




MSP430 DriverLib for MSP430F5xx_6xx Devices

User's Guide

Copyright

Copyright © 2015 Texas Instruments Incorporated. All rights reserved. MSP430 and MSP430Ware are trademarks of Texas Instruments Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

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



Revision Information

This is version 2.10.00.09 of this document, last updated on Mon Apr 13 2015 10:31:31.

Table of Contents

Copyright	1
Revision Information	1
1 Introduction	7
2 Navigating to driverlib through CCS Resource Explorer	9
2.1 Introduction	9
3 How to create a new CCS project that uses Driverlib	24
3.1 Introduction	24
4 How to include driverlib into your existing CCS project	26
4.1 Introduction	26
5 How to create a new IAR project that uses Driverlib	28
5.1 Introduction	28
6 How to include driverlib into your existing IAR project	31
6.1 Introduction	31
7 10-Bit Analog-to-Digital Converter (ADC10_A)	34
7.1 Introduction	34
7.2 API Functions	34
7.3 Programming Example	51
8 12-Bit Analog-to-Digital Converter (ADC12_A)	52
8.1 Introduction	52
8.2 API Functions	52
8.3 Programming Example	70
9 Advanced Encryption Standard (AES)	71
9.1 Introduction	71
9.2 API Functions	71
9.3 Programming Example	79
10 Battery Backup System	80
10.1 Introduction	80
10.2 API Functions	80
11 Comparator (COMP_B)	81
11.1 Introduction	81
11.2 API Functions	81
11.3 Programming Example	91
12 Cyclical Redundancy Check (CRC)	93
12.1 Introduction	93
12.2 API Functions	93
12.3 Programming Example	96
13 12-bit Digital-to-Analog Converter (DAC12_A)	98
13.1 Introduction	98
13.2 API Functions	98
13.3 Programming Example	109
14 Direct Memory Access (DMA)	110
14.1 Introduction	110
14.2 API Functions	110
14.3 Programming Example	122

15	EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)	123
15.1	Introduction	123
15.2	API Functions	123
15.3	Programming Example	132
16	EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)	133
16.1	Introduction	133
16.2	Functions	133
16.3	Programming Example	142
17	EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)	143
17.1	Introduction	143
17.2	Functions	143
17.3	Programming Example	152
18	EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)	153
18.1	Introduction	153
18.2	Master Operations	153
18.3	Slave Operations	154
18.4	API Functions	155
18.5	Programming Example	176
19	FlashCtl - Flash Memory Controller	177
19.1	Introduction	177
19.2	API Functions	177
19.3	Programming Example	183
20	GPIO	184
20.1	Introduction	184
20.2	API Functions	185
20.3	Programming Example	211
21	LDO-PWR	213
21.1	Introduction	213
21.2	API Functions	213
21.3	Programming Example	225
22	32-Bit Hardware Multiplier (MPY32)	227
22.1	Introduction	227
22.2	API Functions	227
22.3	Programming Example	235
23	Port Mapping Controller	236
23.1	Introduction	236
23.2	API Functions	236
23.3	Programming Example	237
24	Power Management Module (PMM)	238
24.1	Introduction	238
24.2	API Functions	240
24.3	Programming Example	251
25	RAM Controller	253
25.1	Introduction	253
25.2	API Functions	253
25.3	Programming Example	255
26	Internal Reference (REF)	256
26.1	Introduction	256
26.2	API Functions	256

26.3	Programming Example	261
27	Real-Time Clock (RTC_A)	263
27.1	Introduction	263
27.2	API Functions	263
27.3	Programming Example	277
28	Real-Time Clock (RTC_B)	278
28.1	Introduction	278
28.2	API Functions	278
28.3	Programming Example	288
29	Real-Time Clock (RTC_C)	290
29.1	Introduction	290
29.2	API Functions	290
29.3	Programming Example	306
30	24-Bit Sigma Delta Converter (SD24_B)	308
30.1	Introduction	308
30.2	API Functions	308
30.3	Programming Example	322
31	SFR Module	323
31.1	Introduction	323
31.2	API Functions	323
31.3	Programming Example	328
32	System Control Module	329
32.1	Introduction	329
32.2	API Functions	329
32.3	Programming Example	337
33	Timer Event Control (TEC)	338
33.1	Introduction	338
33.2	API Functions	338
33.3	Programming Example	348
34	16-Bit Timer_A (TIMER_A)	349
34.1	Introduction	349
34.2	API Functions	350
34.3	Programming Example	364
35	16-Bit Timer_B (TIMER_B)	366
35.1	Introduction	366
35.2	API Functions	367
35.3	Programming Example	382
36	TIMER_D	383
36.1	Introduction	383
36.2	API Functions	384
36.3	Programming Example	409
37	Tag Length Value	411
37.1	Introduction	411
37.2	API Functions	411
37.3	Programming Example	416
38	Unified Clock System (UCS)	417
38.1	Introduction	417
38.2	API Functions	418

38.3	Programming Example	432
39	USCI Universal Asynchronous Receiver/Transmitter (USCI_A_UART)	433
39.1	Introduction	433
39.2	API Functions	433
39.3	Programming Example	441
40	USCI Synchronous Peripheral Interface (USCI_A_SPI)	443
40.1	Introduction	443
40.2	API Functions	443
40.3	Programming Example	451
41	USCI Synchronous Peripheral Interface (USCI_B_SPI)	453
41.1	Introduction	453
41.2	API Functions	453
41.3	Programming Example	461
42	USCI Inter-Integrated Circuit (USCI_B_I2C)	463
42.1	Introduction	463
42.2	Master Operations	463
42.3	Slave Operations	464
42.4	API Functions	465
42.5	Programming Example	481
43	WatchDog Timer (WDT_A)	482
43.1	Introduction	482
43.2	API Functions	482
43.3	Programming Example	485
44	Data Structure Documentation	486
44.1	Data Structures	486
44.2	ADC12_A_configureMemoryParam Struct Reference	488
44.3	Calendar Struct Reference	490
44.4	Comp_B_configureReferenceVoltageParam Struct Reference	491
44.5	Comp_B_initParam Struct Reference	492
44.6	DAC12_A_initParam Struct Reference	495
44.7	DMA_initParam Struct Reference	497
44.8	EUSCI_A_SPI_changeMasterClockParam Struct Reference	499
44.9	EUSCI_A_SPI_initMasterParam Struct Reference	500
44.10	EUSCI_A_SPI_initSlaveParam Struct Reference	502
44.11	EUSCI_A_UART_initParam Struct Reference	503
44.12	EUSCI_B_I2C_initMasterParam Struct Reference	505
44.13	EUSCI_B_I2C_initSlaveParam Struct Reference	507
44.14	EUSCI_B_SPI_changeMasterClockParam Struct Reference	508
44.15	EUSCI_B_SPI_initMasterParam Struct Reference	508
44.16	EUSCI_B_SPI_initSlaveParam Struct Reference	510
44.17	PMAP_initPortsParam Struct Reference	511
44.18	RTC_A_configureCalendarAlarmParam Struct Reference	512
44.19	RTC_B_configureCalendarAlarmParam Struct Reference	513
44.20	RTC_C_configureCalendarAlarmParam Struct Reference	514
44.21	SD24_B_initConverterAdvancedParam Struct Reference	516
44.22	SD24_B_initConverterParam Struct Reference	519
44.23	SD24_B_initParam Struct Reference	520
44.24	Timer_A_initCaptureModeParam Struct Reference	522
44.25	Timer_A_initCompareModeParam Struct Reference	525
44.26	Timer_A_initContinuousModeParam Struct Reference	526

44.27Timer_A_initUpDownModeParam Struct Reference	528
44.28Timer_A_initUpModeParam Struct Reference	530
44.29Timer_A_outputPWMPParam Struct Reference	532
44.30Timer_B_initCaptureModeParam Struct Reference	535
44.31Timer_B_initCompareModeParam Struct Reference	537
44.32Timer_B_initContinuousModeParam Struct Reference	538
44.33Timer_B_initUpDownModeParam Struct Reference	540
44.34Timer_B_initUpModeParam Struct Reference	542
44.35Timer_B_outputPWMPParam Struct Reference	545
44.36Timer_D_combineTDCCRToOutputPWMPParam Struct Reference	547
44.37Timer_D_initCaptureModeParam Struct Reference	549
44.38Timer_D_initCompareModeParam Struct Reference	552
44.39Timer_D_initContinuousModeParam Struct Reference	553
44.40Timer_D_initHighResGeneratorInRegulatedModeParam Struct Reference	555
44.41Timer_D_initUpDownModeParam Struct Reference	558
44.42Timer_D_initUpModeParam Struct Reference	560
44.43Timer_D_outputPWMPParam Struct Reference	563
44.44TEC_initExternalFaultInputParam Struct Reference	565
44.45USCI_A_SPI_changeMasterClockParam Struct Reference	566
44.46USCI_A_SPI_initMasterParam Struct Reference	567
44.47USCI_A_UART_initParam Struct Reference	568
44.48USCI_B_I2C_initMasterParam Struct Reference	570
44.49USCI_B_SPI_changeMasterClockParam Struct Reference	571
44.50USCI_B_SPI_initMasterParam Struct Reference	572
IMPORTANT NOTICE	574

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™

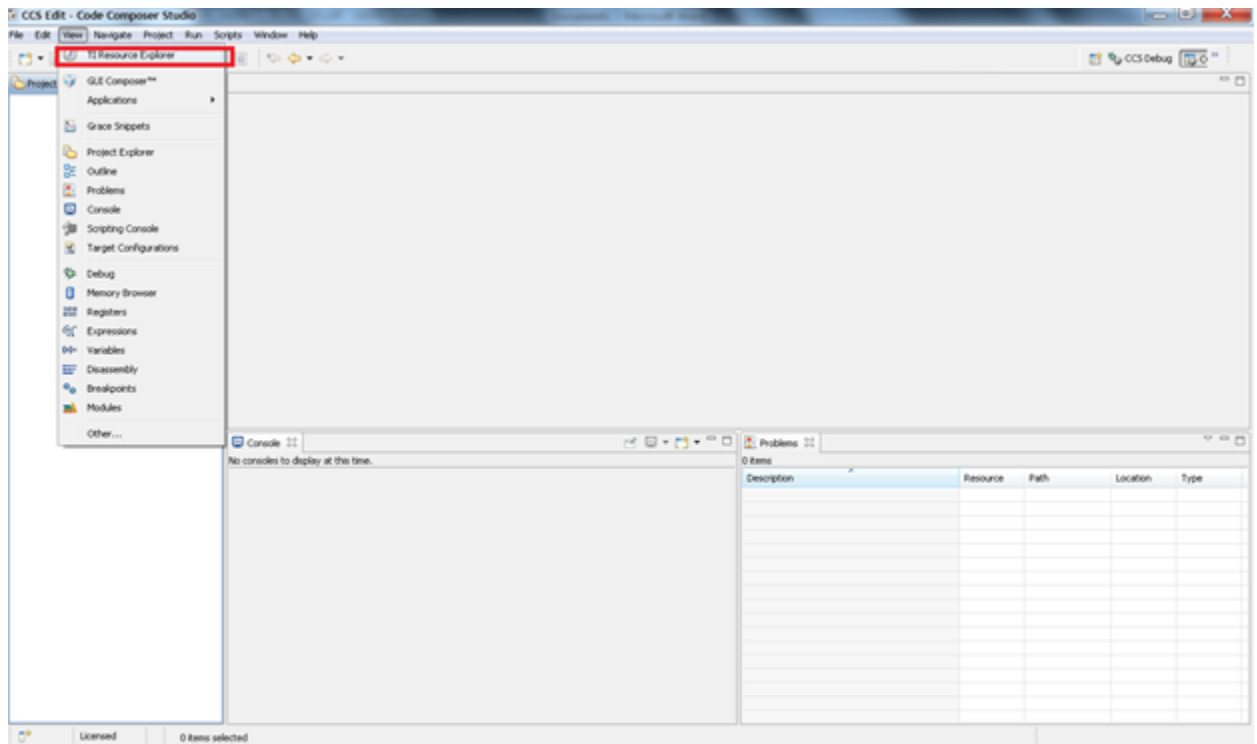
Using assert statements to debug

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.

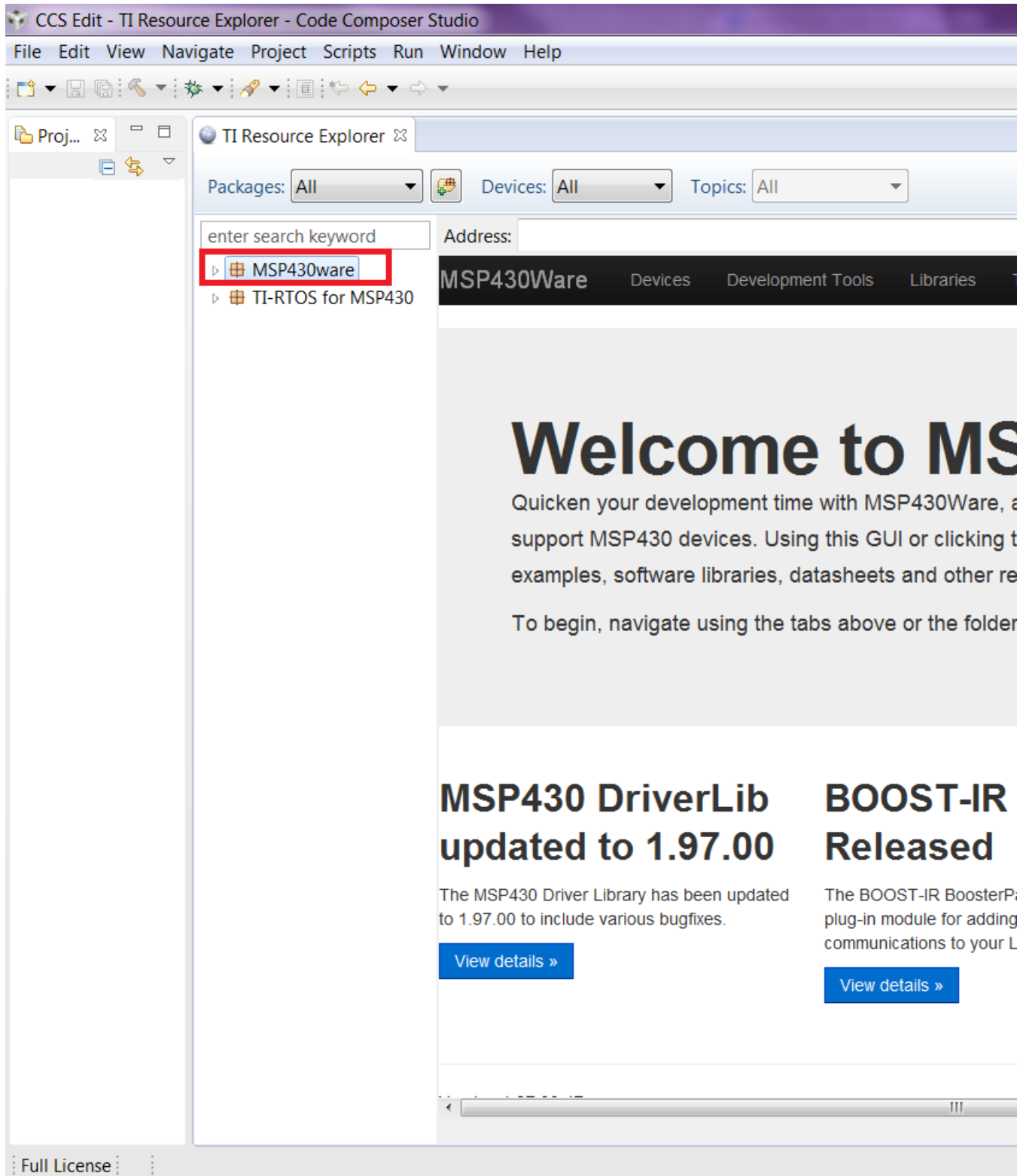
2 Navigating to driverlib through CCS Resource Explorer

2.1 Introduction

In CCS, click View->TI Resource Explorer

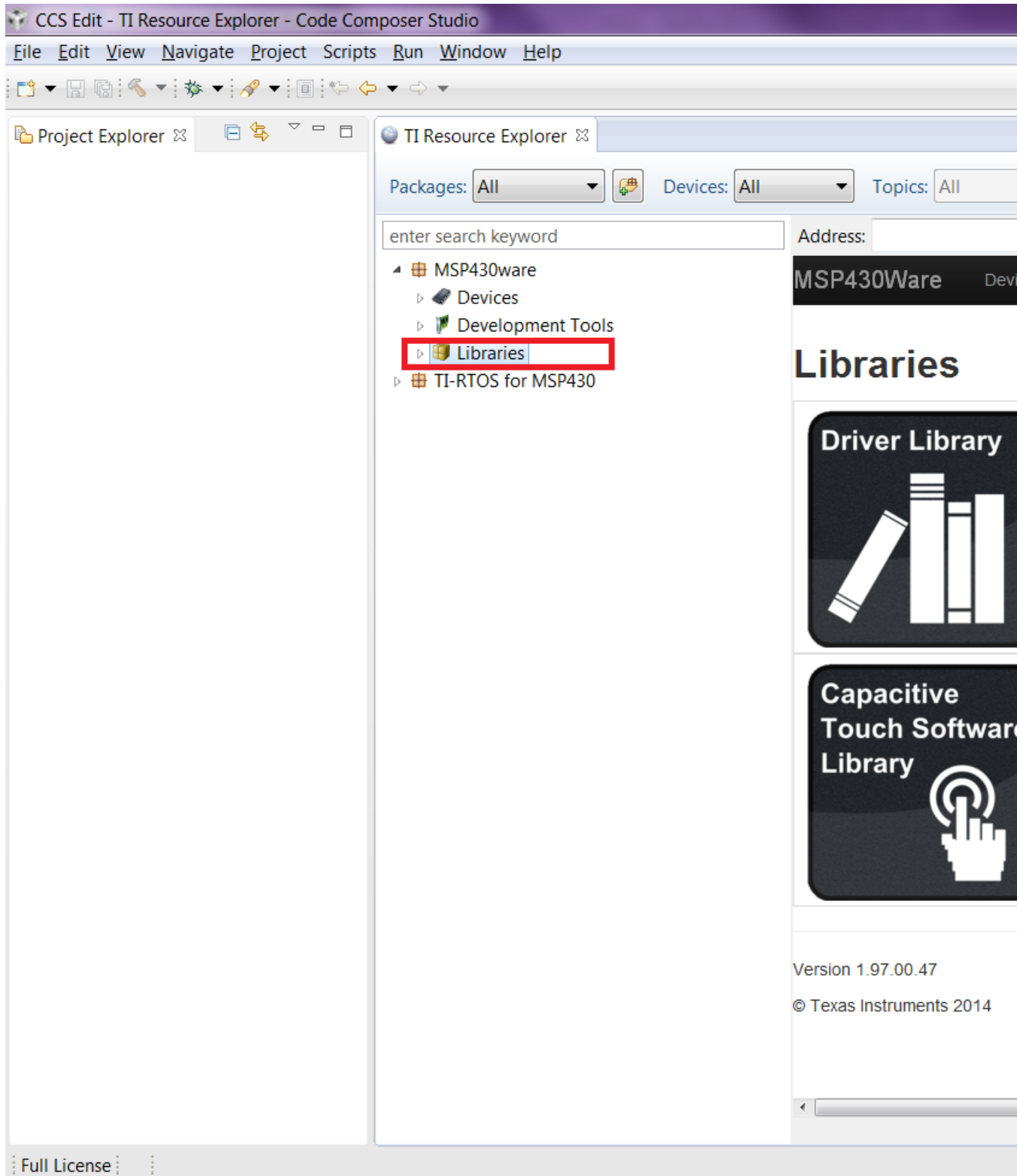


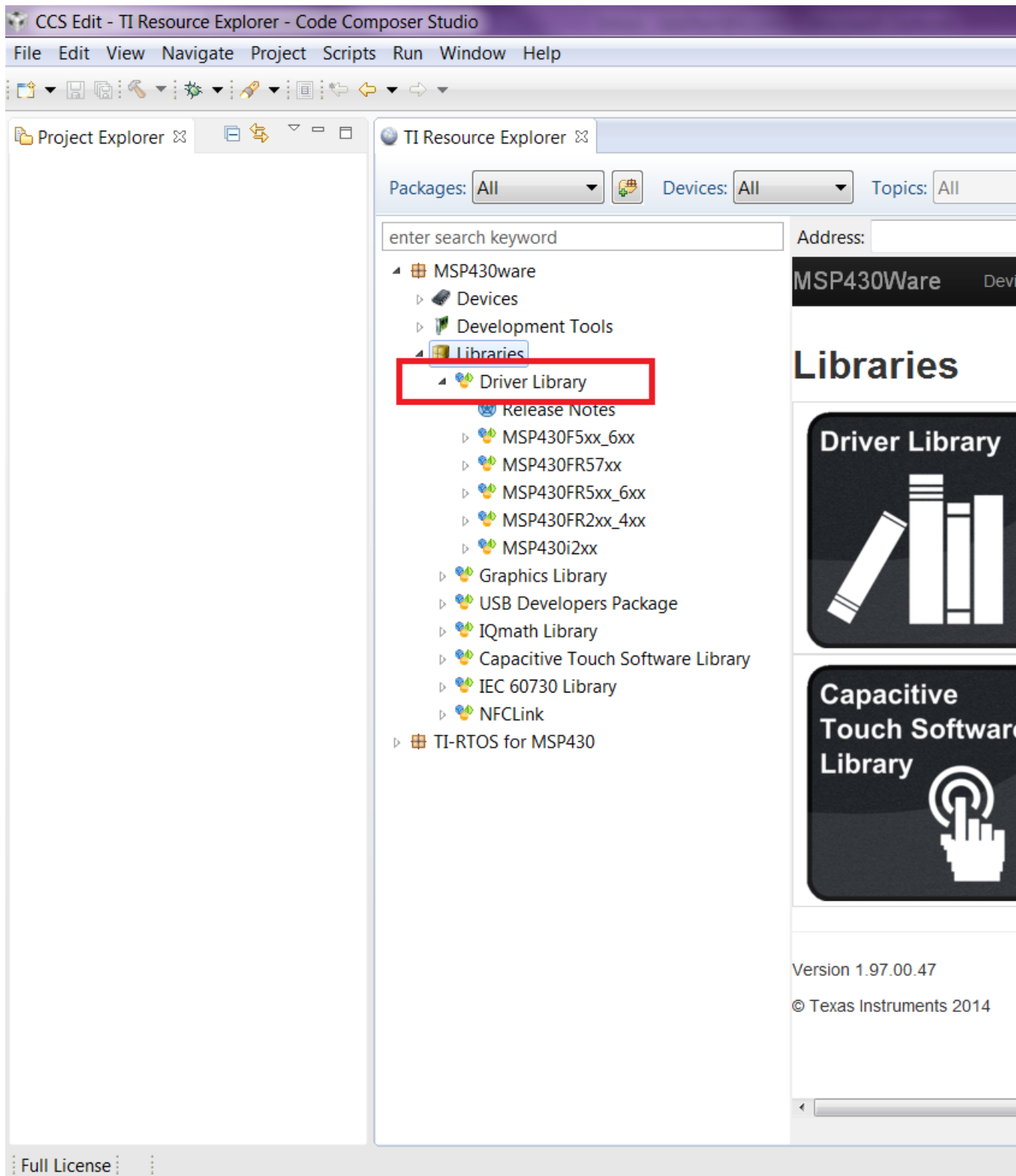
In Resource Explorer View, click on MSP430ware



Clicking MSP430ware takes you to the introductory page. The version of the latest MSP430ware installed is available in this page. In this screenshot the version is 1.30.00.15 The various

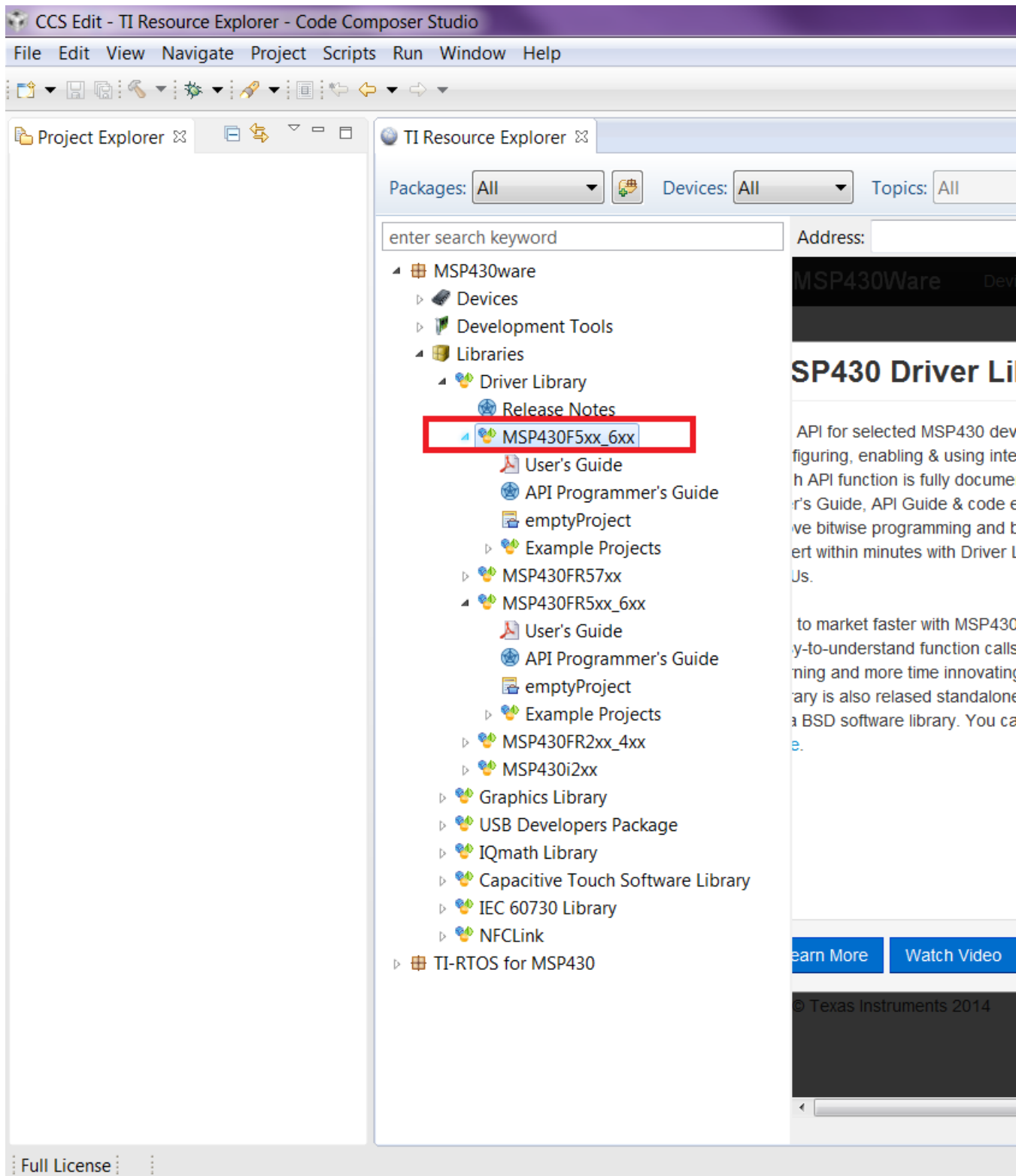
software, collateral, code examples, datasheets and user guides can be navigated by clicking the different topics under MSP430ware. To proceed to driverlib, click on Libraries->Driverlib as shown in the next two screenshots.



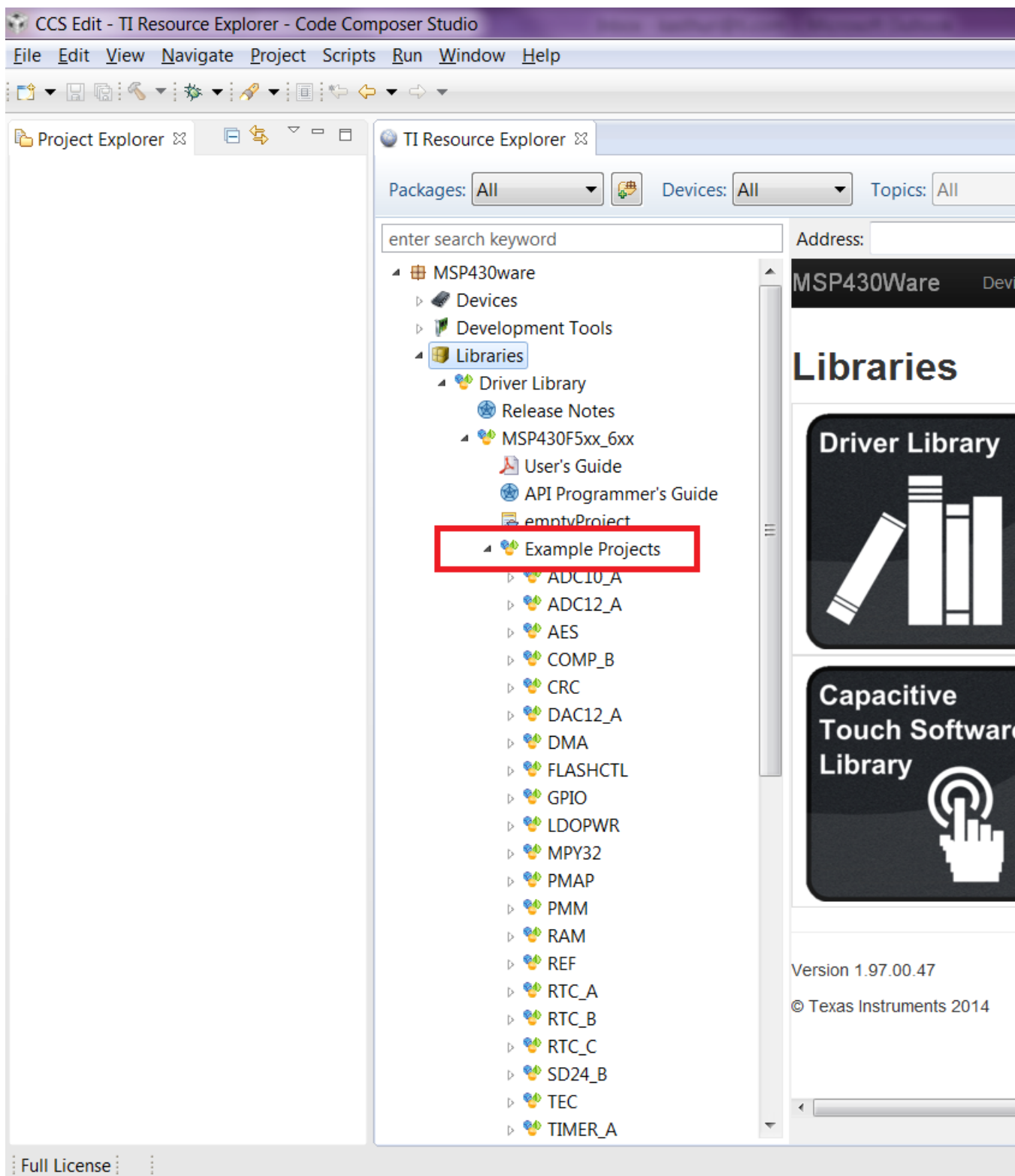


Driverlib is designed per Family. If a common device family user's guide exists for a group of devices, these devices belong to the same 'family'. Currently driverlib is available for the following

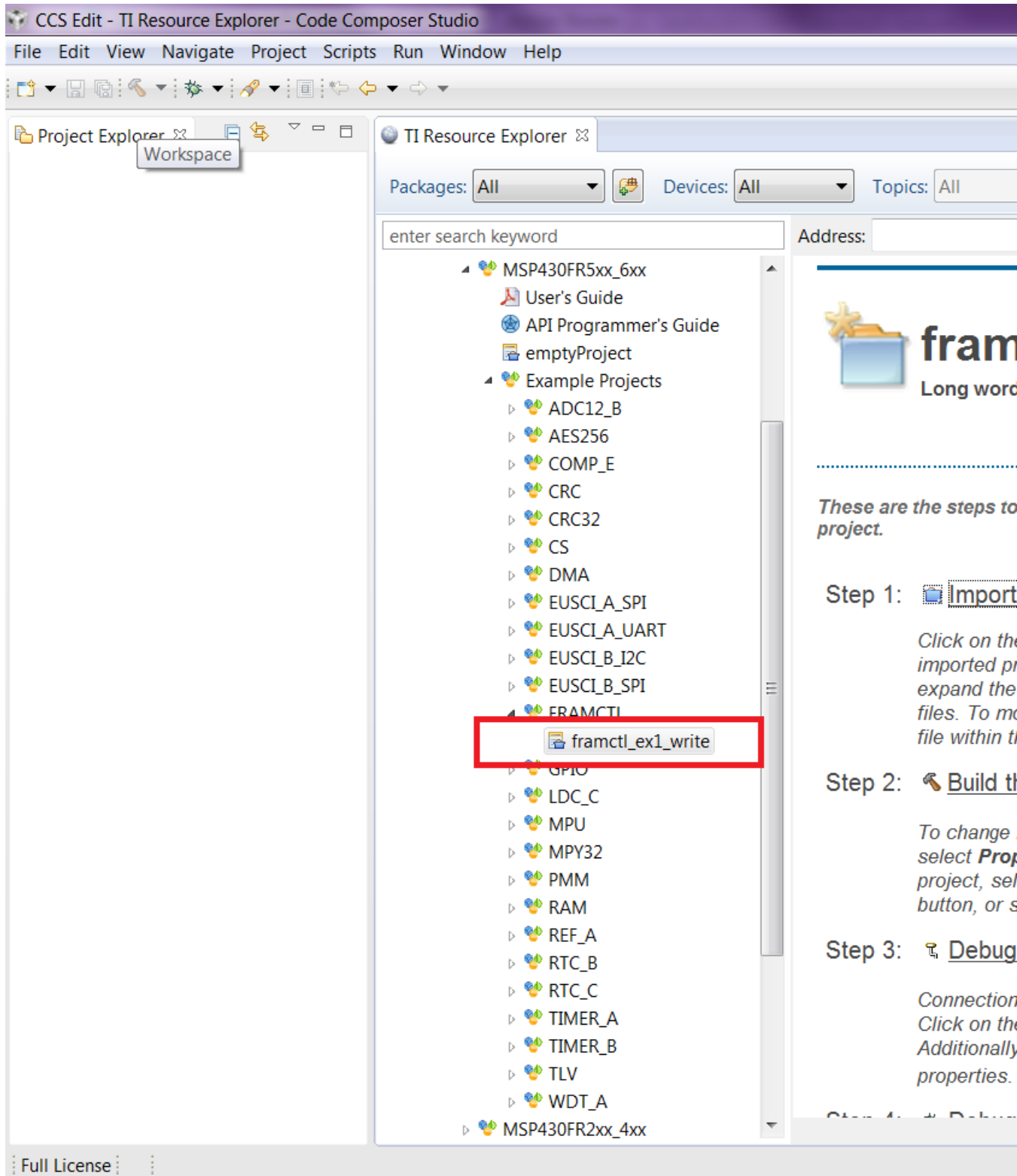
family of devices. MSP430F5xx_6xx MSP430FR57xx MSP430FR5xx_6xx MSP430i2xx



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



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



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

project into CCS”

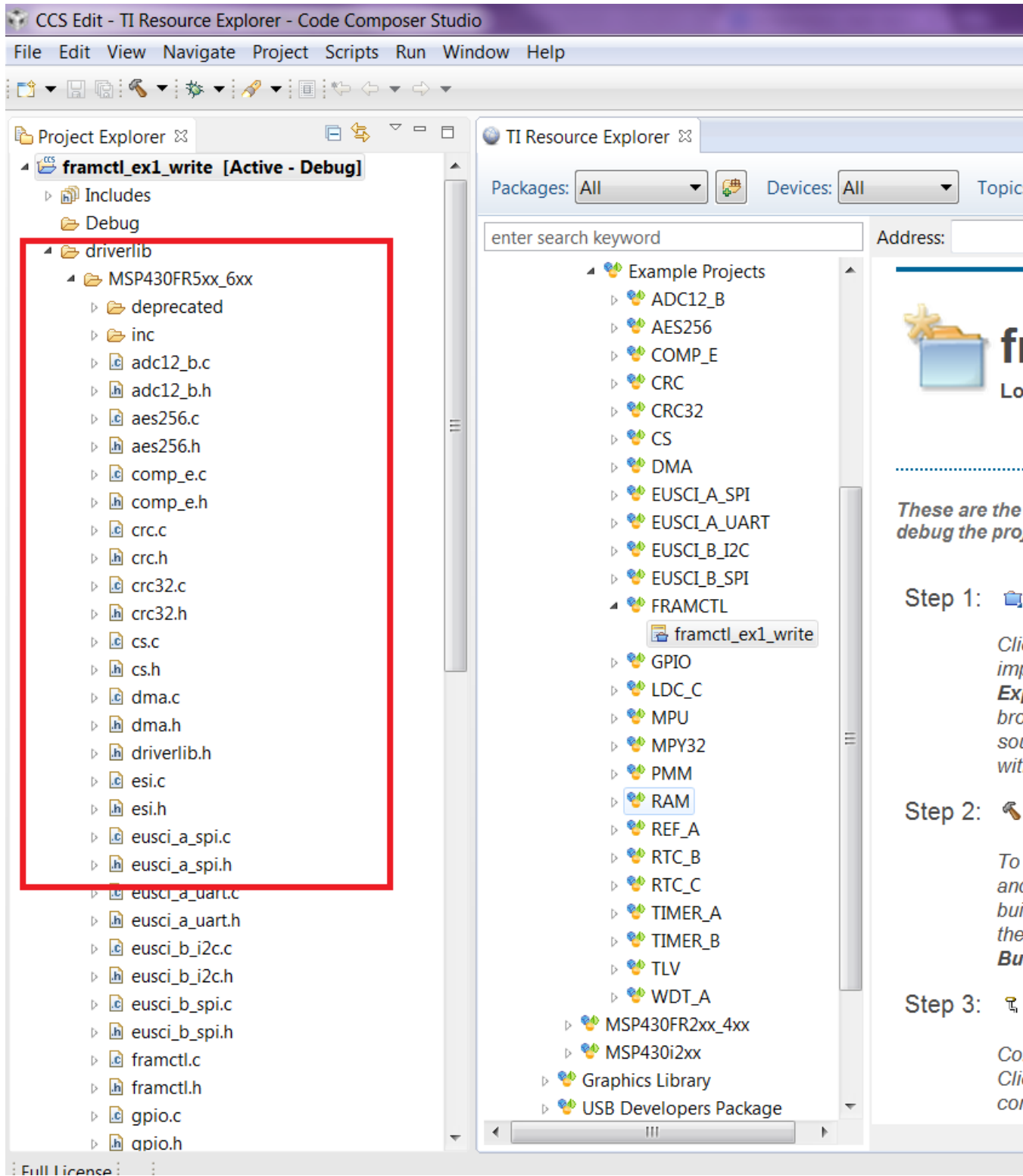
The screenshot shows the CCS Resource Explorer interface. On the left, a tree view displays the project structure under 'MSP430FR5xx_6xx'. The 'FRAMCTL' folder is expanded, and 'framctl_ex1_write' is selected. On the right, the project page for 'framctl_ex1_write' is displayed, featuring a star icon and the title 'Long word writes to FRAM'. Below the title, there is a section with the text 'These are the steps to import the project, build the project, and project.' followed by three numbered steps:

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**. 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.

