAddress, uint16_t groupLatch)
- void TIMER_D_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
- void TIMER_D_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
- void TIMER_D_startCounter (uint16_t baseAddress, uint16_t timerMode)
- void TIMER_D_stop (uint16_t baseAddress)

34.2.1 Detailed Description

The TIMER_D 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.

TimerD configuration and initialization is handled by

- TIMER_D_startCounter(),
- TIMER_D_configureContinuousMode(),
- TIMER_D_configureUpMode(),
- TIMER_D_configureUpDownMode(),
- TIMER_D_startContinuousMode(),
- TIMER_D_startUpMode(),
- TIMER_D_startUpDownMode(),
- TIMER_D_initCapture(),
- TIMER_D_initCompare(),
- TIMER_D_clear(),
- TIMER_D_stop(),
- TIMER_D_configureHighResGeneratorInFreeRunningMode(),
- TIMER_D_configureHighResGeneratorInRegulatedMode(),
- TIMER_D_combineTDCCRToGeneratePWM(),
- TIMER_D_selectLatchingGroup(),
- TIMER_D_selectCounterLength(),
- TIMER_D_initCompareLatchLoadEvent(),
- TIMER_D_disableHighResFastWakeup(),
- TIMER_D_enableHighResFastWakeup(),
- TIMER_D_disableHighResClockEnhancedAccuracy(),
- TIMER_D_enableHighResClockEnhancedAccuracy(),
- TIMER_D_DisableHighResGeneratorForceON(),
- TIMER_D_EnableHighResGeneratorForceON(),
- TIMER_D_selectHighResCoarseClockRange(),
- TIMER_D_selectHighResClockRange()

TimerD outputs are handled by

- TIMER_D_getSynchronizedCaptureCompareInput(),
- TIMER_D_getOutputForOutputModeOutBitValue(),
- TIMER_D_setOutputForOutputModeOutBitValue(),
- TIMER_D_generatePWM(),
- TIMER_D_getCaptureCompareCount(),
- TIMER_D_setCompareValue(),
- TIMER_D_getCaptureCompareLatchCount(),
- TIMER_D_getCaptureCompareInputSignal(),
- TIMER_D_getCounterValue()

The interrupt handler for the TimerD interrupt is managed with

- TIMER_D_enableTimerInterrupt(),
- TIMER_D_disableTimerInterrupt(),
- TIMER_D_getTimerInterruptStatus(),
- TIMER_D_enableCaptureCompareInterrupt(),
- TIMER_D_disableCaptureCompareInterrupt(),
- TIMER_D_getCaptureCompareInterruptStatus(),
- TIMER_D_clearCaptureCompareInterruptFlag()
- TIMER_D_clearTimerInterruptFlag(),
- TIMER_D_enableHighResInterrupt(),
- TIMER_D_disableTimerInterrupt(),
- TIMER_D_getHighResInterruptStatus(),
- TIMER_D_clearHighResInterruptStatus()

Timer_D High Resolution handling APIs

- TIMER_D_getHighResInterruptStatus(),
- TIMER_D_clearHighResInterruptStatus(),
- TIMER_D_disableHighResFastWakeup(),
- TIMER_D_enableHighResFastWakeup(),
- TIMER_D_disableHighResClockEnhancedAccuracy(),
- TIMER_D_enableHighResClockEnhancedAccuracy(),
- TIMER_D_DisableHighResGeneratorForceON(),
- TIMER_D_EnableHighResGeneratorForceON(),
- TIMER_D_selectHighResCoarseClockRange(),
- TIMER_D_selectHighResClockRange(),
- TIMER_D_configureHighResGeneratorInFreeRunningMode(),
- TIMER_D_configureHighResGeneratorInRegulatedMode()

34.2.2 Function Documentation

34.2.2.1 void TIMER_D_clear (uint16_t *baseAddress*)

Reset/Clear the timer clock divider, count direction, count.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits of **TDxCTL0** register.

Returns:
None

34.2.2.2 void TIMER_D_clearCaptureCompareInterruptFlag (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture-compare register being used. Valid values are:

- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Modified bits are **CCIFG** of **TDxCCTLn** register.

Returns:

None

34.2.2.3 void TIMER_D_clearHighResInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Clears High Resolution interrupt status.

Parameters:

baseAddress is the base address of the TIMER_D module.

mask is the mask for the interrupts to clear Mask value is the logical OR of any of the following:

- **TIMER_D_HIGH_RES_FREQUENCY_UNLOCK**
- **TIMER_D_HIGH_RES_FREQUENCY_LOCK**
- **TIMER_D_HIGH_RES_FAIL_HIGH**
- **TIMER_D_HIGH_RES_FAIL_LOW**

Modified bits of **TDxHINT** register.

Returns:

None

34.2.2.4 void TIMER_D_clearTimerInterruptFlag (uint16_t *baseAddress*)

Clears the Timer TDIFG interrupt flag.

Parameters:

baseAddress is the base address of the TIMER_D module.

Modified bits are **TDIFG** of **TDxCTL0** register.

Returns:

None

34.2.2.5 void TIMER_D_combineTDCCRToGeneratePWM (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint16_t *timerPeriod*, uint16_t *combineCCRRegistersCombination*, uint16_t *compareOutputMode*, uint16_t *dutyCycle1*, uint16_t *dutyCycle2*)

DEPRECATED - Combine TDCCR to get PWM.

Parameters:

baseAddress is the base address of the TIMER_D module.

clockSource selects Clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**
- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**
- **TIMER_D_CLOCKSOURCE_DIVIDER_56**
- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:
- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]
- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

timerPeriod is the specified timer period

combineCCRRegistersCombination selects desired CCR registers to combine Valid values are:
- **TIMER_D_COMBINE_CCR1_CCR2**
- **TIMER_D_COMBINE_CCR3_CCR4** - (available on TIMER_D5, TIMER_D7)
- **TIMER_D_COMBINE_CCR5_CCR6** - (available only on TIMER_D7)

compareOutputMode specifies the output mode. Valid values are:
- **TIMER_D_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_D_OUTPUTMODE_SET**
- **TIMER_D_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_D_OUTPUTMODE_SET_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE**
- **TIMER_D_OUTPUTMODE_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE_SET**
- **TIMER_D_OUTPUTMODE_RESET_SET**

dutyCycle1 specifies the dutycycle for the generated waveform

dutyCycle2 specifies the dutycycle for the generated waveform

Modified bits of **TDxCCTLn** register, bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.6 void TIMER_D_combineTDCCRToOutputPWM (uint16_t *baseAddress*, TIMER_D_combineTDCCRToOutputPWMParam ∗ *param*)

Combine TDCCR to get PWM.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for PWM generation using two CCRs.

Modified bits of **TDxCCTLn** register, bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.7 void TIMER_D_configureContinuousMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint16_t *timerInterruptEnable_TDIE*, uint16_t *timerClear*)

DEPRECATED - Configures timer in continuous mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:
baseAddress is the base address of the TIMER_D module.
clockSource selects Clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**
- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**

- **TIMER_D_CLOCKSOURCE_DIVIDER_56**
- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:
- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]
- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

timerInterruptEnable_TDIE is to enable or disable timer interrupt Valid values are:
- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:
- **TIMER_D_DO_CLEAR**
- **TIMER_D_SKIP_CLEAR** [Default]

Modified bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.8 void TIMER_D_configureHighResGeneratorInRegulatedMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint8_t *highResClockMultiplyFactor*, uint8_t *highResClockDivider*)

DEPRECATED - Configures TIMER_D in Regulated mode.

Parameters:

baseAddress is the base address of the TIMER_D module.

clockSource selects Clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**
- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**
- **TIMER_D_CLOCKSOURCE_DIVIDER_56**
- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:
- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]

- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

highResClockMultiplyFactor selects the high resolution multiply factor. Valid values are:

- **TIMER_D_HIGHRES_CLK_MULTIPLY_FACTOR_8x**
- **TIMER_D_HIGHRES_CLK_MULTIPLY_FACTOR_16x**

highResClockDivider selects the high resolution divider. Valid values are:

- **TIMER_D_HIGHRES_CLK_DIVIDER_1**
- **TIMER_D_HIGHRES_CLK_DIVIDER_2**
- **TIMER_D_HIGHRES_CLK_DIVIDER_4**
- **TIMER_D_HIGHRES_CLK_DIVIDER_8**

Modified bits of **TDxHCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.9 void TIMER_D_configureUpDownMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TDIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Configures timer in up down mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:

baseAddress is the base address of the TIMER_D module.

clockSource selects Clock source. Valid values are:

- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:

- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**
- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**
- **TIMER_D_CLOCKSOURCE_DIVIDER_56**
- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:

- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]

- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

timerPeriod is the specified timer period

timerInterruptEnable_TDIE is to enable or disable timer interrupt Valid values are:
- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

captureCompareInterruptEnable_CCR0_CCIE is to enable or disable timer CCR0 captureComapre interrupt. Valid values are:
- **TIMER_D_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_D_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:
- **TIMER_D_DO_CLEAR**
- **TIMER_D_SKIP_CLEAR** [Default]

Modified bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.10 void TIMER_D_configureUpMode (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint16_t *timerPeriod*, uint16_t *timerInterruptEnable_TDIE*, uint16_t *captureCompareInterruptEnable_CCR0_CCIE*, uint16_t *timerClear*)

DEPRECATED - Configures timer in up mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:

baseAddress is the base address of the TIMER_D module.

clockSource selects Clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**
- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**
- **TIMER_D_CLOCKSOURCE_DIVIDER_56**

- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:
- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]
- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

timerPeriod is the specified timer period. This is the value that gets written into the CCR0. Limited to 16 bits [uint16_t]

timerInterruptEnable_TDIE is to enable or disable timer interrupt Valid values are:
- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

captureCompareInterruptEnable_CCR0_CCIE is to enable or disable timer CCR0 captureComapre interrupt. Valid values are:
- **TIMER_D_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_D_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

timerClear decides if timer clock divider, count direction, count need to be reset. Valid values are:
- **TIMER_D_DO_CLEAR**
- **TIMER_D_SKIP_CLEAR** [Default]

Modified bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.11 void TIMER_D_disableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Disable capture compare interrupt.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister is the selected capture compare register Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TDxCCTLn** register.

Returns:
None

34.2.2.12 void TIMER_D_disableHighResClockEnhancedAccuracy (uint16_t *baseAddress*)

Disable High Resolution Clock Enhanced Accuracy.

Parameters:

baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHEAEN** of **TDxHCTL0** register.

Returns:
None

34.2.2.13 void TIMER_D_disableHighResFastWakeup (uint16_t *baseAddress*)

Disable High Resolution fast wakeup.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHFW** of **TDxHCTL0** register.

Returns:
None

34.2.2.14 void TIMER_D_DisableHighResGeneratorForceON (uint16_t *baseAddress*)

Disable High Resolution Clock Enhanced Accuracy.

High-resolution generator is on if the TIMER_D counter

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHRON** of **TDxHCTL0** register.

Returns:
None

34.2.2.15 void TIMER_D_disableHighResInterrupt (uint16_t *baseAddress*, uint16_t *mask*)

Disable High Resolution interrupt.

Parameters:
baseAddress is the base address of the TIMER_D module.
mask is the mask of interrupts to disable Mask value is the logical OR of any of the following:
- **TIMER_D_HIGH_RES_FREQUENCY_UNLOCK**
- **TIMER_D_HIGH_RES_FREQUENCY_LOCK**
- **TIMER_D_HIGH_RES_FAIL_HIGH**
- **TIMER_D_HIGH_RES_FAIL_LOW**

Modified bits of **TDxHINT** register.

Returns:
None

34.2.2.16 void TIMER_D_disableTimerInterrupt (uint16_t *baseAddress*)

Disable timer interrupt.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits of **TDxCTL0** register.

Returns:
None

34.2.2.17 void TIMER_D_enableCaptureCompareInterrupt (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Enable capture compare interrupt.

Parameters:
baseAddress is the base address of the TIMER_D module.
captureCompareRegister is the selected capture compare register Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TDxCCTLn** register.

Returns:
None

34.2.2.18 void TIMER_D_enableHighResClockEnhancedAccuracy (uint16_t *baseAddress*)

Enable High Resolution Clock Enhanced Accuracy.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHEAEN** of **TDxHCTL0** register.

Returns:
None

34.2.2.19 void TIMER_D_enableHighResFastWakeup (uint16_t *baseAddress*)

Enable High Resolution fast wakeup.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHFW** of **TDxHCTL0** register.

Returns:
None

34.2.2.20 void TIMER_D_EnableHighResGeneratorForceON (uint16_t *baseAddress*)

Enable High Resolution Clock Enhanced Accuracy.

High-resolution generator is on in all TIMER_D MCx modes. The PMM remains in high-current mode.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits are **TDHRON** of **TDxHCTL0** register.

Returns:
None

34.2.2.21 void TIMER_D_enableHighResInterrupt (uint16_t *baseAddress*, uint16_t *mask*)

Enable High Resolution interrupt.

Parameters:
baseAddress is the base address of the TIMER_D module.
mask is the mask of interrupts to enable Mask value is the logical OR of any of the following:
- **TIMER_D_HIGH_RES_FREQUENCY_UNLOCK**
- **TIMER_D_HIGH_RES_FREQUENCY_LOCK**
- **TIMER_D_HIGH_RES_FAIL_HIGH**
- **TIMER_D_HIGH_RES_FAIL_LOW**

Modified bits of **TDxHINT** register.

Returns:
None

34.2.2.22 void TIMER_D_enableTimerInterrupt (uint16_t *baseAddress*)

Enable timer interrupt.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits of **TDxCTL0** register.

Returns:
None

34.2.2.23 void TIMER_D_generatePWM (uint16_t *baseAddress*, uint16_t *clockSource*, uint16_t *clockSourceDivider*, uint16_t *clockingMode*, uint16_t *timerPeriod*, uint16_t *compareRegister*, uint16_t *compareOutputMode*, uint16_t *dutyCycle*)

DEPRECATED - Generate a PWM with timer running in up mode.

Parameters:
baseAddress is the base address of the TIMER_D module.
clockSource selects Clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_EXTERNAL_TDCLK** [Default]
- **TIMER_D_CLOCKSOURCE_ACLK**
- **TIMER_D_CLOCKSOURCE_SMCLK**
- **TIMER_D_CLOCKSOURCE_INVERTED_EXTERNAL_TDCLK**

clockSourceDivider is the divider for clock source. Valid values are:
- **TIMER_D_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_D_CLOCKSOURCE_DIVIDER_2**
- **TIMER_D_CLOCKSOURCE_DIVIDER_3**
- **TIMER_D_CLOCKSOURCE_DIVIDER_4**
- **TIMER_D_CLOCKSOURCE_DIVIDER_5**
- **TIMER_D_CLOCKSOURCE_DIVIDER_6**
- **TIMER_D_CLOCKSOURCE_DIVIDER_7**
- **TIMER_D_CLOCKSOURCE_DIVIDER_8**
- **TIMER_D_CLOCKSOURCE_DIVIDER_10**
- **TIMER_D_CLOCKSOURCE_DIVIDER_12**
- **TIMER_D_CLOCKSOURCE_DIVIDER_14**
- **TIMER_D_CLOCKSOURCE_DIVIDER_16**

- **TIMER_D_CLOCKSOURCE_DIVIDER_20**
- **TIMER_D_CLOCKSOURCE_DIVIDER_24**
- **TIMER_D_CLOCKSOURCE_DIVIDER_28**
- **TIMER_D_CLOCKSOURCE_DIVIDER_32**
- **TIMER_D_CLOCKSOURCE_DIVIDER_40**
- **TIMER_D_CLOCKSOURCE_DIVIDER_48**
- **TIMER_D_CLOCKSOURCE_DIVIDER_56**
- **TIMER_D_CLOCKSOURCE_DIVIDER_64**

clockingMode is the selected clock mode register values. Valid values are:
- **TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK** [Default]
- **TIMER_D_CLOCKINGMODE_HIRES_LOCAL_CLOCK**
- **TIMER_D_CLOCKINGMODE_AUXILIARY_CLK**

timerPeriod is the specified timer period

compareRegister selects the compare register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

compareOutputMode specifies the output mode. Valid values are:
- **TIMER_D_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_D_OUTPUTMODE_SET**
- **TIMER_D_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_D_OUTPUTMODE_SET_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE**
- **TIMER_D_OUTPUTMODE_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE_SET**
- **TIMER_D_OUTPUTMODE_RESET_SET**

dutyCycle specifies the dutycycle for the generated waveform

Modified bits of **TDxCCTLn** register, bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.24 uint16_t TIMER_D_getCaptureCompareCount (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get current capturecompare count.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Returns:
current count as uint16_t

34.2.2.25 uint8_t TIMER_D_getCaptureCompareInputSignal (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get current capturecompare input signal.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture register being used. Valid values are:

- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Returns:

One of the following:

- **TIMER_D_CAPTURECOMPARE_INPUT**
- **0x00**
  indicating the current input signal

34.2.2.26 uint32_t TIMER_D_getCaptureCompareInterruptStatus (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *mask*)

Return capture compare interrupt status.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister is the selected capture compare register Valid values are:

- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

mask is the mask for the interrupt status Mask value is the logical OR of any of the following:

- **TIMER_D_CAPTURE_OVERFLOW**
- **TIMER_D_CAPTURECOMPARE_INTERRUPT_FLAG**

Returns:

Logical OR of any of the following:

- **TIMER_D_CAPTURE_OVERFLOW**
- **TIMER_D_CAPTURECOMPARE_INTERRUPT_FLAG**
  indicating the status of the masked flags

34.2.2.27 uint16_t TIMER_D_getCaptureCompareLatchCount (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get current capture compare latch register count.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Returns:
current count as uint16_t

34.2.2.28 uint16_t TIMER_D_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_D_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_D module.