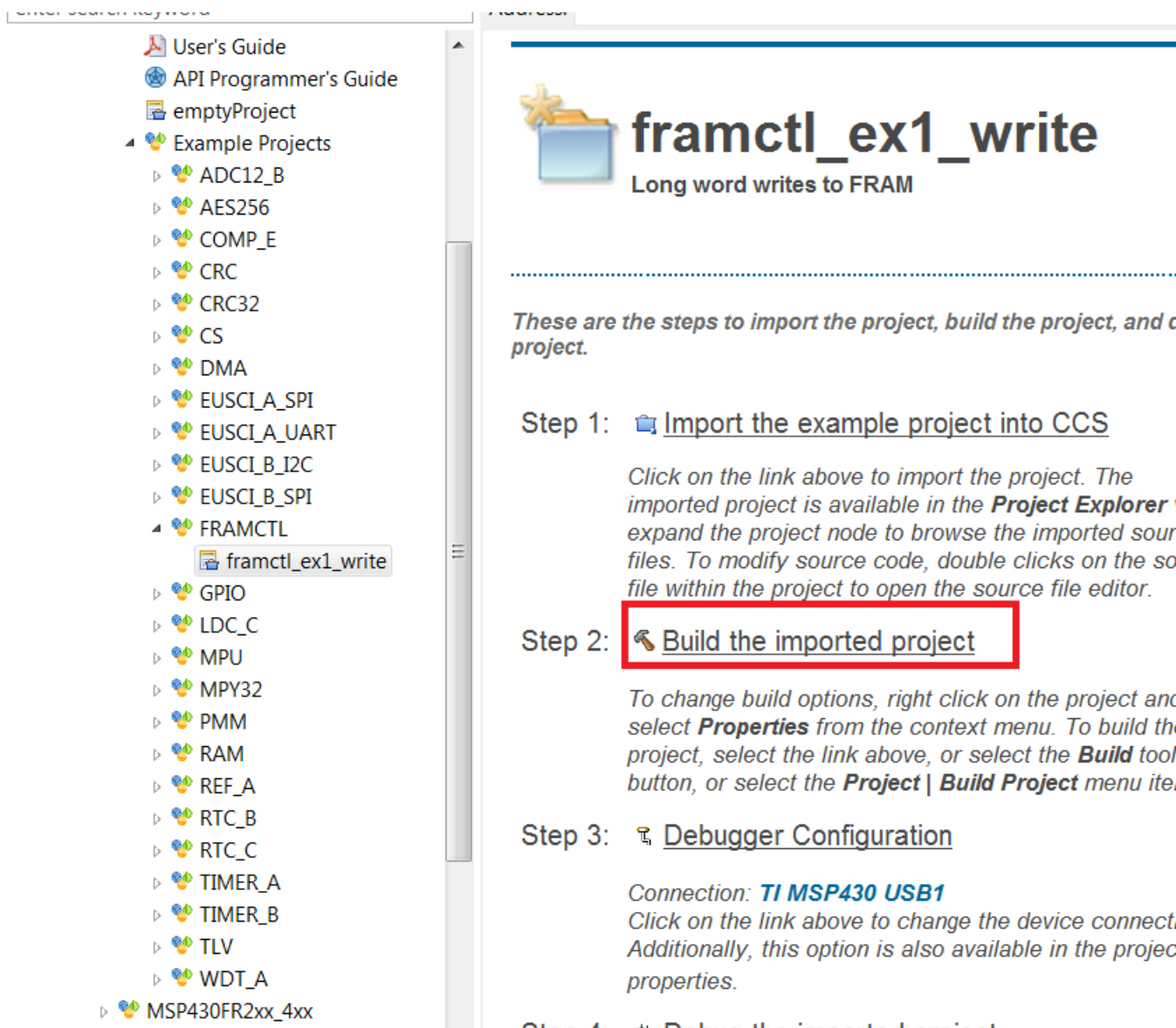
Step 2: [Build the imported project](#)
 To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** tool button, or select the **Project | Build Project** menu item.

Step 3: [Debugger Configuration](#)
 Connection: **TI MSP430 USB1**
 Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

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



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



The screenshot shows the CCS Resource Explorer interface. On the left, a tree view displays various projects under 'Example Projects'. The 'FRAMCTL' project is expanded, and 'framctl_ex1_write' is selected. On the right, the project overview page for 'framctl_ex1_write' is displayed, featuring a title, a subtitle 'Long word writes to FRAM', and a list of steps for importing, building, and configuring the project. The 'Build the imported project' step is highlighted with a red box.

Search for key words

- User's Guide
- API Programmer's Guide
- emptyProject
- Example Projects
 - ADC12_B
 - AES256
 - COMP_E
 - CRC
 - CRC32
 - CS
 - DMA
 - EUSCL_A_SPI
 - EUSCL_A_UART
 - EUSCL_B_I2C
 - EUSCL_B_SPI
 - FRAMCTL
 - framctl_ex1_write**
 - GPIO
 - LDC_C
 - MPU
 - MPY32
 - PMM
 - RAM
 - REF_A
 - RTC_B
 - RTC_C
 - TIMER_A
 - TIMER_B
 - TLV
 - WDT_A
- MSP430FR2xx_4xx

framctl_ex1_write

Long word writes to FRAM

These are the steps to import the project, build the project, and configure 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**. 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](#)

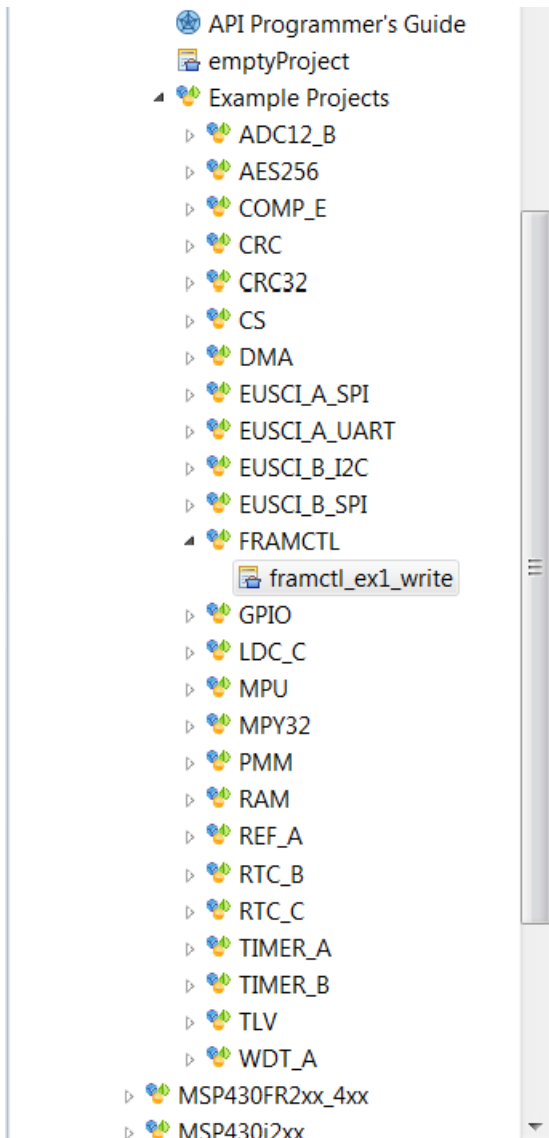
To change build options, right click on the project and select **Properties** from the context menu. To build the project, select the link above, or select the **Build** tool button, or select the **Project | Build Project** menu item.

Step 3: [Debugger Configuration](#)

Connection: **TI MSP430 USB1**

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

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



The screenshot shows the CCS Resource Explorer interface. On the left, a tree view displays the project structure. The 'Example Projects' folder is expanded, and the 'framctl_ex1_write' project is selected. The tree includes various peripheral drivers such as ADC12_B, AES256, COMP_E, CRC, CRC32, CS, DMA, EUSCI_A_SPI, EUSCI_A_UART, EUSCI_B_I2C, EUSCI_B_SPI, GPIO, LDC_C, MPU, MPY32, PMM, RAM, REF_A, RTC_B, RTC_C, TIMER_A, TIMER_B, TLV, and WDT_A. At the bottom, two device folders are visible: 'MSP430FR2xx_4xx' and 'MSP430i2xx'.

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**. 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](#)

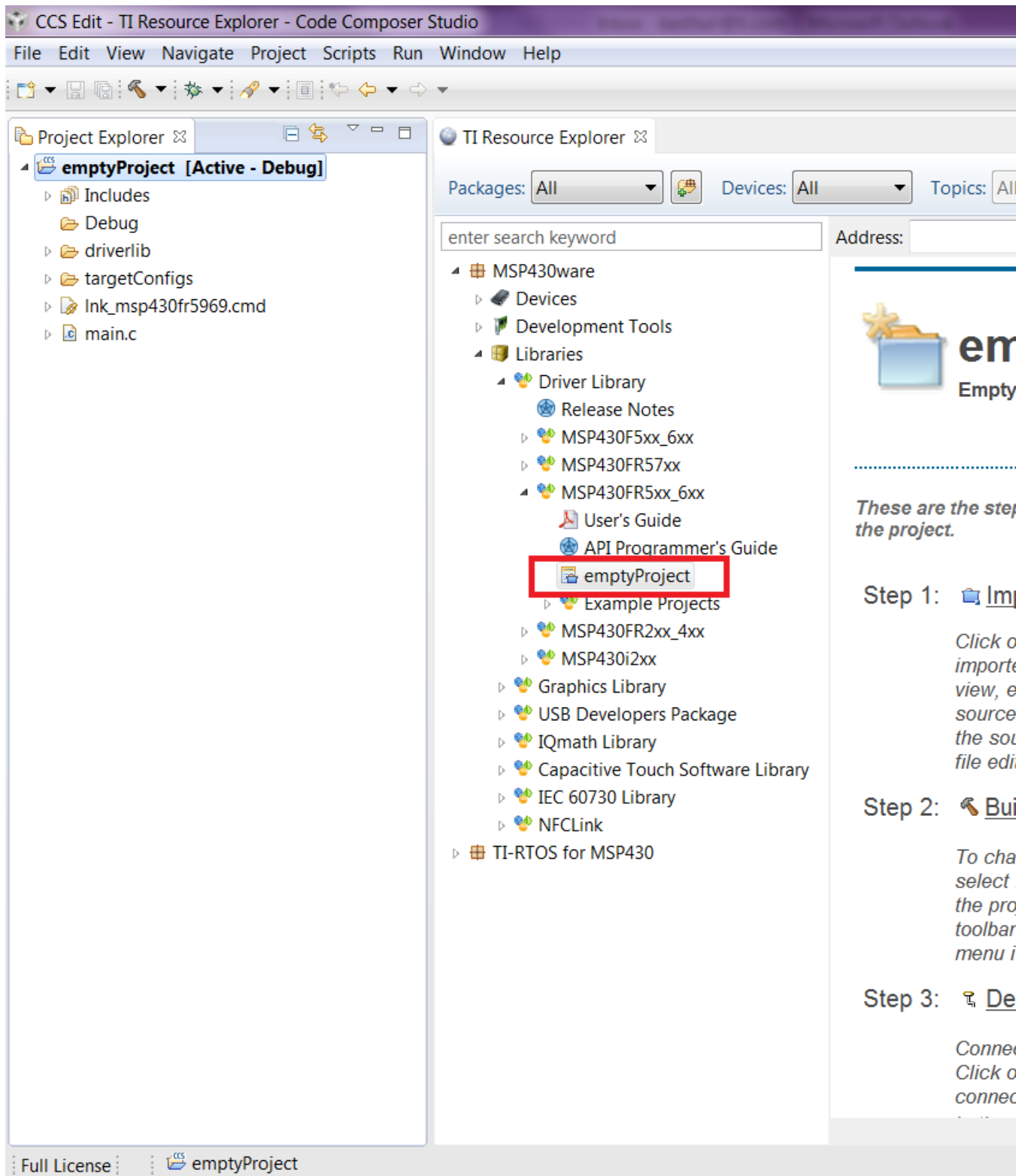
Connection: **TI MSP430 USB1**
Click on the link above to change the device connection. Additionally, this option is also available in the project properties.

Step 4: [Debug the imported project](#)

Click on the link above to launch a debug session for the **framctl_ex1_write** project and switch to the **CCS Debugger 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 stop arrow beside the bug toolbar button and select one of the debug sessions 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.

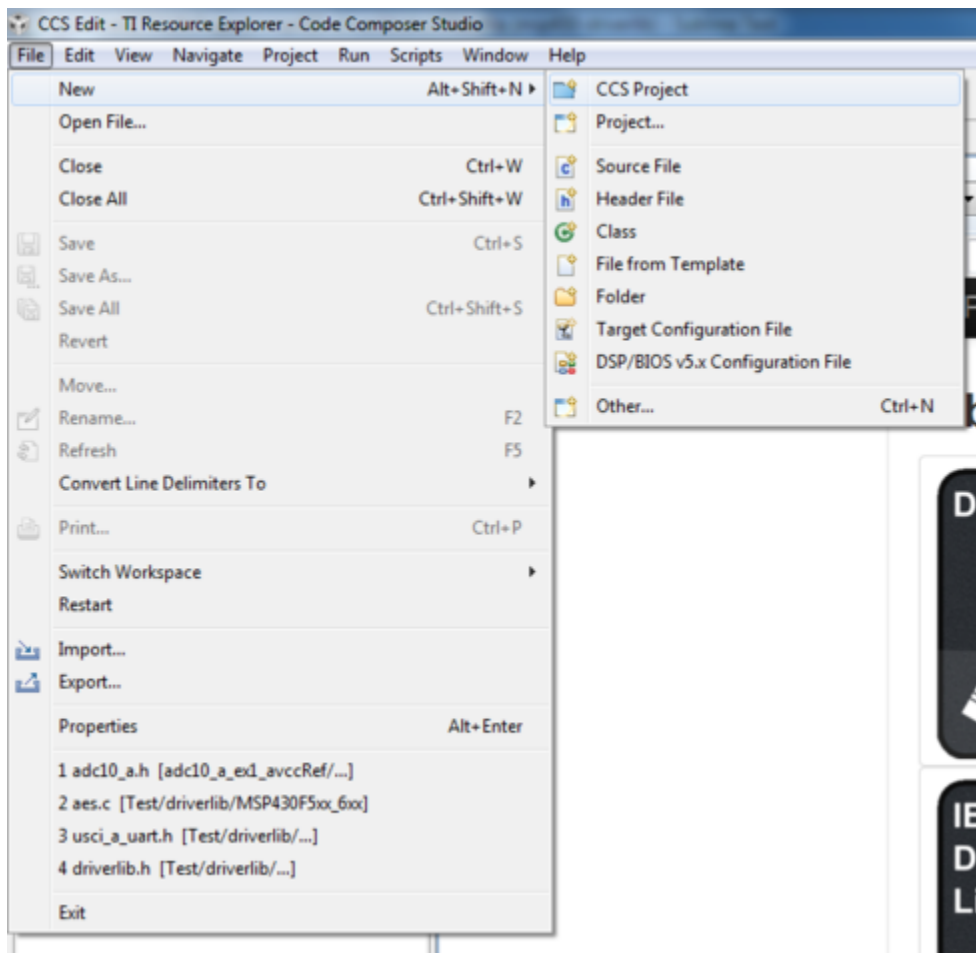


The main.c included with the empty project can be modified to include user code.

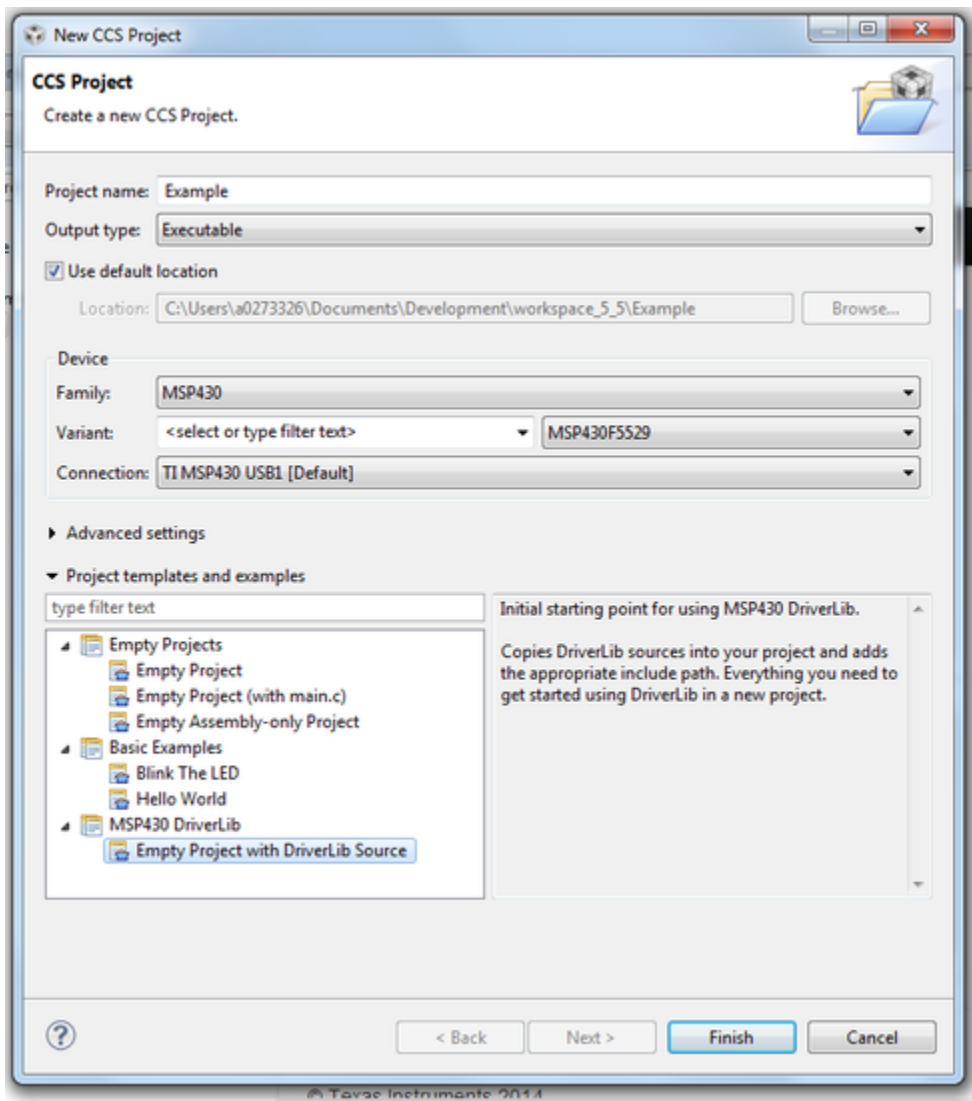
3 How to create a new CCS project that uses Driverlib

3.1 Introduction

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



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



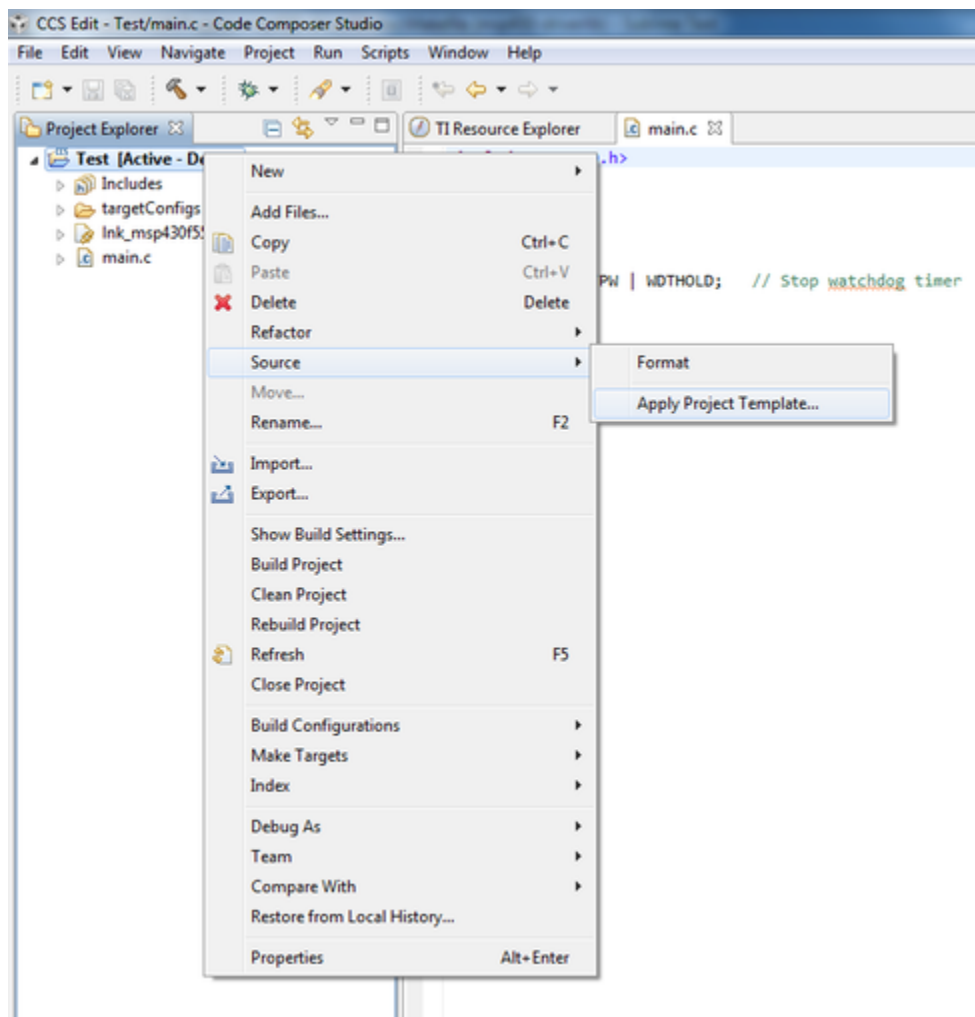
Finally click "Finish" and begin developing with your Driver Library enabled project.

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

4 How to include driverlib into your existing CCS project

4.1 Introduction

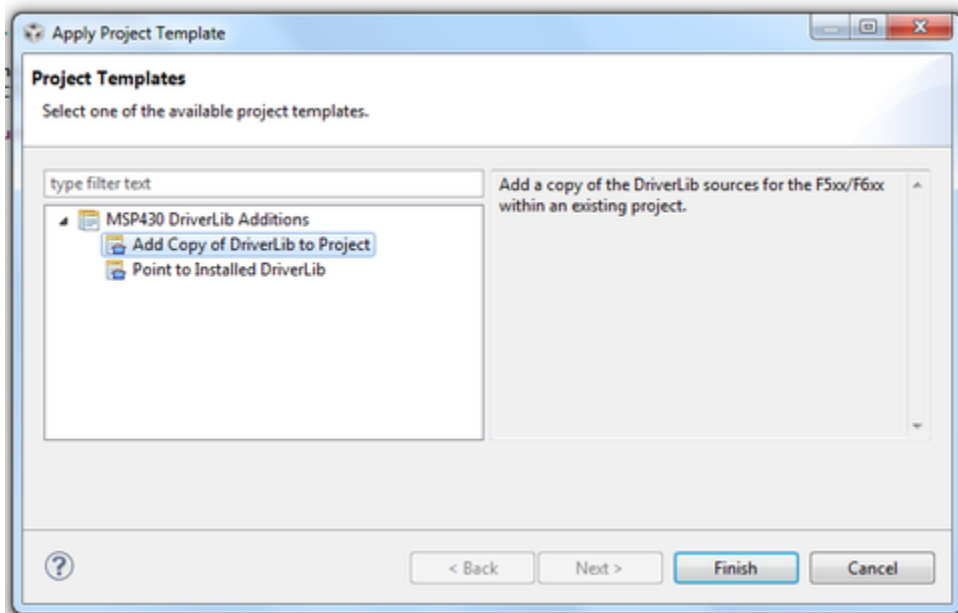
To add driver library to an existing project we recommend using CCS project templates. For driver library to work with project templates CCS must have discovered the driver library RTSC product. For more information refer to the installation steps of the release notes. CCS project templates adds the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



In the "Apply Project Template" dialog box under "MSP430 DriverLib Additions" choose either "Add Local Copy" or "Point to Installed DriverLib" as seen in the screenshot below. Most users will want to add a local copy which copies the DriverLib source into the project and sets the compiler

settings needed.

Pointing to an installed DriverLib is for advanced users who are including a static library in their project and want to add the DriverLib header files to their include path.

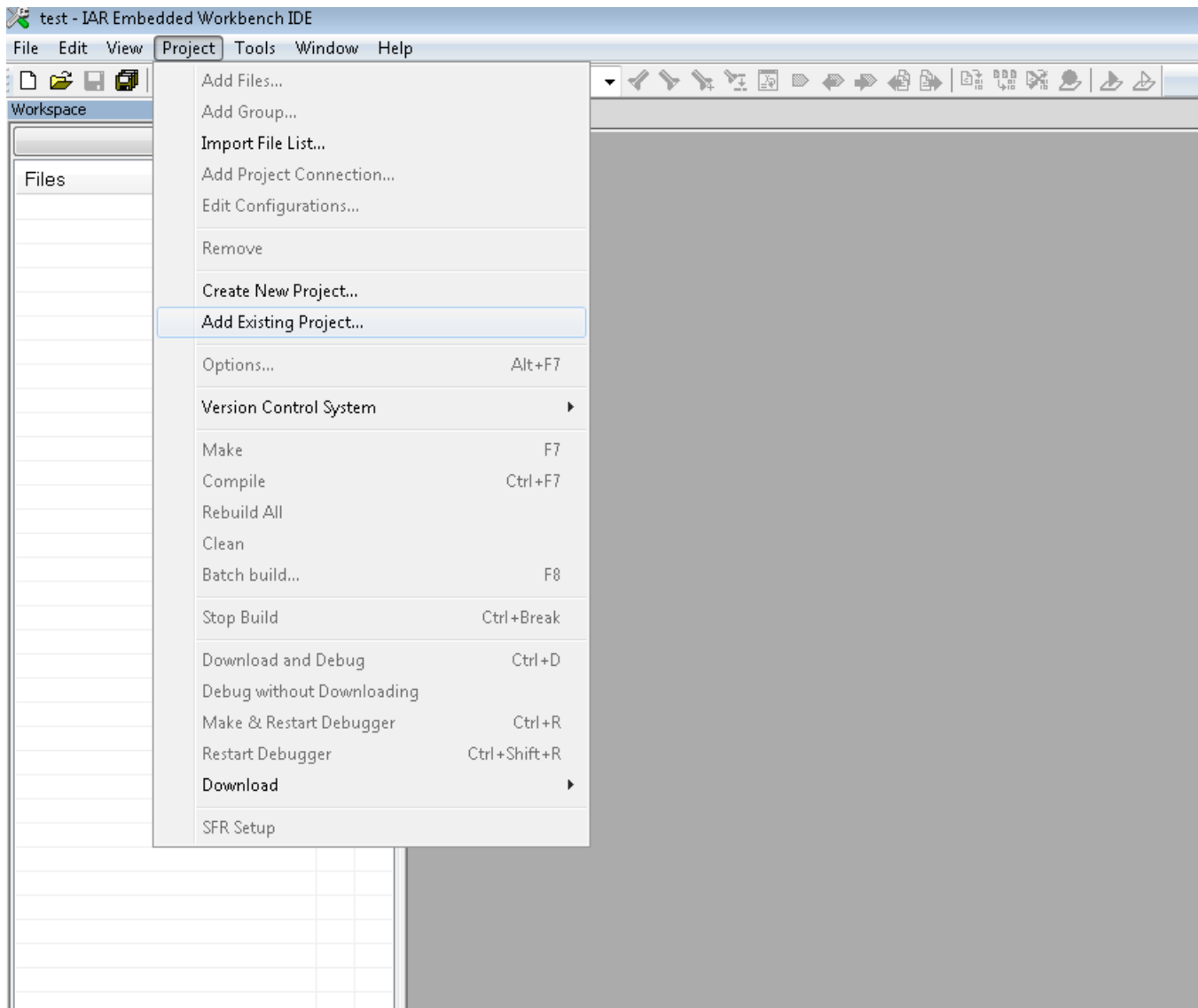


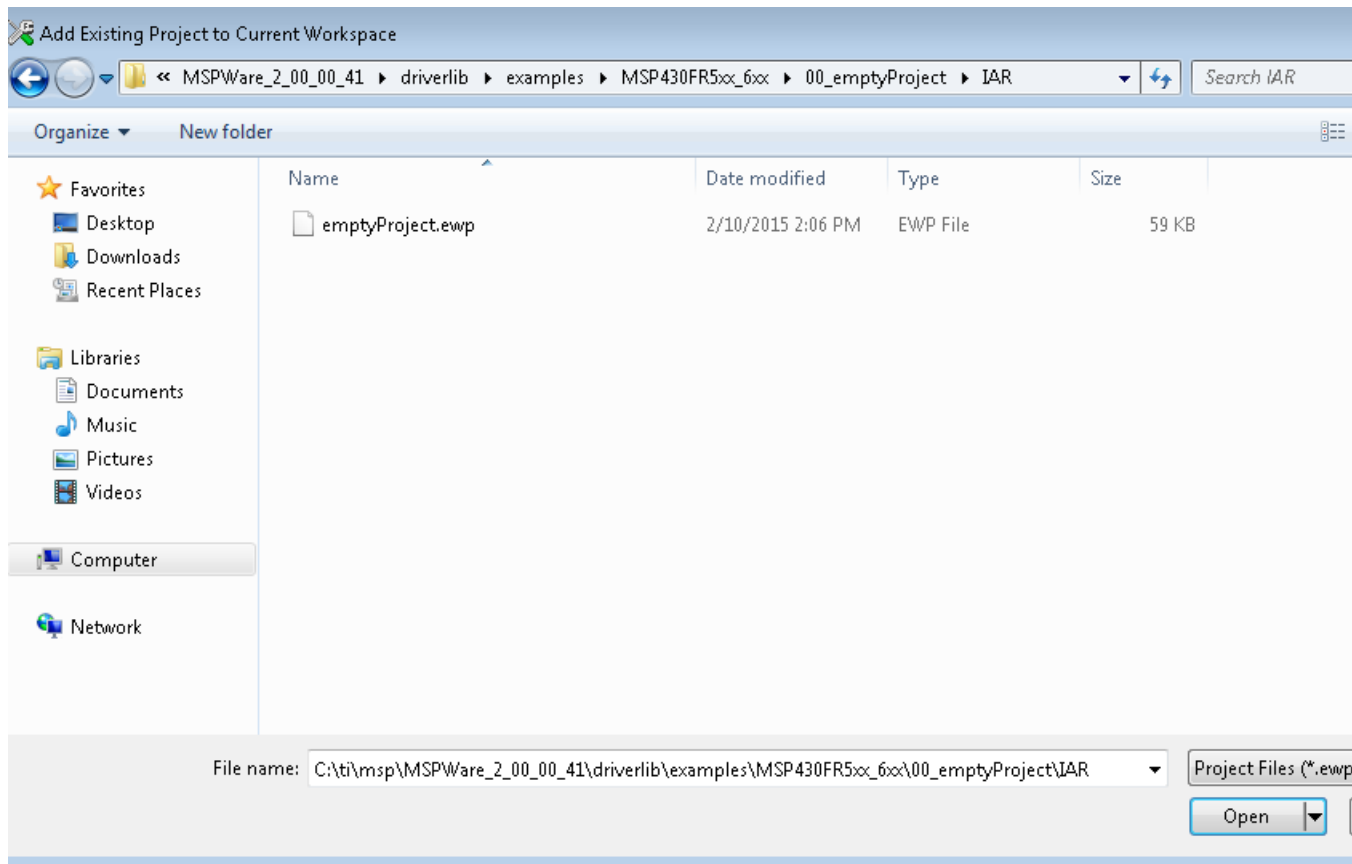
Click "Finish" and start developing with driver library in your project.