Returns:
Majority vote of timer count value

34.2.2.29 uint16_t TIMER_D_getHighResInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Returns High Resolution interrupt status.

Parameters:
baseAddress is the base address of the TIMER_D module.
mask is the mask for the interrupt status Mask value is the logical OR of any of the following:
- **TIMER_D_HIGH_RES_FREQUENCY_UNLOCK**
- **TIMER_D_HIGH_RES_FREQUENCY_LOCK**
- **TIMER_D_HIGH_RES_FAIL_HIGH**
- **TIMER_D_HIGH_RES_FAIL_LOW**

Modified bits of **TDxHINT** register.

Returns:
Logical OR of any of the following:

- **TIMER_D_HIGH_RES_FREQUENCY_UNLOCK**
- **TIMER_D_HIGH_RES_FREQUENCY_LOCK**
- **TIMER_D_HIGH_RES_FAIL_HIGH**
- **TIMER_D_HIGH_RES_FAIL_LOW**
indicating the status of the masked interrupts

34.2.2.30 uint8_t TIMER_D_getOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get output bit for output mode.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

Returns:

One of the following:

- **TIMER_D_OUTPUTMODE_OUTBITVALUE_HIGH**
- **TIMER_D_OUTPUTMODE_OUTBITVALUE_LOW**

34.2.2.31 uint8_t TIMER_D_getSynchronizedCaptureCompareInput (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *synchronized*)

Get synchronized capturecompare input.

Parameters:

baseAddress is the base address of the TIMER_D module.

captureCompareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

synchronized is to select type of capture compare input. Valid values are:
- **TIMER_D_READ_SYNCHRONIZED_CAPTURECOMPAREINPUT**
- **TIMER_D_READ_CAPTURE_COMPARE_INPUT**

Returns:

One of the following:

- **TIMER_D_CAPTURECOMPARE_INPUT_HIGH**
- **TIMER_D_CAPTURECOMPARE_INPUT_LOW**

34.2.2.32 uint32_t TIMER_D_getTimerInterruptStatus (uint16_t *baseAddress*)

Get timer interrupt status.

Parameters:

baseAddress is the base address of the TIMER_D module.

Returns:

One of the following:

- **TIMER_D_INTERRUPT_NOT_PENDING**
- **TIMER_D_INTERRUPT_PENDING**
  indicating the timer interrupt status

34.2.2.33 void TIMER_D_initCapture (uint16_t *baseAddress*, uint16_t *captureRegister*, uint16_t *captureMode*, uint16_t *captureInputSelect*, uint16_t *synchronizeCaptureSource*, uint16_t *captureInterruptEnable*, uint16_t *captureOutputMode*, uint8_t *channelCaptureMode*)

DEPRECATED - Initializes Capture Mode.

Parameters:

baseAddress is the base address of the TIMER_D module.

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_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

captureMode is the capture mode selected. Valid values are:

- **TIMER_D_CAPTUREMODE_NO_CAPTURE** [Default]
- **TIMER_D_CAPTUREMODE_RISING_EDGE**
- **TIMER_D_CAPTUREMODE_FALLING_EDGE**
- **TIMER_D_CAPTUREMODE_RISING_AND_FALLING_EDGE**

captureInputSelect decides the Input Select Valid values are:

- **TIMER_D_CAPTURE_INPUTSELECT_CCIxA** [Default]
- **TIMER_D_CAPTURE_INPUTSELECT_CCIxB**
- **TIMER_D_CAPTURE_INPUTSELECT_GND**
- **TIMER_D_CAPTURE_INPUTSELECT_Vcc**

synchronizeCaptureSource decides if capture source should be synchronized with timer clock Valid values are:

- **TIMER_D_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_D_CAPTURE_SYNCHRONOUS**

Modified bits of **TDxCCTLn** register and bits of **TDxCTL2** register.

Returns:

None

34.2.2.34 void TIMER_D_initCaptureMode (uint16_t *baseAddress*, TIMER_D_initCaptureModeParam ∗ *param*)

Initializes Capture Mode.

Parameters:

baseAddress is the base address of the TIMER_D module.

param is the pointer to struct for capture mode initialization.

Modified bits of **TDxCCTLn** register and bits of **TDxCTL2** register.

Returns:

None

34.2.2.35 void TIMER_D_initCompare (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareInterruptEnable*, uint16_t *compareOutputMode*, uint16_t *compareValue*)

DEPRECATED - Initializes Compare Mode.

Parameters:

baseAddress is the base address of the TIMER_D module.

compareRegister selects the Capture register being used. Valid values are:

- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

compareInterruptEnable is to enable or disable timer captureComapre interrupt. Valid values are:

- **TIMER_D_CAPTURECOMPARE_INTERRUPT_ENABLE**
- **TIMER_D_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]

compareOutputMode specifies the output mode. Valid values are:

- **TIMER_D_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_D_OUTPUTMODE_SET**
- **TIMER_D_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_D_OUTPUTMODE_SET_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE**
- **TIMER_D_OUTPUTMODE_RESET**
- **TIMER_D_OUTPUTMODE_TOGGLE_SET**
- **TIMER_D_OUTPUTMODE_RESET_SET**

compareValue is the count to be compared with in compare mode

Modified bits of **TDxCCTLn** register and bits of **TDxCCRn** register.

Returns:

None

34.2.2.36 void TIMER_D_initCompareLatchLoadEvent (uint16_t *baseAddress*, uint16_t *compareRegister*, uint16_t *compareLatchLoadEvent*)

Selects Compare Latch Load Event.

Parameters:

baseAddress is the base address of the TIMER_D module.

compareRegister selects the compare register being used. Valid values are:

- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

compareLatchLoadEvent selects the latch load event Valid values are:

- **TIMER_D_LATCH_ON_WRITE_TO_TDxCCRn_COMPARE_REGISTER** [Default]
- **TIMER_D_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UP_OR_CONT_MODE**
- **TIMER_D_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UPDOWN_MODE**
- **TIMER_D_LATCH_WHEN_COUNTER_COUNTS_TO_CURRENT_COMPARE_LATCH_VALUE**

Modified bits are **CLLD** of **TDxCCTLn** register.

Returns:
None

34.2.2.37 void TIMER_D_initCompareMode (uint16_t *baseAddress*, TIMER_D_initCompareModeParam ∗ *param*)

Initializes Compare Mode.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for compare mode initialization.

Modified bits of **TDxCCTLn** register and bits of **TDxCCRn** register.

Returns:
None

34.2.2.38 void TIMER_D_initContinuousMode (uint16_t *baseAddress*, TIMER_D_initContinuousModeParam ∗ *param*)

Configures timer in continuous mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for continuous mode initialization.

Modified bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.39 uint8_t TIMER_D_initHighResGeneratorInFreeRunningMode (uint16_t *baseAddress*, uint8_t *desiredHighResFrequency*)

Configures TIMER_D in free running mode.

Parameters:
baseAddress is the base address of the TIMER_D module.
desiredHighResFrequency selects the desired High Resolution frequency used. Valid values are:
- **TIMER_D_HIGHRES_64MHZ**
- **TIMER_D_HIGHRES_128MHZ**
- **TIMER_D_HIGHRES_200MHZ**
- **TIMER_D_HIGHRES_256MHZ**

Modified bits of **TDxHCTL1** register, bits of **TDxHCTL0** register and bits of **TDxCTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAIL

34.2.2.40 void TIMER_D_initHighResGeneratorInRegulatedMode (uint16_t *baseAddress*, TIMER_D_initHighResGeneratorInRegulatedModeParam ∗ *param*)

Configures TIMER_D in Regulated mode.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for high resolution generator in regulated mode.

Modified bits of **TDxHCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.41 void TIMER_D_initUpDownMode (uint16_t *baseAddress*, TIMER_D_initUpDownModeParam ∗ *param*)

Configures timer in up down mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for up-down mode initialization.

Modified bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.42 void TIMER_D_initUpMode (uint16_t *baseAddress*, TIMER_D_initUpModeParam ∗ *param*)

Configures timer in up mode.

This API does not start the timer. Timer needs to be started when required using the TIMER_D_start API.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for up mode initialization.

Modified bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.43 void TIMER_D_outputPWM (uint16_t *baseAddress*, TIMER_D_outputPWMParam ∗ *param*)

Generate a PWM with timer running in up mode.

Parameters:
baseAddress is the base address of the TIMER_D module.
param is the pointer to struct for PWM configuration.

Modified bits of **TDxCCTLn** register, bits of **TDxCCR0** register, bits of **TDxCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns:
None

34.2.2.44 void TIMER_D_selectCounterLength (uint16_t *baseAddress*, uint16_t *counterLength*)

Selects TIMER_D counter length.

Parameters:
baseAddress is the base address of the TIMER_D module.
counterLength selects the value of counter length. Valid values are:
- **TIMER_D_COUNTER_16BIT** [Default]
- **TIMER_D_COUNTER_12BIT**
- **TIMER_D_COUNTER_10BIT**
- **TIMER_D_COUNTER_8BIT**

Modified bits are **CNTL** of **TDxCTL0** register.

Returns:
None

34.2.2.45 void TIMER_D_selectHighResClockRange (uint16_t *baseAddress*, uint16_t *highResClockRange*)

Select High Resolution Clock Range Selection.

Parameters:
baseAddress is the base address of the TIMER_D module.
highResClockRange selects the High Resolution Clock Range. Refer to datasheet for frequency details Valid values are:
- **TIMER_D_CLOCK_RANGE0** [Default]
- **TIMER_D_CLOCK_RANGE1**
- **TIMER_D_CLOCK_RANGE2**

Returns:
None

34.2.2.46 void TIMER_D_selectHighResCoarseClockRange (uint16_t *baseAddress*, uint16_t *highResCoarseClockRange*)

Select High Resolution Coarse Clock Range.

Parameters:
baseAddress is the base address of the TIMER_D module.
highResCoarseClockRange selects the High Resolution Coarse Clock Range Valid values are:
- **TIMER_D_HIGHRES_BELOW_15MHz** [Default]
- **TIMER_D_HIGHRES_ABOVE_15MHz**

Modified bits are **TDHCLKCR** of **TDxHCTL1** register.

Returns:
None

34.2.2.47 void TIMER_D_selectLatchingGroup (uint16_t *baseAddress*, uint16_t *groupLatch*)

Selects TIMER_D Latching Group.

Parameters:
baseAddress is the base address of the TIMER_D module.
groupLatch selects the group latch Valid values are:
- **TIMER_D_GROUP_NONE** [Default]
- **TIMER_D_GROUP_CL12_CL23_CL56**
- **TIMER_D_GROUP_CL123_CL456**
- **TIMER_D_GROUP_ALL**

Modified bits are **TDCLGRP** of **TDxCTL0** register.

Returns:
None

34.2.2.48 void TIMER_D_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_D module.
compareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

compareValue is the count to be compared with in compare mode

Modified bits of **TDxCCRn** register.

Returns:
None

34.2.2.49 void TIMER_D_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_D module.
captureCompareRegister selects the Capture register being used. Valid values are:
- **TIMER_D_CAPTURECOMPARE_REGISTER_0**
- **TIMER_D_CAPTURECOMPARE_REGISTER_1**
- **TIMER_D_CAPTURECOMPARE_REGISTER_2**
- **TIMER_D_CAPTURECOMPARE_REGISTER_3**
- **TIMER_D_CAPTURECOMPARE_REGISTER_4**
- **TIMER_D_CAPTURECOMPARE_REGISTER_5**
- **TIMER_D_CAPTURECOMPARE_REGISTER_6**

outputModeOutBitValue the value to be set for out bit Valid values are:

- **TIMER_D_OUTPUTMODE_OUTBITVALUE_HIGH**
- **TIMER_D_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TDxCCTLn** register.

Returns:
None

34.2.2.50 void TIMER_D_startCounter (uint16_t *baseAddress*, uint16_t *timerMode*)

Starts TIMER_D counter.

NOTE: This function assumes that the timer has been previously configured using TIMER_D_configureContinuousMode, TIMER_D_configureUpMode or TIMER_D_configureUpDownMode.

Parameters:
baseAddress is the base address of the TIMER_DA module.
timerMode selects the mode of the timer Valid values are:
- **TIMER_D_STOP_MODE**
- **TIMER_D_UP_MODE**
- **TIMER_D_CONTINUOUS_MODE** [Default]
- **TIMER_D_UPDOWN_MODE**

Modified bits of **TDxCTL0** register.

Returns:
None

34.2.2.51 void TIMER_D_stop (uint16_t *baseAddress*)

Stops the timer.

Parameters:
baseAddress is the base address of the TIMER_D module.

Modified bits of **TDxCTL0** register.

Returns:
None

34.3 Programming Example

The following example shows some TimerD operations using the APIs

```
{       //Start TimerD
    TIMER_D_configureUpDownMode( TIMER_A1_BASE,
        TIMER_D_CLOCKSOURCE_SMCLK,
        TIMER_D_CLOCKSOURCE_DIVIDER_1,
        TIMER_PERIOD,
        TIMER_D_TAIE_INTERRUPT_DISABLE,
        TIMER_D_CCIE_CCR0_INTERRUPT_DISABLE,
        TIMER_D_DO_CLEAR
        );

    TIMER_D_startCounter( TIMER_A1_BASE,
                TIMER_D_UPDOWN_MODE
                );

    //Initialize compare registers to generate PWM1
    TIMER_D_initCompare(TIMER_A1_BASE,
        TIMER_D_CAPTURECOMPARE_REGISTER_1,
        TIMER_D_CAPTURECOMPARE_INTERRUPT_ENABLE,
        TIMER_D_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE1
        );
    //Initialize compare registers to generate PWM2
    TIMER_D_initCompare(TIMER_A1_BASE,
        TIMER_D_CAPTURECOMPARE_REGISTER_2,
        TIMER_D_CAPTURECOMPARE_INTERRUPT_DISABLE,
        TIMER_D_OUTPUTMODE_TOGGLE_SET,
        DUTY_CYCLE2
        );

    //Enter LPM0
    __bis_SR_register(LPM0_bits);

    //For debugger
    __no_operation();
}
```

35 Tag Length Value

35.1 Introduction

The TLV structure is a table stored in flash memory that contains device-specific information. This table is read-only and is write-protected. It contains important information for using and calibrating the device. A list of the contents of the TLV is available in the device-specific data sheet (in the Device Descriptors section), and an explanation on its functionality is available in the MSP430x5xx/MSP430x6xx Family User?s Guide

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	602
TI Compiler 4.2.1	Size	364
TI Compiler 4.2.1	Speed	368
IAR 5.51.6	None	456
IAR 5.51.6	Size	302
IAR 5.51.6	Speed	354
MSPGCC 4.8.0	None	750
MSPGCC 4.8.0	Size	432
MSPGCC 4.8.0	Speed	706

35.2 API Functions

Functions

- uint16_t TLV_getDeviceType ()
- void TLV_getInfo (uint8_t tag, uint8_t instance, uint8_t ∗length, uint16_t ∗∗data_address)
- uint8_t TLV_getInterrupt (uint8_t tag)
- uint16_t TLV_getMemory (uint8_t instance)
- uint16_t TLV_getPeripheral (uint8_t tag, uint8_t instance)

35.2.1 Detailed Description

The APIs that help in querying the information in the TLV structure are listed

- TLV_getInfo() This function retrieves the value of a tag and the length of the tag.
- TLV_getDeviceType() This function retrieves the unique device ID from the TLV structure.
- TLV_getMemory() The returned value is zero if the end of the memory list is reached.
- TLV_getPeripheral() The returned value is zero if the specified tag value (peripheral) is not available in the device.
- TLV_getInterrupt() The returned value is zero is the specified interrupt vector is not defined.