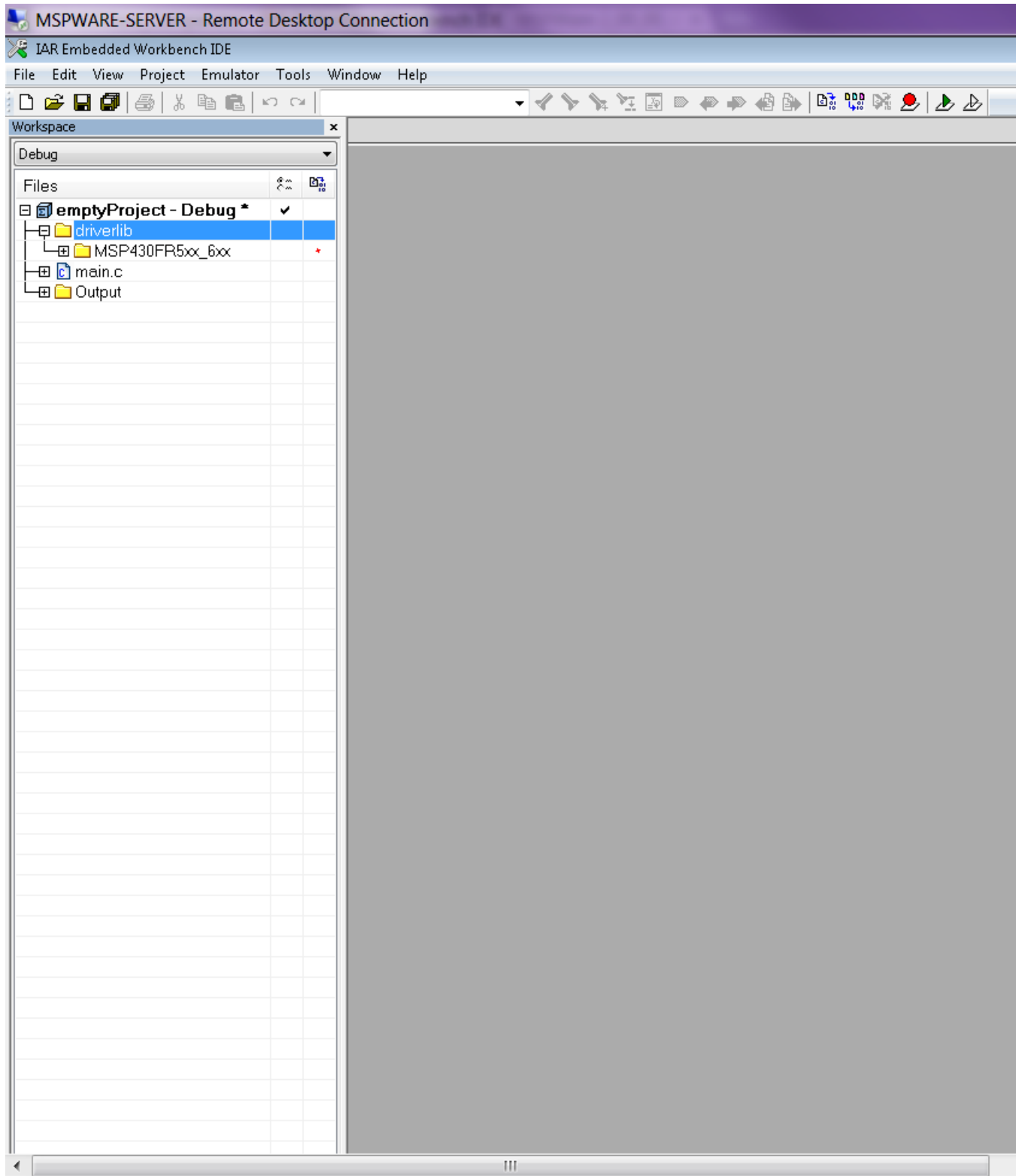
5 How to create a new IAR project that uses Driverlib

5.1 Introduction

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



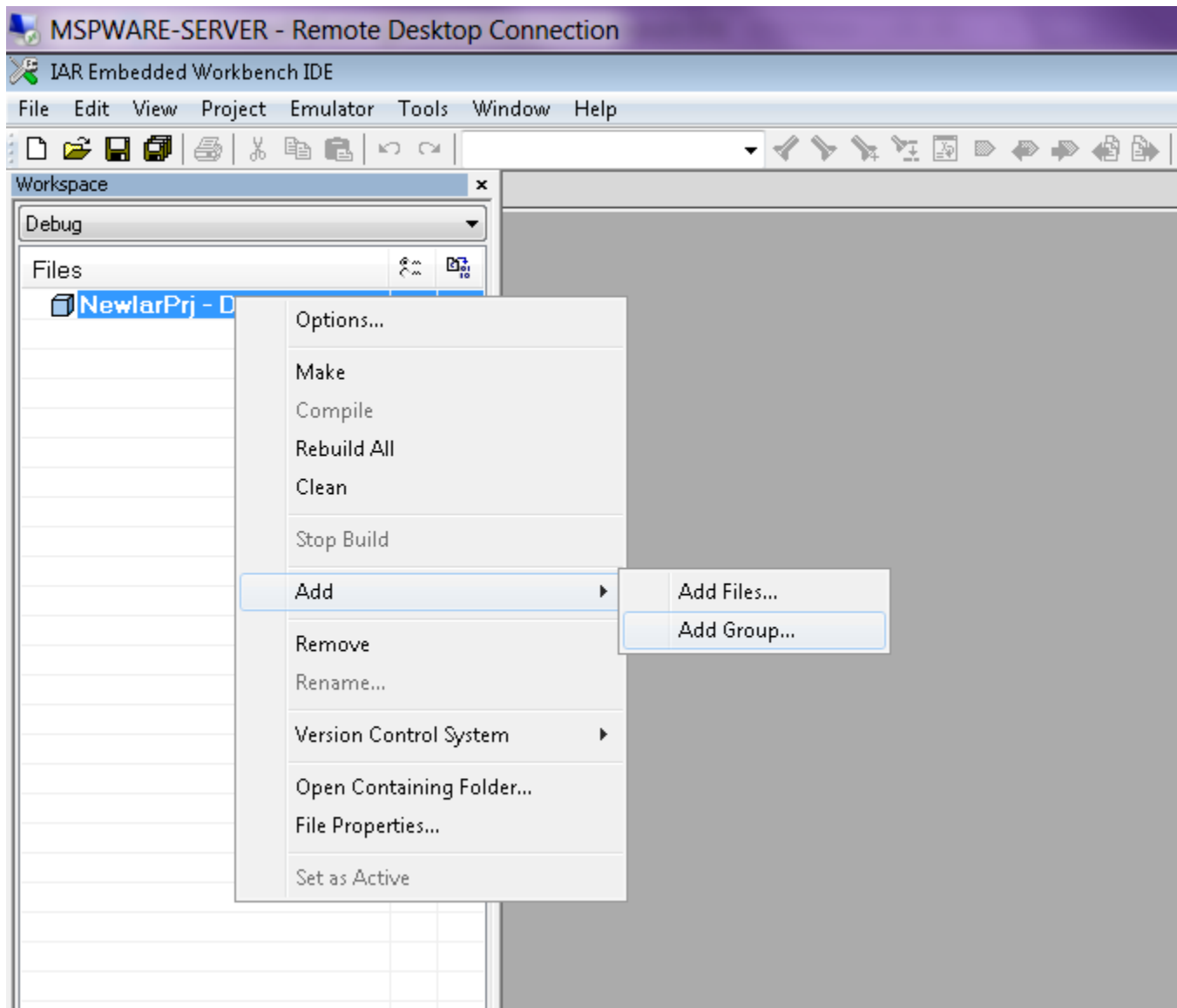




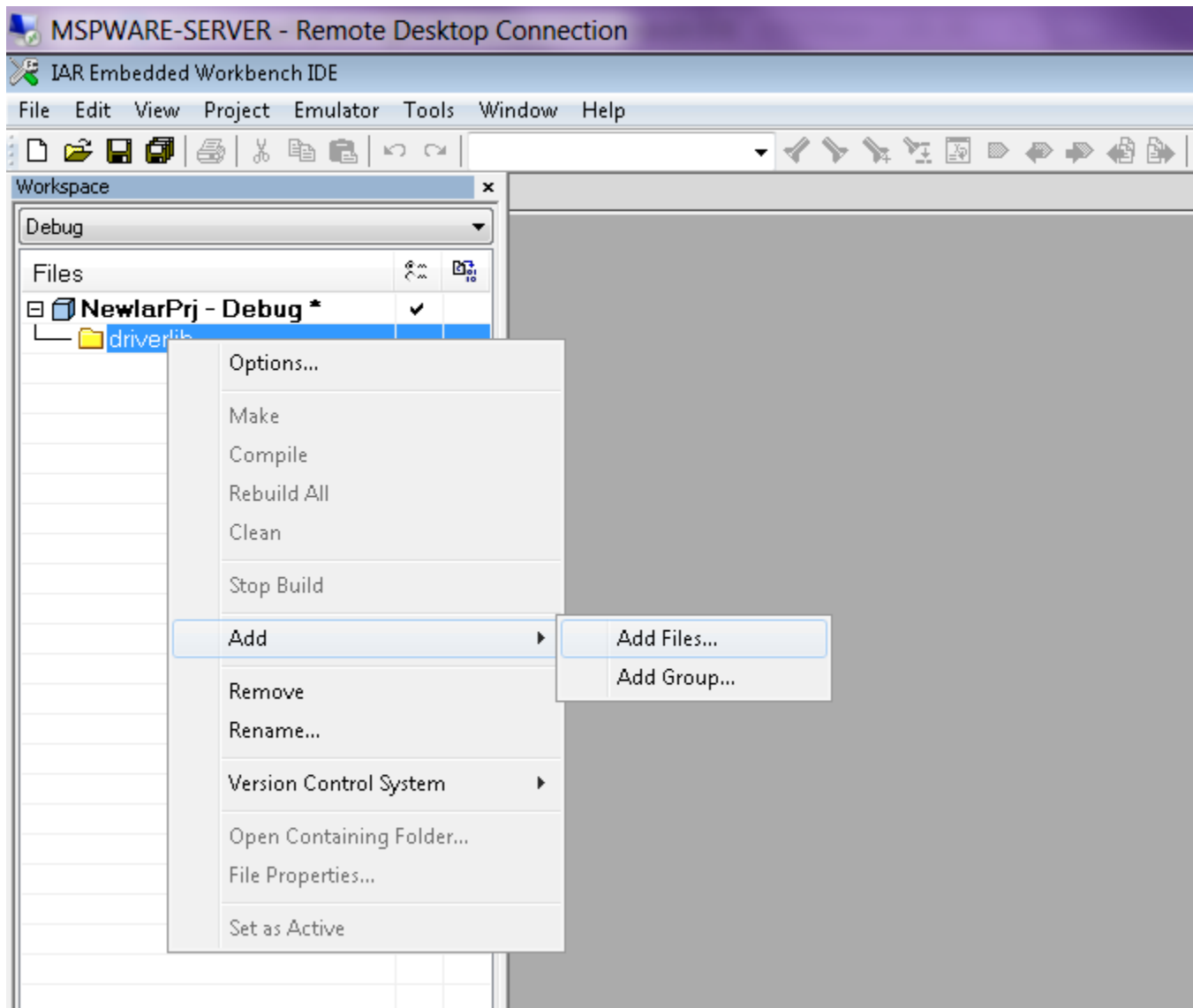
6 How to include driverlib into your existing IAR project

6.1 Introduction

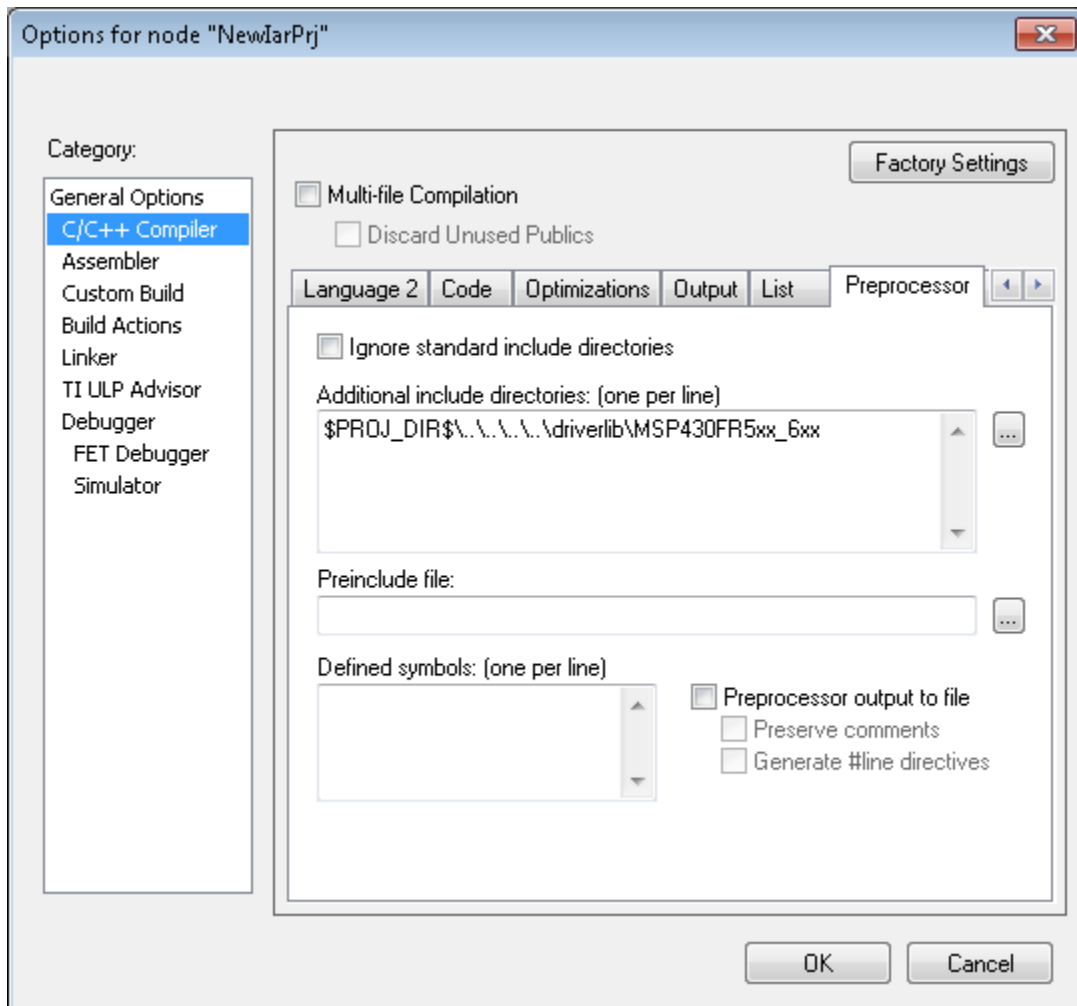
To add driver library to an existing project, right click project click on Add Group - "driverlib"



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



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



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

7 10-Bit Analog-to-Digital Converter (ADC10_A)

Introduction	34
API Functions	34
Programming Example	51

7.1 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 V_{cc} 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.

7.2 API Functions

Functions

- `bool ADC10_A_init` (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
Initializes the ADC10_A Module.
- `void ADC10_A_enable` (uint16_t baseAddress)
Enables the ADC10_A block.
- `void ADC10_A_disable` (uint16_t baseAddress)
Disables the ADC10_A block.
- `void ADC10_A_setupSamplingTimer` (uint16_t baseAddress, uint16_t clockCycleHoldCount, uint16_t multipleSamplesEnabled)
Sets up and enables the Sampling Timer Pulse Mode.
- `void ADC10_A_disableSamplingTimer` (uint16_t baseAddress)
Disables Sampling Timer Pulse Mode.

- void `ADC10_A_configureMemory` (uint16_t baseAddress, uint8_t inputSourceSelect, uint8_t positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect)
Configures the controls of the selected memory buffer.
- void `ADC10_A_enableInterrupt` (uint16_t baseAddress, uint8_t interruptMask)
Enables selected ADC10_A interrupt sources.
- void `ADC10_A_disableInterrupt` (uint16_t baseAddress, uint8_t interruptMask)
Disables selected ADC10_A interrupt sources.
- void `ADC10_A_clearInterrupt` (uint16_t baseAddress, uint8_t interruptFlagMask)
Clears ADC10_A selected interrupt flags.
- uint8_t `ADC10_A_getInterruptStatus` (uint16_t baseAddress, uint8_t interruptFlagMask)
Returns the status of the selected memory interrupt flags.
- void `ADC10_A_startConversion` (uint16_t baseAddress, uint8_t conversionSequenceModeSelect)
Enables/Starts an Analog-to-Digital Conversion.
- void `ADC10_A_disableConversions` (uint16_t baseAddress, bool preempt)
Disables the ADC from converting any more signals.
- int16_t `ADC10_A_getResults` (uint16_t baseAddress)
Returns the raw contents of the specified memory buffer.
- void `ADC10_A_setResolution` (uint16_t baseAddress, uint8_t resolutionSelect)
Use to change the resolution of the converted data.
- void `ADC10_A_setSampleHoldSignalInversion` (uint16_t baseAddress, uint16_t invertedSignal)
Use to invert or un-invert the sample/hold signal.
- void `ADC10_A_setDataReadBackFormat` (uint16_t baseAddress, uint16_t readBackFormat)
Use to set the read-back format of the converted data.
- void `ADC10_A_enableReferenceBurst` (uint16_t baseAddress)
Enables the reference buffer's burst ability.
- void `ADC10_A_disableReferenceBurst` (uint16_t baseAddress)
Disables the reference buffer's burst ability.
- void `ADC10_A_setReferenceBufferSamplingRate` (uint16_t baseAddress, uint16_t samplingRateSelect)
Use to set the reference buffer's sampling rate.
- void `ADC10_A_setWindowComp` (uint16_t baseAddress, uint16_t highThreshold, uint16_t lowThreshold)
Sets the high and low threshold for the window comparator feature.
- uint32_t `ADC10_A_getMemoryAddressForDMA` (uint16_t baseAddress)
Returns the address of the memory buffer for the DMA module.
- uint16_t `ADC10_A_isBusy` (uint16_t baseAddress)
Returns the busy status of the ADC10_A core.

7.2.1 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_configureMemory()`
- `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()`

7.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>interruptFlagMask</i>	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void ADC10_A_configureMemory ( uint16_t baseAddress, uint8_t inputSourceSelect, uint8_t
    positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect )
```

Configures the controls of the selected memory buffer.

Maps an input signal conversion into the memory buffer, as well as the positive and negative reference voltages for each conversion being stored into the memory buffer. If the internal reference is used for the positive reference voltage, the internal REF module has to control the voltage level. Note that if a conversion has been started with the startConversion() function, then a call to disableConversions() is required before this function may be called.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>inputSource</i> ↔ <i>Select</i>	is the input that will store the converted data into the specified memory buffer. Valid values are: <ul style="list-style-type: none"> ■ 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.
<i>positiveRef</i> ↔ <i>Voltage</i> ↔ <i>SourceSelect</i>	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: <ul style="list-style-type: none"> ■ ADC10_A_VREFPOS_AVCC [Default] ■ ADC10_A_VREFPOS_EXT ■ ADC10_A_VREFPOS_INT Modified bits are ADC10SREF of ADC10MCTL0 register.
<i>negativeRef</i> ↔ <i>Voltage</i> ↔ <i>SourceSelect</i>	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: <ul style="list-style-type: none"> ■ ADC10_A_VREFNEG_AVSS ■ ADC10_A_VREFNEG_EXT Modified bits are ADC10SREF of ADC10CTL0 register.

Returns

None

```
void ADC10_A_disable ( uint16_t baseAddress )
```

Disables the ADC10_A block.

This will disable operation of the ADC10_A block.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>preempt</i>	<p>specifies if the current conversion should be pre-empted before the end of the conversion</p> <p>Valid values are:</p> <ul style="list-style-type: none"> ■ 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


```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>interruptMask</i>	is the bit mask of the memory buffer interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

Returns

None

```
void ADC10_A_enable ( uint16_t baseAddress )
```

Enables the ADC10_A block.

This will enable operation of the ADC10_A block.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>interruptMask</i>	is the bit mask of the memory buffer interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>interruptFlagMask</i>	is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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.

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

Returns

The memory address of the memory buffer

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
--------------------	--

Returns

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

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>sampleHold</i> ↔ <i>SignalSource</i> ↔ <i>Select</i>	<p>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:</p> <ul style="list-style-type: none"> ■ ADC10_A_SAMPLEHOLDSOURCE_SC ■ ADC10_A_SAMPLEHOLDSOURCE_1 ■ ADC10_A_SAMPLEHOLDSOURCE_2 ■ ADC10_A_SAMPLEHOLDSOURCE_3 <p>Modified bits are ADC10SHSx of ADC10CTL1 register.</p>
<i>clockSource</i> ↔ <i>Select</i>	<p>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:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC10SSELx of ADC10CTL1 register.</p>
<i>clockSource</i> ↔ <i>Divider</i>	<p>selects the amount that the clock will be divided. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC10DIVx of ADC10CTL1 register; bits ADC10PDIVx of ADC10CTL2 register.</p>

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

`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.

Parameters

<i>baseAddress</i>	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

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>readBackFormat</i>	is the specified format to store the conversions in the memory buffer. Valid values are: <ul style="list-style-type: none"> ■ ADC10_A_UNSIGNED_BINARY [Default] ■ ADC10_A_SIGNED_2SCOMPLEMENT Modified bits are ADC10DF of ADC10CTL2 register.

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>samplingRateSelect</i>	is the specified maximum sampling rate. Valid values are: <ul style="list-style-type: none"> ■ ADC10_A_MAXSAMPLINGRATE_200KSPS [Default] ■ ADC10_A_MAXSAMPLINGRATE_50KSPS Modified bits are ADC10SR of ADC10CTL2 register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>resolutionSelect</i>	determines the resolution of the converted data. Valid values are: <ul style="list-style-type: none"> ■ ADC10_A_RESOLUTION_8BIT ■ ADC10_A_RESOLUTION_10BIT [Default] Modified bits are ADC10RES of ADC10CTL2 register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>invertedSignal</i>	set if the sample/hold signal should be inverted Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>clockCycle↔</i> <i>HoldCount</i>	sets the amount of clock cycles to sample-and- hold for the memory buffer. Valid values are: <ul style="list-style-type: none"> ■ 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.
<i>multiple↔</i> <i>Samples↔</i> <i>Enabled</i>	allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are: <ul style="list-style-type: none"> ■ 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

```
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, ADC10INIIE, ADC10LOIE interrupts to utilize this feature.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>highThreshold</i>	is the upper bound that could trip an interrupt for the window comparator.
<i>lowThreshold</i>	is the lower bound that could trip on interrupt for the window comparator.

Returns

None

```
void ADC10_A_startConversion ( uint16_t baseAddress, uint8_t conversionSequence↵
ModeSelect )
```

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.

Parameters

<i>baseAddress</i>	is the base address of the ADC10_A module.
<i>conversion↵ Sequence↵ ModeSelect</i>	<p>determines the ADC operating mode. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC10CONSEQx of ADC10CTL1 register.</p>

Returns

None

7.3 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.configureMemory (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_A.AIFG0)) );

    // Clear the Interrupt Flag and start another conversion
    ADC10_A.clearInterrupt (ADC10_A.BASE, ADC10_A.AIFG0);
}
```

8 12-Bit Analog-to-Digital Converter (ADC12_A)

Introduction	52
API Functions	52
Programming Example	70

8.1 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.

8.2 API Functions

Functions

- `bool ADC12_A_init` (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
Initializes the ADC12.A Module.
- `void ADC12_A_enable` (uint16_t baseAddress)
Enables the ADC12.A block.
- `void ADC12_A_disable` (uint16_t baseAddress)
Disables the ADC12.A block.
- `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.
- `void ADC12_A_disableSamplingTimer` (uint16_t baseAddress)
Disables Sampling Timer Pulse Mode.

- void `ADC12_A_configureMemory` (uint16_t baseAddress, `ADC12_A_configureMemoryParam` *param)
Configures the controls of the selected memory buffer.
- void `ADC12_A_enableInterrupt` (uint16_t baseAddress, uint32_t interruptMask)
Enables selected ADC12.A interrupt sources.
- void `ADC12_A_disableInterrupt` (uint16_t baseAddress, uint32_t interruptMask)
Disables selected ADC12.A interrupt sources.
- void `ADC12_A_clearInterrupt` (uint16_t baseAddress, uint16_t memoryInterruptFlagMask)
Clears ADC12.A selected interrupt flags.
- uint8_t `ADC12_A_getInterruptStatus` (uint16_t baseAddress, uint16_t memoryInterruptFlagMask)
Returns the status of the selected memory interrupt flags.
- void `ADC12_A_startConversion` (uint16_t baseAddress, uint16_t startingMemoryBufferIndex, uint8_t conversionSequenceModeSelect)
Enables/Starts an Analog-to-Digital Conversion.
- void `ADC12_A_disableConversions` (uint16_t baseAddress, bool preempt)
Disables the ADC from converting any more signals.
- uint16_t `ADC12_A_getResults` (uint16_t baseAddress, uint8_t memoryBufferIndex)
A Signed Integer of the contents of the specified memory buffer.
- void `ADC12_A_setResolution` (uint16_t baseAddress, uint8_t resolutionSelect)
Use to change the resolution of the converted data.
- void `ADC12_A_setSampleHoldSignalInversion` (uint16_t baseAddress, uint16_t invertedSignal)
Use to invert or un-invert the sample/hold signal.
- void `ADC12_A_setDataReadBackFormat` (uint16_t baseAddress, uint8_t readBackFormat)
Use to set the read-back format of the converted data.
- void `ADC12_A_enableReferenceBurst` (uint16_t baseAddress)
Enables the reference buffer's burst ability.
- void `ADC12_A_disableReferenceBurst` (uint16_t baseAddress)
Disables the reference buffer's burst ability.
- void `ADC12_A_setReferenceBufferSamplingRate` (uint16_t baseAddress, uint8_t samplingRateSelect)
Use to set the reference buffer's sampling rate.
- uint32_t `ADC12_A_getMemoryAddressForDMA` (uint16_t baseAddress, uint8_t memoryIndex)
Returns the address of the specified memory buffer for the DMA module.
- uint16_t `ADC12_A_isBusy` (uint16_t baseAddress)
Returns the busy status of the ADC12.A core.

8.2.1 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_configureMemory()`
- `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()

8.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>memory</i> ↔ <i>InterruptFlag</i> ↔ <i>Mask</i>	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void ADC12_A_configureMemory ( uint16_t baseAddress, ADC12_A_configureMemory↔  
Param * 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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>param</i>	is the pointer to struct for memory configuration.

Returns

None

References `ADC12_A_configureMemoryParam::endOfSequence`,
`ADC12_A_configureMemoryParam::inputSourceSelect`,
`ADC12_A_configureMemoryParam::memoryBufferControllIndex`,
`ADC12_A_configureMemoryParam::negativeRefVoltageSourceSelect`, and
`ADC12_A_configureMemoryParam::positiveRefVoltageSourceSelect`.

```
void ADC12_A_disable ( uint16_t baseAddress )
```

Disables the ADC12_A block.

This will disable operation of the ADC12_A block.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>preempt</i>	specifies if the current conversion should be pre-empted before the end of the conversion. Valid values are: <ul style="list-style-type: none"> ■ 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

References ADC12_A.isBusy().

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>interruptMask</i>	Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

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

Returns

None

```
void ADC12_A_enable ( uint16_t baseAddress )
```

Enables the ADC12_A block.

This will enable operation of the ADC12_A block.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>interruptMask</i>	Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>memoryInterruptFlagMask</i>	is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>memoryIndex</i>	is the memory buffer to return the address of. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>memoryBufferIndex</i>	is the specified Memory Buffer to read. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

<i>sampleHold↔</i> <i>SignalSource↔</i> <i>Select</i>	<p>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:</p> <ul style="list-style-type: none"> ■ 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.
<i>clockSource↔</i> <i>Select</i>	<p>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:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC12SSELx of ADC12CTL1 register.</p>
<i>clockSource↔</i> <i>Divider</i>	<p>selects the amount that the clock will be divided. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC12PDIV of ADC12CTL2 register; bits ADC12DIVx of ADC12↔CTL1 register.</p>

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

`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.

Parameters

<i>baseAddress</i>	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

Referenced by ADC12_A.disableConversions().

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>readBackFormat</i>	is the specified format to store the conversions in the memory buffer. Valid values are: <ul style="list-style-type: none"> ■ ADC12_A_UNSIGNED_BINARY [Default] ■ ADC12_A_SIGNED_2SCOMPLEMENT Modified bits are ADC12DF of ADC12CTL2 register.

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>samplingRateSelect</i>	is the specified maximum sampling rate. Valid values are: <ul style="list-style-type: none"> ■ ADC12_A_MAXSAMPLINGRATE_200KSPS [Default] ■ ADC12_A_MAXSAMPLINGRATE_50KSPS Modified bits are ADC12SR of ADC12CTL2 register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
<i>resolutionSelect</i>	determines the resolution of the converted data. Valid values are: <ul style="list-style-type: none"> ■ ADC12_A_RESOLUTION_8BIT ■ ADC12_A_RESOLUTION_10BIT ■ ADC12_A_RESOLUTION_12BIT [Default] Modified bits are ADC12RESx of ADC12CTL2 register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the ADC12_A module.
--------------------	--

<i>invertedSignal</i>	set if the sample/hold signal should be inverted Valid values are: <ul style="list-style-type: none"> ■ 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

```
void ADC12_A_setupSamplingTimer ( uint16_t baseAddress, uint16_t clockCycle↔
  HoldCountLowMem, 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.

Parameters

<p><i>baseAddress</i></p> <p><i>clockCycle</i>↔ <i>HoldCount</i>↔ <i>LowMem</i></p>	<p>is the base address of the ADC12_A module.</p> <p>sets the amount of clock cycles to sample- and-hold for the higher memory buffers 0-7. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC12SHT0x of ADC12CTL0 register.</p>
<p><i>clockCycle</i>↔ <i>HoldCount</i>↔ <i>HighMem</i></p>	<p>sets the amount of clock cycles to sample-and-hold for the higher memory buffers 8-15. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC12SHT1x of ADC12CTL0 register.</p>

<i>multiple</i> ↔ <i>Samples</i> ↔ <i>Enabled</i>	allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are: <ul style="list-style-type: none"> ■ 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

```
void ADC12_A_startConversion ( uint16_t baseAddress, uint16_t starting↔
    MemoryBufferIndex, 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.

Parameters

<p><i>baseAddress</i></p> <p><i>starting</i>↔</p> <p><i>MemoryBuffer</i>↔</p> <p><i>Index</i></p>	<p>is the base address of the ADC12_A module.</p> <p>is the memory buffer that will hold the first or only conversion. Valid values are:</p> <ul style="list-style-type: none"> ■ 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 <p>Modified bits are ADC12STARTADDx of ADC12CTL1 register.</p>
<p><i>conversion</i>↔</p> <p><i>Sequence</i>↔</p> <p><i>ModeSelect</i></p>	<p>determines the ADC operating mode. Valid values are:</p> <ul style="list-style-type: none"> ■ 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. <p>Modified bits are ADC12CONSEQx of ADC12CTL1 register.</p>

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

Returns

None

8.3 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_configureMemoryParam param = {0};
param.memoryBufferControlIndex = ADC12_A_MEMORY_0;
param.inputSourceSelect = ADC12_A_INPUT_A0;
param.positiveRefVoltageSourceSelect = ADC12_A_VREFPOS_AVCC;
param.negativeRefVoltageSourceSelect = ADC12_A_VREFNEG_AVSS;
param.endOfSequence = ADC12_A_NOTENDOFSEQUENCE;
ADC12_A_configureMemory (ADC12_A_BASE, &param);
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, ADC12_IFG0)) );

    // Clear the Interrupt Flag and start another conversion
    ADC12_A_clearInterrupt (ADC12_A_BASE, ADC12_IFG0);
}
```

9 Advanced Encryption Standard (AES)

Introduction	71
API Functions	71
Programming Example	79

9.1 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 128 bit - 168 cycles 192 bit - 204 cycles 256 bit - 234 cycles AES decryption 128 bit - 168 cycles 192 bit - 206 cycles 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

9.2 API Functions

Functions

- uint8_t [AES_setCipherKey](#) (uint16_t baseAddress, const uint8_t *CipherKey)
Loads a 128 bit cipher key to AES module.
- uint8_t [AES_encryptData](#) (uint16_t baseAddress, const uint8_t *Data, uint8_t *encryptedData)
Encrypts a block of data using the AES module.
- uint8_t [AES_decryptData](#) (uint16_t baseAddress, const uint8_t *Data, uint8_t *decryptedData)
Decrypts a block of data using the AES module.
- uint8_t [AES_setDecipherKey](#) (uint16_t baseAddress, const uint8_t *CipherKey)
Sets the decipher key The API.
- void [AES_clearInterrupt](#) (uint16_t baseAddress)
Clears the AES ready interrupt flag.
- uint32_t [AES_getInterruptStatus](#) (uint16_t baseAddress)
Gets the AES ready interrupt flag status.
- void [AES_enableInterrupt](#) (uint16_t baseAddress)

- Enables AES ready interrupt.*
- void `AES_disableInterrupt` (uint16_t baseAddress)
- Disables AES ready interrupt.*
- void `AES_reset` (uint16_t baseAddress)
- Resets AES Module immediately.*
- uint8_t `AES_startEncryptData` (uint16_t baseAddress, const uint8_t *Data, uint8_t *encryptedData)
- Starts an encryption process on the AES module.*
- uint8_t `AES_startDecryptData` (uint16_t baseAddress, const uint8_t *Data)
- Decrypts a block of data using the AES module.*
- uint8_t `AES_startSetDecipherKey` (uint16_t baseAddress, const uint8_t *CipherKey)
- Loads the decipher key.*
- uint8_t `AES_getDataOut` (uint16_t baseAddress, uint8_t *OutputData)
- Reads back the output data from AES module.*
- uint8_t `AES_isBusy` (uint16_t baseAddress)
- Gets the AES module busy status.*
- void `AES_clearErrorFlag` (uint16_t baseAddress)
- Clears the AES error flag.*
- uint32_t `AES_getErrorFlagStatus` (uint16_t baseAddress)
- Gets the AES error flag status.*
- uint8_t `AES_startDecryptDataUsingEncryptionKey` (uint16_t baseAddress, const uint8_t *Data)
- DEPRECATED Starts an decryption process on the AES module.*
- uint8_t `AES_decryptDataUsingEncryptionKey` (uint16_t baseAddress, const uint8_t *Data, uint8_t *decryptedData)
- DEPRECATED Decrypts a block of data using the AES module.*

9.2.1 Detailed Description

The AES module APIs are

- `AES_setCipherKey()`
- `AES_encryptData()`
- `AES_decryptDataUsingEncryptionKey()`
- `AES_setDecipherKey()`
- `AES_decryptData()`
- `AES_reset()`
- `AES_startEncryptData()`
- `AES_startDecryptDataUsingEncryptionKey()`
- `AES_startDecryptData()`
- `AES_startSetDecipherKey()`
- `AES_getDataOut()`

The AES interrupt handler functions

- `AES_enableInterrupt()`
- `AES_disableInterrupt()`
- `AES_clearInterrupt()`
- `AES_getInterruptStatus`

9.2.2 Function Documentation

`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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

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

Returns

None

`void AES_clearInterrupt (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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

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

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an <code>uint8_t</code> array with a length of 16 bytes that contains encrypted data to be decrypted.

<i>decryptedData</i>	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

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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.
<i>decryptedData</i>	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

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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

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

Returns

None

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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

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

Returns

None

```
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\(\)](#)

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.
<i>encryptedData</i>	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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>OutputData</i>	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

```
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](#).

Parameters

<i>baseAddress</i>	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

`uint32_t AES_getInterruptStatus (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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

Returns

`uint32_t` - AES_READY_INTERRUPT or 0x00.

`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.

Parameters

<i>baseAddress</i>	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

`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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
--------------------	--

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

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>CipherKey</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains a 128 bit cipher key.

Returns

STATUS_SUCCESS

`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\(\)](#).

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>CipherKey</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains the initial AES key.

Returns

STATUS_SUCCESS

`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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

Returns

STATUS_SUCCESS

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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains encrypted data to be decrypted.

Returns

STATUS_SUCCESS

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.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>Data</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains data to be encrypted.
<i>encryptedData</i>	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

```
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()`.

Parameters

<i>baseAddress</i>	is the base address of the AES module.
<i>CipherKey</i>	is a pointer to an uint8_t array with a length of 16 bytes that contains the initial AES key.

Returns

STATUS_SUCCESS

9.3 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);
```


10 Battery Backup System

Introduction	80
API Functions	80

10.1 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.

10.2 API Functions

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

- [BattBak_unlockBackupSubSystem\(\)](#)
- [BattBak_enableBackupSupplyToADC\(\)](#)
- [BattBak_disableBackupSupplyToADC\(\)](#)
- [BattBak_switchToBackupSupplyManually\(\)](#)
- [BattBak_disable\(\)](#)

The BATBCK charger is controlled by

- [BattBak_initAndEnableCharger\(\)](#)
- [BattBak_disableCharger\(\)](#)

The backup RAM space is accessed by

- [BattBak_setBackupRAMData\(\)](#)
- [BattBak_getBackupRAMData\(\)](#)

11 Comparator (COMP_B)

Introduction	81
API Functions	81
Programming Example	91

11.1 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.

11.2 API Functions

Functions

- `bool Comp_B_init (uint16_t baseAddress, Comp_B_initParam *param)`
Initializes the Comp_B Module.
- `void Comp_B_configureReferenceVoltage (uint16_t baseAddress, Comp_B_configureReferenceVoltageParam *param)`
Generates a Reference Voltage to the terminal selected during initialization.
- `void Comp_B_enableInterrupt (uint16_t baseAddress, uint16_t interruptMask)`
Enables selected Comp_B interrupt sources.
- `void Comp_B_disableInterrupt (uint16_t baseAddress, uint16_t interruptMask)`
Disables selected Comp_B interrupt sources.
- `void Comp_B_clearInterrupt (uint16_t baseAddress, uint16_t interruptFlagMask)`
Clears Comp_B interrupt flags.
- `uint8_t Comp_B_getInterruptStatus (uint16_t baseAddress, uint16_t interruptFlagMask)`
Gets the current Comp_B interrupt status.
- `void Comp_B_setInterruptEdgeDirection (uint16_t baseAddress, uint16_t edgeDirection)`
Explicitly sets the edge direction that would trigger an interrupt.
- `void Comp_B_toggleInterruptEdgeDirection (uint16_t baseAddress)`
Toggles the edge direction that would trigger an interrupt.
- `void Comp_B_enable (uint16_t baseAddress)`
Turns on the Comp_B module.
- `void Comp_B.disable (uint16_t baseAddress)`
Turns off the Comp_B module.
- `void Comp_B.shortInputs (uint16_t baseAddress)`

- Shorts the two input pins chosen during initialization.*
- void `Comp_B_unshortInputs` (uint16_t baseAddress)
- Disables the short of the two input pins chosen during initialization.*
- 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.*
- void `Comp_B_enableInputBuffer` (uint16_t baseAddress, uint8_t inputPort)
- Enables the input buffer of the selected input port to allow for digital signals.*
- void `Comp_B_swapIO` (uint16_t baseAddress)
- Toggles the bit that swaps which terminals the inputs go to, while also inverting the output of the Comp_B.*
- uint16_t `Comp_B_outputValue` (uint16_t baseAddress)
- Returns the output value of the Comp_B module.*

11.2.1 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_setInterruptEdgeDirection()`
- `Comp_B.toggleInterruptEdgeDirection()`

Auxiliary features of the COMP_B are handled by

- `Comp_B.enableShortOfInputs()`
- `Comp_B.disableShortOfInputs()`
- `Comp_B.disableInputBuffer()`
- `Comp_B.enableInputBuffer()`
- `Comp_B.swapIO()`

11.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>interruptFlag</i> ↔ <i>Mask</i>	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ COMP_B_OUTPUT_FLAG - Output interrupt ■ COMP_B_OUTPUTINVERTED_FLAG - Output interrupt inverted polarity Modified bits of CBINT register.

Returns

None

```
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: $V_{base} * (\text{Numerator} / 32)$. 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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>param</i>	is the pointer to struct for reference voltage configuration.

Returns

None

References `Comp_B_configureReferenceVoltageParam::lowerLimitSupplyVoltageFractionOf32`, `Comp_B_configureReferenceVoltageParam::referenceAccuracy`, `Comp_B_configureReferenceVoltageParam::supplyVoltageReferenceBase`, and `Comp_B_configureReferenceVoltageParam::upperLimitSupplyVoltageFractionOf32`.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>inputPort</i>	is the port in which the input buffer will be disabled. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>interruptMask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ COMP_B_OUTPUT_INT - Output interrupt ■ COMP_B_OUTPUTINVERTED_INT - Output interrupt inverted polarity Modified bits of CBINT register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>inputPort</i>	is the port in which the input buffer will be enabled. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>interruptMask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ COMP_B.OUTPUT_INT - Output interrupt ■ COMP_B.OUTPUTINVERTED_INT - Output interrupt inverted polarity Modified bits of CBINT register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>interruptFlagMask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
bool Comp_B_init ( uint16_t baseAddress, Comp_B_initParam * 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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>param</i>	is the pointer to struct for initialization.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

References [Comp_B_initParam::invertedOutputPolarity](#), [Comp_B_initParam::negativeTerminalInput](#), [Comp_B_initParam::outputFilterEnableAndDelayLevel](#), [Comp_B_initParam::positiveTerminalInput](#), and [Comp_B_initParam::powerModeSelect](#).

```
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.

Parameters

<i>baseAddress</i>	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

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

Explicitly sets the edge direction that would trigger an interrupt.

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

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
<i>edgeDirection</i>	<p>determines which direction the edge would have to go to generate an interrupt based on the non-inverted interrupt flag. Valid values are:</p> <ul style="list-style-type: none"> ■ 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). <p>Modified bits are CBIES of CBCTL1 register.</p>

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

```
void Comp_B_swapIO ( 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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

```
void Comp_B_toggleInterruptEdgeDirection ( uint16_t baseAddress )
```

Toggles the edge direction that would trigger an interrupt.

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

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the COMP_B module.
--------------------	---

Returns

None

11.3 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
*/
CompB_initParam param = {0};
param.positiveTerminalInput = COMP_B.INPUT0;
param.negativeTerminalInput = COMP_B.VREF;
param.powerModeSelect = COMP_B.POWERMODE_NORMALMODE;
param.outputFilterEnableAndDelayLevel = COMP_B.FILTEROUTPUT_DLYLVL1;
param.invertedOutputPolarity = COMP_B.NORMALOUTPUTPOLARITY;
CompB_init(COMP_B.BASE, &param);

// 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
*/
CompB_setReferenceVoltage(COMP_B.BASE,
                        COMP_B.VREFBASE2.5V,
                        32,
                        32
                        );

// Allow power to Comparator module
CompB_enable(COMP_B.BASE);

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

12 Cyclical Redundancy Check (CRC)

Introduction	93
API Functions	93
Programming Example	96

12.1 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).

12.2 API Functions

Functions

- void [CRC.setSeed](#) (uint16_t baseAddress, uint16_t seed)
Sets the seed for the CRC.
- void [CRC.set16BitData](#) (uint16_t baseAddress, uint16_t dataIn)
Sets the 16 bit data to add into the CRC module to generate a new signature.
- void [CRC.set8BitData](#) (uint16_t baseAddress, uint8_t dataIn)
Sets the 8 bit data to add into the CRC module to generate a new signature.
- void [CRC.set16BitDataReversed](#) (uint16_t baseAddress, uint16_t dataIn)
Translates the 16 bit data by reversing the bits in each byte and then sets this data to add into the CRC module to generate a new signature.
- void [CRC.set8BitDataReversed](#) (uint16_t baseAddress, uint8_t dataIn)
Translates the 8 bit data by reversing the bits in each byte and then sets this data to add into the CRC module to generate a new signature.
- uint16_t [CRC.getData](#) (uint16_t baseAddress)
Returns the value currently in the Data register.
- uint16_t [CRC.getResult](#) (uint16_t baseAddress)
Returns the value of the Signature Result.
- uint16_t [CRC.getResultBitsReversed](#) (uint16_t baseAddress)
Returns the bit-wise reversed format of the Signature Result.

12.2.1 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\(\)](#)

12.2.2 Function Documentation

`uint16_t CRC_getData (uint16_t baseAddress)`

Returns the value currently in the Data register.

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

Parameters

<i>baseAddress</i>	is the base address of the CRC module.
--------------------	--

Returns

The value currently in the data register

`uint16_t CRC_getResult (uint16_t baseAddress)`

Returns the value of the Signature Result.

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

Parameters

<i>baseAddress</i>	is the base address of the CRC module.
--------------------	--

Returns

The value currently in the data register

`uint16_t CRC_getResultBitsReversed (uint16_t baseAddress)`

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

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

Parameters

<i>baseAddress</i>	is the base address of the CRC module.
--------------------	--

Returns

The bit-wise reversed format of the Signature Result

`void CRC_set16BitData (uint16_t baseAddress, uint16_t dataIn)`

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

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

Parameters

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

Returns

None

`void CRC_set16BitDataReversed (uint16_t baseAddress, uint16_t dataIn)`

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

This function first reverses the bits in each byte of the data and then generates the new signature from the current signature and new translated data.

Parameters

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

Returns

None

`void CRC_set8BitData (uint16_t baseAddress, uint8_t dataIn)`

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

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

Parameters

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

Returns

None

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

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

This function first reverses the bits in each byte of the data and then generates the new signature from the current signature and new translated data.

Parameters

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

Returns

None

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

Sets the seed for the CRC.

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

Parameters

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

Returns

None

12.3 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);
```

13 12-bit Digital-to-Analog Converter (DAC12_A)

Introduction	98
API Functions	98
Programming Example	109

13.1 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.

13.2 API Functions

Functions

- bool `DAC12_A_init` (uint16_t baseAddress, `DAC12_A_initParam` *param)
Initializes the DAC12_A module with the specified settings.
- void `DAC12_A_setAmplifierSetting` (uint16_t baseAddress, uint8_t submoduleSelect, uint8_t amplifierSetting)
Sets the amplifier settings for the Vref+ and Vout buffers.
- void `DAC12_A_disable` (uint16_t baseAddress, uint8_t submoduleSelect)
Clears the amplifier settings to disable the DAC12_A module.
- void `DAC12_A_enableGrouping` (uint16_t baseAddress)
Enables grouping of two DAC12_A modules in a dual DAC12_A system.
- void `DAC12_A_disableGrouping` (uint16_t baseAddress)
Disables grouping of two DAC12_A modules in a dual DAC12_A system.
- void `DAC12_A_enableInterrupt` (uint16_t baseAddress, uint8_t submoduleSelect)
Enables the DAC12_A module interrupt source.
- void `DAC12_A_disableInterrupt` (uint16_t baseAddress, uint8_t submoduleSelect)
Disables the DAC12_A module interrupt source.
- uint16_t `DAC12_A_getInterruptStatus` (uint16_t baseAddress, uint8_t submoduleSelect)

- Returns the status of the DAC12_A module interrupt flag.*
- void `DAC12_A_clearInterrupt` (uint16_t baseAddress, uint8_t submoduleSelect)
Clears the DAC12_A module interrupt flag.
- void `DAC12_A_calibrateOutput` (uint16_t baseAddress, uint8_t submoduleSelect)
Calibrates the output offset.
- uint16_t `DAC12_A_getCalibrationData` (uint16_t baseAddress, uint8_t submoduleSelect)
Returns the calibrated offset of the output buffer.
- void `DAC12_A_setCalibrationOffset` (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t calibrationOffsetValue)
Returns the calibrated offset of the output buffer.
- void `DAC12_A_enableConversions` (uint16_t baseAddress, uint8_t submoduleSelect)
Enables triggers to start conversions.
- void `DAC12_A_setData` (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t data)
Sets the given data into the buffer to be converted.
- void `DAC12_A_disableConversions` (uint16_t baseAddress, uint8_t submoduleSelect)
Disables triggers to start conversions.
- void `DAC12_A_setResolution` (uint16_t baseAddress, uint8_t submoduleSelect, uint16_t resolutionSelect)
Sets the resolution to be used by the DAC12_A module.
- 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.
- 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.

13.2.1 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\(\)](#)

13.2.2 Function Documentation

```
void DAC12_A_calibrateOutput ( uint16_t baseAddress, uint8_t submoduleSelect )
```

Calibrates the output offset.

This function disables the calibration lock, starts the calibration, waits 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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submoduleSelect</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submoduleSelect</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
--------------------	--

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submoduleSelect</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

Returns

The calibrated offset of the output buffer.

`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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

Returns

The address of the specified memory buffer

`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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ 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

```
bool DAC12_A.init ( uint16_t baseAddress, DAC12_A_initParam * 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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>param</i>	is the pointer to struct for initialization.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

References `DAC12_A_initParam::amplifierSetting`, `DAC12_A_initParam::conversionTriggerSelect`, `DAC12_A_initParam::outputSelect`, `DAC12_A_initParam::outputVoltageMultiplier`, `DAC12_A_initParam::positiveReferenceVoltage`, and `DAC12_A_initParam::submoduleSelect`.

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

<i>amplifierSetting</i>	<p>is the setting of the settling speed and current of the Vref+ and the Vout buffer. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule↔ Select</i>	<p>decides which DAC12_A sub-module to configure. Valid values are:</p> <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1

<i>calibration</i> ↔ <i>OffsetValue</i>	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

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submodule</i> ↔ <i>Select</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1
<i>data</i>	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

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submoduleSelect</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1
<i>inputJustification</i>	is the justification of the data to be converted. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_JUSTIFICATION_RIGHT [Default] ■ DAC12_A_JUSTIFICATION_LEFT Modified bits are DAC12DFJ of DAC12_xCTL1 register.
<i>inputSign</i>	is the sign of the data to be converted. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_UNSIGNED_BINARY [Default] ■ DAC12_A_SIGNED_2SCOMPLEMENT Modified bits are DAC12DF of DAC12_xCTL0 register.

Returns

None

```
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

<i>baseAddress</i>	is the base address of the DAC12_A module.
<i>submoduleSelect</i>	decides which DAC12_A sub-module to configure. Valid values are: <ul style="list-style-type: none"> ■ DAC12_A_SUBMODULE_0 ■ DAC12_A_SUBMODULE_1
<i>resolutionSelect</i>	is the resolution to use for conversions. Valid values are: <ul style="list-style-type: none"> ■ 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

13.3 Programming Example

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

```

DAC12_A_initParam param = {0};
param.submoduleSelect = DAC12_A_SUBMODULE_0;
param.outputSelect = DAC12_A_OUTPUT_1;
param.positiveReferenceVoltage = DAC12_A_VREF_AVCC;
param.outputVoltageMultiplier = DAC12_A_VREFx1;
param.amplifierSetting = DAC12_A_AMP_MEDIN_MEDOUT;
param.conversionTriggerSelect = DAC12_A_TRIGGER_ENCByPASS;
DAC12_A_init(DAC12_A_BASE, &param);

// 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
                );

```

14 Direct Memory Access (DMA)

Introduction	110
API Functions	110
Programming Example	122

14.1 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.

14.2 API Functions

Functions

- void `DMA_init` (`DMA_initParam *param`)
Initializes the specified DMA channel.
- void `DMA_setTransferSize` (`uint8_t channelSelect`, `uint16_t transferSize`)
Sets the specified amount of transfers for the selected DMA channel.
- `uint16_t DMA_getTransferSize` (`uint8_t channelSelect`)
Gets the amount of transfers for the selected DMA channel.
- 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.
- 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.*

 - void [DMA_enableTransfers](#) (uint8_t channelSelect)
Enables transfers to be triggered.
 - void [DMA_disableTransfers](#) (uint8_t channelSelect)
Disables transfers from being triggered.
 - void [DMA_startTransfer](#) (uint8_t channelSelect)
Starts a transfer if using the default trigger source selected in initialization.
 - void [DMA_enableInterrupt](#) (uint8_t channelSelect)
Enables the DMA interrupt for the selected channel.
 - void [DMA_disableInterrupt](#) (uint8_t channelSelect)
Disables the DMA interrupt for the selected channel.
 - uint16_t [DMA_getInterruptStatus](#) (uint8_t channelSelect)
Returns the status of the interrupt flag for the selected channel.
 - void [DMA_clearInterrupt](#) (uint8_t channelSelect)
Clears the interrupt flag for the selected channel.
 - uint16_t [DMA_getNMIAbortStatus](#) (uint8_t channelSelect)
Returns the status of the NMIAbort for the selected channel.
 - void [DMA_clearNMIAbort](#) (uint8_t channelSelect)
Clears the status of the NMIAbort to proceed with transfers for the selected channel.
 - void [DMA_disableTransferDuringReadModifyWrite](#) (void)
Disables the DMA from stopping the CPU during a Read-Modify-Write Operation to start a transfer.
 - void [DMA_enableTransferDuringReadModifyWrite](#) (void)
Enables the DMA to stop the CPU during a Read-Modify-Write Operation to start a transfer.
 - void [DMA_enableRoundRobinPriority](#) (void)
Enables Round Robin prioritization.
 - void [DMA_disableRoundRobinPriority](#) (void)
Disables Round Robin prioritization.
 - void [DMA_enableNMIAbort](#) (void)
Enables a NMI to interrupt a DMA transfer.
 - void [DMA_disableNMIAbort](#) (void)
Disables any NMI from interrupting a DMA transfer.

14.2.1 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\(\)](#) [DMA_getTransferSize\(\)](#)

The DMA interrupts are handled by: [DMA_enableInterrupt\(\)](#) [DMA_disableInterrupt\(\)](#) [DMA_getInterruptStatus\(\)](#) [DMA_clearInterrupt\(\)](#) [DMA_getNMIAbortStatus\(\)](#) [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\(\)](#)

14.2.2 Function Documentation

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.

Parameters

<i>channelSelect</i>	<p>is the specified channel to clear the interrupt flag for. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

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.

Parameters

<i>channelSelect</i>	<p>is the specified channel to clear the NMI Abort flag for. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to disable the interrupt for. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Returns

None

```
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).

Returns

None

```
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.

Returns

None

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to disable transfers for. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>channelSelect</i>	<p>is the specified channel to enable the interrupt for. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

```
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.

Returns

None

```
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.

Returns

None

```
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.

Returns

None

```
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.

Parameters

<i>channelSelect</i>	<p>is the specified channel to enable transfer for. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>channelSelect</i>	<p>is the specified channel to return the interrupt flag status from. Valid values are:</p> <ul style="list-style-type: none"> ■ 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

uint16_t DMA_getNMIAbortStatus (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.

Parameters

<i>channelSelect</i>	is the specified channel to return the status of the NMI Abort flag for. Valid values are: <ul style="list-style-type: none"> ■ 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

uint16_t DMA_getTransferSize (uint8_t *channelSelect*)

Gets the amount of transfers for the selected DMA channel.

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

Parameters

<i>channelSelect</i>	is the specified channel to set source address direction for. Valid values are: <ul style="list-style-type: none"> ■ 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

the amount of transfers

```
void DMA_init ( DMA_initParam * 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.

Parameters

<i>param</i>	is the pointer to struct for initialization.
--------------	--

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

References `DMA_initParam::channelSelect`, `DMA_initParam::transferModeSelect`, `DMA_initParam::transferSize`, `DMA_initParam::transferUnitSelect`, `DMA_initParam::triggerSourceSelect`, and `DMA_initParam::triggerTypeSelect`.

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to set the destination address direction for. Valid values are: <ul style="list-style-type: none"> ■ DMA_CHANNEL_0 ■ DMA_CHANNEL_1 ■ DMA_CHANNEL_2 ■ DMA_CHANNEL_3 ■ DMA_CHANNEL_4 ■ DMA_CHANNEL_5 ■ DMA_CHANNEL_6 ■ DMA_CHANNEL_7
<i>dstAddress</i>	is the address of where the data will be transferred to. Modified bits are DMAxDA of DMAxDA register.
<i>directionSelect</i>	is the specified direction of the destination address after a transfer. Valid values are: <ul style="list-style-type: none"> ■ DMA_DIRECTION_UNCHANGED ■ DMA_DIRECTION_DECREMENT ■ DMA_DIRECTION_INCREMENT Modified bits are DMADSTINCR of DMAxCTL register.

Returns

None

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to set source address direction for. Valid values are: <ul style="list-style-type: none"> ■ DMA_CHANNEL_0 ■ DMA_CHANNEL_1 ■ DMA_CHANNEL_2 ■ DMA_CHANNEL_3 ■ DMA_CHANNEL_4 ■ DMA_CHANNEL_5 ■ DMA_CHANNEL_6 ■ DMA_CHANNEL_7
<i>srcAddress</i>	is the address of where the data will be transferred from. Modified bits are DMAxSA of DMAxSA register.
<i>directionSelect</i>	is the specified direction of the source address after a transfer. Valid values are: <ul style="list-style-type: none"> ■ DMA_DIRECTION_UNCHANGED ■ DMA_DIRECTION_DECREMENT ■ DMA_DIRECTION_INCREMENT Modified bits are DMASRCINCR of DMAxCTL register.

Returns

None

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to set source address direction for. Valid values are: <ul style="list-style-type: none"> ■ DMA_CHANNEL_0 ■ DMA_CHANNEL_1 ■ DMA_CHANNEL_2 ■ DMA_CHANNEL_3 ■ DMA_CHANNEL_4 ■ DMA_CHANNEL_5 ■ DMA_CHANNEL_6 ■ DMA_CHANNEL_7
<i>transferSize</i>	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

```
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.

Parameters

<i>channelSelect</i>	is the specified channel to start transfers for. Valid values are: <ul style="list-style-type: none"> ■ 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

14.3 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_initParam param = {0};
param.channelSelect = DMA_CHANNEL_0;
param.transferModeSelect = DMA_TRANSFER_REPEATED_BLOCK;
param.transferSize = 16;
param.triggerSourceSelect = DMA_TRIGGER_SOURCE_0;
param.transferUnitSelect = DMA_SIZE_SRCWORD_DSTWORD;
param.triggerTypeSelect = DMA_TRIGGER_RISING_EDGE;
DMA_init(&param);
/*
 * 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);
}
```

15 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)

Introduction	123
API Functions	123
Programming Example	132

15.1 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.

15.2 API Functions

Functions

- bool [EUSCI_A_UART_init](#) (uint16_t baseAddress, [EUSCI_A_UART_initParam](#) *param)
Advanced initialization routine for the UART block. The values to be written into the clockPrescaler, firstModReg, secondModReg and overSampling parameters should be pre-computed and passed into the initialization function.
- void [EUSCI_A_UART_transmitData](#) (uint16_t baseAddress, uint8_t transmitData)
Transmits a byte from the UART Module.
- uint8_t [EUSCI_A_UART_receiveData](#) (uint16_t baseAddress)
Receives a byte that has been sent to the UART Module.
- void [EUSCI_A_UART_enableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Enables individual UART interrupt sources.
- void [EUSCI_A_UART_disableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Disables individual UART interrupt sources.
- uint8_t [EUSCI_A_UART_getInterruptStatus](#) (uint16_t baseAddress, uint8_t mask)
Gets the current UART interrupt status.
- void [EUSCI_A_UART_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)

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

15.2.1 Detailed Description

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

The EUSCI_A_UART API is broken into three groups of functions: those that deal with configuration and control of the EUSCI_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 EUSCI_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 EUSCI_UART is handled by the

- [EUSCI_A_UART_transmitData\(\)](#)
- [EUSCI_A_UART_receiveData\(\)](#)

- EUSCI_A_UART_transmitAddress()
- EUSCI_A_UART_transmitBreak()
- EUSCI_A_UART_getTransmitBufferAddress()
- EUSCI_A_UART_getTransmitBufferAddress()

Managing the EUSCI_UART interrupts and status are handled by the

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

15.2.2 Function Documentation

`void EUSCI_A_UART_clearInterrupt (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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>mask</i>	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`void EUSCI_A_UART_disable (uint16_t baseAddress)`

Disables the UART block.

This will disable operation of the UART block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

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

Enables the UART block.

This will enable operation of the UART block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

Returns

Address of RX Buffer

`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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

Returns

Address of TX Buffer

`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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>param</i>	is the pointer to struct for initialization.

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

Returns

STATUS_SUCCESS or STATUS_FAIL of the initialization process

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

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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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

<i>baseAddress</i>	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`.

`void EUSCI_A_UART_resetDormant (uint16_t baseAddress)`

Re-enables UART module from dormant mode.
Not dormant. All received characters set UCRXIFG.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

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

Returns

None

`void EUSCI_A_UART_selectDeglitchTime (uint16_t baseAddress, uint16_t deglitchTime)`

Sets the deglitch time.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>deglitchTime</i>	is the selected deglitch time Valid values are: <ul style="list-style-type: none"> ■ 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

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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

Modified bits of **UCAxCTL1** register.

Returns

None

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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
<i>transmitAddress</i>	is the next byte to be transmitted

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

Returns

None

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

<i>baseAddress</i>	is the base address of the EUSCI_A_UART module.
--------------------	---

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

Returns

None

```
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

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

Modified bits of **UCAxTXBUF** register.

Returns

None

15.3 Programming Example

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

```
// Configure UART
EUSCI_A_UART_initParam param = {0};
param.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_ACLK;
param.clockPrescalar = 15;
param.firstModReg = 0;
param.secondModReg = 68;
param.parity = EUSCI_A_UART_NO_PARITY;
param.msborLsbFirst = EUSCI_A_UART_LSB_FIRST;
param.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT;
param.uartMode = EUSCI_A_UART_MODE;
param.overSampling = EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION;

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

EUSCI_A_UART_enable (EUSCI_A0_BASE);

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

16 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)

Introduction	133
API Functions	133
Programming Example	142

16.1 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.

16.2 Functions

Functions

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

- Gets the current SPI interrupt status.*
- void [EUSCIA_SPI_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Clears the selected SPI interrupt status flag.*
- void [EUSCIA_SPI_enable](#) (uint16_t baseAddress)
 - Enables the SPI block.*
- void [EUSCIA_SPI_disable](#) (uint16_t baseAddress)
 - Disables the SPI block.*
- uint32_t [EUSCIA_SPI_getReceiveBufferAddress](#) (uint16_t baseAddress)
 - Returns the address of the RX Buffer of the SPI for the DMA module.*
- uint32_t [EUSCIA_SPI_getTransmitBufferAddress](#) (uint16_t baseAddress)
 - Returns the address of the TX Buffer of the SPI for the DMA module.*
- uint16_t [EUSCIA_SPI_isBusy](#) (uint16_t baseAddress)
 - Indicates whether or not the SPI bus is busy.*

16.2.1 Detailed Description

To use the module as a master, the user must call [EUSCIA_SPI_initMaster\(\)](#) to configure the SPI Master. This is followed by enabling the SPI module using [EUSCIA_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 [EUSCIA_SPI_transmitData\(\)](#) and then when the receive flag is set, the received data is read using [EUSCIA_SPI_receiveData\(\)](#) and this indicates that an RX/TX operation is complete.

To use the module as a slave, initialization is done using [EUSCIA_SPI_initSlave\(\)](#) and this is followed by enabling the module using [EUSCIA_SPI_enable\(\)](#). Following this, the interrupts may be enabled as needed. When the receive flag is set, data is first transmitted using [EUSCIA_SPI_transmitData\(\)](#) and this is followed by a data reception by [EUSCIA_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

- [EUSCIA_SPI_initMaster\(\)](#)
- [EUSCIA_SPI_initSlave\(\)](#)
- [EUSCIA_SPI_disable\(\)](#)
- [EUSCIA_SPI_enable\(\)](#)
- [EUSCIA_SPI_masterChangeClock\(\)](#)
- [EUSCIA_SPI_isBusy\(\)](#)
- [EUSCIA_SPI_select4PinFunctionality\(\)](#)
- [EUSCIA_SPI_changeClockPhasePolarity\(\)](#)

Data handling is done by

- [EUSCIA_SPI_transmitData\(\)](#)
- [EUSCIA_SPI_receiveData\(\)](#)

Interrupts from the SPI module are managed using

- [EUSCIA_SPI_disableInterrupt\(\)](#)

- [EUSCI_A_SPI.enableInterrupt\(\)](#)
- [EUSCI_A_SPI.getInterruptStatus\(\)](#)
- [EUSCI_A_SPI.clearInterrupt\(\)](#)

DMA related

- [EUSCI_A_SPI.getReceiveBufferAddressForDMA\(\)](#)
- [EUSCI_A_SPI.getTransmitBufferAddressForDMA\(\)](#)

16.2.2 Function Documentation

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT [Default] ■ EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT
<i>clockPolarity</i>	is clock polarity select Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
<i>param</i>	is the pointer to struct for master clock setting.

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

Returns

None

References EUSCI.A.SPI.changeMasterClockParam::clockSourceFrequency, and EUSCI.A.SPI.changeMasterClockParam::desiredSpiClock.

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

Clears the selected SPI interrupt status flag.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
<i>mask</i>	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI.A.SPI_TRANSMIT_INTERRUPT ■ EUSCI.A.SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIFG** register.

Returns

None

```
void EUSCI_A_SPI_disable ( uint16_t baseAddress )
```

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI_A_SPI_TRANSMIT_INTERRUPT ■ EUSCI_A_SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIE** register.

Returns

None

```
void EUSCI_A_SPI_enable ( uint16_t baseAddress )
```

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI_A_SPI_TRANSMIT_INTERRUPT ■ EUSCI_A_SPI_RECEIVE_INTERRUPT

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

Returns

None

`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

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
--------------------	--

Returns

the address of the RX Buffer

`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

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
--------------------	--

Returns

the address of the TX Buffer

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A.SPI Master module.
<i>param</i>	is the pointer to struct for master initialization.

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

Returns

STATUS_SUCCESS

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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A.SPI Slave module.
<i>param</i>	is the pointer to struct for slave initialization.

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

Returns

STATUS_SUCCESS

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

```
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

<i>baseAddress</i>	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

```
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

<i>baseAddress</i>	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.

```
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

<i>baseAddress</i>	is the base address of the EUSCI.A.SPI module.
<i>select4Pin↔ Functionality</i>	selects 4 pin functionality Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_A_SPI module.
<i>transmitData</i>	data to be transmitted from the SPI module

Returns

None

16.3 Programming Example

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

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

//Enable SPI Module
EUSCI_A_SPI_enable(EUSCI_A0_BASE);

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

17 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)

Introduction	143
API Functions	143
Programming Example	152

17.1 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.

17.2 Functions

Functions

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

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

17.2.1 Detailed Description

To use the module as a master, the user must call `EUSCI_B_SPI_masterInit()` to configure the SPI Master. This is followed by enabling the SPI module using `EUSCI_B_SPI_enable()`. The interrupts are then enabled (if needed). It is recommended to enable the SPI module before enabling the interrupts. A data transmit is then initiated using `EUSCI_B_SPI_transmitData()` and then when the receive flag is set, the received data is read using `EUSCI_B_SPI_receiveData()` and this indicates that an RX/TX operation is complete.

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

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

The status and initialization of the SPI module are managed by

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

Data handling is done by

- `EUSCI_B_SPI_transmitData()`
- `EUSCI_B_SPI_receiveData()`

Interrupts from the SPI module are managed using

- `EUSCI_B_SPI_disableInterrupt()`

- [EUSCI_B_SPI.enableInterrupt\(\)](#)
- [EUSCI_B_SPI.getInterruptStatus\(\)](#)
- [EUSCI_B_SPI.clearInterrupt\(\)](#)

DMA related

- [EUSCI_B_SPI.getReceiveBufferAddressForDMA\(\)](#)
- [EUSCI_B_SPI.getTransmitBufferAddressForDMA\(\)](#)

17.2.2 Function Documentation

```
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

<i>baseAddress</i>	is the base address of the EUSCI_B_SPI module.
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT [Default] ■ EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT
<i>clockPolarity</i>	is clock polarity select Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_B_SPI module.
<i>param</i>	is the pointer to struct for master clock setting.

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

Returns

None

References EUSCI.B.SPI.changeMasterClockParam::clockSourceFrequency, and EUSCI.B.SPI.changeMasterClockParam::desiredSpiClock.

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

Clears the selected SPI interrupt status flag.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>mask</i>	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI_B_SPI_TRANSMIT_INTERRUPT ■ EUSCI_B_SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIFG** register.

Returns

None

```
void EUSCI_B_SPI_disable ( uint16_t baseAddress )
```

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI.B.SPI_TRANSMIT_INTERRUPT ■ EUSCI.B.SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIE** register.

Returns

None

```
void EUSCI_B_SPI_enable ( uint16_t baseAddress )
```

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ EUSCI.B.SPI_TRANSMIT_INTERRUPT ■ EUSCI.B.SPI_RECEIVE_INTERRUPT

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
--------------------	--

Returns

the address of the RX Buffer

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
--------------------	--

Returns

the address of the TX Buffer

```
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

<i>baseAddress</i>	is the base address of the EUSCI_B.SPI Master module.
<i>param</i>	is the pointer to struct for master initialization.

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

Returns

STATUS_SUCCESS

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

```
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

<i>baseAddress</i>	is the base address of the EUSCI_B.SPI Slave module.
<i>param</i>	is the pointer to struct for slave initialization.

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

Returns

STATUS_SUCCESS

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

```
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

<i>baseAddress</i>	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

```
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

<i>baseAddress</i>	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.

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>select4Pin↔ Functionality</i>	selects 4 pin functionality Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the EUSCI.B.SPI module.
<i>transmitData</i>	data to be transmitted from the SPI module

Returns

None

17.3 Programming Example

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

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

//Enable SPI Module
EUSCI_B_SPI_enable(EUSCI_B0_BASE);

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

18 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)

Introduction	153
API Functions	155
Programming Example	176

18.1 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.

18.2 Master Operations

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

- **EUSCI_B_I2C.initMaster**
- **EUSCI_B_I2C.setSlaveAddress**
- **EUSCI_B_I2C.setMode**
- **EUSCI_B_I2C.enable**
- **EUSCI_B_I2C.enableInterrupt** (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first initialize the I2C module and configure it as a master with a call to [EUSCI_B_I2C.initMaster\(\)](#). That function will set the clock and data rates. This is followed by a call to set the slave address with which the master intends to communicate with using [EUSCI_B_I2C.setSlaveAddress](#). Then the mode of operation (transmit or receive) is chosen using [EUSCI_B_I2C.setMode](#). The I2C module may now be enabled using [EUSCI_B_I2C.enable](#). It is recommended to enable the [EUSCI_B_I2C](#) module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

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

Master Single Byte Transmission

- [EUSCI_B_I2C.masterSendSingleByte\(\)](#)

Master Multiple Byte Transmission

- [EUSCI_B_I2C.masterSendMultiByteStart\(\)](#)
- [EUSCI_B_I2C.masterSendMultiByteNext\(\)](#)
- [EUSCI_B_I2C.masterSendMultiByteStop\(\)](#)

Master Single Byte Reception

- [EUSCI_B_I2C.masterReceiveSingleByte\(\)](#)

Master Multiple Byte Reception

- [EUSCI_B_I2C.masterMultiByteReceiveStart\(\)](#)
- [EUSCI_B_I2C.masterReceiveMultiByteNext\(\)](#)
- [EUSCI_B_I2C.masterReceiveMultiByteFinish\(\)](#)
- [EUSCI_B_I2C.masterReceiveMultiByteStop\(\)](#)

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

18.3 Slave Operations

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

- [EUSCI_B_I2C.initSlave\(\)](#)
- [EUSCI_B_I2C.setMode\(\)](#)
- [EUSCI_B_I2C.enable\(\)](#)
- [EUSCI_B_I2C.enableInterrupt\(\)](#) (if interrupts are being used) This may be followed by the APIs for transmit or receive as required

The user must first call the [EUSCI_B_I2C.initSlave](#) to initialize the slave module in I2C mode and set the slave address. This is followed by a call to set the mode of operation (transmit or receive).The I2C module may now be enabled using [EUSCI_B_I2C.enable](#). It is recommended to enable the I2C module before enabling the interrupts. Any transmission or reception of data may be initiated at this point after interrupts are enabled (if any).

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

Slave Transmission API

- [EUSCI_B_I2C.slavePutData\(\)](#)

Slave Reception API

- [EUSCI_B_I2C_slaveGetData\(\)](#)

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

18.4 API Functions

Functions

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

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

18.4.1 Detailed Description

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

The I2C master and slave interrupts are handled by

- `EUSCI_B_I2C_enableInterrupt`
- `EUSCI_B_I2C_disableInterrupt`

- EUSCI_B_I2C_clearInterrupt
- EUSCI_B_I2C_getInterruptStatus

Status and initialization functions for the I2C modules are

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

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

- EUSCI_B_I2C_slavePutData
- EUSCI_B_I2C_slaveGetData

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

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

18.4.2 Function Documentation

```
void EUSCI_B_I2C_clearInterrupt ( uint16_t baseAddress, uint16_t mask )
```

Clears I2C interrupt sources.

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

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	<p>is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

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

Disables the I2C block.

This will disable operation of the I2C block.

Parameters

<i>baseAddress</i>	is the base address of the USCI I2C module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

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

Returns

None

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

Enables the I2C block.

This will enable operation of the I2C block.

Parameters

<i>baseAddress</i>	is the base address of the USCI I2C module.
--------------------	---

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

Returns

None

`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

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

uint8_t EUSCI_B_I2C_getMode (uint16_t *baseAddress*)

Gets the mode of the I2C device.

Current I2C transmit/receive mode.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

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

Returns

One of the following:

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

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

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

The address of the I2C RX Buffer

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

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

The address of the I2C TX Buffer

```
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

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

Returns

None

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

```
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

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

Returns

None

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

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

<i>baseAddress</i>	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

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

<i>baseAddress</i>	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

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

<i>baseAddress</i>	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

`uint8_t EUSCI_B_I2C_masterReceiveMultiByteFinish (uint16_t baseAddress)`

Finishes multi-byte reception at the Master end.

This function is used by the Master module to initiate completion of a multi-byte reception. This function receives the current byte and initiates the STOP from master to slave.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

Received byte at Master end.

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

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

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

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is a pointer to the location to store the received byte at master end
<i>timeout</i>	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

`uint8_t EUSCI_B_I2C_masterReceiveMultiByteNext (uint16_t baseAddress)`

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

Returns

Received byte at Master end.

`void EUSCI_B_I2C_masterReceiveMultiByteStop (uint16_t baseAddress)`

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

This function is used by the Master module to initiate STOP

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

`uint8_t EUSCI_B_I2C_masterReceiveSingle (uint16_t baseAddress)`

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

Returns

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

`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

<i>baseAddress</i>	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.

`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

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

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

Finishes multi-byte transmission from Master to Slave.

This function is used by the Master module to send the last byte and STOP. This function transmits the last data byte of a multi-byte transmission to the slave and then sends a stop.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

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

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

This function is used by the Master module to send the last byte and STOP. This function transmits the last data byte of a multi-byte transmission to the slave and then sends a stop.

Parameters

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

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

Continues multi-byte transmission from Master to Slave.

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

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

None

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

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is the next data byte to be transmitted
<i>timeout</i>	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.

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

Starts multi-byte transmission from Master to Slave.

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

```
bool EUSCI_B_I2C_masterSendMultiByteStartWithTimeout ( uint16_t baseAddress, uint8_t txData, uint32_t timeout )
```

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is the first data byte to be transmitted
<i>timeout</i>	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.