35.2.2 Function Documentation

35.2.2.1 uint16_t TLV_getDeviceType (void)

Retrieves the unique device ID from the TLV structure.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	16
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
The device ID is returned as type uint16_t.

35.2.2.2 void TLV_getInfo (uint8_t *tag*, uint8_t *instance*, uint8_t $*$ *length*, uint16_t $**$ *data_address*)

Gets TLV Info.

The TLV structure uses a tag or base address to identify segments of the table where information is stored. Some examples of TLV tags are Peripheral Descriptor, Interrupts, Info Block and Die Record. This function retrieves the value of a tag and the length of the tag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	126
TI Compiler 4.2.1	Size	72
TI Compiler 4.2.1	Speed	76
IAR 5.51.6	None	82
IAR 5.51.6	Size	70
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	168
MSPGCC 4.8.0	Size	100
MSPGCC 4.8.0	Speed	110

Parameters:
tag represents the tag for which the information needs to be retrieved. Valid values are:
- **TLV_TAG_LDTAG**
- **TLV_TAG_PDTAG**
- **TLV_TAG_Reserved3**
- **TLV_TAG_Reserved4**
- **TLV_TAG_BLANK**
- **TLV_TAG_Reserved6**
- **TLV_TAG_Reserved7**
- **TLV_TAG_TAGEND**
- **TLV_TAG_TAGEXT**
- **TLV_TAG_TIMER_D_CAL**

- **TLV_DEVICE_ID_0**
- **TLV_DEVICE_ID_1**
- **TLV_TAG_DIERECORD**
- **TLV_TAG_ADCCAL**
- **TLV_TAG_ADC12CAL**
- **TLV_TAG_ADC10CAL**
- **TLV_TAG_REFCAL**

instance In some cases a specific tag may have more than one instance. For example there may be multiple instances of timer calibration data present under a single Timer Cal tag. This variable specifies the instance for which information is to be retrieved (0, 1, etc.). When only one instance exists; 0 is passed.

length Acts as a return through indirect reference. The function retrieves the value of the TLV tag length. This value is pointed to by ∗length and can be used by the application level once the function is called. If the specified tag is not found then the pointer is null 0.

data_address acts as a return through indirect reference. Once the function is called data_address points to the pointer that holds the value retrieved from the specified TLV tag. If the specified tag is not found then the pointer is null 0.

Returns:
None

35.2.2.3 uint8_t TLV_getInterrupt (uint8_t *tag*)

Get interrupt information from the TLV.

This function is used to retrieve information on available interrupt vectors. It allows the user to check if a specific interrupt vector is defined in a given device.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	160
TI Compiler 4.2.1	Size	98
TI Compiler 4.2.1	Speed	98
IAR 5.51.6	None	128
IAR 5.51.6	Size	82
IAR 5.51.6	Speed	102
MSPGCC 4.8.0	None	200
MSPGCC 4.8.0	Size	106
MSPGCC 4.8.0	Speed	228

Parameters:

tag represents the tag for the interrupt vector. Interrupt vector tags number from 0 to N depending on the number of available interrupts. Refer to the device datasheet for a list of available interrupts.

Returns:
The returned value is zero is the specified interrupt vector is not defined.

35.2.2.4 uint16_t TLV_getMemory (uint8_t *instance*)

Gets memory information.

The Peripheral Descriptor tag is split into two portions a list of the available flash memory blocks followed by a list of available peripherals. This function is used to parse through the first portion and calculate the total flash memory available in a device. The typical usage is to call the TLV_getMemory which returns a non-zero value until the entire memory list has been parsed. When a zero is returned, it indicates that all the memory blocks have been counted and the next address holds the beginning of the device peripheral list.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	70
TI Compiler 4.2.1	Speed	70
IAR 5.51.6	None	92
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	128
MSPGCC 4.8.0	Size	86
MSPGCC 4.8.0	Speed	120

Parameters:

instance In some cases a specific tag may have more than one instance. This variable specifies the instance for which information is to be retrieved (0, 1 etc). When only one instance exists; 0 is passed.

Returns:

The returned value is zero if the end of the memory list is reached.

35.2.2.5 uint16_t TLV_getPeripheral (uint8_t *tag*, uint8_t *instance*)

Gets peripheral information from the TLV.

he Peripheral Descriptor tag is split into two portions a list of the available flash memory blocks followed by a list of available peripherals. This function is used to parse through the second portion and can be used to check if a specific peripheral is present in a device. The function calls TLV_getPeripheral() recursively until the end of the memory list and consequently the beginning of the peripheral list is reached. <

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	186
TI Compiler 4.2.1	Size	116
TI Compiler 4.2.1	Speed	116
IAR 5.51.6	None	146
IAR 5.51.6	Size	90
IAR 5.51.6	Speed	106
MSPGCC 4.8.0	None	238
MSPGCC 4.8.0	Size	134
MSPGCC 4.8.0	Speed	242

Parameters:

tag represents represents the tag for a specific peripheral for which the information needs to be retrieved. In the header file tlv. h specific peripheral tags are pre-defined, for example USCIA_B and TA0 are defined as TLV_PID_USCI_AB and TLV_PID_TA2 respectively. Valid values are:

- **TLV_PID_NO_MODULE** - No Module
- **TLV_PID_PORTMAPPING** - Port Mapping
- **TLV_PID_MSP430CPUXV2** - MSP430CPUXV2
- **TLV_PID_JTAG** - JTAG
- **TLV_PID_SBW** - SBW
- **TLV_PID_EEM_XS** - EEM X-Small
- **TLV_PID_EEM_S** - EEM Small
- **TLV_PID_EEM_M** - EEM Medium
- **TLV_PID_EEM_L** - EEM Large
- **TLV_PID_PMM** - PMM
- **TLV_PID_PMM_FR** - PMM FRAM
- **TLV_PID_FCTL** - Flash
- **TLV_PID_CRC16** - CRC16

- **TLV_PID_CRC16_RB** - CRC16 Reverse
- **TLV_PID_WDT_A** - WDT_A
- **TLV_PID_SFR** - SFR
- **TLV_PID_SYS** - SYS
- **TLV_PID_RAMCTL** - RAMCTL
- **TLV_PID_DMA_1** - DMA 1
- **TLV_PID_DMA_3** - DMA 3
- **TLV_PID_UCS** - UCS
- **TLV_PID_DMA_6** - DMA 6
- **TLV_PID_DMA_2** - DMA 2
- **TLV_PID_PORT1_2** - Port 1 + 2 / A
- **TLV_PID_PORT3_4** - Port 3 + 4 / B
- **TLV_PID_PORT5_6** - Port 5 + 6 / C
- **TLV_PID_PORT7_8** - Port 7 + 8 / D
- **TLV_PID_PORT9_10** - Port 9 + 10 / E
- **TLV_PID_PORT11_12** - Port 11 + 12 / F
- **TLV_PID_PORTU** - Port U
- **TLV_PID_PORTJ** - Port J
- **TLV_PID_TA2** - Timer A2
- **TLV_PID_TA3** - Timer A1
- **TLV_PID_TA5** - Timer A5
- **TLV_PID_TA7** - Timer A7
- **TLV_PID_TB3** - Timer B3
- **TLV_PID_TB5** - Timer B5
- **TLV_PID_TB7** - Timer B7
- **TLV_PID_RTC** - RTC
- **TLV_PID_BT_RTC** - BT + RTC
- **TLV_PID_BBS** - Battery Backup Switch
- **TLV_PID_RTC_B** - RTC_B
- **TLV_PID_TD2** - Timer D2
- **TLV_PID_TD3** - Timer D1
- **TLV_PID_TD5** - Timer D5
- **TLV_PID_TD7** - Timer D7
- **TLV_PID_TEC** - Timer Event Control
- **TLV_PID_RTC_C** - RTC_C
- **TLV_PID_AES** - AES
- **TLV_PID_MPY16** - MPY16
- **TLV_PID_MPY32** - MPY32
- **TLV_PID_MPU** - MPU
- **TLV_PID_USCI_AB** - USCI_AB
- **TLV_PID_USCI_A** - USCI_A
- **TLV_PID_USCI_B** - USCI_B
- **TLV_PID_EUSCI_A** - eUSCI_A
- **TLV_PID_EUSCI_B** - eUSCI_B
- **TLV_PID_REF** - Shared Reference
- **TLV_PID_COMP_B** - COMP_B
- **TLV_PID_COMP_D** - COMP_D
- **TLV_PID_USB** - USB
- **TLV_PID_LCD_B** - LCD_B
- **TLV_PID_LCD_C** - LCD_C
- **TLV_PID_DAC12_A** - DAC12_A
- **TLV_PID_SD16_B_1** - SD16_B 1 Channel
- **TLV_PID_SD16_B_2** - SD16_B 2 Channel
- **TLV_PID_SD16_B_3** - SD16_B 3 Channel
- **TLV_PID_SD16_B_4** - SD16_B 4 Channel

- **TLV_PID_SD16_B_5** - SD16_B 5 Channel
- **TLV_PID_SD16_B_6** - SD16_B 6 Channel
- **TLV_PID_SD16_B_7** - SD16_B 7 Channel
- **TLV_PID_SD16_B_8** - SD16_B 8 Channel
- **TLV_PID_ADC12_A** - ADC12_A
- **TLV_PID_ADC10_A** - ADC10_A
- **TLV_PID_ADC10_B** - ADC10_B
- **TLV_PID_SD16_A** - SD16_A
- **TLV_PID_TI_BSL** - BSL

instance In some cases a specific tag may have more than one instance. For example a device may have more than a single USCI module, each of which is defined by an instance number 0, 1, 2, etc. When only one instance exists; 0 is passed.

Returns:
The returned value is zero if the specified tag value (peripheral) is not available in the device.

35.3 Programming Example

The following example shows some tlv operations using the APIs

```
struct s_TLV_Die_Record * pDIEREC;
unsigned char bDieRecord_bytes;

TLV_getInfo(TLV_TAG_DIERECORD,
                        0,
                        &bDieRecord_bytes,
                        (unsigned int **)&pDIEREC
                        );
```

36 Unified Clock System (UCS)

36.1 Introduction

The UCS is based on five available clock sources (VLO, REFO, XT1, XT2, and DCO) 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.
- XT2 - Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 4-MHz to 32-MHz range, including digital inputs.
- 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.

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 MSP430x5xx 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 UCS, along with improved fail-safe logic, provides a robust clocking scheme for all applications.

Fail-Safe logic The UCS fail-safe logic plays an important part in providing a robust clocking scheme for MSP430x5xx and MSP430x6xx applications. This feature hinges on the ability to detect an oscillator fault for the XT1 in both low- and high-frequency modes (XT1LFOFFG and XT1HFOFFG respectively), the high-frequency XT2 (XT2OFFG), 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 5xx-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 MSP430x5xx/MSP430x6xx Family User?s Guide (SLAU208).

- Low-frequency crystal oscillator 1 (LFXT1) 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 LFXT1 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 LFXT1. As previously described, a fault on the LFXT1 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 LFXT1 (∼300 nA active).
- Modules (WDT_A) In watchdog mode, when SMCLK or ACLK fails, the clock source defaults to the VLOCLK.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1590
TI Compiler 4.2.1	Size	1160
TI Compiler 4.2.1	Speed	1168
IAR 5.51.6	None	1254
IAR 5.51.6	Size	808
IAR 5.51.6	Speed	1190
MSPGCC 4.8.0	None	3292
MSPGCC 4.8.0	Size	1320
MSPGCC 4.8.0	Speed	2216

36.2 API Functions

Functions

- void UCS_bypassXT1 (uint8_t highOrLowFrequency)
- bool UCS_bypassXT1WithTimeout (uint8_t highOrLowFrequency, uint16_t timeout)
- void UCS_bypassXT2 (void)
- bool UCS_bypassXT2WithTimeout (uint16_t timeout)
- uint16_t UCS_clearAllOscFlagsWithTimeout (uint16_t timeout)
- void UCS_clearFaultFlag (uint8_t mask)
- void UCS_clockSignalInit (uint8_t selectedClockSignal, uint16_t clockSource, uint16_t clockSourceDivider)
- void UCS_disableClockRequest (uint8_t selectClock)
- void UCS_enableClockRequest (uint8_t selectClock)
- uint8_t UCS_faultFlagStatus (uint8_t mask)
- uint32_t UCS_getACLK (void)
- uint32_t UCS_getMCLK (void)
- uint32_t UCS_getSMCLK (void)
- void UCS_HFXT1Start (uint16_t xt1drive)
- bool UCS_HFXT1StartWithTimeout (uint16_t xt1drive, uint16_t timeout)
- void UCS_initFLL (uint16_t fsystem, uint16_t ratio)
- void UCS_initFLLSettle (uint16_t fsystem, uint16_t ratio)
- void UCS_LFXT1Start (uint16_t xt1drive, uint8_t xcap)
- bool UCS_LFXT1StartWithTimeout (uint16_t xt1drive, uint8_t xcap, uint16_t timeout)
- void UCS_setExternalClockSource (uint32_t XT1CLK_frequency, uint32_t XT2CLK_frequency)
- void UCS_SMCLKOff (void)
- void UCS_SMCLKOn (void)
- void UCS_XT1Off (void)
- void UCS_XT2Off (void)
- void UCS_XT2Start (uint16_t xt2drive)
- bool UCS_XT2StartWithTimeout (uint16_t xt2drive, uint16_t timeout)

36.2.1 Detailed Description

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

General UCS configuration and initialization is handled by

- UCS_clockSignalInit(),

- UCS_initFLLSettle(),
- UCS_enableClockRequest(),
- UCS_disableClockRequest(),
- UCS_SMCLKOff(),
- UCS_SMCLKOn()

External crystal specific configuration and initialization is handled by

- UCS_setExternalClockSource(),
- UCS_LFXT1Start(),
- UCS_HFXT1Start(),
- UCS_bypassXT1(),
- UCS_LFXT1StartWithTimeout(),
- UCS_HFXT1StartWithTimeout(),
- UCS_bypassXT1WithTimeout(),
- UCS_XT1Off(),
- UCS_XT2Start(),
- UCS_XT2Off(),
- UCS_bypassXT2(),
- UCS_XT2StartWithTimeout(),
- UCS_bypassXT2WithTimeout()
- UCS_clearAllOscFlagsWithTimeout()

UCS_setExternalClockSource must be called if an external crystal XT1 or XT2 is used and the user intends to call UCS_getMCLK, UCS_getSMCLK or UCS_getACLK APIs. If not, it is not necessary to invoke this API.

Failure to invoke UCS_clockSignalInit() sets the clock signals to the default modes ACLK default mode - UCS_XT1CLK_SELECT SMCLK default mode - UCS_DCOCLKDIV_SELECT MCLK default mode - UCS_DCOCLKDIV_SELECT

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

The status and configuration query are done by

- UCS_faultFlagStatus(),
- UCS_clearFaultFlag(),
- UCS_getACLK(),
- UCS_getSMCLK(),
- UCS_getMCLK()

36.2.2 Function Documentation

36.2.2.1 void UCS_bypassXT1 (uint8_t *highOrLowFrequency*)

Bypass the XT1 crystal oscillator.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	66
TI Compiler 4.2.1	Size	54
TI Compiler 4.2.1	Speed	56
IAR 5.51.6	None	56
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	94
MSPGCC 4.8.0	None	188
MSPGCC 4.8.0	Size	62
MSPGCC 4.8.0	Speed	64

Parameters:

highOrLowFrequency selects high frequency or low frequency mode for XT1. Valid values are:
- **UCS_XT1_HIGH_FREQUENCY**
- **UCS_XT1_LOW_FREQUENCY** [Default]

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

None

36.2.2.2 bool UCS_bypassXT1WithTimeout (uint8_t *highOrLowFrequency*, 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	72
TI Compiler 4.2.1	Speed	72
IAR 5.51.6	None	78
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	220
MSPGCC 4.8.0	Size	82
MSPGCC 4.8.0	Speed	90

Parameters:

highOrLowFrequency selects high frequency or low frequency mode for XT1. Valid values are:
- **UCS_XT1_HIGH_FREQUENCY**
- **UCS_XT1_LOW_FREQUENCY** [Default]

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

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

STATUS_SUCCESS or STATUS_FAIL

36.2.2.3 void UCS_bypassXT2 (void)

Bypasses the XT2 crystal oscillator.

Bypasses the XT2 crystal oscillator, which supports crystal frequencies between 4 MHz and 32 MHz. Loops until all oscillator fault flags are cleared, with no timeout.

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	24
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	76
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	28

Returns:
None

36.2.2.4 bool UCS_bypassXT2WithTimeout (uint16_t *timeout*)

Bypasses the XT2 crystal oscillator with timeout.