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

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

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

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>timeout</i>	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.

```
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

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

```
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

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is the data byte to be transmitted
<i>timeout</i>	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.

```
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

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the USCI I2C module.
<i>mode</i>	Mode for the EUSCI_B_I2C module Valid values are: <ul style="list-style-type: none"> ■ EUSCI_B_I2C_TRANSMIT_MODE [Default] ■ EUSCI_B_I2C_RECEIVE_MODE

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the USCI I2C module.
<i>slaveAddress</i>	7-bit slave address

Modified bits of **UCBxI2CSA** register.

Returns

None

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

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Slave module.
--------------------	--

Returns

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

```
void EUSCI_B_I2C_slavePutData ( uint16_t baseAddress, uint8_t transmitData )
```

Transmits a byte from the I2C Module.

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

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

None

18.5 Programming Example

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

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

EUSCI_B_I2C_enable(EUSCI_B0_BASE);

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

19 FlashCtl - Flash Memory Controller

Introduction	177
API Functions	177
Programming Example	183

19.1 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.

19.2 API Functions

Functions

- void [FlashCtl_eraseSegment](#) (uint8_t *flash_ptr)
Erase a single segment of the flash memory.
- void [FlashCtl_eraseBank](#) (uint8_t *flash_ptr)
Erase a single bank of the flash memory.
- void [FlashCtl_performMassErase](#) (uint8_t *flash_ptr)
Erase all flash memory.
- bool [FlashCtl_performEraseCheck](#) (uint8_t *flash_ptr, uint16_t numberOfBytes)
Erase check of the flash memory.
- void [FlashCtl_write8](#) (uint8_t *data_ptr, uint8_t *flash_ptr, uint16_t count)
Write data into the flash memory in byte format, pass by reference.
- void [FlashCtl_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.
- void [FlashCtl_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.
- void [FlashCtl_fillMemory32](#) (uint32_t value, uint32_t *flash_ptr, uint16_t count)
Write data into the flash memory in 32-bit word format, pass by value.
- uint8_t [FlashCtl_getStatus](#) (uint8_t mask)
Check FlashCtl status to see if it is currently busy erasing or programming.
- void [FlashCtl_lockInfoA](#) (void)
Locks the information flash memory segment A.

- void `FlashCtl_unlockInfoA` (void)
Unlocks the information flash memory segment A.

19.2.1 Detailed Description

`FlashCtl_eraseSegment` helps erase a single segment of the flash memory. A pointer to the flash segment being erased is passed on to this function.

`FlashCtl_performEraseCheck` 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.

`FlashCtl_write8` facilitates writing into the flash memory in byte format. `FlashCtl_write16` facilitates writing into the flash memory in word format. `FlashCtl_write32` facilitates writing into the flash memory in long format, pass by reference. `FlashCtl_fillMemory32` facilitates writing into the flash memory in long format, pass by value. `FlashCtl_getStatus` checks if the flash is currently busy erasing or programming. `FlashCtl_lockInfoA` locks segment A of information memory. `FlashCtl_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

- `FlashCtl_eraseSegment()`
- `FlashCtl_eraseBank()`

Flash writes are managed by

- `FlashCtl_write8()`
- `FlashCtl_write16()`
- `FlashCtl_write32()`
- `FlashCtl_fillMemory32()`

The status is given by

- `FlashCtl_getStatus()`
- `FlashCtl_performEraseCheck()`

The segment A of information memory lock/unlock operations are managed by

- `FlashCtl_lockInfoA()`
- `FlashCtl_unlockInfoA()`

19.2.2 Function Documentation

void FlashCtl_eraseBank (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.

Parameters

<i>flash_ptr</i>	is a pointer into the bank to be erased
------------------	---

Returns

None

void FlashCtl_eraseSegment (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.

Parameters

<i>flash_ptr</i>	is the pointer into the flash segment to be erased
------------------	--

Returns

None

void FlashCtl_fillMemory32 (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. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

<i>value</i>	value to fill memory with
<i>flash_ptr</i>	is the pointer into which to write the data
<i>count</i>	number of times to write the value

Returns

None

uint8_t FlashCtl_getStatus (uint8_t *mask*)

Check FlashCtl 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.

Parameters

<i>mask</i>	FLASHCTL status to read Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ FLASHCTL_READY_FOR_NEXT_WRITE ■ FLASHCTL_ACCESS_VIOLATION_INTERRUPT_FLAG ■ FLASHCTL_PASSWORD_WRITTEN_INCORRECTLY ■ FLASHCTL_BUSY
-------------	---

Returns

Logical OR of any of the following:

- **FlashCtl_READY_FOR_NEXT_WRITE**
- **FlashCtl_ACCESS_VIOLATION_INTERRUPT_FLAG**
- **FlashCtl_PASSWORD_WRITTEN_INCORRECTLY**
- **FlashCtl_BUSY**

indicating the status of the FlashCtl

```
void FlashCtl_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.

Returns

None

```
bool FlashCtl_performEraseCheck ( 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).

Parameters

<i>flash_ptr</i>	is the pointer to the starting location of the erase check
<i>numberOfBytes</i>	is the number of bytes to be checked

Returns

STATUS_SUCCESS or STATUS_FAIL

```
void FlashCtl_performMassErase ( 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.

Parameters

<i>flash_ptr</i>	is a pointer into the bank to be erased
------------------	---

Returns

None

```
void FlashCtl_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.

Returns

None

```
void FlashCtl_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. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

<i>data_ptr</i>	is the pointer to the data to be written
<i>flash_ptr</i>	is the pointer into which to write the data
<i>count</i>	number of times to write the value

Returns

None

```
void FlashCtl_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. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

<i>data_ptr</i>	is the pointer to the data to be written
<i>flash_ptr</i>	is the pointer into which to write the data
<i>count</i>	number of times to write the value

Returns

None

```
void FlashCtl_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. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

<i>data_ptr</i>	is the pointer to the data to be written
<i>flash_ptr</i>	is the pointer into which to write the data
<i>count</i>	number of times to write the value

Returns

None

19.3 Programming Example

The following example shows some flash operations using the APIs

```
do{
    FlashCtl_eraseSegment(FlashCtl_BASE,
        (unsigned char *)INFOD_START
    );
    status = FlashCtl_performEraseCheck(FlashCtl_BASE,
        (unsigned char *)INFOD_START,
        128
    );
}while(status == STATUS_FAIL);

//Flash write
FlashCtl_write32(FlashCtl_BASE,
    calibration_data,
    (unsigned long *) (INFOD_START), 1);
```


20 GPIO

Introduction	184
API Functions	185
Programming Example	211

20.1 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()`.

20.2 API Functions

Functions

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

20.2.1 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_clearInterrupt()`
- `GPIO_getInterruptStatus()`
- `GPIO_selectInterruptEdge()`

The GPIO pin state is accessed with

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

20.2.2 Function Documentation

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

This function clears the interrupt flag on the selected pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

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

This function disables the port interrupt on the selected pin.

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

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	<p>is the specified pin in the selected port. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

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

This function enables the port interrupt on the selected pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

`uint8_t GPIO_getInputPinValue (uint8_t selectedPort, uint16_t selectedPins)`

This function gets the input value on the selected pin.

This function gets the input value on the selected pin.

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	is the specified pin in the selected port. Valid values are: <ul style="list-style-type: none">■ 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

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

This function gets the interrupt status of the selected pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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]

```
void GPIO_selectInterruptEdge ( uint8_t selectedPort, uint16_t selectedPins, uint8_t  
edgeSelect )
```

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

This function selects on what edge the port interrupt flag should be set for a transition. Values for *edgeSelect* should be `GPIO_LOW_TO_HIGH_TRANSITION` or `GPIO_HIGH_TO_LOW_TRANSITION`. Please refer to family user's guide for available ports with interrupt capability.

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

<i>edgeSelect</i>	specifies what transition sets the interrupt flag Valid values are: <ul style="list-style-type: none"> ■ GPIO_HIGH_TO_LOW_TRANSITION ■ GPIO_LOW_TO_HIGH_TRANSITION
-------------------	--

Modified bits of **PxIES** register.

Returns

None

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

This function configures the selected Pin as input pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none"> ■ 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
---------------------	--

<i>selectedPins</i>	<p>is the specified pin in the selected port. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

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

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

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

`void GPIO_setAsInputPinWithPullUpResistor (uint8_t selectedPort, uint16_t selectedPins)`

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

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

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	<p>is the specified pin in the selected port. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

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

This function configures the selected Pin as output pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

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

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

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

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	<p>is the specified pin in the selected port. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

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

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

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

```
void GPIO_setDriveStrength ( uint8_t selectedPort, uint16_t selectedPins, uint8_t
    driveStrength )
```

This function sets the drive strength for the selected port pin.

This 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`.

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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
---------------------	--

<i>driveStrength</i>	specifies the drive strength of the pin Valid values are: <ul style="list-style-type: none"> ■ GPIO_REDUCED_OUTPUT_DRIVE_STRENGTH ■ GPIO_FULL_OUTPUT_DRIVE_STRENGTH
----------------------	---

Modified bits of **PxDS** register.

Returns

None

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

This function sets output HIGH on the selected Pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none"> ■ 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
---------------------	--

<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

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

This function sets output LOW on the selected Pin.

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

Parameters

<i>selectedPort</i>	is the selected port. Valid values are: <ul style="list-style-type: none">■ 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
<i>selectedPins</i>	is the specified pin in the selected port. Mask value is the logical OR of any of the following: <ul style="list-style-type: none">■ 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

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

This function toggles the output on the selected Pin.

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

Parameters

<i>selectedPort</i>	<p>is the selected port. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
---------------------	---

<i>selectedPins</i>	<p>is the specified pin in the selected port. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

20.3 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
    );
}
```

21 LDO-PWR

Introduction	213
API Functions	213
Programming Example	225

21.1 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.

21.2 API Functions

Functions

- void [LDOPWR_unlockConfiguration](#) (uint16_t baseAddress)
Unlocks the configuration registers and enables write access.
- void [LDOPWR_lockConfiguration](#) (uint16_t baseAddress)
Locks the configuration registers and disables write access.
- void [LDOPWR_enablePort_U_inputs](#) (uint16_t baseAddress)
Enables Port U inputs.
- void [LDOPWR_disablePort_U_inputs](#) (uint16_t baseAddress)
Disables Port U inputs.
- void [LDOPWR_enablePort_U_outputs](#) (uint16_t baseAddress)
Enables Port U outputs.
- void [LDOPWR_disablePort_U_outputs](#) (uint16_t baseAddress)
Disables Port U outputs.
- uint8_t [LDOPWR_getPort_U1_inputData](#) (uint16_t baseAddress)
Returns PU.1 input data.
- uint8_t [LDOPWR_getPort_U0_inputData](#) (uint16_t baseAddress)
Returns PU.0 input data.
- uint8_t [LDOPWR_getPort_U1_outputData](#) (uint16_t baseAddress)
Returns PU.1 output data.
- uint8_t [LDOPWR_getPort_U0_outputData](#) (uint16_t baseAddress)
Returns PU.0 output data.
- void [LDOPWR_setPort_U1_outputData](#) (uint16_t baseAddress, uint8_t value)
Sets PU.1 output data.
- void [LDOPWR_setPort_U0_outputData](#) (uint16_t baseAddress, uint8_t value)

- Sets PU.0 output data.*
- void `LDOPWR_togglePort_U1_outputData` (uint16_t baseAddress)
- Toggles PU.1 output data.*
- void `LDOPWR_togglePort_U0_outputData` (uint16_t baseAddress)
- Toggles PU.0 output data.*
- void `LDOPWR_enableInterrupt` (uint16_t baseAddress, uint16_t mask)
- Enables LDO-PWR module interrupts.*
- void `LDOPWR_disableInterrupt` (uint16_t baseAddress, uint16_t mask)
- Disables LDO-PWR module interrupts.*
- void `LDOPWR_enable` (uint16_t baseAddress)
- Enables LDO-PWR module.*
- void `LDOPWR_disable` (uint16_t baseAddress)
- Disables LDO-PWR module.*
- uint8_t `LDOPWR_getInterruptStatus` (uint16_t baseAddress, uint16_t mask)
- Returns the interrupt status of LDO-PWR module interrupts.*
- void `LDOPWR_clearInterrupt` (uint16_t baseAddress, uint16_t mask)
- Clears the interrupt status of LDO-PWR module interrupts.*
- uint8_t `LDOPWR_isLDOInputValid` (uint16_t baseAddress)
- Returns if the the LDOI is valid and within bounds.*
- void `LDOPWR_enableOverloadAutoOff` (uint16_t baseAddress)
- Enables the LDO overload auto-off.*
- void `LDOPWR_disableOverloadAutoOff` (uint16_t baseAddress)
- Disables the LDO overload auto-off.*
- uint8_t `LDOPWR_getOverloadAutoOffStatus` (uint16_t baseAddress)
- Returns if the LDOI overload auto-off is enabled or disabled.*

21.2.1 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_clearInterrupt()
- LDOPWR_isLDOInputValid()
- LDOPWR_getOverloadAutoOffStatus()

21.2.2 Function Documentation

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

Clears the interrupt status of LDO-PWR module interrupts.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>mask</i>	mask of interrupts to clear the status of Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ LDOPWR_LDO1_VOLTAGE_GOING_OFF_INTERRUPT ■ LDOPWR_LDO1_VOLTAGE_COMING_ON_INTERRUPT ■ LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT

Modified bits of **LDOPWRCTL** register.

Returns

None

```
void LDOPWR_disable ( uint16_t baseAddress )
```

Disables LDO-PWR module.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOPWRCTL** register.

Returns

None


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

Disables LDO-PWR module interrupts.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>mask</i>	mask of interrupts to disable Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ LDOPWR_LDO1_VOLTAGE_GOING_OFF_INTERRUPT ■ LDOPWR_LDO1_VOLTAGE_COMING_ON_INTERRUPT ■ LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT

Modified bits of **LDOPWRCTL** register.

Returns

None

```
void LDOPWR_disableOverloadAutoOff ( uint16_t baseAddress )
```

Disables the LDO overload auto-off.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOPWRCTL** register.

Returns

None

```
void LDOPWR_disablePort_U_inputs ( uint16_t baseAddress )
```

Disables Port U inputs.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

```
void LDOPWR_disablePort_U_outputs ( uint16_t baseAddress )
```

Disables Port U inputs.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

void LDOPWR_enable (uint16_t *baseAddress*)

Enables LDO-PWR module.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOPWRCTL** register.

Returns

None

void LDOPWR_enableInterrupt (uint16_t *baseAddress*, uint16_t *mask*)

Enables LDO-PWR module interrupts.

Does not clear interrupt flags.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>mask</i>	mask of interrupts to enable Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ LDOPWR_LDOI_VOLTAGE_GOING_OFF_INTERRUPT ■ LDOPWR_LDOI_VOLTAGE_COMING_ON_INTERRUPT ■ LDOPWR_LDO_OVERLOAD_INDICATION_INTERRUPT

Modified bits of **LDOPWRCTL** register.

Returns

None

void LDOPWR_enableOverloadAutoOff (uint16_t *baseAddress*)

Enables the LDO overload auto-off.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOPWRCTL** register.

Returns

None

void LDOPWR_enablePort_U_inputs (uint16_t *baseAddress*)

Enables Port U inputs.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

void LDOPWR_enablePort_U_outputs (uint16_t *baseAddress*)

Enables Port U outputs.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

uint8_t LDOPWR_getInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Returns the interrupt status of LDO-PWR module interrupts.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>mask</i>	mask of interrupts to get the status of Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ LDOPWR_LDO1_VOLTAGE_GOING_OFF_INTERRUPT ■ LDOPWR_LDO1_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

uint8_t LDOPWR_getOverloadAutoOffStatus (uint16_t *baseAddress*)

Returns if the LDOI overload auto-off is enabled or disabled.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_AUTOOFF_ENABLED**
- **LDOPWR_AUTOOFF_DISABLED**

uint8_t LDOPWR_getPort_U0_inputData (uint16_t *baseAddress*)

Returns PU.0 input data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

uint8_t LDOPWR_getPort_U0_outputData (uint16_t *baseAddress*)

Returns PU.0 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

`uint8_t LDOPWR_getPort_U1_inputData (uint16_t baseAddress)`

Returns PU.1 input data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

`uint8_t LDOPWR_getPort_U1_outputData (uint16_t baseAddress)`

Returns PU.1 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_PORTU_PIN_HIGH**
- **LDOPWR_PORTU_PIN_LOW**

`uint8_t LDOPWR_isLDOInputValid (uint16_t baseAddress)`

Returns if the the LDOI is valid and within bounds.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Returns

One of the following:

- **LDOPWR_LDO_INPUT_VALID**
- **LDOPWR_LDO_INPUT_INVALID**

`void LDOPWR_lockConfiguration (uint16_t baseAddress)`

Locks the configuration registers and disables write access.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOKEYPID** register.

Returns

None

```
void LDOPWR_setPort_U0_outputData ( uint16_t baseAddress, uint8_t value )
```

Sets PU.0 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>value</i>	Valid values are: <ul style="list-style-type: none"> ■ LDOPWR_PORTU_PIN_HIGH ■ LDOPWR_PORTU_PIN_LOW

Modified bits of **PUCTL** register.

Returns

None

```
void LDOPWR_setPort_U1_outputData ( uint16_t baseAddress, uint8_t value )
```

Sets PU.1 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
<i>value</i>	Valid values are: <ul style="list-style-type: none"> ■ LDOPWR_PORTU_PIN_HIGH ■ LDOPWR_PORTU_PIN_LOW

Modified bits of **PUCTL** register.

Returns

None

```
void LDOPWR_togglePort_U0_outputData ( uint16_t baseAddress )
```

Toggles PU.0 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

```
void LDOPWR_togglePort_U1_outputData ( uint16_t baseAddress )
```

Toggles PU.1 output data.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **PUCTL** register.

Returns

None

```
void LDOPWR_unlockConfiguration ( uint16_t baseAddress )
```

Unlocks the configuration registers and enables write access.

Parameters

<i>baseAddress</i>	is the base address of the LDOPWR module.
--------------------	---

Modified bits of **LDOKEYPID** register.

Returns

None

21.3 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.clearInterrupt(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);
    }
}
```

22 32-Bit Hardware Multiplier (MPY32)

Introduction	227
API Functions	227
Programming Example	235

22.1 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.

22.2 API Functions

Functions

- void [MPY32_setWriteDelay](#) (uint16_t writeDelaySelect)
Sets the write delay setting for the MPY32 module.
- void [MPY32_enableSaturationMode](#) (void)
Enables Saturation Mode.
- void [MPY32_disableSaturationMode](#) (void)
Disables Saturation Mode.
- uint8_t [MPY32_getSaturationMode](#) (void)
Gets the Saturation Mode.
- void [MPY32_enableFractionalMode](#) (void)
Enables Fraction Mode.
- void [MPY32_disableFractionalMode](#) (void)
Disables Fraction Mode.
- uint8_t [MPY32_getFractionalMode](#) (void)
Gets the Fractional Mode.
- void [MPY32_setOperandOne8Bit](#) (uint8_t multiplicationType, uint8_t operand)
Sets an 8-bit value into operand 1.
- void [MPY32_setOperandOne16Bit](#) (uint8_t multiplicationType, uint16_t operand)
Sets an 16-bit value into operand 1.
- void [MPY32_setOperandOne24Bit](#) (uint8_t multiplicationType, uint32_t operand)
Sets an 24-bit value into operand 1.
- void [MPY32_setOperandOne32Bit](#) (uint8_t multiplicationType, uint32_t operand)
Sets an 32-bit value into operand 1.
- void [MPY32_setOperandTwo8Bit](#) (uint8_t operand)
Sets an 8-bit value into operand 2, which starts the multiplication.
- void [MPY32_setOperandTwo16Bit](#) (uint16_t operand)
Sets an 16-bit value into operand 2, which starts the multiplication.
- void [MPY32_setOperandTwo24Bit](#) (uint32_t operand)
Sets an 24-bit value into operand 2, which starts the multiplication.
- void [MPY32_setOperandTwo32Bit](#) (uint32_t operand)

- Sets an 32-bit value into operand 2, which starts the multiplication.*
- `uint64_t MPY32_getResult` (void)
 - Returns an 64-bit result of the last multiplication operation.*
- `uint16_t MPY32_getSumExtension` (void)
 - Returns the Sum Extension of the last multiplication operation.*
- `uint16_t MPY32_getCarryBitValue` (void)
 - Returns the Carry Bit of the last multiplication operation.*
- `void MPY32_clearCarryBitValue` (void)
 - Clears the Carry Bit of the last multiplication operation.*
- `void MPY32_preloadResult` (`uint64_t` result)
 - Preloads the result register.*

22.2.1 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()`

22.2.2 Function Documentation

`void MPY32_clearCarryBitValue (void)`

Clears the Carry Bit of the last multiplication operation.

This function clears the Carry Bit of the MPY module

Returns

The value of the MPY32 module Carry Bit 0x0 or 0x1.

`void MPY32_disableFractionalMode (void)`

Disables Fraction Mode.

This function disables fraction mode.

Returns

None

`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.

Returns

None

`void MPY32_enableFractionalMode (void)`

Enables Fraction Mode.

This function enables fraction mode.

Returns

None

`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.

Returns

None

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.

Returns

The value of the MPY32 module Carry Bit 0x0 or 0x1.

uint8_t MPY32_getFractionalMode (void)

Gets the Fractional Mode.

This function gets the current fractional mode.

Returns

Gets the fractional mode Return one of the following:

- **MPY32_FRACTIONAL_MODE_DISABLED**
- **MPY32_FRACTIONAL_MODE_ENABLED**

Gets the Fractional Mode

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

Returns

The 64-bit result is returned as a uint64_t type

uint8_t MPY32_getSaturationMode (void)

Gets the Saturation Mode.

This function gets the current saturation mode.

Returns

Gets the Saturation Mode Return one of the following:

- **MPY32_SATURATION_MODE_DISABLED**
- **MPY32_SATURATION_MODE_ENABLED**

Gets the Saturation Mode

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.

Returns

The value of the MPY32 module Sum Extension.

void MPY32_preloadResult (uint64_t *result*)

Preloads the result register.

This function Preloads the result register

Parameters

<i>result</i>	value to preload the result register to
---------------	---

Returns

None

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.

Parameters

<i>multiplicationType</i>	<p>is the type of multiplication to perform once the second operand is set. Valid values are:</p> <ul style="list-style-type: none"> ■ MPY32_MULTIPLY_UNSIGNED ■ MPY32_MULTIPLY_SIGNED ■ MPY32_MULTIPLYACCUMULATE_UNSIGNED ■ MPY32_MULTIPLYACCUMULATE_SIGNED
---------------------------	--

<i>operand</i>	is the 16-bit value to load into the 1st operand.
----------------	---

Returns

None

```
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.

Parameters

<i>multiplicationType</i>	is the type of multiplication to perform once the second operand is set. Valid values are: <ul style="list-style-type: none"> ■ MPY32_MULTIPLY_UNSIGNED ■ MPY32_MULTIPLY_SIGNED ■ MPY32_MULTIPLYACCUMULATE_UNSIGNED ■ MPY32_MULTIPLYACCUMULATE_SIGNED
<i>operand</i>	is the 24-bit value to load into the 1st operand.

Returns

None

```
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.

Parameters

<i>multiplicationType</i>	is the type of multiplication to perform once the second operand is set. Valid values are: <ul style="list-style-type: none"> ■ MPY32_MULTIPLY_UNSIGNED ■ MPY32_MULTIPLY_SIGNED ■ MPY32_MULTIPLYACCUMULATE_UNSIGNED ■ MPY32_MULTIPLYACCUMULATE_SIGNED
---------------------------	---

<i>operand</i>	is the 32-bit value to load into the 1st operand.
----------------	---

Returns

None

```
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.

Parameters

<i>multiplicationType</i>	is the type of multiplication to perform once the second operand is set. Valid values are: <ul style="list-style-type: none"> ■ MPY32_MULTIPLY_UNSIGNED ■ MPY32_MULTIPLY_SIGNED ■ MPY32_MULTIPLYACCUMULATE_UNSIGNED ■ MPY32_MULTIPLYACCUMULATE_SIGNED
<i>operand</i>	is the 8-bit value to load into the 1st operand.

Returns

None

```
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.

Parameters

<i>operand</i>	is the 16-bit value to load into the 2nd operand.
----------------	---

Returns

None

```
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.

Parameters

<i>operand</i>	is the 24-bit value to load into the 2nd operand.
----------------	---

Returns

None

```
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.

Parameters

<i>operand</i>	is the 32-bit value to load into the 2nd operand.
----------------	---

Returns

None

```
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.

Parameters

<i>operand</i>	is the 8-bit value to load into the 2nd operand.
----------------	--

Returns

None

```
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 unpredictable results if registers are changed before the results are ready.

23 Port Mapping Controller

Introduction	236
API Functions	236
Programming Example	237

23.1 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.

23.2 API Functions

Functions

- void `PMAP_initPorts` (uint16_t baseAddress, `PMAP_initPortsParam` *param)
This function configures the MSP430 Port Mapper.

23.2.1 Detailed Description

The MSP430ware API that configures Port Mapping is `PMAP_initPorts()`

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

23.2.2 Function Documentation

```
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.

Returns

None

References PMAP_initPortsParam::numberOfPorts, PMAP_initPortsParam::portMapping, PMAP_initPortsParam::portMapReconfigure, and PMAP_initPortsParam::PxMAPy.

23.3 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_initPorts(P4MAP_BASE,
    (const unsigned char *)port_mapping,
    (unsigned char *)&P4MAP01,
    1,
    PMAP_DISABLE_RECONFIGURATION
);
```

24 Power Management Module (PMM)

Introduction	238
API Functions	240
Programming Example	251

24.1 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:

1. Decrement SvmL and SVSL levels.
2. 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 μ s, not 5 μ 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 μ 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_disableSVLSvmL()` 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.

24.2 API Functions

Functions

- void `PMM_enableSvsL` (void)
Enables the low-side SVS circuitry.
- void `PMM_disableSvsL` (void)
Disables the low-side SVS circuitry.
- void `PMM_enableSvmL` (void)
Enables the low-side SVM circuitry.
- void `PMM_disableSvmL` (void)
Disables the low-side SVM circuitry.
- void `PMM_enableSvsH` (void)
Enables the high-side SVS circuitry.
- void `PMM_disableSvsH` (void)
Disables the high-side SVS circuitry.
- void `PMM_enableSvmH` (void)
Enables the high-side SVM circuitry.
- void `PMM_disableSvmH` (void)
Disables the high-side SVM circuitry.
- void `PMM_enableSvsLSvmL` (void)
Enables the low-side SVS and SVM circuitry.
- void `PMM_disableSvsLSvmL` (void)
Disables the low-side SVS and SVM circuitry.
- void `PMM_enableSvsHSvmH` (void)
Enables the high-side SVS and SVM circuitry.
- void `PMM_disableSvsHSvmH` (void)
Disables the high-side SVS and SVM circuitry.
- void `PMM_enableSvsLReset` (void)
Enables the POR signal generation when a low-voltage event is registered by the low-side SVS.
- void `PMM_disableSvsLReset` (void)
Disables the POR signal generation when a low-voltage event is registered by the low-side SVS.
- void `PMM_enableSvmLInterrupt` (void)
Enables the interrupt generation when a low-voltage event is registered by the low-side SVM.
- void `PMM_disableSvmLInterrupt` (void)
Disables the interrupt generation when a low-voltage event is registered by the low-side SVM.
- void `PMM_enableSvsHReset` (void)
Enables the POR signal generation when a low-voltage event is registered by the high-side SVS.
- void `PMM_disableSvsHReset` (void)
Disables the POR signal generation when a low-voltage event is registered by the high-side SVS.
- void `PMM_enableSvmHInterrupt` (void)
Enables the interrupt generation when a low-voltage event is registered by the high-side SVM.
- void `PMM_disableSvmHInterrupt` (void)
Disables the interrupt generation when a low-voltage event is registered by the high-side SVM.
- void `PMM_clearPMMIFGS` (void)
Clear all interrupt flags for the PMM.
- void `PMM_enableSvsLInLPMFastWake` (void)
Enables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4.
- void `PMM_enableSvsLInLPMSlowWake` (void)
Enables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4.
- void `PMM_disableSvsLInLPMFastWake` (void)
Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4.

- void `PMM_disableSvsLInLPMSlowWake` (void)
Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4.
- void `PMM_enableSvsHInLPMNormPerf` (void)
Enables supervisor high side in LPM with $tpd = 20 \text{ ?s}(1)$
- void `PMM_enableSvsHInLPMFullPerf` (void)
Enables supervisor high side in LPM with $tpd = 2.5 \text{ ?s}(1)$
- void `PMM_disableSvsHInLPMNormPerf` (void)
Disables supervisor high side in LPM with $tpd = 20 \text{ ?s}(1)$
- void `PMM_disableSvsHInLPMFullPerf` (void)
Disables supervisor high side in LPM with $tpd = 2.5 \text{ ?s}(1)$
- void `PMM_optimizeSvsLInLPMFastWake` (void)
Optimized to provide twake-up-fast from LPM2, LPM3, and LPM4 with least power.
- void `PMM_optimizeSvsHInLPMFullPerf` (void)
Optimized to provide $tpd = 2.5 \text{ ?s}(1)$ in LPM with least power.
- `uint16_t` `PMM_setVCoreUp` (`uint8_t` level)
Increase Vcore by one level.
- `uint16_t` `PMM_setVCoreDown` (`uint8_t` level)
Decrease Vcore by one level.
- `bool` `PMM_setVCore` (`uint8_t` level)
Set Vcore to expected level.
- `uint16_t` `PMM_getInterruptStatus` (`uint16_t` mask)
Returns interrupt status.

24.2.1 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_enableSvsLInLPMFastWake()` Enables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

`PMM_enableSvsLInLPMSlowWake()` Enables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_disableSvsLInLPMFastWake() Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4

PMM_disableSvsLInLPMSlowWake() Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4

PMM_enableSvsHInLPMNormPerf() Enables supervisor high side in LPM with tpd = 20 ?s(1)

PMM_enableSvsHInLPMFullPerf() Enables supervisor high side in LPM with tpd = 2.5 ?s(1)

PMM_disableSvsHInLPMNormPerf() Disables supervisor high side in LPM with tpd = 20 ?s(1)

PMM_disableSvsHInLPMFullPerf() Disables supervisor high side in LPM with tpd = 2.5 ?s(1)

PMM_optimizeSvsLInLPMFastWake() Optimized to provide twake-up-fast from LPM2, LPM3, and LPM4 with least power

PMM_optimizeSvsHInLPMFullPerf() 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.

24.2.2 Function Documentation

void PMM_clearPMMIFGS (void)

Clear all interrupt flags for the PMM.

Modified bits of **PMMCTL0** register and bits of **PMMIFG** register.

Returns

None

void PMM_disableSvmH (void)

Disables the high-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns

None

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

void PMM_disableSvmL (void)

Disables the low-side SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.**Returns**

None

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

void PMM_disableSvsH (void)

Disables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.**Returns**

None

void PMM_disableSvsHInLPMFullPerf (void)Disables supervisor high side in LPM with $t_{pd} = 2.5 \mu s(1)$ Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.**Returns**

None

void PMM_disableSvsHInLPMNormPerf (void)Disables supervisor high side in LPM with $t_{pd} = 20 \mu s(1)$ Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns

None

void PMM_disableSvsHReset (void)

Disables the POR signal generation when a low-voltage event is registered by the high-side SVS.
Modified bits of **PMMCTLO** register and bits of **PMMIE** register.

Returns

None

void PMM_disableSvsHSvmH (void)

Disables the high-side SVS and SVM circuitry.
Modified bits of **PMMCTLO** register and bits of **SVSMHCTL** register.

Returns

None

void PMM_disableSvsL (void)

Disables the low-side SVS circuitry.
Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

void PMM_disableSvsLInLPMFastWake (void)

Disables supervisor low side in LPM with twake-up-fast from LPM2, LPM3, and LPM4.
Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

void PMM_disableSvsLInLPMSlowWake (void)

Disables supervisor low side in LPM with twake-up-slow from LPM2, LPM3, and LPM4.
Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

void PMM_disableSvsLReset (void)

Disables the POR signal generation when a low-voltage event is registered by the low-side SVS.
Modified bits of **PMMCTLO** register and bits of **PMMIE** register.

Returns

None

void PMM_disableSvsLSvmL (void)

Disables the low-side SVS and SVM circuitry.
Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

void PMM_enableSvmH (void)

Enables the high-side SVM circuitry.
Modified bits of **PMMCTLO** register and bits of **SVSMHCTL** register.

Returns

None

void PMM_enableSvmHInterrupt (void)

Enables the interrupt generation when a low-voltage event is registered by the high-side SVM.
Modified bits of **PMMCTLO** register and bits of **PMMIE** register.

Returns

None

void PMM_enableSvmL (void)

Enables the low-side SVM circuitry.
Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

```
void PMM_enableSvmLInterrupt ( void )
```

Enables the interrupt generation when a low-voltage event is registered by the low-side SVM.

Modified bits of **PMMCTLO** register and bits of **PMMIE** register.

Returns

None

```
void PMM_enableSvsH ( void )
```

Enables the high-side SVS circuitry.

Modified bits of **PMMCTLO** register and bits of **SVSMHCTL** register.

Returns

None

```
void PMM_enableSvsHInLPMFullPerf ( void )
```

Enables supervisor high side in LPM with $t_{pd} = 2.5 \mu s(1)$

Modified bits of **PMMCTLO** register and bits of **SVSMHCTL** register.

Returns

None

```
void PMM_enableSvsHInLPMNormPerf ( void )
```

Enables supervisor high side in LPM with $t_{pd} = 20 \mu s(1)$

Modified bits of **PMMCTLO** register and bits of **SVSMHCTL** register.

Returns

None

```
void PMM_enableSvsHReset ( void )
```

Enables the POR signal generation when a low-voltage event is registered by the high-side SVS.

Modified bits of **PMMCTLO** register and bits of **PMMIE** register.

Returns

None

```
void PMM_enableSvsHSvmH ( void )
```

Enables the high-side SVS and SVM circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMHCTL** register.

Returns

None

```
void PMM_enableSvsL ( void )
```

Enables the low-side SVS circuitry.

Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns

None

```
void PMM_enableSvsLInLPMFastWake ( 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

```
void PMM_enableSvsLInLPMSlowWake ( 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

```
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

```
void PMM_enableSvsLSvmL ( void )
```

Enables the low-side SVS and SVM circuitry.

Modified bits of **PMMCTLO** register and bits of **SVSMLCTL** register.

Returns

None

```
uint16_t PMM_getInterruptStatus ( uint16_t mask )
```

Returns interrupt status.

Parameters

<i>mask</i>	<p>is the mask for specifying the required flag Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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_PMPORIFG**
- **PMM_SVSHIFG**
- **PMM_SVSLIFG**
- **PMM_PMMLPM5IFG**
indicating the status of the masked interrupts

void PMM_optimizeSvsHInLPMFullPerf (void)

Optimized to provide $t_{pd} = 2.5 \mu s(1)$ in LPM with least power.
Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns

None

void PMM_optimizeSvsLInLPMFastWake (void)

Optimized to provide $t_{wake-up-fast}$ from LPM2, LPM3, and LPM4 with least power.
Modified bits of **PMMCTL0** register and bits of **SVSMLCTL** register.

Returns

None

bool PMM_setVCore (uint8_t level)

Set Vcore to expected level.

Parameters

<i>level</i>	level to which Vcore needs to be decreased/increased Valid values are: <ul style="list-style-type: none"> ■ 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

References PMM_setVCoreDown(), and PMM_setVCoreUp().

uint16_t PMM_setVCoreDown (uint8_t *level*)

Decrease Vcore by one level.

Parameters

<i>level</i>	level to which Vcore needs to be decreased Valid values are: <ul style="list-style-type: none"> ■ PMM_CORE_LEVEL_0 [Default] ■ PMM_CORE_LEVEL_1 ■ PMM_CORE_LEVEL_2 ■ PMM_CORE_LEVEL_3
--------------	---

Modified bits of **PMMCTLO** register, bits of **PMMIFG** register, bits of **PMMRIE** register, bits of **SVSMHCTL** register and bits of **SVSMLCTL** register.

Returns

STATUS_SUCCESS

Referenced by PMM_setVCore().

uint16_t PMM_setVCoreUp (uint8_t *level*)

Increase Vcore by one level.

Parameters

<i>level</i>	level to which Vcore needs to be increased Valid values are: <ul style="list-style-type: none"> ■ PMM_CORE_LEVEL_0 [Default] ■ PMM_CORE_LEVEL_1 ■ PMM_CORE_LEVEL_2 ■ PMM_CORE_LEVEL_3
--------------	---

Modified bits of **PMMCTLO** register, bits of **PMMIFG** register, bits of **PMMRIE** register, bits of **SVSMHCTL** register and bits of **SVSMLCTL** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

Referenced by PMM_setVCore().

24.3 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_CORE_LEVEL_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);
}
```

25 RAM Controller

Introduction	253
API Functions	253
Programming Example	255

25.1 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.

25.2 API Functions

Functions

- void [RAM.setSectorOff](#) (uint8_t sector)
Set specified RAM sector off.
- uint8_t [RAM.getSectorState](#) (uint8_t sector)
Get RAM sector ON/OFF status.

25.2.1 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

25.2.2 Function Documentation

uint8_t [RAM.getSectorState](#) (uint8_t *sector*)

Get RAM sector ON/OFF status.

Parameters

<i>sector</i>	is specified sector Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`void RAM_setSectorOff (uint8_t sector)`

Set specified RAM sector off.

Parameters

<i>sector</i>	is specified sector to be set off. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RAM_SECTOR0 ■ RAM_SECTOR1 ■ RAM_SECTOR2 ■ RAM_SECTOR3 ■ RAM_SECTOR4 ■ RAM_SECTOR5 ■ RAM_SECTOR6 ■ RAM_SECTOR7
---------------	---

Modified bits of **RCCTL0** register.

Returns

None

25.3 Programming Example

The following example shows some RAM Controller operations using the APIs

```

//Start timer
Timer_A.clearTimerInterrupt(TIMER_A0.BASE);

Timer_A.initUpModeParam param = {0};
param.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
param.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
param.timerPeriod = 25000;
param.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_DISABLE;
param.captureCompareInterruptEnable_CCR0_CCIE =
    TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE;
param.timerClear = TIMER_A_DO_CLEAR;
param.startTimer = true;
Timer_A.initUpMode(TIMER_A0.BASE, &param);

//RAM controller sector off
RAM.setSectorOff(RAM.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);
}

```


26 Internal Reference (REF)

Introduction	256
API Functions	256
Programming Example	261

26.1 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.

26.2 API Functions

Functions

- void [Ref_setReferenceVoltage](#) (uint16_t baseAddress, uint8_t referenceVoltageSelect)
Sets the reference voltage for the voltage generator.
- void [Ref_disableTempSensor](#) (uint16_t baseAddress)
Disables the internal temperature sensor to save power consumption.
- void [Ref_enableTempSensor](#) (uint16_t baseAddress)
Enables the internal temperature sensor.
- void [Ref_enableReferenceVoltageOutput](#) (uint16_t baseAddress)
Outputs the reference voltage to an output pin.
- void [Ref_disableReferenceVoltageOutput](#) (uint16_t baseAddress)
Disables the reference voltage as an output to a pin.
- void [Ref_enableReferenceVoltage](#) (uint16_t baseAddress)
Enables the reference voltage to be used by peripherals.
- void [Ref_disableReferenceVoltage](#) (uint16_t baseAddress)
Disables the reference voltage.
- uint16_t [Ref_getBandgapMode](#) (uint16_t baseAddress)
Returns the bandgap mode of the Ref module.
- bool [Ref_isBandgapActive](#) (uint16_t baseAddress)
Returns the active status of the bandgap in the Ref module.
- uint16_t [Ref_isRefGenBusy](#) (uint16_t baseAddress)
Returns the busy status of the reference generator in the Ref module.
- bool [Ref_isRefGenActive](#) (uint16_t baseAddress)
Returns the active status of the reference generator in the Ref module.