Bypasses the XT2 crystal oscillator, which supports crystal frequencies between 4 MHz and 32 MHz. Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	48
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	40
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	36
MSPGCC 4.8.0	None	106
MSPGCC 4.8.0	Size	48
MSPGCC 4.8.0	Speed	48

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

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:
STATUS_SUCCESS or STATUS_FAIL

36.2.2.5 uint16_t UCS_clearAllOscFlagsWithTimeout (uint16_t *timeout*)

Clears all the Oscillator Flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	40
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	36
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	84
MSPGCC 4.8.0	Size	36
MSPGCC 4.8.0	Speed	36

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

Returns:
Logical OR of any of the following:

- **UCS_XT2OFFG** XT2 oscillator fault flag
- **UCS_XT1HFOFFG** XT1 oscillator fault flag (HF mode)
- **UCS_XT1LFOFFG** XT1 oscillator fault flag (LF mode)
- **UCS_DCOFFG** DCO fault flag
indicating the status of the oscillator fault flags

36.2.2.6 void UCS_clearFaultFlag (uint8_t *mask*)

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
mask is the masked interrupt flag status to be returned. mask parameter can be any one of the following Valid values are:

- **UCS_XT2OFFG** - XT2 oscillator fault flag
- **UCS_XT1HFOFFG** - XT1 oscillator fault flag (HF mode)
- **UCS_XT1LFOFFG** - XT1 oscillator fault flag (LF mode)
- **UCS_DCOFFG** - DCO fault flag

Modified bits of **UCSCTL7** register.

Returns:
None

36.2.2.7 void UCS_clockSignalInit (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. Refer MSP430Ware documentation for UCS module or Device Family User's Guide for details of default clock signal states.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	208
TI Compiler 4.2.1	Size	136
TI Compiler 4.2.1	Speed	144
IAR 5.51.6	None	150
IAR 5.51.6	Size	134
IAR 5.51.6	Speed	134
MSPGCC 4.8.0	None	498
MSPGCC 4.8.0	Size	168
MSPGCC 4.8.0	Speed	172

Parameters:

selectedClockSignal selected clock signal Valid values are:
- **UCS_ACLK**
- **UCS_MCLK**
- **UCS_SMCLK**
- **UCS_FLLREF**

clockSource is clock source for the selectedClockSignal Valid values are:
- **UCS_XT1CLK_SELECT**
- **UCS_VLOCLK_SELECT**
- **UCS_REFOCLK_SELECT**
- **UCS_DCOCLK_SELECT**
- **UCS_DCOCLKDIV_SELECT**
- **UCS_XT2CLK_SELECT**

clockSourceDivider selected the clock divider to calculate clocksignal from clock source. Valid values are:
- **UCS_CLOCK_DIVIDER_1** [Default]
- **UCS_CLOCK_DIVIDER_2**
- **UCS_CLOCK_DIVIDER_4**
- **UCS_CLOCK_DIVIDER_8**
- **UCS_CLOCK_DIVIDER_12** - [Valid only for UCS_FLLREF]
- **UCS_CLOCK_DIVIDER_16**
- **UCS_CLOCK_DIVIDER_32** - [Not valid for UCS_FLLREF]

Modified bits of **UCSCTL5** register, bits of **UCSCTL4** register and bits of **UCSCTL3** register.

Returns:

None

36.2.2.8 void UCS_disableClockRequest (uint8_t *selectClock*)

Disables conditional module requests.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	46
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

selectClock selects specific request disable Valid values are:

- **UCS_ACLK**
- **UCS_SMCLK**
- **UCS_MCLK**
- **UCS_MODOSC**

Modified bits of **UCSCTL8** register.

Returns:

None

36.2.2.9 void UCS_enableClockRequest (uint8_t *selectClock*)

Enables conditional module requests.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	32
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

selectClock selects specific request enables Valid values are:

- **UCS_ACLK**
- **UCS_SMCLK**
- **UCS_MCLK**
- **UCS_MODOSC**

Modified bits of **UCSCTL8** register.

Returns:

None

36.2.2.10 uint8_t UCS_faultFlagStatus (uint8_t *mask*)

Gets the current UCS fault flag status.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

mask is the masked interrupt flag status to be returned. Mask parameter can be either any of the following selection. Valid values are:
- **UCS_XT2OFFG** - XT2 oscillator fault flag
- **UCS_XT1HFOFFG** - XT1 oscillator fault flag (HF mode)
- **UCS_XT1LFOFFG** - XT1 oscillator fault flag (LF mode)
- **UCS_DCOFFG** - DCO fault flag

36.2.2.11 uint32_t UCS_getACLK (void)

Get the current ACLK frequency.

Get the current ACLK frequency. The user of this API must ensure that UCS_setExternalClockSource API was invoked before in case XT1 or XT2 is being used.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	50
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	42
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	286

Returns:

Current ACLK frequency in Hz

36.2.2.12 uint32_t UCS_getMCLK (void)

Get the current MCLK frequency.

Get the current MCLK frequency. The user of this API must ensure that UCS_setExternalClockSource API was invoked before in case XT1 or XT2 is being used.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	278

Returns:
Current MCLK frequency in Hz

36.2.2.13 uint32_t UCS_getSMCLK (void)

Get the current SMCLK frequency.

Get the current SMCLK frequency. The user of this API must ensure that UCS_setExternalClockSource API was invoked before in case XT1 or XT2 is being used.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	36
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	286

Returns:
Current SMCLK frequency in Hz

36.2.2.14 void UCS_HFXT1Start (uint16_t *xt1drive*)

Initializes the XT1 crystal oscillator in low 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	64
TI Compiler 4.2.1	Size	56
TI Compiler 4.2.1	Speed	56
IAR 5.51.6	None	56
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	68
MSPGCC 4.8.0	None	166
MSPGCC 4.8.0	Size	58
MSPGCC 4.8.0	Speed	60

Parameters:

xt1drive is the target drive strength for the XT1 crystal oscillator. Valid values are:

- **UCS_XT1_DRIVE0**
- **UCS_XT1_DRIVE1**
- **UCS_XT1_DRIVE2**
- **UCS_XT1_DRIVE3** [Default]

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

None

36.2.2.15 bool UCS_HFXT1StartWithTimeout (uint16_t *xt1drive*, 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 data sheet for appropriate drive settings.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	84
TI Compiler 4.2.1	Size	68
TI Compiler 4.2.1	Speed	68
IAR 5.51.6	None	72
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	60
MSPGCC 4.8.0	None	196
MSPGCC 4.8.0	Size	76
MSPGCC 4.8.0	Speed	70

Parameters:

xt1drive is the target drive strength for the XT1 crystal oscillator. Valid values are:

- **UCS_XT1_DRIVE0**
- **UCS_XT1_DRIVE1**
- **UCS_XT1_DRIVE2**
- **UCS_XT1_DRIVE3** [Default]

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

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

STATUS_SUCCESS or STATUS_FAIL

36.2.2.16 void UCS_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.

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 no timeout. If the frequency is greater than 16 MHz, the function sets the MCLK and SMCLK source to the undivided DCO frequency. Otherwise, the function sets the MCLK and SMCLK source to the DCOCLKDIV frequency.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	300
TI Compiler 4.2.1	Size	224
TI Compiler 4.2.1	Speed	224
IAR 5.51.6	None	228
IAR 5.51.6	Size	220
IAR 5.51.6	Speed	258
MSPGCC 4.8.0	None	496
MSPGCC 4.8.0	Size	264
MSPGCC 4.8.0	Speed	354

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 **UCSCTL0** register, bits of **UCSCTL4** register, bits of **UCSCTL7** register, bits of **UCSCTL1** register, bits of **SFRIFG1** register and bits of **UCSCTL2** register.

Returns:

None

36.2.2.17 void UCS_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 16 MHz, the function sets the MCLK and SMCLK source to the undivided DCO frequency. 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	78
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	58
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	86
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	90

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 **UCSCTL0** register, bits of **UCSCTL4** register, bits of **UCSCTL7** register, bits of **UCSCTL1** register, bits of **SFRIFG1** register and bits of **UCSCTL2** register.

Returns:

None

36.2.2.18 void UCS_LFXT1Start (uint16_t *xt1drive*, uint8_t *xcap*)

Initializes 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	70
TI Compiler 4.2.1	Speed	68
IAR 5.51.6	None	68
IAR 5.51.6	Size	34
IAR 5.51.6	Speed	82
MSPGCC 4.8.0	None	180
MSPGCC 4.8.0	Size	72
MSPGCC 4.8.0	Speed	74

Parameters:

xt1drive is the target drive strength for the XT1 crystal oscillator. Valid values are:

- **UCS_XT1_DRIVE0**
- **UCS_XT1_DRIVE1**
- **UCS_XT1_DRIVE2**
- **UCS_XT1_DRIVE3** [Default]
  Modified bits are **XT1DRIVE** of **UCSCTL6** register.

xcap is the selected capacitor value. This parameter selects the capacitors applied to the LF crystal (XT1) or resonator in the LF mode. The effective capacitance (seen by the crystal) is Ceff. (CXIN + 2 pF)/2. It is assumed that CXIN = CXOUT and that a parasitic capacitance of 2 pF is added by the package and the printed circuit board. For details about the typical internal and the effective capacitors, refer to the device-specific data sheet. Valid values are:

- **UCS_XCAP_0**
- **UCS_XCAP_1**
- **UCS_XCAP_2**
- **UCS_XCAP_3** [Default]

Modified bits are **XCAP** of **UCSCTL6** register.

Returns:

None

36.2.2.19 bool UCS_LFXT1StartWithTimeout (uint16_t *xt1drive*, uint8_t *xcap*, 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	82
TI Compiler 4.2.1	Speed	80
IAR 5.51.6	None	84
IAR 5.51.6	Size	40
IAR 5.51.6	Speed	74
MSPGCC 4.8.0	None	210
MSPGCC 4.8.0	Size	90
MSPGCC 4.8.0	Speed	84

Parameters:

xt1drive is the target drive strength for the XT1 crystal oscillator. Valid values are:
- **UCS_XT1_DRIVE0**
- **UCS_XT1_DRIVE1**
- **UCS_XT1_DRIVE2**
- **UCS_XT1_DRIVE3** [Default]

xcap is the selected capacitor value. This parameter selects the capacitors applied to the LF crystal (XT1) or resonator in the LF mode. The effective capacitance (seen by the crystal) is Ceff. (CXIN + 2 pF)/2. It is assumed that CXIN = CXOUT and that a parasitic capacitance of 2 pF is added by the package and the printed circuit board. For details about the typical internal and the effective capacitors, refer to the device-specific data sheet. Valid values are:
- **UCS_XCAP_0**
- **UCS_XCAP_1**
- **UCS_XCAP_2**
- **UCS_XCAP_3** [Default]

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

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

STATUS_SUCCESS or STATUS_FAIL

36.2.2.20 void UCS_setExternalClockSource (uint32_t *XT1CLK_frequency*, uint32_t *XT2CLK_frequency*)

Sets the external clock source.

This function sets the external clock sources XT1 and XT2 crystal oscillator frequency values. This function must be called if an external crystal XT1 or XT2 is used and the user intends to call UCS_getMCLK, UCS_getSMCLK or UCS_getACLK APIs. If not, it is not necessary to invoke this API.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	44
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	18
IAR 5.51.6	Size	18
IAR 5.51.6	Speed	18
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:

XT1CLK_frequency is the XT1 crystal frequencies in Hz

XT2CLK_frequency is the XT2 crystal frequencies in Hz

Returns:
None

36.2.2.21 void UCS_SMCLKOff (void)

Turns off SMCLK using the SMCLKOFF bit.

Modified bits of **UCSCTL6** register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

36.2.2.22 void UCS_SMCLKOn (void)

Turns ON SMCLK using the SMCLKOFF bit.

Modified bits of **UCSCTL6** register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

36.2.2.23 void UCS_XT1Off (void)

Stops the XT1 oscillator using the XT1OFF bit.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	6
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	18
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Returns:
None

36.2.2.24 void UCS_XT2Off (void)

Stops the XT2 oscillator using the XT2OFF bit.

Modified bits of **UCSCTL6** register.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	8
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Returns:
None

36.2.2.25 void UCS_XT2Start (uint16_t *xt2drive*)

Initializes the XT2 crystal oscillator.

Initializes the XT2 crystal oscillator, which supports crystal frequencies between 4 MHz and 32 MHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared, with no timeout. See the device-specific data sheet for appropriate drive settings.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	60
TI Compiler 4.2.1	Size	52
TI Compiler 4.2.1	Speed	52
IAR 5.51.6	None	52
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	36
MSPGCC 4.8.0	None	150
MSPGCC 4.8.0	Size	54
MSPGCC 4.8.0	Speed	56

Parameters:

xt2drive is the target drive strength for the XT2 crystal oscillator. Valid values are:
- **UCS_XT2DRIVE_4MHZ_8MHZ**
- **UCS_XT2DRIVE_8MHZ_16MHZ**
- **UCS_XT2DRIVE_16MHZ_24MHZ**
- **UCS_XT2DRIVE_24MHZ_32MHZ** [Default]

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

None

36.2.2.26 bool UCS_XT2StartWithTimeout (uint16_t *xt2drive*, uint16_t *timeout*)

Initializes the XT2 crystal oscillator with timeout.

Initializes the XT2 crystal oscillator, which supports crystal frequencies between 4 MHz and 32 MHz, depending on the selected drive strength. Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero. See the device-specific data sheet for appropriate drive settings.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	80
TI Compiler 4.2.1	Size	64
TI Compiler 4.2.1	Speed	64
IAR 5.51.6	None	68
IAR 5.51.6	Size	26
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	180
MSPGCC 4.8.0	Size	74
MSPGCC 4.8.0	Speed	72

Parameters:

xt2drive is the target drive strength for the XT2 crystal oscillator. Valid values are:
- **UCS_XT2DRIVE_4MHZ_8MHZ**
- **UCS_XT2DRIVE_8MHZ_16MHZ**
- **UCS_XT2DRIVE_16MHZ_24MHZ**
- **UCS_XT2DRIVE_24MHZ_32MHZ** [Default]

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

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns:

STATUS_SUCCESS or STATUS_FAIL

36.3 Programming Example

The following example shows some UCS operations using the APIs

```
// Set DCO FLL reference = REFO
  UCS_clockSignalInit(UCS_BASE,
                      UCS_FLLREF,
                      UCS_REFOCLK_SELECT,
                      UCS_CLOCK_DIVIDER_1
                       );
```

```
 // Set ACLK = REFO
UCS_clockSignalInit(UCS_BASE,
                      UCS_ACLK,
                      UCS_REFOCLK_SELECT,
                      UCS_CLOCK_DIVIDER_1
                       );

 // Set Ratio and Desired MCLK Frequency  and initialize DCO
 UCS_initFLLSettle(  UCS_BASE,
                       UCS_MCLK_DESIRED_FREQUENCY_IN_KHZ,
                       UCS_MCLK_FLLREF_RATIO
                     );

 //Verify if the Clock settings are as expected
 clockValue = UCS_getSMCLK (UCS_BASE);

 while(1);
```

37 USCI Universal Asynchronous Receiver/Transmitter (USCI_A_UART)

37.1 Introduction

The MSP430Ware library for USCI_A_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

The modes of operations supported by the USCI_A_UART and the library include

- USCI_A_UART mode
- Idle-line multiprocessor mode
- Address-bit multiprocessor mode
- USCI_A_UART mode with automatic baud-rate detection

In USCI_A_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.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	800
TI Compiler 4.2.1	Size	468
TI Compiler 4.2.1	Speed	468
IAR 5.51.6	None	620
IAR 5.51.6	Size	454
IAR 5.51.6	Speed	500
MSPGCC 4.8.0	None	1412
MSPGCC 4.8.0	Size	492
MSPGCC 4.8.0	Speed	726

37.2 API Functions

Functions

- void USCI_A_UART_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void USCI_A_UART_disable (uint16_t baseAddress)
- void USCI_A_UART_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void USCI_A_UART_enable (uint16_t baseAddress)
- void USCI_A_UART_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t USCI_A_UART_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t USCI_A_UART_getReceiveBufferAddressForDMA (uint16_t baseAddress)
- uint32_t USCI_A_UART_getTransmitBufferAddressForDMA (uint16_t baseAddress)
- bool USCI_A_UART_init (uint16_t baseAddress, USCI_A_UART_initParam ∗param)
- bool USCI_A_UART_initAdvance (uint16_t baseAddress, uint8_t selectClockSource, uint16_t clockPrescalar, uint8_t firstModReg, uint8_t secondModReg, uint8_t parity, uint8_t msborLsbFirst, uint8_t numberofStopBits, uint8_t uartMode, uint8_t overSampling)
- uint8_t USCI_A_UART_queryStatusFlags (uint16_t baseAddress, uint8_t mask)
- uint8_t USCI_A_UART_receiveData (uint16_t baseAddress)
- void USCI_A_UART_resetDormant (uint16_t baseAddress)
- void USCI_A_UART_setDormant (uint16_t baseAddress)
- void USCI_A_UART_transmitAddress (uint16_t baseAddress, uint8_t transmitAddress)
- void USCI_A_UART_transmitBreak (uint16_t baseAddress)
- void USCI_A_UART_transmitData (uint16_t baseAddress, uint8_t transmitData)

37.2.1 Detailed Description

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

The USCI_A_UART API is broken into three groups of functions: those that deal with configuration and control of the USCI_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 USCI_A_UART are handled by the

- USCI_A_UART_init()
- USCI_A_UART_initAdvance()
- USCI_A_UART_enable()
- USCI_A_UART_disable()
- USCI_A_UART_setDormant()
- USCI_A_UART_resetDormant()

Sending and receiving data via the USCI_A_UART is handled by the

- USCI_A_UART_transmitData()
- USCI_A_UART_receiveData()
- USCI_A_UART_transmitAddress()
- USCI_A_UART_transmitBreak()

Managing the USCI_A_UART interrupts and status are handled by the

- USCI_A_UART_enableInterrupt()
- USCI_A_UART_disableInterrupt()

- USCI_A_UART_getInterruptStatus()
- USCI_A_UART_clearInterruptFlag()
- USCI_A_UART_queryStatusFlags()

DMA related

- USCI_A_UART_getReceiveBufferAddressForDMA()
- USCI_A_UART_getTransmitBufferAddressForDMA()

37.2.2 Function Documentation

37.2.2.1 void USCI_A_UART_clearInterruptFlag (uint16_t *baseAddress*, uint8_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the USCI_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:

- **USCI_A_UART_RECEIVE_INTERRUPT_FLAG** - Receive interrupt flag
- **USCI_A_UART_TRANSMIT_INTERRUPT_FLAG** - Transmit interrupt flag

Modified bits of **UCAxIFG** register.