26.2.1 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\(\)](#)

26.2.2 Function Documentation

`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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFON** of **REFCTL0** register.

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFOUT** of **REFCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFTCOFF** of **REFCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFON** of **REFCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFOUT** of **REFCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Modified bits are **REFTCOFF** of **REFCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	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 bandgap mode of the module

`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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Returns

One of the following:

- **Ref.ACTIVE** if active
- **Ref.INACTIVE** if not active
indicating the bandgap active status of the module

`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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
--------------------	--

Returns

One of the following:

- **Ref.ACTIVE** if active
- **Ref.INACTIVE** if not active
indicating the reference generator active status of the module

`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.

Parameters

<i>baseAddress</i>	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 changes to be made to the Ref module controls indicating the reference generator busy status of the module

```
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.

Parameters

<i>baseAddress</i>	is the base address of the REF module.
<i>referenceVoltageSelect</i>	is the desired voltage to generate for a reference voltage. Valid values are: <ul style="list-style-type: none"> ■ REF_VREF1_5V [Default] ■ REF_VREF2_0V ■ REF_VREF2_5V Modified bits are REFVSEL of REFCTL0 register.

Returns

None

26.3 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, ADC12_IFG0));

    __no_operation(); // SET BREAKPOINT HERE
}
}
```

27 Real-Time Clock (RTC_A)

Introduction	263
API Functions	263
Programming Example	277

27.1 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.

27.2 API Functions

Functions

- void [RTC_A_startClock](#) (uint16_t baseAddress)
Starts the RTC.
- void [RTC_A_holdClock](#) (uint16_t baseAddress)
Holds the RTC.
- void [RTC_A_setCalibrationFrequency](#) (uint16_t baseAddress, uint16_t frequencySelect)
Allows and Sets the frequency output to RTCCLK pin for calibration measurement.
- void [RTC_A_setCalibrationData](#) (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
Sets the specified calibration for the RTC.
- void [RTC_A_initCounter](#) (uint16_t baseAddress, uint16_t clockSelect, uint16_t counterSizeSelect)
Initializes the settings to operate the RTC in Counter mode.
- void [RTC_A_initCalendar](#) (uint16_t baseAddress, [Calendar](#) *CalendarTime, uint16_t formatSelect)
Initializes the settings to operate the RTC in calendar mode.
- [Calendar](#) [RTC_A_getCalendarTime](#) (uint16_t baseAddress)
Returns the Calendar Time stored in the Calendar registers of the RTC.
- void [RTC_A_configureCalendarAlarm](#) (uint16_t baseAddress, [RTC_A_configureCalendarAlarmParam](#) *param)
Sets and Enables the desired Calendar Alarm settings.
- void [RTC_A_setCalendarEvent](#) (uint16_t baseAddress, uint16_t eventSelect)
Sets a single specified Calendar interrupt condition.
- uint32_t [RTC_A_getCounterValue](#) (uint16_t baseAddress)
Returns the value of the Counter register.

- void [RTC_A_setCounterValue](#) (uint16_t baseAddress, uint32_t counterValue)
Sets the value of the Counter register.
- void [RTC_A_initCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint16_t prescaleClockSelect, uint16_t prescaleDivider)
Initializes the Prescaler for Counter mode.
- void [RTC_A_holdCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect)
Holds the selected Prescaler.
- void [RTC_A_startCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect)
Starts the selected Prescaler.
- void [RTC_A_definePrescaleEvent](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
Sets up an interrupt condition for the selected Prescaler.
- uint8_t [RTC_A_getPrescaleValue](#) (uint16_t baseAddress, uint8_t prescaleSelect)
Returns the selected prescaler value.
- void [RTC_A_setPrescaleValue](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
Sets the selected prescaler value.
- void [RTC_A_enableInterrupt](#) (uint16_t baseAddress, uint8_t interruptMask)
Enables selected RTC interrupt sources.
- void [RTC_A_disableInterrupt](#) (uint16_t baseAddress, uint8_t interruptMask)
Disables selected RTC interrupt sources.
- uint8_t [RTC_A_getInterruptStatus](#) (uint16_t baseAddress, uint8_t interruptFlagMask)
Returns the status of the selected interrupts flags.
- void [RTC_A_clearInterrupt](#) (uint16_t baseAddress, uint8_t interruptFlagMask)
Clears selected RTC interrupt flags.

27.2.1 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.initCalender\(\)](#)
- [RTC_A.getCalenderTime\(\)](#)

The RTC_A counter mode is initialized and setup by

- [RTC_A.initCounter\(\)](#)
- [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\(\)](#)

27.2.2 Function Documentation

`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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>interruptFlagMask</i>	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>param</i>	is the pointer to struct for calendar alarm configuration.

Returns

None

References RTC_A_configureCalendarAlarmParam::dayOfMonthAlarm, RTC_A_configureCalendarAlarmParam::dayOfWeekAlarm, RTC_A_configureCalendarAlarmParam::hoursAlarm, and RTC_A_configureCalendarAlarmParam::minutesAlarm.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to define an interrupt for. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1
<i>prescaleEventDivider</i>	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: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>interruptMask</i>	is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_A_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>interruptMask</i>	is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_A_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
--------------------	--

Returns

A [Calendar](#) structure containing the current time.

References `Calendar::DayOfMonth`, `Calendar::DayOfWeek`, `Calendar::Hours`, `Calendar::Minutes`, `Calendar::Month`, `Calendar::Seconds`, and `Calendar::Year`.

`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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
--------------------	--

Returns

The raw value of the full 32-bit Counter Register.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>interruptFlagMask</i>	is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_A_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to obtain the value of. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1

Returns

The value of the specified prescaler count register

```
void RTC_A_holdClock ( uint16_t baseAddress )
```

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to hold. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1

Returns

None

```
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. To prevent potential erroneous alarm conditions from occurring, the alarm should be disabled by clearing the RTCAIE, RTCAIFG and AE bits with APIs: [RTC_A_disableInterrupt\(\)](#), [RTC_A_clearInterrupt\(\)](#) and [RTC_A_configureCalendarAlarm\(\)](#) before calendar initialization.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>CalendarTime</i>	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 0-23 DayOfWeek between 0-6 DayOfMonth between 1-31 Year between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.
<i>formatSelect</i>	is the format for the Calendar registers to use. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_FORMAT_BINARY [Default] ■ RTC_A_FORMAT_BCD Modified bits are RTCBCD of RTCCTL1 register.

Returns

None

References `Calendar::DayOfMonth`, `Calendar::DayOfWeek`, `Calendar::Hours`, `Calendar::Minutes`, `Calendar::Month`, `Calendar::Seconds`, and `Calendar::Year`.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>clockSelect</i>	is the selected clock for the counter mode to use. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_CLOCKSELECT_ACLK [Default] ■ RTC_A_CLOCKSELECT_SMCLK ■ RTC_A_CLOCKSELECT_RT1PS - use Prescaler 1 as source to RTC Modified bits are RTCSEL of RTCCTL1 register.

<i>counterSize</i> ↔ <i>Select</i>	is the size of the counter. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to initialize. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1
<i>prescaleClock</i> ↔ <i>Select</i>	is the clock to drive the selected prescaler. Valid values are: <ul style="list-style-type: none"> ■ 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.
<i>prescaleDivider</i>	is the divider for the selected clock source. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>eventSelect</i>	is the condition selected. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_CALENDAREVENT_MINUTECHANGE - assert interrupt on every minute ■ RTC_A_CALENDAREVENT_HOURLCHANGE - 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>offsetDirection</i>	is the direction that the calibration offset will go. Valid values are: <ul style="list-style-type: none"> ■ 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.

<i>offsetValue</i>	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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>frequencySelect</i>	is the frequency output to RTCCLK. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>counterValue</i>	is the value to set the Counter register to; a valid value may be any 32-bit integer.

Returns

None

```
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\(\)](#).

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to set the value for. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1
<i>prescaleCounterValue</i>	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

```
void RTC_A_startClock ( uint16_t baseAddress )
```

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_A module.
<i>prescaleSelect</i>	is the prescaler to start. Valid values are: <ul style="list-style-type: none"> ■ RTC_A_PRESCALE_0 ■ RTC_A_PRESCALE_1

Returns

None

27.3 Programming Example

The following example shows how to initialize and use the RTC API to setup Calendar 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();
```

28 Real-Time Clock (RTC_B)

Introduction	278
API Functions	278
Programming Example	288

28.1 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.

28.2 API Functions

Functions

- void [RTC_B_startClock](#) (uint16_t baseAddress)
Starts the RTC.
- void [RTC_B_holdClock](#) (uint16_t baseAddress)
Holds the RTC.
- void [RTC_B_setCalibrationFrequency](#) (uint16_t baseAddress, uint16_t frequencySelect)
Allows and Sets the frequency output to RTCCLK pin for calibration measurement.
- void [RTC_B_setCalibrationData](#) (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
Sets the specified calibration for the RTC.
- void [RTC_B_initCalendar](#) (uint16_t baseAddress, [Calendar](#) *CalendarTime, uint16_t formatSelect)
Initializes the settings to operate the RTC in calendar mode.
- [Calendar](#) [RTC_B_getCalendarTime](#) (uint16_t baseAddress)
Returns the [Calendar](#) Time stored in the [Calendar](#) registers of the RTC.
- void [RTC_B_configureCalendarAlarm](#) (uint16_t baseAddress, [RTC_B_configureCalendarAlarmParam](#) *param)
Sets and Enables the desired [Calendar](#) Alarm settings.
- void [RTC_B_setCalendarEvent](#) (uint16_t baseAddress, uint16_t eventSelect)
Sets a single specified [Calendar](#) interrupt condition.
- void [RTC_B_definePrescaleEvent](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
Sets up an interrupt condition for the selected Prescaler.
- uint8_t [RTC_B_getPrescaleValue](#) (uint16_t baseAddress, uint8_t prescaleSelect)
Returns the selected prescaler value.

- void `RTC_B_setPrescaleValue` (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
Sets the selected prescaler value.
- void `RTC_B_enableInterrupt` (uint16_t baseAddress, uint8_t interruptMask)
Enables selected RTC interrupt sources.
- void `RTC_B_disableInterrupt` (uint16_t baseAddress, uint8_t interruptMask)
Disables selected RTC interrupt sources.
- uint8_t `RTC_B_getInterruptStatus` (uint16_t baseAddress, uint8_t interruptFlagMask)
Returns the status of the selected interrupts flags.
- void `RTC_B_clearInterrupt` (uint16_t baseAddress, uint8_t interruptFlagMask)
Clears selected RTC interrupt flags.
- uint16_t `RTC_B_convertBCDToBinary` (uint16_t baseAddress, uint16_t valueToConvert)
Convert the given BCD value to binary format.
- uint16_t `RTC_B_convertBinaryToBCD` (uint16_t baseAddress, uint16_t valueToConvert)
Convert the given binary value to BCD format.

28.2.1 Detailed Description

The RTC_B API is broken into 5 groups of functions: clock settings, calendar 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 calendar 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()`

28.2.2 Function Documentation

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>interruptFlagMask</i> <i>Mask</i>	<p>is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ RTC_B_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>param</i>	is the pointer to struct for calendar alarm configuration.

Returns

None

References RTC_B_configureCalendarAlarmParam::dayOfMonthAlarm, RTC_B_configureCalendarAlarmParam::dayOfWeekAlarm, RTC_B_configureCalendarAlarmParam::hoursAlarm, and RTC_B_configureCalendarAlarmParam::minutesAlarm.

`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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>valueToConvert</i>	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

`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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>valueToConvert</i>	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

`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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>prescaleSelect</i>	is the prescaler to define an interrupt for. Valid values are: <ul style="list-style-type: none"> ■ RTC_B_PRESCALE_0 ■ RTC_B_PRESCALE_1
<i>prescaleEvent</i> ↔ <i>Divider</i>	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: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>interruptMask</i>	is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_B_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>interruptMask</i>	is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_B_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

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

<i>baseAddress</i>	is the base address of the RTC_B module.
--------------------	--

Returns

A **Calendar** structure containing the current time.

References Calendar::DayOfMonth, Calendar::DayOfWeek, Calendar::Hours, Calendar::Minutes, Calendar::Month, Calendar::Seconds, and Calendar::Year.

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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>interruptFlagMask</i>	is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>prescaleSelect</i>	is the prescaler to obtain the value of. Valid values are: <ul style="list-style-type: none"> ■ RTC_B_PRESCALE_0 ■ RTC_B_PRESCALE_1

Returns

The value of the specified prescaler count register

`void RTC_B_holdClock (uint16_t baseAddress)`

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Parameters

<i>baseAddress</i>	is the base address of the RTC_B module.
--------------------	--

Returns

None

```
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. To prevent potential erroneous alarm conditions from occurring, the alarm should be disabled by clearing the RTCAIE, RTCAIFG and AE bits with APIs: [RTC_B_disableInterrupt\(\)](#), [RTC_B_clearInterrupt\(\)](#) and [RTC_B_configureCalendarAlarm\(\)](#) before calendar initialization.

Parameters

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>CalendarTime</i>	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 0-23 DayOfWeek between 0-6 DayOfMonth between 1-31 Year between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.
<i>formatSelect</i>	is the format for the Calendar registers to use. Valid values are: <ul style="list-style-type: none"> ■ RTC_B_FORMAT_BINARY [Default] ■ RTC_B_FORMAT_BCD Modified bits are RTCBCD of RTCCTL1 register.

Returns

None

References [Calendar::DayOfMonth](#), [Calendar::DayOfWeek](#), [Calendar::Hours](#), [Calendar::Minutes](#), [Calendar::Month](#), [Calendar::Seconds](#), and [Calendar::Year](#).

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>eventSelect</i>	is the condition selected. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>offsetDirection</i>	is the direction that the calibration offset will go. Valid values are: <ul style="list-style-type: none"> ■ 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.
<i>offsetValue</i>	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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>frequencySelect</i>	is the frequency output to RTCCLK. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
<i>prescaleSelect</i>	is the prescaler to set the value for. Valid values are: <ul style="list-style-type: none"> ■ <code>RTC_B_PRESCALE_0</code> ■ <code>RTC_B_PRESCALE_1</code>
<i>prescaleCounterValue</i>	is the specified value to set the prescaler to. Valid values are any integer between 0-255. Modified bits are <code>RTxPS</code> of <code>RTxPS</code> register.

Returns

None

```
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

<i>baseAddress</i>	is the base address of the RTC_B module.
--------------------	--

Returns

None

28.3 Programming Example

The following example shows how to initialize and use the RTC API to setup Calendar 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();
```

29 Real-Time Clock (RTC_C)

Introduction	290
API Functions	290
Programming Example	306

29.1 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.

29.2 API Functions

Functions

- void [RTC_C.startClock](#) (uint16_t baseAddress)
Starts the RTC.
- void [RTC_C.holdClock](#) (uint16_t baseAddress)
Holds the RTC.
- void [RTC_C.setCalibrationFrequency](#) (uint16_t baseAddress, uint16_t frequencySelect)
Allows and Sets the frequency output to RTCCLK pin for calibration measurement.
- void [RTC_C.setCalibrationData](#) (uint16_t baseAddress, uint8_t offsetDirection, uint8_t offsetValue)
Sets the specified calibration for the RTC.
- void [RTC_C.initCounter](#) (uint16_t baseAddress, uint16_t clockSelect, uint16_t counterSizeSelect)
Initializes the settings to operate the RTC in Counter mode.
- bool [RTC_C.setTemperatureCompensation](#) (uint16_t baseAddress, uint16_t offsetDirection, uint8_t offsetValue)
Sets the specified temperature compensation for the RTC.
- void [RTC_C.initCalendar](#) (uint16_t baseAddress, [Calendar](#) *CalendarTime, uint16_t formatSelect)
Initializes the settings to operate the RTC in calendar mode.
- [Calendar](#) [RTC_C.getCalendarTime](#) (uint16_t baseAddress)
Returns the Calendar Time stored in the Calendar registers of the RTC.
- void [RTC_C.configureCalendarAlarm](#) (uint16_t baseAddress, [RTC_C.configureCalendarAlarmParam](#) *param)

- Sets and Enables the desired [Calendar Alarm](#) settings.*
- void [RTC_C_setCalendarEvent](#) (uint16_t baseAddress, uint16_t eventSelect)
 - Sets a single specified [Calendar](#) interrupt condition.*
- uint32_t [RTC_C_getCounterValue](#) (uint16_t baseAddress)
 - Returns the value of the Counter register.*
- void [RTC_C_setCounterValue](#) (uint16_t baseAddress, uint32_t counterValue)
 - Sets the value of the Counter register.*
- void [RTC_C_initCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint16_t prescaleClockSelect, uint16_t prescaleDivider)
 - Initializes the Prescaler for Counter mode.*
- void [RTC_C_holdCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect)
 - Holds the selected Prescaler.*
- void [RTC_C_startCounterPrescale](#) (uint16_t baseAddress, uint8_t prescaleSelect)
 - Starts the selected Prescaler.*
- void [RTC_C_definePrescaleEvent](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleEventDivider)
 - Sets up an interrupt condition for the selected Prescaler.*
- uint8_t [RTC_C_getPrescaleValue](#) (uint16_t baseAddress, uint8_t prescaleSelect)
 - Returns the selected prescaler value.*
- void [RTC_C_setPrescaleValue](#) (uint16_t baseAddress, uint8_t prescaleSelect, uint8_t prescaleCounterValue)
 - Sets the selected Prescaler value.*
- void [RTC_C_enableInterrupt](#) (uint16_t baseAddress, uint8_t interruptMask)
 - Enables selected RTC interrupt sources.*
- void [RTC_C_disableInterrupt](#) (uint16_t baseAddress, uint8_t interruptMask)
 - Disables selected RTC interrupt sources.*
- uint8_t [RTC_C_getInterruptStatus](#) (uint16_t baseAddress, uint8_t interruptFlagMask)
 - Returns the status of the selected interrupts flags.*
- void [RTC_C_clearInterrupt](#) (uint16_t baseAddress, uint8_t interruptFlagMask)
 - Clears selected RTC interrupt flags.*
- uint16_t [RTC_C_convertBCDToBinary](#) (uint16_t baseAddress, uint16_t valueToConvert)
 - Convert the given BCD value to binary format.*
- uint16_t [RTC_C_convertBinaryToBCD](#) (uint16_t baseAddress, uint16_t valueToConvert)
 - Convert the given binary value to BCD format.*

29.2.1 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()`

29.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>interruptFlag</i> ↔ <i>Mask</i>	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_C_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
void RTC_C_configureCalendarAlarm ( uint16_t baseAddress, RTC_C_configure↔  
CalendarAlarmParam * 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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>param</i>	is the pointer to struct for calendar alarm configuration.

Returns

None

References RTC_C_configureCalendarAlarmParam::dayOfMonthAlarm, RTC_C_configureCalendarAlarmParam::dayOfWeekAlarm, RTC_C_configureCalendarAlarmParam::hoursAlarm, and RTC_C_configureCalendarAlarmParam::minutesAlarm.

`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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>valueToConvert</i>	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

`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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>valueToConvert</i>	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

`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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>prescaleSelect</i>	is the prescaler to define an interrupt for. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1
<i>prescaleEvent</i> ↔ <i>Divider</i>	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: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>interruptMask</i>	is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_C_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>interruptMask</i>	is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_C_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
--------------------	--

Returns

A **Calendar** structure containing the current time.

References Calendar::DayOfMonth, Calendar::DayOfWeek, Calendar::Hours, Calendar::Minutes, Calendar::Month, Calendar::Seconds, and Calendar::Year.

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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
--------------------	--

Returns

The raw value of the full 32-bit Counter Register.

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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>interruptFlag</i> ↔ <i>Mask</i>	is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ RTC_C_TIME_EVENT_INTERRUPT - asserts when counter overflows in counter mode or when Calendar event condition defined by <code>defineCalendarEvent()</code> 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
--------------------	--

<i>prescaleSelect</i>	is the prescaler to obtain the value of. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1
-----------------------	---

Returns

The value of the specified prescaler count register

```
void RTC_C_holdClock ( uint16_t baseAddress )
```

Holds the RTC.

This function sets the RTC main hold bit to disable RTC functionality.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>prescaleSelect</i>	is the prescaler to hold. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1

Returns

None

```
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. To prevent potential erroneous alarm conditions from occurring, the alarm should be disabled by clearing the RTCAIE, RTCAIFG and AE bits with APIs: [RTC_C_disableInterrupt\(\)](#), [RTC_C_clearInterrupt\(\)](#) and [RTC_C_configureCalendarAlarm\(\)](#) before calendar initialization.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>CalendarTime</i>	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 0-23 DayOfWeek between 0-6 DayOfMonth between 1-31 Year between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.
<i>formatSelect</i>	is the format for the Calendar registers to use. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_FORMAT_BINARY [Default] ■ RTC_C_FORMAT_BCD Modified bits are RTCBCD of RTCCTL1 register.

Returns

None

References [Calendar::DayOfMonth](#), [Calendar::DayOfWeek](#), [Calendar::Hours](#), [Calendar::Minutes](#), [Calendar::Month](#), [Calendar::Seconds](#), and [Calendar::Year](#).

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>clockSelect</i>	is the selected clock for the counter mode to use. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_CLOCKSELECT_32KHZ_OSC ■ RTC_C_CLOCKSELECT_RT1PS Modified bits are RTCSSEL of RTCCTL1 register.

<i>counterSize</i> ↔ <i>Select</i>	is the size of the counter. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>prescaleSelect</i>	is the prescaler to initialize. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1
<i>prescaleClock</i> ↔ <i>Select</i>	is the clock to drive the selected prescaler. Valid values are: <ul style="list-style-type: none"> ■ 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.
<i>prescaleDivider</i>	is the divider for the selected clock source. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>eventSelect</i>	is the condition selected. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_CALENDAREVENT_MINUTECHANGE - assert interrupt on every minute ■ RTC_C_CALENDAREVENT_HOURLCHANGE - 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

```
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).

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>offsetDirection</i>	is the direction that the calibration offset will go. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_CALIBRATION_DOWN1PPM - calibrate at steps of -1 ■ RTC_C_CALIBRATION_UP1PPM - calibrate at steps of +1 Modified bits are RTC0CAL s of RTC0CAL register.

<i>offsetValue</i>	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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>frequencySelect</i>	is the frequency output to RTCCLK. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>counterValue</i>	is the value to set the Counter register to; a valid value may be any 32-bit integer.

Returns

None

```
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\(\)](#).

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>prescaleSelect</i>	is the prescaler to set the value for. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1
<i>prescaleCounterValue</i>	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

```
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).

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>offsetDirection</i>	is the direction that the calibration offset will go Valid values are: <ul style="list-style-type: none"> ■ RTC_C_COMPENSATION_DOWN1PPM ■ RTC_C_COMPENSATION_UP1PPM Modified bits are RTCTCMPS of RTCTCMP register.
<i>offsetValue</i>	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

```
void RTC_C_startClock ( uint16_t baseAddress )
```

Starts the RTC.

This function clears the RTC main hold bit to allow the RTC to function.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
--------------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the RTC_C module.
<i>prescaleSelect</i>	is the prescaler to start. Valid values are: <ul style="list-style-type: none"> ■ RTC_C_PRESCALE_0 ■ RTC_C_PRESCALE_1

Returns

None

29.3 Programming Example

The following example shows how to initialize and use the RTC_C API to setup Calendar 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();
```

30 24-Bit Sigma Delta Converter (SD24_B)

Introduction	308
API Functions	308
Programming Example	322

30.1 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.

30.2 API Functions

Functions

- void `SD24_B_init` (uint16_t baseAddress, `SD24_B_initParam` *param)
Initializes the SD24.B Module.
- void `SD24_B_initConverter` (uint16_t baseAddress, `SD24_B_initConverterParam` *param)
Configure SD24.B converter.
- void `SD24_B_initConverterAdvanced` (uint16_t baseAddress, `SD24_B_initConverterAdvancedParam` *param)
Configure SD24.B converter - Advanced Configure.
- void `SD24_B_setConverterDataFormat` (uint16_t baseAddress, uint8_t converter, uint8_t dataFormat)
Set SD24.B converter data format.
- void `SD24_B_startGroupConversion` (uint16_t baseAddress, uint8_t group)
Start Conversion Group.
- void `SD24_B_stopGroupConversion` (uint16_t baseAddress, uint8_t group)
Stop Conversion Group.
- void `SD24_B_startConverterConversion` (uint16_t baseAddress, uint8_t converter)
Start Conversion for Converter.

- void [SD24_B_stopConverterConversion](#) (uint16_t baseAddress, uint8_t converter)
Stop Conversion for Converter.
- void [SD24_B_configureDMATrigger](#) (uint16_t baseAddress, uint16_t interruptFlag)
Configures the converter that triggers a DMA transfer.
- void [SD24_B_setInterruptDelay](#) (uint16_t baseAddress, uint8_t converter, uint8_t sampleDelay)
Configures the delay for an interrupt to trigger.
- void [SD24_B_setOversampling](#) (uint16_t baseAddress, uint8_t converter, uint16_t oversampleRatio)
Configures the oversampling ratio for a converter.
- void [SD24_B_setGain](#) (uint16_t baseAddress, uint8_t converter, uint8_t gain)
Configures the gain for the converter.
- uint32_t [SD24_B_getResults](#) (uint16_t baseAddress, uint8_t converter)
Returns the results for a converter.
- uint16_t [SD24_B_getHighWordResults](#) (uint16_t baseAddress, uint8_t converter)
Returns the high word results for a converter.
- void [SD24_B_enableInterrupt](#) (uint16_t baseAddress, uint8_t converter, uint16_t mask)
Enables interrupts for the SD24.B Module.
- void [SD24_B_disableInterrupt](#) (uint16_t baseAddress, uint8_t converter, uint16_t mask)
Disables interrupts for the SD24.B Module.
- void [SD24_B_clearInterrupt](#) (uint16_t baseAddress, uint8_t converter, uint16_t mask)
Clears interrupts for the SD24.B Module.
- uint16_t [SD24_B_getInterruptStatus](#) (uint16_t baseAddress, uint8_t converter, uint16_t mask)
Returns the interrupt status for the SD24.B Module.

30.2.1 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\(\)](#)

30.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	is the selected converter. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>mask</i>	is the bit mask of the converter interrupt sources to clear. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ SD24_B_CONVERTER_INTERRUPT ■ SD24_B_CONVERTER_OVERFLOW_INTERRUPT Modified bits are SD24OVIFGx of SD24BIFG register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>interruptFlag</i>	selects the converter interrupt that triggers a DMA transfer. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	is the selected converter. Valid values are: <ul style="list-style-type: none"> ■ 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

<i>mask</i>	is the bit mask of the converter interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ SD24_B_CONVERTER_INTERRUPT ■ SD24_B_CONVERTER_OVERFLOW_INTERRUPT Modified bits are SD24OVIEx of SD24BIE register.
-------------	---

Modified bits of **SD24BIE** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	is the selected converter. Valid values are: <ul style="list-style-type: none"> ■ 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

<i>mask</i>	is the bit mask of the converter interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ SD24_B_CONVERTER_INTERRUPT ■ SD24_B_CONVERTER_OVERFLOW_INTERRUPT Modified bits are SD24OVIEx of SD24BIE register.
-------------	--

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter who's results will be returned Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	is the selected converter. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>mask</i>	is the bit mask of the converter interrupt sources to return. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter who's results will be returned Valid values are: <ul style="list-style-type: none"> ■ 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

```
void SD24_B_init ( uint16_t baseAddress, SD24_B_initParam * 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: $f_{sd24} = f_{clk} / (divider * predivider)$

Parameters

<i>baseAddress</i>	is the base address of the SD24_B module.
<i>param</i>	is the pointer to struct for initialization.

Returns

None

References SD24_B_initParam::clockDivider, SD24_B_initParam::clockPreDivider, SD24_B_initParam::clockSourceSelect, and SD24_B_initParam::referenceSelect.

```
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\(\)](#).

Parameters

<i>baseAddress</i>	is the base address of the SD24_B module.
<i>param</i>	is the pointer to struct for converter configuration.

Returns

None

References SD24_B_initConverterParam::alignment, SD24_B_initConverterParam::conversionMode, SD24_B_initConverterParam::converter, and SD24_B_initConverterParam::startSelect.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>param</i>	is the pointer to struct for converter advanced configuration.

Returns

None

References [SD24_B_initConverterAdvancedParam::alignment](#), [SD24_B_initConverterAdvancedParam::conversionMode](#), [SD24_B_initConverterAdvancedParam::converter](#), [SD24_B_initConverterAdvancedParam::dataFormat](#), [SD24_B_initConverterAdvancedParam::gain](#), [SD24_B_initConverterAdvancedParam::oversampleRatio](#), [SD24_B_initConverterAdvancedParam::sampleDelay](#), and [SD24_B_initConverterAdvancedParam::startSelect](#).

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be configured. Check datasheet for available converters on device. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>dataFormat</i>	selects how the data format of the results Valid values are: <ul style="list-style-type: none"> ■ SD24_B_DATA_FORMAT_BINARY [Default] ■ SD24_B_DATA_FORMAT_2COMPLEMENT Modified bits are SD24DFx of SD24BCCTLx register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be configured Valid values are: <ul style="list-style-type: none"> ■ 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

<i>gain</i>	selects the gain for the converter Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be stopped Valid values are: <ul style="list-style-type: none"> ■ 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

<i>sampleDelay</i>	selects the delay for the interrupt Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be configured Valid values are: <ul style="list-style-type: none"> ■ 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

<i>oversampleRatio</i>	selects oversampling ratio for the converter Valid values are: <ul style="list-style-type: none"> ■ 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

```
void SD24_B_startConverterConversion ( uint16_t baseAddress, uint8_t converter )
```

Start Conversion for Converter.

This function starts a single converter.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be started Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>group</i>	selects the group that will be started Valid values are: <ul style="list-style-type: none"> ■ SD24.B.GROUP0 ■ SD24.B.GROUP1 ■ SD24.B.GROUP2 ■ SD24.B.GROUP3 Modified bits are SD24DGRP_xSC of SD24BCTL1 register.

Returns

None

```
void SD24_B_stopConverterConversion ( uint16_t baseAddress, uint8_t converter )
```

Stop Conversion for Converter.

This function stops a single converter.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>converter</i>	selects the converter that will be stopped Valid values are: <ul style="list-style-type: none"> ■ 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 SD24BCCTL_x register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SD24.B module.
<i>group</i>	selects the group that will be stopped Valid values are: <ul style="list-style-type: none"> ■ SD24.B_GROUP0 ■ SD24.B_GROUP1 ■ SD24.B_GROUP2 ■ SD24.B_GROUP3 Modified bits are SD24DGRP_xSC of SD24BCTL1 register.

Returns

None

30.3 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_initParam initParam = {0};
initParam.clockSourceSelect = SD24_B_CLOCKSOURCE_SMCLK; // Select SMCLK as SD24.B clock
    source
initParam.clockPreDivider = SD24_B_PRECLOCKDIVIDER_1;
initParam.clockDivider = SD24_B_CLOCKDIVIDER_1;
initParam.referenceSelect = SD24_B_REF_INTERNAL; // Select internal REF
SD24_B_init(SD24_BASE, &initParam);

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 bit to start conversion

    // 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
}

```

31 SFR Module

Introduction	323
API Functions	323
Programming Example	328

31.1 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 \sim RST/NMI pin

The SFR module can enable interrupts to be generated from other peripherals of the device.

31.2 API Functions

Functions

- void [SFR_enableInterrupt](#) (uint8_t interruptMask)
Enables selected SFR interrupt sources.
- void [SFR_disableInterrupt](#) (uint8_t interruptMask)
Disables selected SFR interrupt sources.
- uint8_t [SFR_getInterruptStatus](#) (uint8_t interruptFlagMask)
Returns the status of the selected SFR interrupt flags.
- void [SFR_clearInterrupt](#) (uint8_t interruptFlagMask)
Clears the selected SFR interrupt flags.
- void [SFR_setResetPinPullResistor](#) (uint16_t pullResistorSetup)
Sets the pull-up/down resistor on the \sim RST/NMI pin.
- void [SFR_setNMIEdge](#) (uint16_t edgeDirection)
Sets the edge direction that will assert an NMI from a signal on the \sim RST/NMI pin if NMI function is active.
- void [SFR_setResetNMIPinFunction](#) (uint8_t resetPinFunction)
Sets the function of the \sim RST/NMI pin.

31.2.1 Detailed Description

The SFR API is broken into 2 groups: the SFR interrupts and the SFR \sim RST/NMI pin control

The SFR interrupts are handled by

- [SFR_enableInterrupt\(\)](#)
- [SFR_disableInterrupt\(\)](#)
- [SFR_getInterruptStatus\(\)](#)
- [SFR_clearInterrupt\(\)](#)

The SFR \sim RST/NMI pin is controlled by

- [SFR_setResetPinPullResistor\(\)](#)
- [SFR_setNMIEdge\(\)](#)
- [SFR_setResetNMIPinFunction\(\)](#)

31.2.2 Function Documentation

`void SFR_clearInterrupt (uint8_t interruptFlagMask)`

Clears the selected SFR interrupt flags.

This function clears the status of the selected SFR interrupt flags.

Parameters

<i>interruptFlagMask</i>	<p>is the bit mask of interrupt flags that should be cleared Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

`void SFR_disableInterrupt (uint8_t interruptMask)`

Disables selected SFR interrupt sources.

This function disables the selected SFR interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Parameters

<i>interruptMask</i>	<p>is the bit mask of interrupts that will be disabled. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>interruptMask</i>	<p>is the bit mask of interrupts that will be enabled. Mask value is the logical OR of any of the following:</p> <ul style="list-style-type: none"> ■ 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

`uint8_t SFR_getInterruptStatus (uint8_t interruptFlagMask)`

Returns the status of the selected SFR interrupt flags.

This function returns the status of the selected SFR interrupt flags in a bit mask format matching that passed into the `interruptFlagMask` parameter.

Parameters

<i>interruptFlagMask</i>	<p>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:</p> <ul style="list-style-type: none"> ■ 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

`void SFR_setNMIEdge (uint16_t edgeDirection)`

Sets the edge direction that will assert an NMI from a signal on the \sim RST/NMI pin if NMI function is active.

This function sets the edge direction that will assert an NMI from a signal on the \sim RST/NMI pin if the NMI function is active. To activate the NMI function of the \sim RST/NMI use the [SFR_setResetNMIpinFunction\(\)](#) passing `SFR_RESETPINFUNC_NMI` into the `resetPinFunction` parameter.

Parameters

<i>edgeDirection</i>	is the direction that the signal on the \sim RST/NMI pin should go to signal an interrupt, if enabled. Valid values are: <ul style="list-style-type: none"> ■ SFR_NMI_RISINGEDGE [Default] ■ SFR_NMI_FALLINGEDGE Modified bits are SYSNMIES of SFRRPCR register.
----------------------	--

Returns

None

```
void SFR_setResetNMIPinFunction ( uint8_t resetPinFunction )
```

Sets the function of the \sim RST/NMI pin.

This function sets the functionality of the \sim RST/NMI pin, whether in reset mode which will assert a reset if a low signal is observed on that pin, or an NMI which will assert an interrupt from an edge of the signal dependent on the setting of the *edgeDirection* parameter in [SFR_setNMIEdge\(\)](#).

Parameters

<i>resetPin↔ Function</i>	is the function that the \sim RST/NMI pin should take on. Valid values are: <ul style="list-style-type: none"> ■ SFR_RESETPINFUNC_RESET [Default] ■ SFR_RESETPINFUNC_NMI Modified bits are SYSNMI of SFRRPCR register.
-------------------------------	--

Returns

None

```
void SFR_setResetPinPullResistor ( uint16_t pullResistorSetup )
```

Sets the pull-up/down resistor on the \sim RST/NMI pin.

This function sets the pull-up/down resistors on the \sim RST/NMI pin to the settings from the *pullResistorSetup* parameter.