Returns:

None

37.2.2.2 void USCI_A_UART_disable (uint16_t *baseAddress*)

Disables the UART block.

This will disable operation of the UART block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI_A_UART module.

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

Returns:
None

37.2.2.3 void USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	70
TI Compiler 4.2.1	Size	30
TI Compiler 4.2.1	Speed	30
IAR 5.51.6	None	40
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	136
MSPGCC 4.8.0	Size	32
MSPGCC 4.8.0	Speed	32

Parameters:
baseAddress is the base address of the USCI_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:
- **USCI_A_UART_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable
- **USCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable

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

Returns:
None

37.2.2.4 void USCI_A_UART_enable (uint16_t *baseAddress*)

Enables the UART block.

This will enable operation of the UART block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI_A_UART module.

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

Returns:
None

37.2.2.5 void USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	32
IAR 5.51.6	Size	26
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	94
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:
baseAddress is the base address of the USCI_A_UART module.
mask is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following:

- **USCI_A_UART_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable
- **USCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable

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

Returns:
None

37.2.2.6 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the USCI_A_UART module.
mask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:
- **USCI_A_UART_RECEIVE_INTERRUPT_FLAG** - Receive interrupt flag
- **USCI_A_UART_TRANSMIT_INTERRUPT_FLAG** - Transmit interrupt flag

Modified bits of **UCAxIFG** register.

Returns:
Logical OR of any of the following:

- **USCI_A_UART_RECEIVE_INTERRUPT_FLAG** Receive interrupt flag
- **USCI_A_UART_TRANSMIT_INTERRUPT_FLAG** Transmit interrupt flag
indicating the status of the masked flags

37.2.2.7 uint32_t USCI_A_UART_getReceiveBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the USCI_A_UART module.

Returns:

Address of RX Buffer

37.2.2.8 uint32_t USCI_A_UART_getTransmitBufferAddressForDMA (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.

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the USCI_A_UART module.

Returns:

Address of TX Buffer

37.2.2.9 bool USCI_A_UART_init (uint16_t *baseAddress*, USCI_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 USCI_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`

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	248
TI Compiler 4.2.1	Size	148
TI Compiler 4.2.1	Speed	148
IAR 5.51.6	None	202
IAR 5.51.6	Size	158
IAR 5.51.6	Speed	158
MSPGCC 4.8.0	None	522
MSPGCC 4.8.0	Size	158
MSPGCC 4.8.0	Speed	266

Parameters:

baseAddress is the base address of the USCI_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

37.2.2.10 bool USCI_A_UART_initAdvance (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint16_t *clockPrescalar*, uint8_t *firstModReg*, uint8_t *secondModReg*, uint8_t *parity*, uint8_t *msborLsbFirst*, uint8_t *numberofStopBits*, uint8_t *uartMode*, uint8_t *overSampling*)

DEPRECATED - 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 USCI_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`

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	106
TI Compiler 4.2.1	Size	118
TI Compiler 4.2.1	Speed	118
IAR 5.51.6	None	108
IAR 5.51.6	Size	100
IAR 5.51.6	Speed	100
MSPGCC 4.8.0	None	122
MSPGCC 4.8.0	Size	98
MSPGCC 4.8.0	Speed	230

Parameters:

baseAddress is the base address of the USCI_A_UART module.

selectClockSource selects Clock source. Valid values are:
- **USCI_A_UART_CLOCKSOURCE_SMCLK**
- **USCI_A_UART_CLOCKSOURCE_ACLK**

clockPrescalar is the value to be written into UCBRx bits

firstModReg is First modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device Users Guide. This value is written into UCBRFx bits of UCAxMCTLW.

secondModReg is Second modulation stage register setting. This value is a pre-calculated value which can be obtained from the Device Users Guide. This value is written into UCBRSx bits of UCAxMCTLW.

parity is the desired parity. Valid values are:
- **USCI_A_UART_NO_PARITY** [Default]
- **USCI_A_UART_ODD_PARITY**
- **USCI_A_UART_EVEN_PARITY**

msborLsbFirst controls direction of receive and transmit shift register. Valid values are:
- **USCI_A_UART_MSB_FIRST**
- **USCI_A_UART_LSB_FIRST** [Default]

numberofStopBits indicates one/two STOP bits Valid values are:
- **USCI_A_UART_ONE_STOP_BIT** [Default]
- **USCI_A_UART_TWO_STOP_BITS**

uartMode selects the mode of operation Valid values are:
- **USCI_A_UART_MODE** [Default]
- **USCI_A_UART_IDLE_LINE_MULTI_PROCESSOR_MODE**
- **USCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR_MODE**
- **USCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION_MODE**

overSampling indicates low frequency or oversampling baud generation Valid values are:
- **USCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION**
- **USCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

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

37.2.2.11 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the USCI_A_UART module.
mask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:
- **USCI_A_UART_LISTEN_ENABLE**
- **USCI_A_UART_FRAMING_ERROR**
- **USCI_A_UART_OVERRUN_ERROR**
- **USCI_A_UART_PARITY_ERROR**
- **USCI_A_UART_BREAK_DETECT**
- **USCI_A_UART_RECEIVE_ERROR**
- **USCI_A_UART_ADDRESS_RECEIVED**
- **USCI_A_UART_IDLELINE**
- **USCI_A_UART_BUSY**

Modified bits of **UCAxSTAT** register.

Returns:
Logical OR of any of the following:

- **USCI_A_UART_LISTEN_ENABLE**
- **USCI_A_UART_FRAMING_ERROR**
- **USCI_A_UART_OVERRUN_ERROR**
- **USCI_A_UART_PARITY_ERROR**
- **USCI_A_UART_BREAK_DETECT**

- **USCI_A_UART_RECEIVE_ERROR**
- **USCI_A_UART_ADDRESS_RECEIVED**
- **USCI_A_UART_IDLELINE**
- **USCI_A_UART_BUSY**
  indicating the status of the masked interrupt flags

37.2.2.12 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	32
IAR 5.51.6	Size	32
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:
baseAddress is the base address of the USCI_A_UART module.

Modified bits of **UCAxRXBUF** register.

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

37.2.2.13 void USCI_A_UART_resetDormant (uint16_t *baseAddress*)

Re-enables UART module from dormant mode.

Not dormant. All received characters set UCRXIFG.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI_A_UART module.

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

Returns:
None

37.2.2.14 void USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI_A_UART module.

Modified bits of **UCAxCTL1** register.

Returns:
None

37.2.2.15 void USCI_A_UART_transmitAddress (uint16_t *baseAddress*, uint8_t *transmitAddress*)

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	18
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the USCI_A_UART module.
transmitAddress is the next byte to be transmitted

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

Returns:
None

37.2.2.16 void USCI_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, USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	58
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	44
IAR 5.51.6	None	72
IAR 5.51.6	Size	38
IAR 5.51.6	Speed	64
MSPGCC 4.8.0	None	98
MSPGCC 4.8.0	Size	56
MSPGCC 4.8.0	Speed	50

Parameters:

baseAddress is the base address of the USCI_A_UART module.

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

Returns:

None

37.2.2.17 void USCI_A_UART_transmitData (uint16_t *baseAddress*, uint8_t *transmitData*)

Transmits a byte from the UART Module.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	36
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	62
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:

baseAddress is the base address of the USCI_A_UART module.

transmitData data to be transmitted from the UART module

Modified bits of **UCAxTXBUF** register.

Returns:

None

37.3 Programming Example

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

```
if ( STATUS_FAIL == USCI_A_UART_init (USCI_A0_BASE,
                                 USCI_A_UART_CLOCKSOURCE_SMCLK,
                                 UCS_getSMCLK(UCS_BASE),
                                 BAUD_RATE,
                                 USCI_A_UART_NO_PARITY,
                                 USCI_A_UART_LSB_FIRST,
                                 USCI_A_UART_ONE_STOP_BIT,
                                 USCI_A_UART_MODE,
                                 USCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION ))
  {
        return;
  }

  //Enable USCI_A_UART module for operation
  USCI_A_UART_enable (USCI_A0_BASE);

  //Enable Receive Interrupt
  USCI_A_UART_enableInterrupt (USCI_A0_BASE,
                       UCRXIE);

  //Transmit data
  USCI_A_UART_transmitData(USCI_A0_BASE,
                  transmitData++
                    );

  // Enter LPM3, interrupts enabled
  __bis_SR_register(LPM3_bits + GIE);
  __no_operation();
}

//*****************************************************************************
//
// This is the USCI_A0 interrupt vector service routine.
//
//*****************************************************************************
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    // Vector 2 - RXIFG
     case 2:
        // Echo back RXed character, confirm TX buffer is ready first

        // USCI_A0 TX buffer ready?
        while (!USCI_A_UART_interruptStatus(USCI_A0_BASE,
                                     UCTXIFG)
               );

        //Receive echoed data
        receivedData = USCI_A_UART_receiveData(USCI_A0_BASE);

        //Transmit next data
        USCI_A_UART_transmitData(USCI_A0_BASE,
                          transmitData++
                            );

        break;
     default: break;
  }
```

```
}
```

38 USCI Synchronous Peripheral Interface (USCI_A_SPI)

38.1 Introduction

The Serial Peripheral Interface Bus or USCI_A_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 3-wire USCI_A_SPI communication

The USCI_A_SPI module can be configured as either a master or a slave device.

The USCI_A_SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	700
TI Compiler 4.2.1	Size	396
TI Compiler 4.2.1	Speed	392
IAR 5.51.6	None	506
IAR 5.51.6	Size	436
IAR 5.51.6	Speed	420
MSPGCC 4.8.0	None	1172
MSPGCC 4.8.0	Size	398
MSPGCC 4.8.0	Speed	404

38.2 API Functions

Functions

- void USCI_A_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint8_t clockPhase, uint8_t clockPolarity)
- void USCI_A_SPI_changeMasterClock (uint16_t baseAddress, USCI_A_SPI_changeMasterClockParam ∗param)
- void USCI_A_SPI_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void USCI_A_SPI_disable (uint16_t baseAddress)
- void USCI_A_SPI_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void USCI_A_SPI_enable (uint16_t baseAddress)
- void USCI_A_SPI_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t USCI_A_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t USCI_A_SPI_getReceiveBufferAddressForDMA (uint16_t baseAddress)
- uint32_t USCI_A_SPI_getTransmitBufferAddressForDMA (uint16_t baseAddress)
- bool USCI_A_SPI_initMaster (uint16_t baseAddress, USCI_A_SPI_initMasterParam ∗param)

- uint8_t USCI_A_SPI_isBusy (uint16_t baseAddress)
- void USCI_A_SPI_masterChangeClock (uint16_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- bool USCI_A_SPI_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t clockSourceFrequency, uint32_t desiredSpiClock, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
- uint8_t USCI_A_SPI_receiveData (uint16_t baseAddress)
- bool USCI_A_SPI_slaveInit (uint16_t baseAddress, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
- void USCI_A_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)

38.2.1 Detailed Description

To use the module as a master, the user must call USCI_A_SPI_masterInit() to configure the USCI_A_SPI Master. This is followed by enabling the USCI_A_SPI module using USCI_A_SPI_enable(). The interrupts are then enabled (if needed). **It** is recommended to enable the USCI_A_SPI module before enabling the interrupts. A data transmit is then initiated using USCI_A_SPI_transmitData() and then when the receive flag is set, the received data is read using USCI_A_SPI_receiveData() and this indicates that an RX/TX operation is complete.

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

The USCI_A_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 USCI_A_SPI module are managed by

- USCI_A_SPI_masterInit()
- USCI_A_SPI_slaveInit()
- USCI_A_SPI_disable()
- USCI_A_SPI_enable()
- USCI_A_SPI_masterChangeClock()
- USCI_A_SPI_isBusy()

Data handling is done by

- USCI_A_SPI_transmitData()
- USCI_A_SPI_receiveData()

Interrupts from the USCI_A_SPI module are managed using

- USCI_A_SPI_disableInterrupt()
- USCI_A_SPI_enableInterrupt()
- USCI_A_SPI_getInterruptStatus()
- USCI_A_SPI_clearInterruptFlag()

DMA related

- USCI_A_SPI_getReceiveBufferAddressForDMA()
- USCI_A_SPI_getTransmitBufferAddressForDMA()

38.2.2 Function Documentation

38.2.2.1 void USCI_A_SPI_changeClockPhasePolarity (uint16_t *baseAddress*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

Changes the SPI clock phase and polarity.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	56
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	40
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	128
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

baseAddress is the base address of the I2C Master module.

clockPhase is clock phase select. Valid values are:
- **USCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:
- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCCKPL** and **UCCKPH** of **UCAxCTL0** register.

Returns:

None

38.2.2.2 void USCI_A_SPI_changeMasterClock (uint16_t *baseAddress*, USCI_A_SPI_changeMasterClockParam ∗ *param*)

Initializes the SPI Master clock.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	54
TI Compiler 4.2.1	Size	40
TI Compiler 4.2.1	Speed	40
IAR 5.51.6	None	50
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	98
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

Parameters:

baseAddress is the base address of the I2C Master module.

param is the pointer to struct for master clock setting.

Modified bits of **UCAxBRW** register.

Returns:

None

38.2.2.3 void USCI_A_SPI_clearInterruptFlag (uint16_t *baseAddress*, uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the SPI module.
mask is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following:
- **USCI_A_SPI_TRANSMIT_INTERRUPT**
- **USCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns:
None

38.2.2.4 void USCI_A_SPI_disable (uint16_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI SPI module.

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

Returns:
None

38.2.2.5 void USCI_A_SPI_disableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Disables individual SPI interrupt sources.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the 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:
- **USCI_A_SPI_TRANSMIT_INTERRUPT**
- **USCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIE** register.

Returns:
None

38.2.2.6 void USCI_A_SPI_enable (uint16_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI SPI module.

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

Returns:
None

38.2.2.7 void USCI_A_SPI_enableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Enables individual SPI interrupt sources.

Enables the indicated SPI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. **Does not clear interrupt flags.**

Parameters:

baseAddress **is the base address of the 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:**

- **USCI_A_SPI_TRANSMIT_INTERRUPT**
- **USCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of UCAxIE register.

Returns:

None

38.2.2.8 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the SPI module.

mask is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following:

- **USCI_A_SPI_TRANSMIT_INTERRUPT**
- **USCI_A_SPI_RECEIVE_INTERRUPT**

Returns:

The current interrupt status as the mask of the set flags Return Logical OR of any of the following:

- **USCI_A_SPI_TRANSMIT_INTERRUPT**
- **USCI_A_SPI_RECEIVE_INTERRUPT**
  indicating the status of the masked interrupts

38.2.2.9 uint32_t USCI_A_SPI_getReceiveBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the SPI module.

Returns:
the address of the RX Buffer

38.2.2.10 uint32_t USCI_A_SPI_getTransmitBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the SPI module.

Returns:
the address of the TX Buffer

38.2.2.11 bool USCI_A_SPI_initMaster (uint16_t *baseAddress*, USCI_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 USCI_A_SPI_enable()

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	84
TI Compiler 4.2.1	Speed	84
IAR 5.51.6	None	104
IAR 5.51.6	Size	94
IAR 5.51.6	Speed	94
MSPGCC 4.8.0	None	230
MSPGCC 4.8.0	Size	90
MSPGCC 4.8.0	Speed	90

Parameters:

baseAddress is the base address of the I2C Master module.

param is the pointer to struct for master initialization.

Modified bits are **UCCKPH**, **UCCKPL**, **UC7BIT** and **UCMSB** of **UCAxCTL0** register; bits **UCSSELx** and **UCSWRST** of **UCAxCTL1** register.

Returns:

STATUS_SUCCESS

38.2.2.12 uint8_t USCI_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

Code Metrics:

Compiler	**Optimization**	**Code Size**
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the SPI module.

Returns:

USCI_A_SPI_BUSY if the SPI module transmitting or receiving is busy; otherwise, returns USCI_A_SPI_NOT_BUSY. Return one of the following:

- **USCI_A_SPI_BUSY**
- **USCI_A_SPI_NOT_BUSY**
  indicating if the USCI_A_SPI is busy

38.2.2.13 void USCI_A_SPI_masterChangeClock (uint16_t *baseAddress*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*)

DEPRECATED - Initializes the SPI Master clock.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	90
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	66
IAR 5.51.6	Size	58
IAR 5.51.6	Speed	42
MSPGCC 4.8.0	None	92
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	60

Parameters:

baseAddress is the base address of the I2C Master module.

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

Modified bits of **UCAxBRW** register.

Returns:

None

38.2.2.14 bool USCI_A_SPI_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*, uint8_t *msbFirst*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

DEPRECATED - 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 USCI_A_SPI_enable()

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	100
TI Compiler 4.2.1	Size	106
TI Compiler 4.2.1	Speed	106
IAR 5.51.6	None	98
IAR 5.51.6	Size	94
IAR 5.51.6	Speed	94
MSPGCC 4.8.0	None	118
MSPGCC 4.8.0	Size	88
MSPGCC 4.8.0	Speed	84

Parameters:

baseAddress is the base address of the I2C Master module.

selectClockSource selects Clock source. Valid values are:

- **USCI_A_SPI_CLOCKSOURCE_ACLK**
- **USCI_A_SPI_CLOCKSOURCE_SMCLK**

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

msbFirst controls the direction of the receive and transmit shift register. Valid values are:

- **USCI_A_SPI_MSB_FIRST**
- **USCI_A_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:

- **USCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:

- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCCKPH**, **UCCKPL**, **UC7BIT** and **UCMSB** of **UCAxCTL0** register; bits **UCSSELx** and **UCSWRST** of **UCAxCTL1** register.

Returns:
STATUS_SUCCESS

38.2.2.15 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the SPI module.

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

38.2.2.16 bool USCI_A_SPI_slaveInit (uint16_t *baseAddress*, uint8_t *msbFirst*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

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 USCI_A_SPI_enable()

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	64
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	40
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	134
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the SPI Slave module.

msbFirst controls the direction of the receive and transmit shift register. Valid values are:

- **USCI_A_SPI_MSB_FIRST**
- **USCI_A_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:

- **USCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:

- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCMSB**, **UCMST**, **UC7BIT**, **UCCKPL**, **UCCKPH** and **UCMODE** of **UCAxCTL0** register; bits **UCSWRST** of **UCAxCTL1** register.

Returns:

STATUS_SUCCESS

38.2.2.17 void USCI_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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the SPI module.

transmitData data to be transmitted from the SPI module

Returns:

None

38.3 Programming Example

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

```
//Initialize Master
returnValue = USCI_A_SPI_masterInit(USCI_A0_BASE, SMCLK, CLK_getSMClk(),
                              USCI_SPICLK, MSB_FIRST,
                              CLOCK_POLARITY_INACTIVITYHIGH
                           );

if(STATUS_FAIL == returnValue)
```

```
  {
    return;
  }

  //Enable USCI_A_SPI module
  USCI_A_SPI_enable(USCI_A0_BASE);

  //Enable Receive interrupt
  USCI_A_SPI_enableInterrupt(USCI_A0_BASE, UCRXIE);

  //Configure port pins to reset slave

  // Wait for slave to initialize
  __delay_cycles(100);

  // Initialize data values
  transmitData = 0x00;

  // USCI_A0 TX buffer ready?
  while (!USCI_A_SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

  //Transmit Data to slave
  USCI_A_SPI_transmitData(USCI_A0_BASE, transmitData);

  // CPU off, enable interrupts
  __bis_SR_register(LPM0_bits + GIE);
}

//******************************************************************************
//
// This is the USCI_B0 interrupt vector service routine.
//
//******************************************************************************
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    // Vector 2 - RXIFG
    case 2:
    // USCI_A0 TX buffer ready?
    while (!USCI_A_SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

    receiveData = USCI_A_SPI_receiveData(USCI_A0_BASE);

    // Increment data
    transmitData++;

    // Send next value
    USCI_A_SPI_transmitData(USCI_A0_BASE, transmitData);

    //Delay between transmissions for slave to process information
    __delay_cycles(40);

    break;
     default: break;
  }
}
```

39 USCI Synchronous Peripheral Interface (USCI_B_SPI)

39.1 Introduction

The Serial Peripheral Interface Bus or USCI_B_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 3-wire USCI_B_SPI communication

The USCI_B_SPI module can be configured as either a master or a slave device.

The USCI_B_SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	694
TI Compiler 4.2.1	Size	392
TI Compiler 4.2.1	Speed	388
IAR 5.51.6	None	498
IAR 5.51.6	Size	428
IAR 5.51.6	Speed	412
MSPGCC 4.8.0	None	1162
MSPGCC 4.8.0	Size	392
MSPGCC 4.8.0	Speed	392

39.2 API Functions

Functions

- void USCI_B_SPI_changeClockPhasePolarity (uint16_t baseAddress, uint8_t clockPhase, uint8_t clockPolarity)
- void USCI_B_SPI_changeMasterClock (uint16_t baseAddress, USCI_B_SPI_changeMasterClockParam ∗param)
- void USCI_B_SPI_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void USCI_B_SPI_disable (uint16_t baseAddress)
- void USCI_B_SPI_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void USCI_B_SPI_enable (uint16_t baseAddress)
- void USCI_B_SPI_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t USCI_B_SPI_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t USCI_B_SPI_getReceiveBufferAddressForDMA (uint16_t baseAddress)
- uint32_t USCI_B_SPI_getTransmitBufferAddressForDMA (uint16_t baseAddress)
- bool USCI_B_SPI_initMaster (uint16_t baseAddress, USCI_B_SPI_initMasterParam ∗param)

- uint8_t USCI_B_SPI_isBusy (uint16_t baseAddress)
- void USCI_B_SPI_masterChangeClock (uint16_t baseAddress, uint32_t clockSourceFrequency, uint32_t desiredSpiClock)
- bool USCI_B_SPI_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t clockSourceFrequency, uint32_t desiredSpiClock, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
- uint8_t USCI_B_SPI_receiveData (uint16_t baseAddress)
- bool USCI_B_SPI_slaveInit (uint16_t baseAddress, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
- void USCI_B_SPI_transmitData (uint16_t baseAddress, uint8_t transmitData)

39.2.1 Detailed Description

To use the module as a master, the user must call USCI_B_SPI_masterInit() to configure the USCI_B_SPI Master. This is followed by enabling the USCI_B_SPI module using USCI_B_SPI_enable(). The interrupts are then enabled (if needed). **It** is recommended to enable the USCI_B_SPI module before enabling the interrupts. A data transmit is then initiated using USCI_B_SPI_transmitData() and then when the receive flag is set, the received data is read using USCI_B_SPI_receiveData() and this indicates that an RX/TX operation is complete.

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

The USCI_B_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 USCI_B_SPI module are managed by

- USCI_B_SPI_masterInit()
- USCI_B_SPI_slaveInit()
- USCI_B_SPI_disable()
- USCI_B_SPI_enable()
- USCI_B_SPI_masterChangeClock()
- USCI_B_SPI_isBusy()

Data handling is done by

- USCI_B_SPI_transmitData()
- USCI_B_SPI_receiveData()

Interrupts from the USCI_B_SPI module are managed using

- USCI_B_SPI_disableInterrupt()
- USCI_B_SPI_enableInterrupt()
- USCI_B_SPI_getInterruptStatus()
- USCI_B_SPI_clearInterruptFlag()

DMA related

- USCI_B_SPI_getReceiveBufferAddressForDMA()
- USCI_B_SPI_getTransmitBufferAddressForDMA()

39.2.2 Function Documentation

39.2.2.1 void USCI_B_SPI_changeClockPhasePolarity (uint16_t *baseAddress*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

Changes the SPI clock phase and polarity.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	56
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	40
IAR 5.51.6	Size	30
IAR 5.51.6	Speed	30
MSPGCC 4.8.0	None	128
MSPGCC 4.8.0	Size	26
MSPGCC 4.8.0	Speed	26

Parameters:

baseAddress is the base address of the I2C Master module.

clockPhase is clock phase select. Valid values are:

- **USCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:

- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCCKPL** and **UCCKPH** of **UCAxCTL0** register.

Returns:

None

39.2.2.2 void USCI_B_SPI_changeMasterClock (uint16_t *baseAddress*, USCI_B_SPI_changeMasterClockParam ∗ *param*)

Initializes the SPI Master clock.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	54
TI Compiler 4.2.1	Size	40
TI Compiler 4.2.1	Speed	40
IAR 5.51.6	None	50
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	46
MSPGCC 4.8.0	None	98
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

Parameters:

baseAddress is the base address of the I2C Master module.

param is the pointer to struct for master clock setting.

Modified bits of **UCAxBRW** register.

Returns:

None

39.2.2.3 void USCI_B_SPI_clearInterruptFlag (uint16_t *baseAddress*, uint8_t *mask*)

Clears the selected SPI interrupt status flag.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the SPI module.
mask is the masked interrupt flag to be cleared. Valid values are:
- **USCI_B_SPI_TRANSMIT_INTERRUPT**
- **USCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCBxIFG** register.

Returns:
None

39.2.2.4 void USCI_B_SPI_disable (uint16_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI SPI module.

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

Returns:
None

39.2.2.5 void USCI_B_SPI_disableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Disables individual SPI interrupt sources.

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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the SPI module.

mask is the bit mask of the interrupt sources to be disabled. Valid values are:
- **USCI_B_SPI_TRANSMIT_INTERRUPT**
- **USCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCBxIE** register.

Returns:

None

39.2.2.6 void USCI_B_SPI_enable (uint16_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the USCI SPI module.

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

Returns:

None

39.2.2.7 void USCI_B_SPI_enableInterrupt (uint16_t *baseAddress*, uint8_t *mask*)

Enables individual SPI interrupt sources.

Enables the indicated SPI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. **Does not clear interrupt flags.**

Parameters:

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

mask **is the bit mask of the interrupt sources to be enabled. Valid values are:**

- **USCI_B_SPI_TRANSMIT_INTERRUPT**
- **USCI_B_SPI_RECEIVE_INTERRUPT**

Modified bits of UCBxIE register.

Returns:

None

39.2.2.8 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the SPI module.

mask is the masked interrupt flag status to be returned. Valid values are:

- **USCI_B_SPI_TRANSMIT_INTERRUPT**
- **USCI_B_SPI_RECEIVE_INTERRUPT**

Returns:

The current interrupt status as the mask of the set flags Return Logical OR of any of the following:

- **USCI_B_SPI_TRANSMIT_INTERRUPT**
- **USCI_B_SPI_RECEIVE_INTERRUPT**
  indicating the status of the masked interrupts

39.2.2.9 uint32_t USCI_B_SPI_getReceiveBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the SPI module.

Returns:
The address of the SPI RX buffer

39.2.2.10 uint32_t USCI_B_SPI_getTransmitBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the SPI module.

Returns:
The address of the SPI TX buffer

39.2.2.11 bool USCI_B_SPI_initMaster (uint16_t *baseAddress*, USCI_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 USCI_B_SPI_enable()

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	108
TI Compiler 4.2.1	Size	80
TI Compiler 4.2.1	Speed	80
IAR 5.51.6	None	96
IAR 5.51.6	Size	86
IAR 5.51.6	Speed	86
MSPGCC 4.8.0	None	220
MSPGCC 4.8.0	Size	84
MSPGCC 4.8.0	Speed	84

Parameters:

baseAddress is the base address of the I2C Master module.

param is the pointer to struct for master initialization.

Modified bits are **UCSSELx** and **UCSWRST** of **UCBxCTL1** register; bits **UCCKPH**, **UCCKPL**, **UC7BIT** and **UCMSB** of **UCBxCTL0** register.

Returns:

STATUS_SUCCESS

39.2.2.12 uint8_t USCI_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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	10
TI Compiler 4.2.1	Speed	10
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the SPI module.

Returns:

USCI_B_SPI_BUSY if the SPI module transmitting or receiving is busy; otherwise, returns USCI_B_SPI_NOT_BUSY. Return one of the following:

- **USCI_B_SPI_BUSY**
- **USCI_B_SPI_NOT_BUSY**
  indicating if the USCI_B_SPI is busy

39.2.2.13 void USCI_B_SPI_masterChangeClock (uint16_t *baseAddress*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*)

DEPRECATED - Initializes the SPI Master clock.At the end of this function call, SPI module is left enabled.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	90
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	66
IAR 5.51.6	Size	58
IAR 5.51.6	Speed	42
MSPGCC 4.8.0	None	92
MSPGCC 4.8.0	Size	50
MSPGCC 4.8.0	Speed	60

Parameters:

baseAddress is the base address of the I2C Master module.

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

Modified bits of **UCAxBRW** register.

Returns:

None

39.2.2.14 bool USCI_B_SPI_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *clockSourceFrequency*, uint32_t *desiredSpiClock*, uint8_t *msbFirst*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

DEPRECATED - 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 USCI_B_SPI_enable()

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	100
TI Compiler 4.2.1	Size	106
TI Compiler 4.2.1	Speed	106
IAR 5.51.6	None	98
IAR 5.51.6	Size	94
IAR 5.51.6	Speed	94
MSPGCC 4.8.0	None	118
MSPGCC 4.8.0	Size	88
MSPGCC 4.8.0	Speed	78

Parameters:

baseAddress is the base address of the I2C Master module.

selectClockSource selects Clock source. Valid values are:

- **USCI_B_SPI_CLOCKSOURCE_ACLK**
- **USCI_B_SPI_CLOCKSOURCE_SMCLK**

clockSourceFrequency is the frequency of the selected clock source

desiredSpiClock is the desired clock rate for SPI communication

msbFirst controls the direction of the receive and transmit shift register. Valid values are:

- **USCI_B_SPI_MSB_FIRST**
- **USCI_B_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:

- **USCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:

- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCSSELx** and **UCSWRST** of **UCBxCTL1** register; bits **UCCKPH**, **UCCKPL**, **UC7BIT** and **UCMSB** of **UCBxCTL0** register.

Returns:
STATUS_SUCCESS

39.2.2.15 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the SPI module.

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

39.2.2.16 bool USCI_B_SPI_slaveInit (uint16_t *baseAddress*, uint8_t *msbFirst*, uint8_t *clockPhase*, uint8_t *clockPolarity*)

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 USCI_B_SPI_enable()

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	64
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	40
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	28
MSPGCC 4.8.0	None	134
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the SPI Slave module.

msbFirst controls the direction of the receive and transmit shift register. Valid values are:

- **USCI_B_SPI_MSB_FIRST**
- **USCI_B_SPI_LSB_FIRST** [Default]

clockPhase is clock phase select. Valid values are:

- **USCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **USCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

clockPolarity Valid values are:

- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Modified bits are **UCSWRST** of **UCBxCTL1** register; bits **UCMSB**, **UCMST**, **UC7BIT**, **UCCKPL**, **UCCKPH** and **UCMODE** of **UCBxCTL0** register.