Parameters

<i>pullResistor↔ Setup</i>	is the selection of how the pull-up/down resistor on the \sim RST/NMI pin should be setup or disabled. Valid values are: <ul style="list-style-type: none"> ■ SFR_RESISTORDISABLE ■ SFR_RESISTORENABLE_PULLUP [Default] ■ SFR_RESISTORENABLE_PULLDOWN Modified bits are SYSRSTUP of SFRRPCR register.
--------------------------------	---

Returns

None

31.3 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));
```

32 System Control Module

Introduction	329
API Functions	329
Programming Example	337

32.1 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.

32.2 API Functions

Functions

- void [SysCtl.enableDedicatedJTAGPins](#) (void)
Sets the JTAG pins to be exclusively for JTAG until a BOR occurs.
- uint8_t [SysCtl.getBSLEntryIndication](#) (void)
Returns the indication of a BSL entry sequence from the Spy-Bi-Wire.
- void [SysCtl.enablePMMAccessProtect](#) (void)
Enables PMM Access Protection.
- void [SysCtl.enableRAMBasedInterruptVectors](#) (void)
Enables RAM-based Interrupt Vectors.
- void [SysCtl.disableRAMBasedInterruptVectors](#) (void)
Disables RAM-based Interrupt Vectors.
- void [SysCtl.enableBSLProtect](#) (void)
Enables BSL memory protection.
- void [SysCtl.disableBSLProtect](#) (void)
Disables BSL memory protection.
- void [SysCtl.enableBSLMemory](#) (void)
Enables BSL memory.
- void [SysCtl.disableBSLMemory](#) (void)
Disables BSL memory.
- void [SysCtl.setRAMAssignedToBSL](#) (uint8_t BSLRAMAssignment)
Sets RAM assignment to BSL area.
- void [SysCtl.setBSLSize](#) (uint8_t BSLSizeSelect)
Sets the size of the BSL in Flash.
- void [SysCtl.initJTAGMailbox](#) (uint8_t mailboxSizeSelect, uint8_t autoClearInboxFlagSelect)
Initializes JTAG Mailbox with selected properties.
- uint8_t [SysCtl.getJTAGMailboxFlagStatus](#) (uint8_t mailboxFlagMask)
Returns the status of the selected JTAG Mailbox flags.
- void [SysCtl.clearJTAGMailboxFlagStatus](#) (uint8_t mailboxFlagMask)
Clears the status of the selected JTAG Mailbox flags.
- uint16_t [SysCtl.getJTAGInboxMessage16Bit](#) (uint8_t inboxSelect)
Returns the contents of the selected JTAG Inbox in a 16 bit format.
- uint32_t [SysCtl.getJTAGInboxMessage32Bit](#) (void)

Returns the contents of JTAG Inboxes in a 32 bit format.

- void `SysCtl_setJTAGOutgoingMessage16Bit` (uint8_t outboxSelect, uint16_t outgoingMessage)
Sets a 16 bit outgoing message in to the selected JTAG Outbox.
- void `SysCtl_setJTAGOutgoingMessage32Bit` (uint32_t outgoingMessage)
Sets a 32 bit message in to both JTAG Outboxes.

32.2.1 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

- `SysCtl_enableDedicatedJTAGPins()`
- `SysCtl_getBSLEntryIndication()`
- `SysCtl_enablePMMAccessProtect()`
- `SysCtl_enableRAMBasedInterruptVectors()`
- `SysCtl_disableRAMBasedInterruptVectors()`

The BSL controls are handled by

- `SysCtl_enableBSLProtect()`
- `SysCtl_disableBSLProtect()`
- `SysCtl_disableBSLMemory()`
- `SysCtl_enableBSLMemory()`
- `SysCtl_setRAMAssignedToBSL()`
- `SysCtl_setBSLSize()`

The JTAG Mailbox controls are handled by

- `SysCtl_initJTAGMailbox()`
- `SysCtl_getJTAGMailboxFlagStatus()`
- `SysCtl_getJTAGInboxMessage16Bit()`
- `SysCtl_getJTAGInboxMessage32Bit()`
- `SysCtl_setJTAGOutgoingMessage16Bit()`
- `SysCtl_setJTAGOutgoingMessage32Bit()`
- `SysCtl_clearJTAGMailboxFlagStatus()`

32.2.2 Function Documentation

`void SysCtl_clearJTAGMailboxFlagStatus (uint8_t mailboxFlagMask)`

Clears the status of the selected JTAG Mailbox flags.

This function clears the selected JTAG Mailbox flags.

Parameters

<i>mailboxFlag</i> ↔ <i>Mask</i>	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: <ul style="list-style-type: none"> ■ SYSCTL_JTAGOUTBOX_FLAG0 - flag for JTAG outbox 0 ■ SYSCTL_JTAGOUTBOX_FLAG1 - flag for JTAG outbox 1 ■ SYSCTL_JTAGINBOX_FLAG0 - flag for JTAG inbox 0 ■ SYSCTL_JTAGINBOX_FLAG1 - flag for JTAG inbox 1
-------------------------------------	--

Returns

None

```
void SysCtl_disableBSLMemory ( void )
```

Disables BSL memory.

This function disables BSL memory, which makes BSL memory act like vacant memory.

Returns

None

```
void SysCtl_disableBSLProtect ( void )
```

Disables BSL memory protection.

This function disables protection on the BSL memory.

Returns

None

```
void SysCtl_disableRAMBasedInterruptVectors ( void )
```

Disables RAM-based Interrupt Vectors.

This function disables the interrupt vectors from being generated at the top of the RAM.

Returns

None

```
void SysCtl_enableBSLMemory ( void )
```

Enables BSL memory.

This function enables BSL memory, which allows BSL memory to be addressed

Returns

None

void SysCtl_enableBSLProtect (void)

Enables BSL memory protection.

This function enables protection on the BSL memory, which prevents any reading, programming, or erasing of the BSL memory.

Returns

None

void SysCtl_enableDedicatedJTAGPins (void)

Sets the JTAG pins to be exclusively for JTAG until a BOR occurs.

This function sets the JTAG pins to be exclusively used for the JTAG, and not to be shared with the GPIO pins. This setting can only be cleared when a BOR occurs.

Returns

None

void SysCtl_enablePMMAccessProtect (void)

Enables PMM Access Protection.

This function enables the PMM Access Protection, which will lock any changes on the PMM control registers until a BOR occurs.

Returns

None

void SysCtl_enableRAMBasedInterruptVectors (void)

Enables RAM-based Interrupt Vectors.

This function enables RAM-base Interrupt Vectors, which means that interrupt vectors are generated with the end address at the top of RAM, instead of the top of the lower 64kB of flash.

Returns

None

uint8_t SysCtl_getBSLEntryIndication (void)

Returns the indication of a BSL entry sequence from the Spy-Bi-Wire.

This function returns the indication of a BSL entry sequence from the Spy- Bi-Wire.

Returns

One of the following:

- **SysCtl_BSLENTY_INDICATED**
- **SysCtl_BSLENTY_NOTINDICATED**
indicating if a BSL entry sequence was detected

uint16_t SysCtl_getJTAGInboxMessage16Bit (uint8_t *inboxSelect*)

Returns the contents of the selected JTAG Inbox in a 16 bit format.

This function returns the message contents of the selected JTAG inbox. If the auto clear settings for the Inbox flags were set, then using this function will automatically clear the corresponding JTAG inbox flag.

Parameters

<i>inboxSelect</i>	is the chosen JTAG inbox that the contents of should be returned Valid values are: <ul style="list-style-type: none"> ■ SYSCTL_JTAGINBOX_0 - return contents of JTAG inbox 0 ■ SYSCTL_JTAGINBOX_1 - return contents of JTAG inbox 1
--------------------	---

Returns

The contents of the selected JTAG inbox in a 16 bit format.

uint32_t SysCtl_getJTAGInboxMessage32Bit (void)

Returns the contents of JTAG Inboxes in a 32 bit format.

This function returns the message contents of both JTAG inboxes in a 32 bit format. This function should be used if 32-bit messaging has been set in the SYS_initJTAGMailbox() function. If the auto clear settings for the Inbox flags were set, then using this function will automatically clear both JTAG inbox flags.

Returns

The contents of both JTAG messages in a 32 bit format.

uint8_t SysCtl_getJTAGMailboxFlagStatus (uint8_t *mailboxFlagMask*)

Returns the status of the selected JTAG Mailbox flags.

This function will return the status of the selected JTAG Mailbox flags in bit mask format matching that passed into the mailboxFlagMask parameter.

Parameters

<i>mailboxFlag</i> ↔ <i>Mask</i>	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: <ul style="list-style-type: none"> ■ SYSCTL_JTAGOUTBOX_FLAG0 - flag for JTAG outbox 0 ■ SYSCTL_JTAGOUTBOX_FLAG1 - flag for JTAG outbox 1 ■ SYSCTL_JTAGINBOX_FLAG0 - flag for JTAG inbox 0 ■ SYSCTL_JTAGINBOX_FLAG1 - flag for JTAG inbox 1
-------------------------------------	---

Returns

A bit mask of the status of the selected mailbox flags.

```
void SysCtl_initJTAGMailbox ( uint8_t mailboxSizeSelect, uint8_t autoClearInboxFlagSelect
)
```

Initializes JTAG Mailbox with selected properties.

This function sets the specified settings for the JTAG Mailbox system. The settings that can be set are the size of the JTAG messages, and the auto-clearing of the inbox flags. If the inbox flags are set to auto-clear, then the inbox flags will be cleared upon reading of the inbox message buffer, otherwise they will have to be reset by software using the `SYS_clearJTAGMailboxFlagStatus()` function.

Parameters

<i>mailboxSize</i> ↔ <i>Select</i>	is the size of the JTAG Mailboxes, whether 16- or 32-bits. Valid values are: <ul style="list-style-type: none"> ■ SYSCTL_JTAGMBSIZE_16BIT [Default] - the JTAG messages will take up only one JTAG mailbox (i. e. an outgoing message will take up only 1 outbox of the JTAG mailboxes) ■ SYSCTL_JTAGMBSIZE_32BIT - the JTAG messages will be contained within both JTAG mailboxes (i. e. an outgoing message will take up both Outboxes of the JTAG mailboxes) Modified bits are JMBMODE of SYSJMBC register.
---------------------------------------	--

<i>autoClear</i> ↔ <i>InboxFlagSelect</i>	<p>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:</p> <ul style="list-style-type: none"> ■ SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1AUTO [Default] - both JTAG inbox flags will be reset automatically when the corresponding inbox is read from. ■ SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1SW - only JTAG inbox 0 flag is reset automatically, while JTAG inbox 1 is reset with the ■ SYSCTL_JTAGINBOX0SW_JTAGINBOX1AUTO - only JTAG inbox 1 flag is reset automatically, while JTAG inbox 0 is reset with the ■ SYSCTL_JTAGINBOX0SW_JTAGINBOX1SW - both JTAG inbox flags will need to be reset manually by the <p>Modified bits are JMBCLR0OFF and JMBCLR1OFF of SYSJMBC register.</p>
--	--

Returns

None

```
void SysCtl_setBSLSize ( uint8_t BSLSizeSelect )
```

Sets the size of the BSL in Flash.

This function sets the size of the BSL in Flash memory.

Parameters

<i>BSLSizeSelect</i>	<p>is the amount of segments the BSL should take. Valid values are:</p> <ul style="list-style-type: none"> ■ SYSCTL_BSLSIZE_SEG3 ■ SYSCTL_BSLSIZE_SEGS23 ■ SYSCTL_BSLSIZE_SEGS123 ■ SYSCTL_BSLSIZE_SEGS1234 [Default] <p>Modified bits are SYSBSLSIZE of SYSBSLC register.</p>
----------------------	--

Returns

None

```
void SysCtl_setJTAGOutgoingMessage16Bit ( uint8_t outboxSelect, uint16_t  
outgoingMessage )
```

Sets a 16 bit outgoing message in to the selected JTAG Outbox.

This function sets the outgoing message in the selected JTAG outbox. The corresponding JTAG outbox flag is cleared after this function, and set after the JTAG has read the message.

Parameters

<i>outboxSelect</i>	is the chosen JTAG outbox that the message should be set it. Valid values are: <ul style="list-style-type: none"> ■ SYSCTL_JTAGOUTBOX_0 - set the contents of JTAG outbox 0 ■ SYSCTL_JTAGOUTBOX_1 - set the contents of JTAG outbox 1
<i>outgoing↔ Message</i>	is the message to send to the JTAG. Modified bits are MSGHI and MSGLO of SYSJMBOX register.

Returns

None

```
void SysCtl_setJTAGOutgoingMessage32Bit ( uint32_t outgoingMessage )
```

Sets a 32 bit message in to both JTAG Outboxes.

This function sets the 32-bit outgoing message in both JTAG outboxes. The JTAG outbox flags are cleared after this function, and set after the JTAG has read the message.

Parameters

<i>outgoing↔ Message</i>	is the message to send to the JTAG. Modified bits are MSGHI and MSGLO of SYSJMBOX register.
------------------------------	---

Returns

None

```
void SysCtl_setRAMAssignedToBSL ( uint8_t BSLRAMAssignment )
```

Sets RAM assignment to BSL area.

This function allows RAM to be assigned to BSL, based on the selection of the *BSLRAMAssignment* parameter.

Parameters

<i>BSLRAM↔ Assignment</i>	is the selection of if the BSL should be placed in RAM or not. Valid values are: <ul style="list-style-type: none"> ■ SYSCTL_BSLRAMASSIGN_NORAM [Default] ■ SYSCTL_BSLRAMASSIGN_LOWEST16BYTES Modified bits are SYSBSLR of SYSBSLC register.
-------------------------------	--

Returns

None

32.3 Programming Example

The following example shows how to initialize and use the SYS API

```
SysCtl.enableBSLProtect();
```

33 Timer Event Control (TEC)

Introduction	338
API Functions	338
Programming Example	348

33.1 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

33.2 API Functions

Functions

- void [TEC_initExternalClearInput](#) (uint16_t baseAddress, uint8_t signalType, uint8_t signalHold, uint8_t polarityBit)
Configures the Timer Event Control External Clear Input.
- void [TEC_initExternalFaultInput](#) (uint16_t baseAddress, [TEC_initExternalFaultInputParam](#) *param)
Configures the Timer Event Control External Fault Input.
- void [TEC_enableExternalFaultInput](#) (uint16_t baseAddress, uint8_t channelEventBlock)
Enable the Timer Event Control External fault input.
- void [TEC_disableExternalFaultInput](#) (uint16_t baseAddress, uint8_t channelEventBlock)
Disable the Timer Event Control External fault input.
- void [TEC_enableExternalClearInput](#) (uint16_t baseAddress)
Enable the Timer Event Control External Clear Input.
- void [TEC_disableExternalClearInput](#) (uint16_t baseAddress)
Disable the Timer Event Control External Clear Input.
- void [TEC_enableAuxiliaryClearSignal](#) (uint16_t baseAddress)
Enable the Timer Event Control Auxiliary Clear Signal.
- void [TEC_disableAuxiliaryClearSignal](#) (uint16_t baseAddress)
Disable the Timer Event Control Auxiliary Clear Signal.
- void [TEC_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)

- Clears the Timer Event Control Interrupt flag.*
- uint8_t [TEC_getInterruptStatus](#) (uint16_t baseAddress, uint8_t mask)
 - Gets the current Timer Event Control interrupt status.*
- void [TEC_enableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Enables individual Timer Event Control interrupt sources.*
- void [TEC_disableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Disables individual Timer Event Control interrupt sources.*
- uint8_t [TEC_getExternalFaultStatus](#) (uint16_t baseAddress, uint8_t mask)
 - Gets the current Timer Event Control External Fault Status.*
- void [TEC_clearExternalFaultStatus](#) (uint16_t baseAddress, uint8_t mask)
 - Clears the Timer Event Control External Fault Status.*
- uint8_t [TEC_getExternalClearStatus](#) (uint16_t baseAddress)
 - Gets the current Timer Event Control External Clear Status.*
- void [TEC_clearExternalClearStatus](#) (uint16_t baseAddress)
 - Clears the Timer Event Control External Clear Status.*

33.2.1 Detailed Description

The tec configuration is handled by

- [TEC_configureExternalClearInput\(\)](#)
- [TEC_initExternalFaultInput\(\)](#)
- [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_clearInterrupt\(\)](#)
- [TEC_getInterruptStatus\(\)](#)
- [TEC_enableInterrupt\(\)](#)
- [TEC_disableInterrupt\(\)](#)
- [TEC_getExternalFaultStatus\(\)](#)
- [TEC_clearExternalFaultStatus\(\)](#)
- [TEC_getExternalClearStatus\(\)](#)
- [TEC_clearExternalClearStatus\(\)](#)

33.2.2 Function Documentation

void [TEC_clearExternalClearStatus](#) (uint16_t *baseAddress*)

Clears the Timer Event Control External Clear Status.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
--------------------	--

Modified bits of **TECxINT** register.

Returns

None

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

Clears the Timer Event Control External Fault Status.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the masked status flag be cleared Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

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

Clears the Timer Event Control Interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void TEC_disableAuxiliaryClearSignal ( uint16_t baseAddress )
```

Disable the Timer Event Control Auxiliary Clear Signal.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
--------------------	--

Modified bits of **TECxCTL2** register.

Returns

None

```
void TEC_disableExternalClearInput ( uint16_t baseAddress )
```

Disable the Timer Event Control External Clear Input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
--------------------	--

Modified bits of **TECxCTL2** register.

Returns

None

```
void TEC_disableExternalFaultInput ( uint16_t baseAddress, uint8_t channelEventBlock )
```

Disable the Timer Event Control External fault input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>channelEventBlock</i>	selects the channel event block Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void TEC_enableAuxiliaryClearSignal ( uint16_t baseAddress )
```

Enable the Timer Event Control Auxiliary Clear Signal.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
--------------------	--

Modified bits of **TECxCTL2** register.

Returns

None

```
void TEC_enableExternalClearInput ( uint16_t baseAddress )
```

Enable the Timer Event Control External Clear Input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
--------------------	--

Modified bits of **TECxCTL2** register.

Returns

None

```
void TEC_enableExternalFaultInput ( uint16_t baseAddress, uint8_t channelEventBlock )
```

Enable the Timer Event Control External fault input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>channelEvent↔ Block</i>	selects the channel event block Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
uint8_t TEC_getExternalClearStatus ( uint16_t baseAddress )
```

Gets the current Timer Event Control External Clear Status.

Parameters

<i>baseAddress</i>	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

```
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

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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

<i>baseAddress</i>	is the base address of the TEC module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void TEC_initExternalClearInput ( uint16_t baseAddress, uint8_t signalType, uint8_t
    signalHold, uint8_t polarityBit )
```

Configures the Timer Event Control External Clear Input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>signalType</i>	is the selected signal type Valid values are: <ul style="list-style-type: none"> ■ TEC_EXTERNAL_CLEAR_SIGNALTYPE_EDGE_SENSITIVE [Default] ■ TEC_EXTERNAL_CLEAR_SIGNALTYPE_LEVEL_SENSITIVE
<i>signalHold</i>	is the selected signal hold Valid values are: <ul style="list-style-type: none"> ■ TEC_EXTERNAL_CLEAR_SIGNAL_NOT_HELD [Default] ■ TEC_EXTERNAL_CLEAR_SIGNAL_HELD
<i>polarityBit</i>	is the selected signal type Valid values are: <ul style="list-style-type: none"> ■ 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

```
void TEC_initExternalFaultInput ( uint16_t baseAddress, TEC_initExternalFaultInputParam
    * param )
```

Configures the Timer Event Control External Fault Input.

Parameters

<i>baseAddress</i>	is the base address of the TEC module.
<i>param</i>	is the pointer to struct for external fault input initialization.

Modified bits of **TECxCTL2** register.

Returns

None

References `TEC_initExternalFaultInputParam::polarityBit`,
`TEC_initExternalFaultInputParam::selectedExternalFault`,
`TEC_initExternalFaultInputParam::signalHold`, and `TEC_initExternalFaultInputParam::signalType`.

33.3 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_SENSITIVE,
        TEC_EXTERNAL_CLEAR_SIGNAL_NOT_HELD,
        TEC_EXTERNAL_CLEAR_POLARITY_RISING_EDGE_OR_HIGH_LEVEL
    );
    TEC_enableExternalClearInput(TEC1.BASE);
}
```

34 16-Bit Timer_A (TIMER_A)

Introduction	349
API Functions	350
Programming Example	364

34.1 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.

34.2 API Functions

Functions

- void `Timer_A_startCounter` (uint16_t baseAddress, uint16_t timerMode)
Starts Timer_A counter.
- void `Timer_A_initContinuousMode` (uint16_t baseAddress, `Timer_A_initContinuousModeParam` *param)
Configures Timer_A in continuous mode.
- void `Timer_A_initUpMode` (uint16_t baseAddress, `Timer_A_initUpModeParam` *param)
Configures Timer_A in up mode.
- void `Timer_A_initUpDownMode` (uint16_t baseAddress, `Timer_A_initUpDownModeParam` *param)
Configures Timer_A in up down mode.
- void `Timer_A_initCaptureMode` (uint16_t baseAddress, `Timer_A_initCaptureModeParam` *param)
Initializes Capture Mode.
- void `Timer_A_initCompareMode` (uint16_t baseAddress, `Timer_A_initCompareModeParam` *param)
Initializes Compare Mode.
- void `Timer_A_enableInterrupt` (uint16_t baseAddress)
Enable timer interrupt.
- void `Timer_A_disableInterrupt` (uint16_t baseAddress)
Disable timer interrupt.
- uint32_t `Timer_A_getInterruptStatus` (uint16_t baseAddress)
Get timer interrupt status.
- void `Timer_A_enableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Enable capture compare interrupt.
- void `Timer_A_disableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Disable capture compare interrupt.
- uint32_t `Timer_A_getCaptureCompareInterruptStatus` (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
Return capture compare interrupt status.
- void `Timer_A_clear` (uint16_t baseAddress)
Reset/Clear the timer clock divider, count direction, count.
- uint8_t `Timer_A_getSynchronizedCaptureCompareInput` (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
Get synchronized capturecompare input.
- uint8_t `Timer_A_getOutputForOutputModeOutBitValue` (uint16_t baseAddress, uint16_t captureCompareRegister)
Get output bit for output mode.
- uint16_t `Timer_A_getCaptureCompareCount` (uint16_t baseAddress, uint16_t captureCompareRegister)
Get current capturecompare count.
- void `Timer_A_setOutputForOutputModeOutBitValue` (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
Set output bit for output mode.
- void `Timer_A_outputPWM` (uint16_t baseAddress, `Timer_A_outputPWMPParam` *param)
Generate a PWM with timer running in up mode.
- void `Timer_A_stop` (uint16_t baseAddress)

- Stops the timer.*
- void `Timer_A_setCompareValue` (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
- Sets the value of the capture-compare register.*
- void `Timer_A_clearTimerInterrupt` (uint16_t baseAddress)
- Clears the Timer TAIFG interrupt flag.*
- void `Timer_A_clearCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
- Clears the capture-compare interrupt flag.*
- uint16_t `Timer_A_getCounterValue` (uint16_t baseAddress)
- Reads the current timer count value.*

34.2.1 Detailed Description

The TIMER_A API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TIMER_A configuration and initialization is handled by

- `Timer_A_startCounter()`
- `Timer_A_initUpMode()`
- `Timer_A_initUpDownMode()`
- `Timer_A_initContinuousMode()`
- `Timer_A_initCaptureMode()`
- `Timer_A_initCompareMode()`
- `Timer_A_clear()`
- `Timer_A.stop()`

TIMER_A outputs are handled by

- `Timer_A_getSynchronizedCaptureCompareInput()`
- `Timer_A_getOutputForOutputModeOutBitValue()`
- `Timer_A_setOutputForOutputModeOutBitValue()`
- `Timer_A_outputPWM()`
- `Timer_A_getCaptureCompareCount()`
- `Timer_A.setCompareValue()`
- `Timer_A.getCounterValue()`

The interrupt handler for the TIMER_A interrupt is managed with

- `Timer_A.enableInterrupt()`
- `Timer_A.disableInterrupt()`
- `Timer_A.getInterruptStatus()`
- `Timer_A.enableCaptureCompareInterrupt()`
- `Timer_A.disableCaptureCompareInterrupt()`
- `Timer_A.getCaptureCompareInterruptStatus()`
- `Timer_A.clearCaptureCompareInterrupt()`
- `Timer_A.clearTimerInterrupt()`

34.2.2 Function Documentation

`void Timer_A_clear (uint16_t baseAddress)`

Reset/Clear the timer clock divider, count direction, count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Modified bits of **TAxCTL** register.

Returns

None

```
void Timer_A_clearCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Clears the capture-compare interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture← Compare← Register</i>	selects the Capture-compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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 **TAxCTLn** register.

Returns

None

```
void Timer_A_clearTimerInterrupt ( uint16_t baseAddress )
```

Clears the Timer TAIFG interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Modified bits are **TAIFG** of **TAxCTL** register.

Returns

None

```
void Timer_A_disableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Disable capture compare interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_A_disableInterrupt ( uint16_t baseAddress )
```

Disable timer interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Modified bits of **TAxCTL** register.

Returns

None

```
void Timer_A_enableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t  
captureCompareRegister )
```

Enable capture compare interrupt.

Does not clear interrupt flags

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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 **TAxCTLn** register.

Returns

None

```
void Timer_A_enableInterrupt ( uint16_t baseAddress )
```

Enable timer interrupt.

Does not clear interrupt flags

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Modified bits of **TAxCTL** register.

Returns

None

```
uint16_t Timer_A_getCaptureCompareCount ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Get current capturecompare count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	Valid values are: <ul style="list-style-type: none"> ■ 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

```
uint32_t Timer_A_getCaptureCompareInterruptStatus ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint16_t mask )
```

Return capture compare interrupt status.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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

<i>mask</i>	is the mask for the interrupt status Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`uint16_t Timer_A_getCounterValue (uint16_t baseAddress)`

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The `TIMER_A_THRESHOLD` #define in the corresponding header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Returns

Majority vote of timer count value

`uint32_t Timer_A_getInterruptStatus (uint16_t baseAddress)`

Get timer interrupt status.

Parameters

<i>baseAddress</i>	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

`uint8_t Timer_A_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)`

Get output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	Valid values are: <ul style="list-style-type: none"> ■ 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**

`uint8_t Timer_A_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)`

Get synchronized capturecompare input.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	Valid values are: <ul style="list-style-type: none"> ■ 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

<i>synchronized</i>	Valid values are: <ul style="list-style-type: none"> ■ 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**

```
void Timer_A_initCaptureMode ( uint16_t baseAddress, Timer_A_initCaptureModeParam *  
param )
```

Initializes Capture Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for capture mode initialization.

Modified bits of **TAxCTLn** register.

Returns

None

References `Timer_A_initCaptureModeParam::captureInputSelect`,
`Timer_A_initCaptureModeParam::captureInterruptEnable`,
`Timer_A_initCaptureModeParam::captureMode`,
`Timer_A_initCaptureModeParam::captureOutputMode`,
`Timer_A_initCaptureModeParam::captureRegister`, and
`Timer_A_initCaptureModeParam::synchronizeCaptureSource`.

```
void Timer_A_initCompareMode ( uint16_t baseAddress, Timer_A_initCompareModeParam  
* param )
```

Initializes Compare Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for compare mode initialization.

Modified bits of **TAxCCRn** register and bits of **TAxCTLn** register.

Returns

None

References `Timer_A_initCompareModeParam::compareInterruptEnable`,
`Timer_A_initCompareModeParam::compareOutputMode`,
`Timer_A_initCompareModeParam::compareRegister`, and
`Timer_A_initCompareModeParam::compareValue`.


```
void Timer_A_initContinuousMode ( uint16_t baseAddress, Timer_A_initContinuous↔  
    ModeParam * param )
```

Configures Timer_A in continuous mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for continuous mode initialization.

Modified bits of **TAxCTL** register.

Returns

None

References Timer_A_initContinuousModeParam::clockSource, Timer_A_initContinuousModeParam::clockSourceDivider, Timer_A_initContinuousModeParam::startTimer, Timer_A_initContinuousModeParam::timerClear, and Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE.

```
void Timer_A_initUpDownMode ( uint16_t baseAddress, Timer_A_initUpDownModeParam
* param )
```

Configures Timer_A in up down mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for up-down mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_A_initUpDownModeParam::clockSource, Timer_A_initUpDownModeParam::clockSourceDivider, Timer_A_initUpDownModeParam::startTimer, Timer_A_initUpDownModeParam::timerClear, Timer_A_initUpDownModeParam::timerInterruptEnable_TAIE, and Timer_A_initUpDownModeParam::timerPeriod.

```
void Timer_A_initUpMode ( uint16_t baseAddress, Timer_A_initUpModeParam * param )
```

Configures Timer_A in up mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for up mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_A_initUpModeParam::clockSource, Timer_A_initUpModeParam::clockSourceDivider,

Timer_A_initUpModeParam::startTimer, Timer_A_initUpModeParam::timerClear,
 Timer_A_initUpModeParam::timerInterruptEnable_TAIE, and
 Timer_A_initUpModeParam::timerPeriod.

```
void Timer_A_outputPWM ( uint16_t baseAddress, Timer_A_outputPWMPParam * param )
```

Generate a PWM with timer running in up mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>param</i>	is the pointer to struct for PWM configuration.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register, bits of **TAxCCR0** register and bits of **TAxCCTLn** register.

Returns

None

References Timer_A_outputPWMPParam::clockSource,
 Timer_A_outputPWMPParam::clockSourceDivider,
 Timer_A_outputPWMPParam::compareOutputMode, Timer_A_outputPWMPParam::compareRegister,
 Timer_A_outputPWMPParam::dutyCycle, and Timer_A_outputPWMPParam::timerPeriod.

```
void Timer_A_setCompareValue ( uint16_t baseAddress, uint16_t compareRegister,  

  uint16_t compareValue )
```

Sets the value of the capture-compare register.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>compareRegister</i>	selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

<i>compareValue</i>	is the count to be compared with in compare mode
---------------------	--

Modified bits of **TAxCCRn** register.

Returns

None

```
void Timer_A_setOutputForOutputModeOutBitValue ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint8_t outputModeOutBitValue )
```

Set output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
<i>capture← Compare← Register</i>	Valid values are: <ul style="list-style-type: none"> ■ 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
<i>outputMode← OutBitValue</i>	is the value to be set for out bit Valid values are: <ul style="list-style-type: none"> ■ TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH ■ TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW

Modified bits of **TAxCCTLn** register.

Returns

None

```
void Timer_A_startCounter ( uint16_t baseAddress, uint16_t timerMode )
```

Starts Timer_A counter.

This function assumes that the timer has been previously configured using `Timer_A_initContinuousMode`, `Timer_A_initUpMode` or `Timer_A_initUpDownMode`.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

<i>timerMode</i>	mode to put the timer in Valid values are: <ul style="list-style-type: none"> ■ TIMER_A_STOP_MODE ■ TIMER_A_UP_MODE ■ TIMER_A_CONTINUOUS_MODE [Default] ■ TIMER_A_UPDOWN_MODE
------------------	---

Modified bits of **TAxCTL** register.

Returns

None

```
void Timer_A_stop ( uint16_t baseAddress )
```

Stops the timer.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_A module.
--------------------	--

Modified bits of **TAxCTL** register.

Returns

None

34.3 Programming Example

The following example shows some TIMER_A operations using the APIs

```
{ //Start TIMER_A
  Timer_A_initContinuousModeParam initContParam = {0};
  initContParam.clockSource = TIMER_A_CLOCKSOURCE_SMCLK;
  initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_DISABLE;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
  Timer_A_initContinuousMode(TIMER_A1_BASE, &initContParam);

  //Initiaze compare mode
  Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
    TIMER_A_CAPTURECOMPARE_REGISTER_0
  );

  Timer_A_initCompareModeParam initCompParam = {0};
  initCompParam.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_0;
  initCompParam.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE;
  initCompParam.compareOutputMode = TIMER_A_OUTPUTMODE_OUTBITVALUE;
  initCompParam.compareValue = COMPARE_VALUE;
  Timer_A_initCompareMode(TIMER_A1_BASE, &initCompParam);

  Timer_A_startCounter( TIMER_A1_BASE,
    TIMER_A_CONTINUOUS_MODE
  );

  //Enter LPM0
  _bis_SR_register(LPM0_bits);

  //For debugger
```

```
    __no_operation();  
}
```

35 16-Bit Timer_B (TIMER_B)

Introduction	366
API Functions	367
Programming Example	382

35.1 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.

35.2 API Functions

Functions

- void `Timer_B_startCounter` (uint16_t baseAddress, uint16_t timerMode)
Starts Timer_B counter.
- void `Timer_B_initContinuousMode` (uint16_t baseAddress, `Timer_B_initContinuousModeParam` *param)
Configures Timer_B in continuous mode.
- void `Timer_B_initUpMode` (uint16_t baseAddress, `Timer_B_initUpModeParam` *param)
Configures Timer_B in up mode.
- void `Timer_B_initUpDownMode` (uint16_t baseAddress, `Timer_B_initUpDownModeParam` *param)
Configures Timer_B in up down mode.
- void `Timer_B_initCaptureMode` (uint16_t baseAddress, `Timer_B_initCaptureModeParam` *param)
Initializes Capture Mode.
- void `Timer_B_initCompareMode` (uint16_t baseAddress, `Timer_B_initCompareModeParam` *param)
Initializes Compare Mode.
- void `Timer_B_enableInterrupt` (uint16_t baseAddress)
Enable Timer_B interrupt.
- void `Timer_B_disableInterrupt` (uint16_t baseAddress)
Disable Timer_B interrupt.
- uint32_t `Timer_B_getInterruptStatus` (uint16_t baseAddress)
Get Timer_B interrupt status.
- void `Timer_B_enableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Enable capture compare interrupt.
- void `Timer_B_disableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Disable capture compare interrupt.
- uint32_t `Timer_B_getCaptureCompareInterruptStatus` (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
Return capture compare interrupt status.
- void `Timer_B_clear` (uint16_t baseAddress)
Reset/Clear the Timer_B clock divider, count direction, count.
- uint8_t `Timer_B_getSynchronizedCaptureCompareInput` (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
Get synchronized capturecompare input.
- uint8_t `Timer_B_getOutputForOutputModeOutBitValue` (uint16_t baseAddress, uint16_t captureCompareRegister)
Get output bit for output mode.

- uint16_t `Timer_B.getCaptureCompareCount` (uint16_t baseAddress, uint16_t captureCompareRegister)
Get current capturecompare count.
- void `Timer_B.setOutputForOutputModeOutBitValue` (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
Set output bit for output mode.
- void `Timer_B.outputPWM` (uint16_t baseAddress, `Timer_B.outputPWMPParam` *param)
Generate a PWM with Timer_B running in up mode.
- void `Timer_B.stop` (uint16_t baseAddress)
Stops the Timer_B.
- void `Timer_B.setCompareValue` (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
Sets the value of the capture-compare register.
- void `Timer_B.clearTimerInterrupt` (uint16_t baseAddress)
Clears the Timer_B TBIFG interrupt flag.
- void `Timer_B.clearCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Clears the capture-compare interrupt flag.
- void `Timer_B.selectCounterLength` (uint16_t baseAddress, uint16_t counterLength)
Selects Timer_B counter length.
- void `Timer_B.selectLatchingGroup` (uint16_t baseAddress, uint16_t groupLatch)
Selects Timer_B Latching Group.
- void `Timer_B.initCompareLatchLoadEvent` (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
Selects Compare Latch Load Event.
- uint16_t `Timer_B.getCounterValue` (uint16_t baseAddress)
Reads the current timer count value.

35.2.1 Detailed Description

The TIMER_B API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

TIMER_B configuration and initialization is handled by

- `Timer_B.startCounter()`
- `Timer_B.initUpMode()`
- `Timer_B.initUpDownMode()`
- `Timer_B.initContinuousMode()`
- `Timer_B.initCapture()`
- `Timer_B.initCompare()`
- `Timer_B.clear()`
- `Timer_B.stop()`
- `Timer_B.initCompareLatchLoadEvent()`
- `Timer_B.selectLatchingGroup()`
- `Timer_B.selectCounterLength()`

TIMER_B outputs are handled by

- `Timer_B_getSynchronizedCaptureCompareInput()`
- `Timer_B_getOutputForOutputModeOutBitValue()`
- `Timer_B_setOutputForOutputModeOutBitValue()`
- `Timer_B_generatePWM()`
- `Timer_B_getCaptureCompareCount()`
- `Timer_B_setCompareValue()`
- `Timer_B_getCounterValue()`

The interrupt handler for the TIMER_B interrupt is managed with

- `Timer_B_enableInterrupt()`
- `Timer_B_disableInterrupt()`
- `Timer_B_getInterruptStatus()`
- `Timer_B_enableCaptureCompareInterrupt()`
- `Timer_B_disableCaptureCompareInterrupt()`
- `Timer_B_getCaptureCompareInterruptStatus()`
- `Timer_B_clearCaptureCompareInterrupt()`
- `Timer_B_clearTimerInterrupt()`

35.2.2 Function Documentation

`void Timer_B_clear (uint16_t baseAddress)`

Reset/Clear the Timer_B clock divider, count direction, count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

Modified bits of **TBxCTL** register.

Returns

None

`void Timer_B_clearCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)`

Clears the capture-compare interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

<i>capture← Compare← Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_B_clearTimerInterrupt ( uint16_t baseAddress )
```

Clears the Timer_B TBIFG interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

Modified bits are **TBIFG** of **TBxCTL** register.