Returns:

STATUS_SUCCESS

39.2.2.17 void USCI_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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the SPI module.

transmitData data to be transmitted from the SPI module

Returns:

None

39.3 Programming Example

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

```
//Initialize Master
returnValue = USCI_B_SPI_masterInit(USCI_A0_BASE, SMCLK, CLK_getSMClk(),
                               USCI_SPICLK, MSB_FIRST,
                               CLOCK_POLARITY_INACTIVITYHIGH
                            );

if(STATUS_FAIL == returnValue)
```

```
  {
    return;
  }

  //Enable USCI_B_SPI module
  USCI_B_SPI_enable(USCI_A0_BASE);

  //Enable Receive interrupt
  USCI_B_SPI_enableInterrupt(USCI_A0_BASE, UCRXIE);

  //Configure port pins to reset slave

  // Wait for slave to initialize
  __delay_cycles(100);

  // Initialize data values
  transmitData = 0x00;

  // USCI_A0 TX buffer ready?
  while (!USCI_B_SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

  //Transmit Data to slave
  USCI_B_SPI_transmitData(USCI_A0_BASE, transmitData);

  // CPU off, enable interrupts
  __bis_SR_register(LPM0_bits + GIE);
}

//******************************************************************************
//
// This is the USCI_B0 interrupt vector service routine.
//
//******************************************************************************
#pragma vector=USCI_B0_VECTOR
__interrupt void USCI_B0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    // Vector 2 - RXIFG
    case 2:
    // USCI_A0 TX buffer ready?
    while (!USCI_B_SPI_interruptStatus(USCI_A0_BASE, UCTXIFG));

    receiveData = USCI_B_SPI_receiveData(USCI_A0_BASE);

    // Increment data
    transmitData++;

    // Send next value
    USCI_B_SPI_transmitData(USCI_A0_BASE, transmitData);

    //Delay between transmissions for slave to process information
    __delay_cycles(40);

    break;
     default: break;
  }
}
```

40 USCI Inter-Integrated Circuit (USCI_B_I2C)

40.1 Introduction

The Inter-Integrated Circuit (USCI_B_I2C) API provides a set of functions for using the MSP430Ware USCI_B_I2C modules. Functions are provided to initialize the USCI_B_I2C modules, to send and receive data, obtain status, and to manage interrupts for the USCI_B_I2C modules.

The USCI_B_I2C module provide the ability to communicate to other IC devices over an USCI_B_I2C bus. The USCI_B_I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the USCI_B_I2C bus can be designated as either a master or a slave. The MSP430Ware USCI_B_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. Finally, the MSP430Ware USCI_B_I2C modules can operate at two speeds: Standard (100 kb/s) and Fast (400 kb/s).

USCI_B_I2C module can generate interrupts. The USCI_B_I2C module configured as a master will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The USCI_B_I2C module configured as a slave will generate interrupts when data has been sent or requested by a master.

40.1.1 Master Operations

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

- **USCI_B_I2C_masterInit()**
- **USCI_B_I2C_setSlaveAddress()**
- **USCI_B_I2C_setMode()**
- **USCI_B_I2C_enable()**
- **USCI_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 USCI_B_I2C module and configure it as a master with a call to USCI_B_I2C_masterInit(). 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 USCI_B_I2C_setSlaveAddress. Then the mode of operation (transmit or receive) is chosen using USCI_B_I2C_setMode. The USCI_B_I2C module may now be enabled using USCI_B_I2C_enable. It is recommended to enable the USCI_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. APIs that include a time-out can be used to avoid being stuck in an infinite loop if the device is stuck waiting for an IFG flag to be set.

Master Single Byte Transmission

- USCI_B_I2C_masterSendSingleByte()

Master Multiple Byte Transmission

- USCI_B_I2C_masterMultiByteSendStart()
- USCI_B_I2C_masterMultiByteSendNext()
- USCI_B_I2C_masterMultiByteSendFinish()
- USCI_B_I2C_masterMultiByteSendStop()

Master Single Byte Reception

- USCI_B_I2C_masterSingleReceiveStart()

- USCI_B_I2C_masterSingleReceive()

Master Multiple Byte Reception

- USCI_B_I2C_masterMultiByteReceiveStart()
- USCI_B_I2C_masterMultiByteReceiveNext()
- USCI_B_I2C_masterMultiByteReceiveFinish()
- USCI_B_I2C_masterMultiByteReceiveStop()

Master Single Byte Transmission with Time-out

- USCI_B_I2C_masterSendSingleByteWithTimeout()

Master Multiple Byte Transmission with Time-out

- USCI_B_I2C_masterMultiByteSendStartWithTimeout()
- USCI_B_I2C_masterMultiByteSendNextWithTimeout()
- USCI_B_I2C_masterMultiByteReceiveFinishWithTimeout()
- USCI_B_I2C_masterMultiByteSendStopWithTimeout()

Master Single Byte Reception with Time-out USCI_B_I2C_masterSingleReceiveStartWithTimeout()

For the interrupt-driven transaction, the user must register an interrupt handler for the USCI_B_I2C devices and enable the USCI_B_I2C interrupt.

40.1.2 Slave Operations

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

- **USCI_B_I2C_slaveInit()**
- **USCI_B_I2C_setMode()**
- **USCI_B_I2C_enable()**
- **USCI_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 USCI_B_I2C_slaveInit to initialize the slave module in USCI_B_I2C mode and set the slave address. This is followed by a call to set the mode of operation (transmit or receive).The USCI_B_I2C module may now be enabled using USCI_B_I2C_enable() It is recommended to enable the USCI_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.

Slave Transmission API

- USCI_B_I2C_slaveDataPut()

Slave Reception API

- USCI_B_I2C_slaveDataGet()

For the interrupt-driven transaction, the user must register an interrupt handler for the USCI_B_I2C devices and enable the USCI_B_I2C interrupt.

This driver is contained in `usci_b_i2c.c`, with `usci_b_i2c.h` containing the API definitions for use by applications.

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	1818
TI Compiler 4.2.1	Size	1014
TI Compiler 4.2.1	Speed	1014
IAR 5.51.6	None	1428
IAR 5.51.6	Size	984
IAR 5.51.6	Speed	1160
MSPGCC 4.8.0	None	3016
MSPGCC 4.8.0	Size	1174
MSPGCC 4.8.0	Speed	1146

40.2 API Functions

Functions

- void USCI_B_I2C_clearInterruptFlag (uint16_t baseAddress, uint8_t mask)
- void USCI_B_I2C_disable (uint16_t baseAddress)
- void USCI_B_I2C_disableInterrupt (uint16_t baseAddress, uint8_t mask)
- void USCI_B_I2C_enable (uint16_t baseAddress)
- void USCI_B_I2C_enableInterrupt (uint16_t baseAddress, uint8_t mask)
- uint8_t USCI_B_I2C_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
- uint32_t USCI_B_I2C_getReceiveBufferAddressForDMA (uint16_t baseAddress)
- uint32_t USCI_B_I2C_getTransmitBufferAddressForDMA (uint16_t baseAddress)
- void USCI_B_I2C_initMaster (uint16_t baseAddress, USCI_B_I2C_initMasterParam ∗param)
- uint8_t USCI_B_I2C_isBusBusy (uint16_t baseAddress)
- uint8_t USCI_B_I2C_isBusy (uint16_t baseAddress)
- void USCI_B_I2C_masterInit (uint16_t baseAddress, uint8_t selectClockSource, uint32_t i2cClk, uint32_t dataRate)
- uint8_t USCI_B_I2C_masterIsStartSent (uint16_t baseAddress)
- uint8_t USCI_B_I2C_masterIsStopSent (uint16_t baseAddress)
- uint8_t USCI_B_I2C_masterMultiByteReceiveFinish (uint16_t baseAddress)
- bool USCI_B_I2C_masterMultiByteReceiveFinishWithTimeout (uint16_t baseAddress, uint8_t ∗rxData, uint32_t timeout)
- uint8_t USCI_B_I2C_masterMultiByteReceiveNext (uint16_t baseAddress)
- void USCI_B_I2C_masterMultiByteReceiveStart (uint16_t baseAddress)
- void USCI_B_I2C_masterMultiByteReceiveStop (uint16_t baseAddress)
- void USCI_B_I2C_masterMultiByteSendFinish (uint16_t baseAddress, uint8_t txData)
- bool USCI_B_I2C_masterMultiByteSendFinishWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void USCI_B_I2C_masterMultiByteSendNext (uint16_t baseAddress, uint8_t txData)
- bool USCI_B_I2C_masterMultiByteSendNextWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void USCI_B_I2C_masterMultiByteSendStart (uint16_t baseAddress, uint8_t txData)
- bool USCI_B_I2C_masterMultiByteSendStartWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void USCI_B_I2C_masterMultiByteSendStop (uint16_t baseAddress)
- bool USCI_B_I2C_masterMultiByteSendStopWithTimeout (uint16_t baseAddress, uint32_t timeout)
- void USCI_B_I2C_masterSendSingleByte (uint16_t baseAddress, uint8_t txData)
- bool USCI_B_I2C_masterSendSingleByteWithTimeout (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- void USCI_B_I2C_masterSendStart (uint16_t baseAddress)
- uint8_t USCI_B_I2C_masterSingleReceive (uint16_t baseAddress)
- void USCI_B_I2C_masterSingleReceiveStart (uint16_t baseAddress)
- bool USCI_B_I2C_masterSingleReceiveStartWithTimeout (uint16_t baseAddress, uint32_t timeout)
- void USCI_B_I2C_setMode (uint16_t baseAddress, uint8_t mode)
- void USCI_B_I2C_setSlaveAddress (uint16_t baseAddress, uint8_t slaveAddress)
- uint8_t USCI_B_I2C_slaveDataGet (uint16_t baseAddress)
- void USCI_B_I2C_slaveDataPut (uint16_t baseAddress, uint8_t transmitData)
- void USCI_B_I2C_slaveInit (uint16_t baseAddress, uint8_t slaveAddress)

40.2.1 Detailed Description

The USCI_B_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 USCI_B_I2C master and slave interrupts and status are handled by

- USCI_B_I2C_enableInterrupt()
- USCI_B_I2C_disableInterrupt()
- USCI_B_I2C_clearInterruptFlag()
- USCI_B_I2C_getInterruptStatus()
- USCI_B_I2C_masterIsStopSent()
- USCI_B_I2C_masterIsStartSent()

Status and initialization functions for the USCI_B_I2C modules are

- USCI_B_I2C_masterInit()
- USCI_B_I2C_enable()
- USCI_B_I2C_disable()
- USCI_B_I2C_isBusBusy()
- USCI_B_I2C_isBusy()
- USCI_B_I2C_slaveInit()
- USCI_B_I2C_interruptStatus()
- USCI_B_I2C_setSlaveAddress()
- USCI_B_I2C_setMode()

Sending and receiving data from the USCI_B_I2C slave module is handled by

- USCI_B_I2C_slaveDataPut()
- USCI_B_I2C_slaveDataGet()

Sending and receiving data from the USCI_B_I2C slave module is handled by

- USCI_B_I2C_masterSendSingleByte()
- USCI_B_I2C_masterMultiByteSendStart()
- USCI_B_I2C_masterMultiByteSendNext()
- USCI_B_I2C_masterMultiByteSendFinish()
- USCI_B_I2C_masterMultiByteSendStop()
- USCI_B_I2C_masterMultiByteReceiveStart()
- USCI_B_I2C_masterMultiByteReceiveNext()
- USCI_B_I2C_masterMultiByteReceiveFinish()
- USCI_B_I2C_masterMultiByteReceiveStop()
- USCI_B_I2C_masterSingleReceiveStart()
- USCI_B_I2C_masterSingleReceive()
- USCI_B_I2C_getReceiveBufferAddressForDMA()
- USCI_B_I2C_getTransmitBufferAddressForDMA()

DMA related

- USCI_B_I2C_getReceiveBufferAddressForDMA()
- USCI_B_I2C_getTransmitBufferAddressForDMA()

40.2.2 Function Documentation

40.2.2.1 void USCI_B_I2C_clearInterruptFlag (uint16_t *baseAddress*, uint8_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the I2C Slave module.

mask is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following:

- **USCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **USCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **USCI_B_I2C_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_B_I2C_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **USCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt

Modified bits of **UCBxIFG** register.

Returns:

None

40.2.2.2 void USCI_B_I2C_disable (uint16_t *baseAddress*)

Disables the I2C block.

This will disable operation of the I2C block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:

baseAddress is the base address of the USCI I2C module.

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

Returns:
None

40.2.2.3 void USCI_B_I2C_disableInterrupt (uint16_t *baseAddress*, uint8_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	60
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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:

- **USCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **USCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **USCI_B_I2C_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_B_I2C_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **USCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt

Modified bits of **UCBxIE** register.

Returns:
None

40.2.2.4 void USCI_B_I2C_enable (uint16_t *baseAddress*)

Enables the I2C block.

This will enable operation of the I2C block.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the USCI I2C module.

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

Returns:
None

40.2.2.5 void USCI_B_I2C_enableInterrupt (uint16_t *baseAddress*, uint8_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. Does not clear interrupt flags.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	4
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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:
- **USCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **USCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **USCI_B_I2C_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_B_I2C_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **USCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt

Modified bits of **UCBxIE** register.

Returns:
None

40.2.2.6 uint8_t USCI_B_I2C_getInterruptStatus (uint16_t *baseAddress*, uint8_t *mask*)

Gets the current I2C interrupt status.

This returns the interrupt status for the I2C module based on which flag is passed. mask parameter can be logic OR of any of the following selection.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	10
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	34
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

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:

- **USCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt
- **USCI_B_I2C_START_INTERRUPT** - START condition interrupt
- **USCI_B_I2C_RECEIVE_INTERRUPT** - Receive interrupt
- **USCI_B_I2C_TRANSMIT_INTERRUPT** - Transmit interrupt
- **USCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt
- **USCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt

Returns:

the masked status of the interrupt flag Return Logical OR of any of the following:

- **USCI_B_I2C_STOP_INTERRUPT** STOP condition interrupt
- **USCI_B_I2C_START_INTERRUPT** START condition interrupt
- **USCI_B_I2C_RECEIVE_INTERRUPT** Receive interrupt
- **USCI_B_I2C_TRANSMIT_INTERRUPT** Transmit interrupt
- **USCI_B_I2C_NAK_INTERRUPT** Not-acknowledge interrupt
- **USCI_B_I2C_ARBITRATIONLOST_INTERRUPT** Arbitration lost interrupt
  indicating the status of the masked interrupts

40.2.2.7 uint32_t USCI_B_I2C_getReceiveBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:

baseAddress is the base address of the I2C module.

Returns:

the address of the RX Buffer

40.2.2.8 uint32_t USCI_B_I2C_getTransmitBufferAddressForDMA (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	18
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	8
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	24
MSPGCC 4.8.0	Size	8
MSPGCC 4.8.0	Speed	8

Parameters:
baseAddress is the base address of the I2C module.

Returns:
the address of the TX Buffer

40.2.2.9 void USCI_B_I2C_initMaster (uint16_t *baseAddress*, USCI_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 USCI_B_I2C_enable is invoked. If the parameter *dataRate* is USCI_B_I2C_SET_DATA_RATE_400KBPS, then the master block will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	52
TI Compiler 4.2.1	Speed	52
IAR 5.51.6	None	64
IAR 5.51.6	Size	60
IAR 5.51.6	Speed	60
MSPGCC 4.8.0	None	98
MSPGCC 4.8.0	Size	52
MSPGCC 4.8.0	Speed	52

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

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

Returns:
None

40.2.2.10 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	12
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:
baseAddress is the base address of the I2C module.

Returns:
Returns USCI_B_I2C_BUS_BUSY if the I2C Master is busy; otherwise, returns USCI_B_I2C_BUS_NOT_BUSY. Return one of the following:

- **USCI_B_I2C_BUS_BUSY**
- **USCI_B_I2C_BUS_NOT_BUSY**
  indicating if the USCI_B_I2C is busy

40.2.2.11 uint8_t USCI_B_I2C_isBusy (uint16_t *baseAddress*)

DEPRECATED - Function may be removed in future release. Indicates whether or not the I2C module is busy.

This function returns an indication of whether or not the I2C module is busy transmitting or receiving data. This function checks if the Transmit or receive flag is set.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	26
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	22
MSPGCC 4.8.0	None	38
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	22

Parameters:
baseAddress is the base address of the I2C module.

Returns:
Returns USCI_B_I2C_BUS_BUSY if the I2C module is busy; otherwise, returns USCI_B_I2C_BUS_NOT_BUSY. Return one of the following:

- **USCI_B_I2C_BUS_BUSY**
- **USCI_B_I2C_BUS_NOT_BUSY**
  indicating if the USCI_B_I2C is busy

40.2.2.12 void USCI_B_I2C_masterInit (uint16_t *baseAddress*, uint8_t *selectClockSource*, uint32_t *i2cClk*, uint32_t *dataRate*)

DEPRECATED - 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 USCI_B_I2C_enable is invoked. If the parameter *dataRate* is USCI_B_I2C_SET_DATA_RATE_400KBPS, then the master block will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	82
TI Compiler 4.2.1	Size	72
TI Compiler 4.2.1	Speed	72
IAR 5.51.6	None	72
IAR 5.51.6	Size	68
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	86
MSPGCC 4.8.0	Size	70
MSPGCC 4.8.0	Speed	44

Parameters:

baseAddress is the base address of the I2C Master module.

selectClockSource is the clocksource. Valid values are:

- **USCI_B_I2C_CLOCKSOURCE_ACLK**
- **USCI_B_I2C_CLOCKSOURCE_SMCLK**

i2cClk is the rate of the clock supplied to the I2C module.

dataRate set up for selecting data transfer rate. Valid values are:

- **USCI_B_I2C_SET_DATA_RATE_400KBPS**
- **USCI_B_I2C_SET_DATA_RATE_100KBPS**

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

Returns:

None

40.2.2.13 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C module.

Returns:
Returns USCI_B_I2C_START_SEND_COMPLETE if the I2C Master finished sending START; otherwise, returns USCI_B_I2C_SENDING_START. Return one of the following:

- **USCI_B_I2C_SENDING_START**
- **USCI_B_I2C_START_SEND_COMPLETE**

40.2.2.14 uint8_t USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	6
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	22
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C module.

Returns:
Returns USCI_B_I2C_STOP_SEND_COMPLETE if the I2C Master finished sending STOP; otherwise, returns USCI_B_I2C_SENDING_STOP. Return one of the following:

- **USCI_B_I2C_SENDING_STOP**
- **USCI_B_I2C_STOP_SEND_COMPLETE**

40.2.2.15 uint8_t USCI_B_I2C_masterMultiByteReceiveFinish (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 does the following:
- Receives the current byte and initiates the STOP from Master to Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	42
TI Compiler 4.2.1	Size	24
TI Compiler 4.2.1	Speed	24
IAR 5.51.6	None	36
IAR 5.51.6	Size	26
IAR 5.51.6	Speed	34
MSPGCC 4.8.0	None	70
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:
baseAddress is the base address of the I2C Master module.

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

Returns:
Received byte at Master end.

40.2.2.16 bool USCI_B_I2C_masterMultiByteReceiveFinishWithTimeout (uint16_t *baseAddress*, uint8_t ∗ *rxData*, 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 does the following:
- Receives the current byte and initiates the STOP from Master to Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	138
TI Compiler 4.2.1	Size	76
TI Compiler 4.2.1	Speed	76
IAR 5.51.6	None	96
IAR 5.51.6	Size	82
IAR 5.51.6	Speed	94
MSPGCC 4.8.0	None	182
MSPGCC 4.8.0	Size	92
MSPGCC 4.8.0	Speed	98

Parameters:
baseAddress is the base address of the I2C Master module.
rxData 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 **UCBxCTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.17 uint8_t USCI_B_I2C_masterMultiByteReceiveNext (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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C Master module.

Returns:
Received byte at Master end.

40.2.2.18 void USCI_B_I2C_masterMultiByteReceiveStart (uint16_t *baseAddress*)

Starts multi-byte reception at the Master end.

This function is used by the Master module initiate reception of a single byte. This function does the following: - Sends START

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	24
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	16
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:
baseAddress is the base address of the I2C Master module.

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

Returns:
None

40.2.2.19 void USCI_B_I2C_masterMultiByteReceiveStop (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

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	0
IAR 5.51.6	Speed	0
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C Master module.

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

Returns:
None

40.2.2.20 void USCI_B_I2C_masterMultiByteSendFinish (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 does the following: - Transmits the last data byte of a multi-byte transmission to the Slave; - Sends STOP

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	28
TI Compiler 4.2.1	Speed	28
IAR 5.51.6	None	54
IAR 5.51.6	Size	42
IAR 5.51.6	Speed	42
MSPGCC 4.8.0	None	100
MSPGCC 4.8.0	Size	38
MSPGCC 4.8.0	Speed	38

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 **UCBxCTL1** register.

Returns:

None

40.2.2.21 bool USCI_B_I2C_masterMultiByteSendFinishWithTimeout (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 does the following: - Transmits the last data byte of a multi-byte transmission to the Slave; - Sends STOP

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	136
TI Compiler 4.2.1	Size	82
TI Compiler 4.2.1	Speed	82
IAR 5.51.6	None	114
IAR 5.51.6	Size	98
IAR 5.51.6	Speed	104
MSPGCC 4.8.0	None	198
MSPGCC 4.8.0	Size	94
MSPGCC 4.8.0	Speed	96

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 **UCBxCTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.22 void USCI_B_I2C_masterMultiByteSendNext (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 does the following: -Transmits each data byte of a multi-byte transmission to the Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	36
IAR 5.51.6	Size	4
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	62
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

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

40.2.2.23 bool USCI_B_I2C_masterMultiByteSendNextWithTimeout (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 does the following: -Transmits each data byte of a multi-byte transmission to the Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	74
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	44
IAR 5.51.6	None	68
IAR 5.51.6	Size	62
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	108
MSPGCC 4.8.0	Size	64
MSPGCC 4.8.0	Speed	64

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.

40.2.2.24 void USCI_B_I2C_masterMultiByteSendStart (uint16_t *baseAddress*, uint8_t *txData*)

Starts multi-byte transmission from Master to Slave.

This function is used by the Master module to send a single byte. This function does the following: - Sends START; - Transmits the first data byte of a multi-byte transmission to the Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	68
TI Compiler 4.2.1	Size	36
TI Compiler 4.2.1	Speed	36
IAR 5.51.6	None	60
IAR 5.51.6	Size	28
IAR 5.51.6	Speed	50
MSPGCC 4.8.0	None	150
MSPGCC 4.8.0	Size	46
MSPGCC 4.8.0	Speed	46

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 **UCBxIFG** register, bits of **UCBxCTL1** register and bits of **UCBxIE** register.

Returns:
None

40.2.2.25 bool USCI_B_I2C_masterMultiByteSendStartWithTimeout (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 send a single byte. This function does the following: - Sends START; - Transmits the first data byte of a multi-byte transmission to the Slave

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	114
TI Compiler 4.2.1	Size	66
TI Compiler 4.2.1	Speed	66
IAR 5.51.6	None	92
IAR 5.51.6	Size	66
IAR 5.51.6	Speed	80
MSPGCC 4.8.0	None	196
MSPGCC 4.8.0	Size	84
MSPGCC 4.8.0	Speed	82

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

Returns:

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.26 void USCI_B_I2C_masterMultiByteSendStop (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 does the following: - Sends a STOP after current transmission is complete

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	28
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	32
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	56
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:

baseAddress is the base address of the I2C Master module.

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

Returns:

None

40.2.2.27 bool USCI_B_I2C_masterMultiByteSendStopWithTimeout (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 does the following: - Sends a STOP after current transmission is complete

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	74
TI Compiler 4.2.1	Size	44
TI Compiler 4.2.1	Speed	44
IAR 5.51.6	None	64
IAR 5.51.6	Size	44
IAR 5.51.6	Speed	58
MSPGCC 4.8.0	None	114
MSPGCC 4.8.0	Size	56
MSPGCC 4.8.0	Speed	52

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 **UCBxCTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.28 void USCI_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 does the following: - Sends START; - Transmits the byte to the Slave; - Sends STOP

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	92
TI Compiler 4.2.1	Size	50
TI Compiler 4.2.1	Speed	50
IAR 5.51.6	None	88
IAR 5.51.6	Size	52
IAR 5.51.6	Speed	66
MSPGCC 4.8.0	None	218
MSPGCC 4.8.0	Size	60
MSPGCC 4.8.0	Speed	60

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 **UCBxIFG** register, bits of **UCBxCTL1** register and bits of **UCBxIE** register.

Returns:
None

40.2.2.29 bool USCI_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 does the following: - Sends START; - Transmits the byte to the Slave; - Sends STOP

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	182
TI Compiler 4.2.1	Size	104
TI Compiler 4.2.1	Speed	104
IAR 5.51.6	None	148
IAR 5.51.6	Size	110
IAR 5.51.6	Speed	120
MSPGCC 4.8.0	None	312
MSPGCC 4.8.0	Size	118
MSPGCC 4.8.0	Speed	118

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 **UCBxIFG** register, bits of **UCBxCTL1** register and bits of **UCBxIE** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.30 void USCI_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 STOP

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	16
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	26
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C Master module.

Returns:
None

40.2.2.31 uint8_t USCI_B_I2C_masterSingleReceive (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	26
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	32
IAR 5.51.6	Size	16
IAR 5.51.6	Speed	24
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:
baseAddress is the base address of the I2C module.

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

40.2.2.32 void USCI_B_I2C_masterSingleReceiveStart (uint16_t *baseAddress*)

Initiates a single byte Reception at the Master End.

This function sends a START and STOP immediately to indicate Single byte reception

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	54
TI Compiler 4.2.1	Size	34
TI Compiler 4.2.1	Speed	34
IAR 5.51.6	None	40
IAR 5.51.6	Size	22
IAR 5.51.6	Speed	32
MSPGCC 4.8.0	None	102
MSPGCC 4.8.0	Size	30
MSPGCC 4.8.0	Speed	30

Parameters:
baseAddress is the base address of the I2C Master module.

Modified bits are **GIE** of **SR** register; bits **UCTXSTT** and **UCTXSTP** of **UCBxCTL1** register.

Returns:
None

40.2.2.33 bool USCI_B_I2C_masterSingleReceiveStartWithTimeout (uint16_t *baseAddress*, uint32_t *timeout*)

Initiates a single byte Reception at the Master End with timeout.

This function sends a START and STOP immediately to indicate Single byte reception

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	98
TI Compiler 4.2.1	Size	60
TI Compiler 4.2.1	Speed	60
IAR 5.51.6	None	78
IAR 5.51.6	Size	46
IAR 5.51.6	Speed	70
MSPGCC 4.8.0	None	166
MSPGCC 4.8.0	Size	74
MSPGCC 4.8.0	Speed	72

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 **GIE** of **SR** register; bits **UCTXSTT** and **UCTXSTP** of **UCBxCTL1** register.

Returns:
STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

40.2.2.34 void USCI_B_I2C_setMode (uint16_t *baseAddress*, uint8_t *mode*)

Sets the mode of the I2C device.

When the receive parameter is set to USCI_B_I2C_TRANSMIT_MODE, the address will indicate that the I2C module is in receive mode; otherwise, the I2C module is in send mode.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	30
TI Compiler 4.2.1	Size	12
TI Compiler 4.2.1	Speed	12
IAR 5.51.6	None	14
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	58
MSPGCC 4.8.0	Size	12
MSPGCC 4.8.0	Speed	12

Parameters:

baseAddress is the base address of the I2C Master module.

mode indicates whether module is in transmit/receive mode Valid values are:
- **USCI_B_I2C_TRANSMIT_MODE**
- **USCI_B_I2C_RECEIVE_MODE** [Default]

Returns:

None

40.2.2.35 void USCI_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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	22
TI Compiler 4.2.1	Size	8
TI Compiler 4.2.1	Speed	8
IAR 5.51.6	None	14
IAR 5.51.6	Size	6
IAR 5.51.6	Speed	6
MSPGCC 4.8.0	None	30
MSPGCC 4.8.0	Size	10
MSPGCC 4.8.0	Speed	10

Parameters:

baseAddress is the base address of the I2C Master module.

slaveAddress 7-bit slave address

Modified bits of **UCBxI2CSA** register; bits **UCSWRST** of **UCBxCTL1** register.

Returns:

None

40.2.2.36 uint8_t USCI_B_I2C_slaveDataGet (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	14
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	8
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	20
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

Parameters:
baseAddress is the base address of the I2C module.

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

40.2.2.37 void USCI_B_I2C_slaveDataPut (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 Modified bit is UCBxTXBUF register

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	20
TI Compiler 4.2.1	Size	6
TI Compiler 4.2.1	Speed	6
IAR 5.51.6	None	12
IAR 5.51.6	Size	2
IAR 5.51.6	Speed	2
MSPGCC 4.8.0	None	28
MSPGCC 4.8.0	Size	6
MSPGCC 4.8.0	Speed	6

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

Modified bits of **UCBxTXBUF** register.

Returns:
None

40.2.2.38 void USCI_B_I2C_slaveInit (uint16_t *baseAddress*, uint8_t *slaveAddress*)

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 USCI_B_I2C_enable is invoked.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	46
TI Compiler 4.2.1	Size	26
TI Compiler 4.2.1	Speed	26
IAR 5.51.6	None	38
IAR 5.51.6	Size	26
IAR 5.51.6	Speed	26
MSPGCC 4.8.0	None	88
MSPGCC 4.8.0	Size	28
MSPGCC 4.8.0	Speed	28

Parameters:

baseAddress is the base address of the I2C Slave module.

slaveAddress 7-bit slave address

Modified bits of **UCBxI2COA** register; bits **UCSWRST** of **UCBxCTL1** register; bits **UCMODE_3** and **UCSYNC** of **UCBxCTL0** register.

Returns:

None

40.3 Programming Example

The following example shows how to use the USCI_B_I2C API to send data as a master.

```
  // Initialize Master
USCI_B_I2C_masterInit(USCI_B0_BASE, SMCLK, CLK_getSMClk(), USCI_B_I2C_SET_DATA_RATE_400KBPS);

// Specify slave address
USCI_B_I2C_setSlaveAddress(USCI_B0_BASE, SLAVE_ADDRESS);

// Set in transmit mode
USCI_B_I2C_setMode(USCI_B0_BASE, USCI_B_I2C_TRANSMIT_MODE);

//Enable USCI_B_I2C Module to start operations
USCI_B_I2C_enable(USCI_B0_BASE);

while (1)
{
  // Send single byte data.
  USCI_B_I2C_masterSendSingleByte(USCI_B0_BASE, transmitData);

  // Delay until transmission completes
  while(USCI_B_I2C_busBusy(USCI_B0_BASE));

  // Increment transmit data counter
  transmitData++;
}
```

41 WatchDog Timer (WDT_A)

41.1 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.

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

T

he following code metrics were performed with the TI Compiler 4.2.1 compiler, IAR 5.51.6 compiler and MSPGCC 4.8.0 compiler with different optimization settings. Users may see different code sizes depending on their project settings so it is best to perform your benchmarks within your project. These sizes contain all functions of the peripheral but only functions that are used will be linked into the application and added to the total code size. To see individual API code metrics see the specific API below.

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	172
TI Compiler 4.2.1	Size	90
TI Compiler 4.2.1	Speed	90
IAR 5.51.6	None	140
IAR 5.51.6	Size	54
IAR 5.51.6	Speed	54
MSPGCC 4.8.0	None	236
MSPGCC 4.8.0	Size	104
MSPGCC 4.8.0	Speed	104

41.2 API Functions

Functions

- void WDT_A_hold (uint16_t baseAddress)
- void WDT_A_intervalTimerInit (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
- void WDT_A_resetTimer (uint16_t baseAddress)
- void WDT_A_start (uint16_t baseAddress)
- void WDT_A_watchdogTimerInit (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)

41.2.1 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_watchdogTimerInit()
- WDT_A_intervalTimerInit()

41.2.2 Function Documentation

41.2.2.1 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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	24
MSPGCC 4.8.0	Speed	24

Parameters:
baseAddress is the base address of the WDT_A module.

Returns:
None

41.2.2.2 void WDT_A_intervalTimerInit (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	26
IAR 5.51.6	Size	10
IAR 5.51.6	Speed	10
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

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.

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

41.2.2.3 void WDT_A_resetTimer (uint16_t *baseAddress*)

Resets the timer counter of the Watchdog Timer.

This function resets the watchdog timer to 0x0000h.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	32
TI Compiler 4.2.1	Size	18
TI Compiler 4.2.1	Speed	18
IAR 5.51.6	None	28
IAR 5.51.6	Size	8
IAR 5.51.6	Speed	8
MSPGCC 4.8.0	None	48
MSPGCC 4.8.0	Size	22
MSPGCC 4.8.0	Speed	22

Parameters:
baseAddress is the base address of the WDT_A module.

Returns:
None

41.2.2.4 void WDT_A_start (uint16_t *baseAddress*)

Starts the Watchdog Timer.

This function starts the watchdog timer functionality to start counting again.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	34
TI Compiler 4.2.1	Size	20
TI Compiler 4.2.1	Speed	20
IAR 5.51.6	None	30
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	50
MSPGCC 4.8.0	Size	18
MSPGCC 4.8.0	Speed	18

Parameters:
baseAddress is the base address of the WDT_A module.

Returns:
None

41.2.2.5 void WDT_A_watchdogTimerInit (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.

Code Metrics:

Compiler	Optimization	Code Size
TI Compiler 4.2.1	None	36
TI Compiler 4.2.1	Size	16
TI Compiler 4.2.1	Speed	16
IAR 5.51.6	None	26
IAR 5.51.6	Size	12
IAR 5.51.6	Speed	12
MSPGCC 4.8.0	None	44
MSPGCC 4.8.0	Size	20
MSPGCC 4.8.0	Speed	20

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.

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

41.3 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_intervalTimerInit(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();
```

IMPORTANT NOTICE

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