Returns

None

```
void Timer_B_disableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t  
captureCompareRegister )
```

Disable capture compare interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>capture← Compare← Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_B_disableInterrupt ( uint16_t baseAddress )
```

Disable Timer_B interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

Modified bits of **TBxCTL** register.

Returns

None

```
void Timer_B_enableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Enable capture compare interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>capture↔ Compare↔ Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_B_enableInterrupt ( uint16_t baseAddress )
```

Enable Timer_B interrupt.

Enables Timer_B interrupt. Does not clear interrupt flags.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

Modified bits of **TBxCTL** register.

Returns

None

```
uint16_t Timer_B_getCaptureCompareCount ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Get current capturecompare count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>capture← Compare← Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
uint32_t Timer_B_getCaptureCompareInterruptStatus ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint16_t mask )
```

Return capture compare interrupt status.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>mask</i>	is the mask for the interrupt status Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`uint16_t Timer_B_getCounterValue (uint16_t baseAddress)`

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The `Timer_B_THRESHOLD` #define in the associated header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

<i>baseAddress</i>	is the base address of the Timer module.
--------------------	--

Returns

Majority vote of timer count value

`uint32_t Timer_B_getInterruptStatus (uint16_t baseAddress)`

Get Timer_B interrupt status.

Parameters

<i>baseAddress</i>	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

uint8_t Timer_B_getOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>captureCompareRegister</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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**

uint8_t Timer_B_getSynchronizedCaptureCompareInput (uint16_t *baseAddress*, uint16_t *captureCompareRegister*, uint16_t *synchronized*)

Get synchronized capturecompare input.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

<i>capture← Compare← Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>synchronized</i>	selects the type of capture compare input Valid values are: <ul style="list-style-type: none"> ■ 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**

```
void Timer_B_initCaptureMode ( uint16_t baseAddress, Timer_B_initCaptureModeParam *  
param )
```

Initializes Capture Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for capture mode initialization.

Modified bits of **TBxCCTLn** register.

Returns

None

References `Timer_B_initCaptureModeParam::captureInputSelect`,
`Timer_B_initCaptureModeParam::captureInterruptEnable`,
`Timer_B_initCaptureModeParam::captureMode`,
`Timer_B_initCaptureModeParam::captureOutputMode`,
`Timer_B_initCaptureModeParam::captureRegister`, and
`Timer_B_initCaptureModeParam::synchronizeCaptureSource`.

```
void Timer_B_initCompareLatchLoadEvent ( uint16_t baseAddress, uint16_t  
compareRegister, uint16_t compareLatchLoadEvent )
```

Selects Compare Latch Load Event.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>compare</i> ↔ <i>Register</i>	selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>compareLatch</i> ↔ <i>LoadEvent</i>	selects the latch load event Valid values are: <ul style="list-style-type: none"> ■ TIMER_B_LATCH_ON_WRITE_TO_TBxCCTLn_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_LAT↔ CH_VALUE

Modified bits are **CLLD** of **TBxCCTLn** register.

Returns

None

```
void Timer_B_initCompareMode ( uint16_t baseAddress, Timer_B_initCompareModeParam
* param )
```

Initializes Compare Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for compare mode initialization.

Modified bits of **TBxCCTLn** register and bits of **TBxCCTLn** register.

Returns

None

References `Timer_B_initCompareModeParam::compareInterruptEnable`,
`Timer_B_initCompareModeParam::compareOutputMode`,
`Timer_B_initCompareModeParam::compareRegister`, and
`Timer_B_initCompareModeParam::compareValue`.

```
void Timer_B_initContinuousMode ( uint16_t baseAddress, Timer_B_initContinuous↔  

ModeParam * param )
```

Configures Timer_B in continuous mode.

This API does not start the timer. Timer needs to be started when required using the `Timer_B_startCounter` API.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for continuous mode initialization.

Modified bits of **TBxCTL** register.

Returns

None

References `Timer_B_initContinuousModeParam::clockSource`,
`Timer_B_initContinuousModeParam::clockSourceDivider`,
`Timer_B_initContinuousModeParam::startTimer`, `Timer_B_initContinuousModeParam::timerClear`,
and `Timer_B_initContinuousModeParam::timerInterruptEnable_TBIE`.

```
void Timer_B_initUpDownMode ( uint16_t baseAddress, Timer_B_initUpDownModeParam  

* param )
```

Configures Timer_B in up down mode.

This API does not start the timer. Timer needs to be started when required using the `Timer_B_startCounter` API.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for up-down mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References `Timer_B_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE`,
`Timer_B_initUpDownModeParam::clockSource`,
`Timer_B_initUpDownModeParam::clockSourceDivider`,
`Timer_B_initUpDownModeParam::startTimer`, `Timer_B_initUpDownModeParam::timerClear`,

Timer_B_initUpDownModeParam::timerInterruptEnable_TBIE, and
Timer_B_initUpDownModeParam::timerPeriod.

```
void Timer_B_initUpMode ( uint16_t baseAddress, Timer_B_initUpModeParam * param )
```

Configures Timer_B in up mode.

This API does not start the timer. Timer needs to be started when required using the
Timer_B.startCounter API.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for up mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE,
Timer_B_initUpModeParam::clockSource, Timer_B_initUpModeParam::clockSourceDivider,
Timer_B_initUpModeParam::startTimer, Timer_B_initUpModeParam::timerClear,
Timer_B_initUpModeParam::timerInterruptEnable_TBIE, and
Timer_B_initUpModeParam::timerPeriod.

```
void Timer_B_outputPWM ( uint16_t baseAddress, Timer_B_outputPWMPParam * param )
```

Generate a PWM with Timer_B running in up mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>param</i>	is the pointer to struct for PWM configuration.

Modified bits of **TBxCCTLn** register, bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits
of **TBxCCR0** register.

Returns

None

References Timer_B_outputPWMPParam::clockSource,
Timer_B_outputPWMPParam::clockSourceDivider,
Timer_B_outputPWMPParam::compareOutputMode, Timer_B_outputPWMPParam::compareRegister,
Timer_B_outputPWMPParam::dutyCycle, and Timer_B_outputPWMPParam::timerPeriod.

```
void Timer_B_selectCounterLength ( uint16_t baseAddress, uint16_t counterLength )
```

Selects Timer_B counter length.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>counterLength</i>	selects the value of counter length. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_B_selectLatchingGroup ( uint16_t baseAddress, uint16_t groupLatch )
```

Selects Timer_B Latching Group.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>groupLatch</i>	selects the latching group. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_B_setCompareValue ( uint16_t baseAddress, uint16_t compareRegister,
                               uint16_t compareValue )
```

Sets the value of the capture-compare register.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>compareRegister</i>	selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>compareValue</i>	is the count to be compared with in compare mode

Modified bits of **TBxCCRn** register.

Returns

None

```
void Timer_B_setOutputForOutputModeOutBitValue ( uint16_t baseAddress, uint16_t
                                                    captureCompareRegister, uint8_t outputModeOutBitValue )
```

Set output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>capture← Compare← Register</i>	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>outputMode← OutBitValue</i>	the value to be set for out bit Valid values are: <ul style="list-style-type: none"> ■ TIMER_B_OUTPUTMODE_OUTBITVALUE_HIGH ■ TIMER_B_OUTPUTMODE_OUTBITVALUE_LOW

Modified bits of **TBxCCTLn** register.

Returns

None

```
void Timer_B_startCounter ( uint16_t baseAddress, uint16_t timerMode )
```

Starts Timer_B counter.

This function assumes that the timer has been previously configured using `Timer_B_initContinuousMode`, `Timer_B_initUpMode` or `Timer_B_initUpDownMode`.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
<i>timerMode</i>	selects the mode of the timer Valid values are: <ul style="list-style-type: none"> ■ TIMER_B_STOP_MODE ■ TIMER_B_UP_MODE ■ TIMER_B_CONTINUOUS_MODE [Default] ■ TIMER_B_UPDOWN_MODE

Modified bits of **TBxCTL** register.

Returns

None

```
void Timer_B_stop ( uint16_t baseAddress )
```

Stops the Timer_B.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_B module.
--------------------	--

Modified bits of **TBxCTL** register.

Returns

None

35.3 Programming Example

The following example shows some TIMER_B operations using the APIs

```

{ //Start timer in continuous mode sourced by SMCLK
  Timer_B_initContinuousModeParam initContParam = {0};
  initContParam.clockSource = TIMER_B_CLOCKSOURCE_SMCLK;
  initContParam.clockSourceDivider = TIMER_B_CLOCKSOURCE_DIVIDER_1;
  initContParam.timerInterruptEnable_TBIE = TIMER_B_TBIE_INTERRUPT_DISABLE;
  initContParam.timerClear = TIMER_B_DO_CLEAR;
  initContParam.startTimer = false;
  Timer_B_initContinuousMode(TIMER_B0_BASE, &initContParam);

  //Initiaze compare mode
  Timer_B_clearCaptureCompareInterrupt(TIMER_B0_BASE,
    TIMER_B_CAPTURECOMPARE_REGISTER_0);

  Timer_B_initCompareModeParam initCompParam = {0};
  initCompParam.compareRegister = TIMER_B_CAPTURECOMPARE_REGISTER_0;
  initCompParam.compareInterruptEnable = TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE;
  initCompParam.compareOutputMode = TIMER_B_OUTPUTMODE_OUTBITVALUE;
  initCompParam.compareValue = COMPARE_VALUE;
  Timer_B_initCompareMode(TIMER_B0_BASE, &initCompParam);

  Timer_B_startCounter( TIMER_B0_BASE,
    TIMER_B_CONTINUOUS_MODE
  );
}

```

36 TIMER_D

Introduction	383
API Functions	384
Programming Example	409

36.1 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 TDCCR_x 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.

36.2 API Functions

Functions

- void `Timer_D_startCounter` (uint16_t baseAddress, uint16_t timerMode)
Starts Timer_D counter.
- void `Timer_D_initContinuousMode` (uint16_t baseAddress, `Timer_D_initContinuousModeParam` *param)
Configures timer in continuous mode.
- void `Timer_D_initUpMode` (uint16_t baseAddress, `Timer_D_initUpModeParam` *param)
Configures timer in up mode.
- void `Timer_D_initUpDownMode` (uint16_t baseAddress, `Timer_D_initUpDownModeParam` *param)
Configures timer in up down mode.
- void `Timer_D_initCaptureMode` (uint16_t baseAddress, `Timer_D_initCaptureModeParam` *param)
Initializes Capture Mode.
- void `Timer_D_initCompareMode` (uint16_t baseAddress, `Timer_D_initCompareModeParam` *param)
Initializes Compare Mode.
- void `Timer_D_enableTimerInterrupt` (uint16_t baseAddress)
Enable timer interrupt.
- void `Timer_D_enableHighResInterrupt` (uint16_t baseAddress, uint16_t mask)
Enable High Resolution interrupt.
- void `Timer_D_disableTimerInterrupt` (uint16_t baseAddress)
Disable timer interrupt.
- void `Timer_D_disableHighResInterrupt` (uint16_t baseAddress, uint16_t mask)
Disable High Resolution interrupt.
- uint32_t `Timer_D_getTimerInterruptStatus` (uint16_t baseAddress)
Get timer interrupt status.
- void `Timer_D_enableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)
Enable capture compare interrupt.
- void `Timer_D_disableCaptureCompareInterrupt` (uint16_t baseAddress, uint16_t captureCompareRegister)

- Disable capture compare interrupt.*

 - uint32_t [Timer_D.getCaptureCompareInterruptStatus](#) (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)

Return capture compare interrupt status.
- uint16_t [Timer_D.getHighResInterruptStatus](#) (uint16_t baseAddress, uint16_t mask)

Returns High Resolution interrupt status.
- void [Timer_D.clear](#) (uint16_t baseAddress)

Reset/Clear the timer clock divider, count direction, count.
- void [Timer_D.clearHighResInterrupt](#) (uint16_t baseAddress, uint16_t mask)

Clears High Resolution interrupt status.
- uint8_t [Timer_D.getSynchronizedCaptureCompareInput](#) (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)

Get synchronized capturecompare input.
- uint8_t [Timer_D.getOutputForOutputModeOutBitValue](#) (uint16_t baseAddress, uint16_t captureCompareRegister)

Get output bit for output mode.
- uint16_t [Timer_D.getCaptureCompareCount](#) (uint16_t baseAddress, uint16_t captureCompareRegister)

Get current capturecompare count.
- uint16_t [Timer_D.getCaptureCompareLatchCount](#) (uint16_t baseAddress, uint16_t captureCompareRegister)

Get current capture compare latch register count.
- uint8_t [Timer_D.getCaptureCompareInputSignal](#) (uint16_t baseAddress, uint16_t captureCompareRegister)

Get current capturecompare input signal.
- void [Timer_D.setOutputForOutputModeOutBitValue](#) (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)

Set output bit for output mode.
- void [Timer_D.outputPWM](#) (uint16_t baseAddress, [Timer_D.outputPWMPParam](#) *param)

Generate a PWM with timer running in up mode.
- void [Timer_D.stop](#) (uint16_t baseAddress)

Stops the timer.
- void [Timer_D.setCompareValue](#) (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)

Sets the value of the capture-compare register.
- void [Timer_D.clearTimerInterrupt](#) (uint16_t baseAddress)

Clears the Timer TDIFG interrupt flag.
- void [Timer_D.clearCaptureCompareInterrupt](#) (uint16_t baseAddress, uint16_t captureCompareRegister)

Clears the capture-compare interrupt flag.
- uint8_t [Timer_D.initHighResGeneratorInFreeRunningMode](#) (uint16_t baseAddress, uint8_t desiredHighResFrequency)

Configures Timer_D in free running mode.
- void [Timer_D.initHighResGeneratorInRegulatedMode](#) (uint16_t baseAddress, [Timer_D.initHighResGeneratorInRegulatedModeParam](#) *param)

Configures Timer_D in Regulated mode.
- void [Timer_D.combineTDCCRToOutputPWM](#) (uint16_t baseAddress, [Timer_D.combineTDCCRToOutputPWMPParam](#) *param)

Combine TDCCR to get PWM.
- void [Timer_D.selectLatchingGroup](#) (uint16_t baseAddress, uint16_t groupLatch)

Selects Timer_D Latching Group.
- void [Timer_D.selectCounterLength](#) (uint16_t baseAddress, uint16_t counterLength)

Selects Timer_D counter length.

- void `Timer_D_initCompareLatchLoadEvent` (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
Selects Compare Latch Load Event.
- void `Timer_D_disableHighResFastWakeup` (uint16_t baseAddress)
Disable High Resolution fast wakeup.
- void `Timer_D_enableHighResFastWakeup` (uint16_t baseAddress)
Enable High Resolution fast wakeup.
- void `Timer_D_disableHighResClockEnhancedAccuracy` (uint16_t baseAddress)
Disable High Resolution Clock Enhanced Accuracy.
- void `Timer_D_enableHighResClockEnhancedAccuracy` (uint16_t baseAddress)
Enable High Resolution Clock Enhanced Accuracy.
- void `Timer_D_disableHighResGeneratorForceON` (uint16_t baseAddress)
Disable High Resolution Clock Enhanced Accuracy.
- void `Timer_D_enableHighResGeneratorForceON` (uint16_t baseAddress)
Enable High Resolution Clock Enhanced Accuracy.
- void `Timer_D_selectHighResCoarseClockRange` (uint16_t baseAddress, uint16_t highResCoarseClockRange)
Select High Resolution Coarse Clock Range.
- void `Timer_D_selectHighResClockRange` (uint16_t baseAddress, uint16_t highResClockRange)
Select High Resolution Clock Range Selection.
- uint16_t `Timer_D_getCounterValue` (uint16_t baseAddress)
Reads the current timer count value.

36.2.1 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_initContinuousMode()`,
- `Timer_D_initUpMode()`,
- `Timer_D_initUpDownMode()`,
- `Timer_D_initCaptureMode()`,
- `Timer_D_initCompareMode()`,
- `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.outputPWM()`,
- `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.clearCaptureCompareInterrupt()`
- `Timer_D.clearTimerInterrupt()`,
- `Timer_D.enableHighResInterrupt()`,
- `Timer_D.disableHighResInterrupt()`,
- `Timer_D.getHighResInterruptStatus()`,
- `Timer_D.clearHighResInterrupt()`

Timer_D High Resolution handling APIs

- `Timer_D.getHighResInterruptStatus()`,
- `Timer_D.clearHighResInterrupt()`,
- `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()`

36.2.2 Function Documentation

void Timer_D_clear (uint16_t *baseAddress*)

Reset/Clear the timer clock divider, count direction, count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits of **TDxCTL0** register.

Returns

None

```
void Timer_D_clearCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Clears the capture-compare interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture← Compare← Register</i>	selects the Capture-compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

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

Clears High Resolution interrupt status.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>mask</i>	is the mask for the interrupts to clear Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_clearTimerInterrupt ( uint16_t baseAddress )
```

Clears the Timer TDIFG interrupt flag.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits are **TDIFG** of **TDxCTL0** register.

Returns

None

```
void Timer_D_combineTDCCRToOutputPWM ( uint16_t baseAddress,
Timer_D_combineTDCCRToOutputPWMPParam * param )
```

Combine TDCCR to get PWM.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>param</i>	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

References `Timer_D_combineTDCCRToOutputPWMPParam::clockingMode`,
`Timer_D_combineTDCCRToOutputPWMPParam::clockSource`,
`Timer_D_combineTDCCRToOutputPWMPParam::clockSourceDivider`,
`Timer_D_combineTDCCRToOutputPWMPParam::combineCCRRegistersCombination`,
`Timer_D_combineTDCCRToOutputPWMPParam::compareOutputMode`,
`Timer_D_combineTDCCRToOutputPWMPParam::dutyCycle1`,
`Timer_D_combineTDCCRToOutputPWMPParam::dutyCycle2`, and
`Timer_D_combineTDCCRToOutputPWMPParam::timerPeriod`.

```
void Timer_D_disableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Disable capture compare interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_disableHighResClockEnhancedAccuracy ( uint16_t baseAddress )
```

Disable High Resolution Clock Enhanced Accuracy.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
--------------------	--

Modified bits are **TDHEAEN** of **TDxHCTL0** register.

Returns

None

```
void Timer_D_disableHighResFastWakeup ( uint16_t baseAddress )
```

Disable High Resolution fast wakeup.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
--------------------	--

Modified bits are **TDHFW** of **TDxHCTL0** register.

Returns

None

```
void Timer_D_disableHighResGeneratorForceON ( uint16_t baseAddress )
```

Disable High Resolution Clock Enhanced Accuracy.

High-resolution generator is on if the Timer_D counter

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits are **TDHRON** of **TDxHCTL0** register.

Returns

None

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

Disable High Resolution interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>mask</i>	is the mask of interrupts to disable Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_disableTimerInterrupt ( uint16_t baseAddress )
```

Disable timer interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits of **TDxCTL0** register.

Returns

None

```
void Timer_D_enableCaptureCompareInterrupt ( uint16_t baseAddress, uint16_t captureCompareRegister )
```

Enable capture compare interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_enableHighResClockEnhancedAccuracy ( uint16_t baseAddress )
```

Enable High Resolution Clock Enhanced Accuracy.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits are **TDHEAEN** of **TDxHCTL0** register.

Returns

None

```
void Timer_D_enableHighResFastWakeup ( uint16_t baseAddress )
```

Enable High Resolution fast wakeup.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits are **TDHFW** of **TDxHCTL0** register.

Returns

None

```
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

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits are **TDHRON** of **TDxHCTL0** register.

Returns

None

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

Enable High Resolution interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>mask</i>	is the mask of interrupts to enable Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_enableTimerInterrupt ( uint16_t baseAddress )
```

Enable timer interrupt.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	--

Modified bits of **TDxCTL0** register.

Returns

None

```
uint16_t Timer_D_getCaptureCompareCount ( uint16_t baseAddress, uint16_t captureCompareRegister )
```

Get current capturecompare count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
uint8_t Timer_D_getCaptureCompareInputSignal ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Get current capturecompare input signal.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

```
uint32_t Timer_D_getCaptureCompareInterruptStatus ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint16_t mask )
```

Return capture compare interrupt status.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	is the selected capture compare register Valid values are: <ul style="list-style-type: none"> ■ 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
<i>mask</i>	is the mask for the interrupt status Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
uint16_t Timer_D_getCaptureCompareLatchCount ( uint16_t baseAddress, uint16_t
captureCompareRegister )
```

Get current capture compare latch register count.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

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

<i>baseAddress</i>	is the base address of the <code>TIMER_D</code> module.
--------------------	---

Returns

Majority vote of timer count value

uint16_t Timer_D_getHighResInterruptStatus (uint16_t *baseAddress*, uint16_t *mask*)

Returns High Resolution interrupt status.

Parameters

<i>baseAddress</i>	is the base address of the <code>TIMER_D</code> module.
<i>mask</i>	is the mask for the interrupt status Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ <code>TIMER_D_HIGH_RES_FREQUENCY_UNLOCK</code> ■ <code>TIMER_D_HIGH_RES_FREQUENCY_LOCK</code> ■ <code>TIMER_D_HIGH_RES_FAIL_HIGH</code> ■ <code>TIMER_D_HIGH_RES_FAIL_LOW</code>

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

uint8_t Timer_D_getOutputForOutputModeOutBitValue (uint16_t *baseAddress*, uint16_t *captureCompareRegister*)

Get output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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.OUTPUTVALUE.HIGH**
- **Timer.D.OUTPUTMODE.OUTPUTVALUE.LOW**

```
uint8_t Timer_D_getSynchronizedCaptureCompareInput ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint16_t synchronized )
```

Get synchronized capturecompare input.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>capture</i> ↔ <i>Compare</i> ↔ <i>Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

<i>synchronized</i>	is to select type of capture compare input. Valid values are: <ul style="list-style-type: none"> ■ 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**

```
uint32_t Timer_D_getTimerInterruptStatus ( uint16_t baseAddress )
```

Get timer interrupt status.

Parameters

<i>baseAddress</i>	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

```
void Timer_D_initCaptureMode ( uint16_t baseAddress, Timer_D_initCaptureModeParam  
* param )
```

Initializes Capture Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>param</i>	is the pointer to struct for capture mode initialization.

Modified bits of **TDxCCTLn** register and bits of **TDxCTL2** register.

Returns

None

References `Timer_D_initCaptureModeParam::captureInputSelect`,
`Timer_D_initCaptureModeParam::captureInterruptEnable`,
`Timer_D_initCaptureModeParam::captureMode`,
`Timer_D_initCaptureModeParam::captureOutputMode`,
`Timer_D_initCaptureModeParam::captureRegister`,
`Timer_D_initCaptureModeParam::channelCaptureMode`, and
`Timer_D_initCaptureModeParam::synchronizeCaptureSource`.

```
void Timer_D_initCompareLatchLoadEvent ( uint16_t baseAddress, uint16_t  
    compareRegister, uint16_t compareLatchLoadEvent )
```

Selects Compare Latch Load Event.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>compare</i> ↔ <i>Register</i>	selects the compare register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>compareLatch</i> ↔ <i>LoadEvent</i>	selects the latch load event Valid values are: <ul style="list-style-type: none"> ■ TIMER_D_LATCH_ON_WRITE_TO_TDxCCTLn_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_LAT↔ CH_VALUE

Modified bits are **CLLD** of **TDxCCTLn** register.

Returns

None

```
void Timer_D_initCompareMode ( uint16_t baseAddress, Timer_D_initCompareMode↔  
Param * param )
```

Initializes Compare Mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>param</i>	is the pointer to struct for compare mode initialization.

Modified bits of **TDxCCTLn** register and bits of **TDxCCTLn** register.

Returns

None

References `Timer_D.initCompareModeParam::compareInterruptEnable`,
`Timer_D.initCompareModeParam::compareOutputMode`,
`Timer_D.initCompareModeParam::compareRegister`, and
`Timer_D.initCompareModeParam::compareValue`.

```
void Timer_D_initContinuousMode ( uint16_t baseAddress, Timer_D_initContinuous↔  

ModeParam * 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

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>param</i>	is the pointer to struct for continuous mode initialization.

Modified bits of **TDxCTL0** register and bits of **TDxCTL1** register.

Returns

None

References `Timer_D.initContinuousModeParam::clockingMode`,
`Timer_D.initContinuousModeParam::clockSource`,
`Timer_D.initContinuousModeParam::clockSourceDivider`,
`Timer_D.initContinuousModeParam::timerClear`, and
`Timer_D.initContinuousModeParam::timerInterruptEnable_TDIE`.

```
uint8_t Timer_D_initHighResGeneratorInFreeRunningMode ( uint16_t baseAddress, uint8_t  

desiredHighResFrequency )
```

Configures `Timer_D` in free running mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>desiredHigh</i> ↔ <i>ResFrequency</i>	selects the desired High Resolution frequency used. Valid values are: <ul style="list-style-type: none"> ■ 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

References TLV_getInfo().

```
void Timer_D_initHighResGeneratorInRegulatedMode ( uint16_t baseAddress,
Timer_D_initHighResGeneratorInRegulatedModeParam * param )
```

Configures Timer_D in Regulated mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>param</i>	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

References `Timer_D_initHighResGeneratorInRegulatedModeParam::clockingMode`,
`Timer_D_initHighResGeneratorInRegulatedModeParam::clockSource`,
`Timer_D_initHighResGeneratorInRegulatedModeParam::clockSourceDivider`,
`Timer_D_initHighResGeneratorInRegulatedModeParam::highResClockDivider`, and
`Timer_D_initHighResGeneratorInRegulatedModeParam::highResClockMultiplyFactor`.

```
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

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>param</i>	is the pointer to struct for up-down mode initialization.

Modified bits of **TDxCCR0** register, bits of **TDxCCCTL0** register, bits of **TDxCTL0** register and bits of **TDxCTL1** register.**Returns**

None

References `Timer_D_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE`,
`Timer_D_initUpDownModeParam::clockingMode`, `Timer_D_initUpDownModeParam::clockSource`,
`Timer_D_initUpDownModeParam::clockSourceDivider`,
`Timer_D_initUpDownModeParam::timerClear`,
`Timer_D_initUpDownModeParam::timerInterruptEnable_TDIE`, and
`Timer_D_initUpDownModeParam::timerPeriod`.

```
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

<i>baseAddress</i>	is the base address of the <code>TIMER_D</code> module.
<i>param</i>	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

References `Timer_D_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE`, `Timer_D_initUpModeParam::clockingMode`, `Timer_D_initUpModeParam::clockSource`, `Timer_D_initUpModeParam::clockSourceDivider`, `Timer_D_initUpModeParam::timerClear`, `Timer_D_initUpModeParam::timerInterruptEnable_TDIE`, and `Timer_D_initUpModeParam::timerPeriod`.

```
void Timer_D_outputPWM ( uint16_t baseAddress, Timer_D_outputPWMPParam * param )
```

Generate a PWM with timer running in up mode.

Parameters

<i>baseAddress</i>	is the base address of the <code>TIMER_D</code> module.
<i>param</i>	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

References `Timer_D_outputPWMPParam::clockingMode`, `Timer_D_outputPWMPParam::clockSource`, `Timer_D_outputPWMPParam::clockSourceDivider`, `Timer_D_outputPWMPParam::compareOutputMode`, `Timer_D_outputPWMPParam::compareRegister`, `Timer_D_outputPWMPParam::dutyCycle`, and `Timer_D_outputPWMPParam::timerPeriod`.

```
void Timer_D_selectCounterLength ( uint16_t baseAddress, uint16_t counterLength )
```

Selects `Timer_D` counter length.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>counterLength</i>	selects the value of counter length. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_selectHighResClockRange ( uint16_t baseAddress, uint16_t
highResClockRange )
```

Select High Resolution Clock Range Selection.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>highResClock↔ Range</i>	selects the High Resolution Clock Range. Refer to datasheet for frequency details Valid values are: <ul style="list-style-type: none"> ■ TIMER_D_CLOCK_RANGE0 [Default] ■ TIMER_D_CLOCK_RANGE1 ■ TIMER_D_CLOCK_RANGE2

Returns

None

```
void Timer_D_selectHighResCoarseClockRange ( uint16_t baseAddress, uint16_t
highResCoarseClockRange )
```

Select High Resolution Coarse Clock Range.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
<i>highRes↔ CoarseClock↔ Range</i>	selects the High Resolution Coarse Clock Range Valid values are: <ul style="list-style-type: none"> ■ TIMER_D_HIGHRES_BELOW_15MHz [Default] ■ TIMER_D_HIGHRES_ABOVE_15MHz

Modified bits are **TDHCLKCR** of **TDxHCTL1** register.

Returns

None

```
void Timer_D_selectLatchingGroup ( uint16_t baseAddress, uint16_t groupLatch )
```

Selects Timer_D Latching Group.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>groupLatch</i>	selects the group latch Valid values are: <ul style="list-style-type: none"> ■ 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

```
void Timer_D_setCompareValue ( uint16_t baseAddress, uint16_t compareRegister,  
uint16_t compareValue )
```

Sets the value of the capture-compare register.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>compareRegister</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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

<i>compareValue</i>	is the count to be compared with in compare mode
---------------------	--

Modified bits of **TDxCCRn** register.

Returns

None

```
void Timer_D_setOutputForOutputModeOutBitValue ( uint16_t baseAddress, uint16_t
captureCompareRegister, uint8_t outputModeOutBitValue )
```

Set output bit for output mode.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_D module.
<i>capture← Compare← Register</i>	selects the Capture register being used. Valid values are: <ul style="list-style-type: none"> ■ 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
<i>outputMode← OutBitValue</i>	the value to be set for out bit Valid values are: <ul style="list-style-type: none"> ■ TIMER_D_OUTPUTMODE_OUTBITVALUE_HIGH ■ TIMER_D_OUTPUTMODE_OUTBITVALUE_LOW

Modified bits of **TDxCCTLn** register.

Returns

None

```
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_initContinuousMode`, `Timer_D_initUpMode` or `Timer_D_initUpDownMode`.

Parameters

<i>baseAddress</i>	is the base address of the TIMER_DA module.
--------------------	---

<i>timerMode</i>	selects the mode of the timer Valid values are: <ul style="list-style-type: none"> ■ TIMER_D_STOP_MODE ■ TIMER_D_UP_MODE ■ TIMER_D_CONTINUOUS_MODE [Default] ■ TIMER_D_UPDOWN_MODE
------------------	--

Modified bits of **TDxCTL0** register.

Returns

None

```
void Timer_D_stop ( uint16_t baseAddress )
```

Stops the timer.

Parameters

<i>baseAddress</i>	is the base address of the TIMER.D module.
--------------------	---

Modified bits of **TDxCTL0** register.

Returns

None

36.3 Programming Example

The following example shows some TimerD operations using the APIs

```
{ //Start TimerD
//Start timer in continuous mode sourced by SMCLK
Timer_D_initContinuousModeParam initContparam = {0};
initContparam.clockSource = TIMER_D_CLOCKSOURCE_SMCLK;
initContparam.clockSourceDivider = TIMER_D_CLOCKSOURCE_DIVIDER_1;
initContparam.clockingMode = TIMER_D_CLOCKINGMODE_EXTERNAL_CLOCK;
initContparam.timerInterruptEnable_TDIE = TIMER_D_TDIE_INTERRUPT_DISABLE;
initContparam.timerClear = TIMER_D_DO_CLEAR;
Timer_D_initContinuousMode(TIMER_D0_BASE, &initContparam);

Timer_D_startCounter(TIMER_D0_BASE,
    TIMER_D_CONTINUOUS_MODE
);

//Initiaze compare mode
Timer_D_clearCaptureCompareInterrupt(TIMER_D0_BASE,
    TIMER_D_CAPTURECOMPARE_REGISTER_0);

Timer_D_initCompareModeParam initCompParam = {0};
initCompParam.compareRegister = TIMER_D_CAPTURECOMPARE_REGISTER_0;
initCompParam.compareInterruptEnable = TIMER_D_CAPTURECOMPARE_INTERRUPT_ENABLE;
initCompParam.compareOutputMode = TIMER_D_OUTPUTMODE_OUTBITVALUE;
initCompParam.compareValue = 50000;
Timer_D_initCompareMode(TIMER_D0_BASE, &initCompParam);

//Enter LPM0
_bis_SR_register(LPM0_bits);

//For debugger
```

```
    _no_operation();  
}
```

37 Tag Length Value

Introduction	411
API Functions	411
Programming Example	416

37.1 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

37.2 API Functions

Functions

- void [TLV_getInfo](#) (uint8_t tag, uint8_t instance, uint8_t *length, uint16_t **data_address)
Gets TLV Info.
- uint16_t [TLV_getDeviceType](#) ()
Retrieves the unique device ID from the TLV structure.
- uint16_t [TLV_getMemory](#) (uint8_t instance)
Gets memory information.
- uint16_t [TLV_getPeripheral](#) (uint8_t tag, uint8_t instance)
Gets peripheral information from the TLV.
- uint8_t [TLV_getInterrupt](#) (uint8_t tag)
Get interrupt information from the TLV.

37.2.1 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 if the specified interrupt vector is not defined.

37.2.2 Function Documentation

`uint16_t TLV_getDeviceType (void)`

Retrieves the unique device ID from the TLV structure.

Returns

The device ID is returned as type `uint16_t`.

`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.

Parameters

<i>tag</i>	<p>represents the tag for which the information needs to be retrieved. Valid values are:</p> <ul style="list-style-type: none"> ■ TLV_TAG_LDRTAG ■ 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
------------	--

<i>instance</i>	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.
<i>length</i>	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.
<i>data_address</i>	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

Referenced by Timer_D_initHighResGeneratorInFreeRunningMode(), TLV_getInterrupt(), TLV_getMemory(), and TLV_getPeripheral().

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.

Parameters

<i>tag</i>	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 if the specified interrupt vector is not defined.

References TLV_getInfo(), and TLV_getMemory().

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.

Parameters

<i>instance</i>	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.

References TLV_getInfo().

Referenced by TLV_getInterrupt(), and TLV_getPeripheral().

uint16_t TLV_getPeripheral (uint8_t *tag*, uint8_t *instance*)

Gets peripheral information from the TLV.

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 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. <

Parameters

<i>tag</i>	<p>represents represents the tag for a specific peripheral for which the information needs to be retrieved. In the header file <code>tlv.h</code> specific peripheral tags are pre-defined, for example <code>USCIA_B</code> and <code>TA0</code> are defined as <code>TLV_PID_USCIA_B</code> and <code>TLV_PID_TA2</code> respectively. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
------------	---

Returns

The returned value is zero if the specified tag value (peripheral) is not available in the device.

References TLV_getInfo(), and TLV_getMemory().

37.3 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  
            );
```

38 Unified Clock System (UCS)

Introduction	417
API Functions	418
Programming Example	432

38.1 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 $\pm 3\%$.
- 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 ($\sim 3 \mu\text{A}$ active) versus the LFXT1 ($\sim 300 \text{ nA}$ active).
- Modules (WDT.A) In watchdog mode, when SMCLK or ACLK fails, the clock source defaults to the VLOCLK.

38.2 API Functions

Macros

- #define **CC430_DEVICE**
- #define **NOT_CC430_DEVICE**

Functions

- void **UCS_setExternalClockSource** (uint32_t XT1CLK_frequency, uint32_t XT2CLK_frequency)
Sets the external clock source.
- void **UCS_initClockSignal** (uint8_t selectedClockSignal, uint16_t clockSource, uint16_t clockSourceDivider)
Initializes a clock signal.
- void **UCS_turnOnLFXT1** (uint16_t xt1drive, uint8_t xcap)
Initializes the XT1 crystal oscillator in low frequency mode.
- void **UCS_turnOnHFXT1** (uint16_t xt1drive)
Initializes the XT1 crystal oscillator in high frequency mode.
- void **UCS_bypassXT1** (uint8_t highOrLowFrequency)
Bypass the XT1 crystal oscillator.
- bool **UCS_turnOnLFXT1WithTimeout** (uint16_t xt1drive, uint8_t xcap, uint16_t timeout)
Initializes the XT1 crystal oscillator in low frequency mode with timeout.
- bool **UCS_turnOnHFXT1WithTimeout** (uint16_t xt1drive, uint16_t timeout)
Initializes the XT1 crystal oscillator in high frequency mode with timeout.
- bool **UCS_bypassXT1WithTimeout** (uint8_t highOrLowFrequency, uint16_t timeout)
Bypasses the XT1 crystal oscillator with time out.
- void **UCS_turnOffXT1** (void)
Stops the XT1 oscillator using the XT1OFF bit.
- void **UCS_turnOnXT2** (uint16_t xt2drive)
Initializes the XT2 crystal oscillator.
- void **UCS_bypassXT2** (void)
Bypasses the XT2 crystal oscillator.
- bool **UCS_turnOnXT2WithTimeout** (uint16_t xt2drive, uint16_t timeout)
Initializes the XT2 crystal oscillator with timeout.

- `bool UCS.bypassXT2WithTimeout` (uint16_t timeout)
Bypasses the XT2 crystal oscillator with timeout.
- `void UCS.turnOffXT2` (void)
Stops the XT2 oscillator using the XT2OFF bit.
- `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.
- `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.
- `void UCS.enableClockRequest` (uint8_t selectClock)
Enables conditional module requests.
- `void UCS.disableClockRequest` (uint8_t selectClock)
Disables conditional module requests.
- `uint8_t UCS.getFaultFlagStatus` (uint8_t mask)
Gets the current UCS fault flag status.
- `void UCS.clearFaultFlag` (uint8_t mask)
Clears the current UCS fault flag status for the masked bit.
- `void UCS.turnOffSMCLK` (void)
Turns off SMCLK using the SMCLKOFF bit.
- `void UCS.turnOnSMCLK` (void)
Turns ON SMCLK using the SMCLKOFF bit.
- `uint32_t UCS.getACLK` (void)
Get the current ACLK frequency.
- `uint32_t UCS.getSMCLK` (void)
Get the current SMCLK frequency.
- `uint32_t UCS.getMCLK` (void)
Get the current MCLK frequency.
- `uint16_t UCS.clearAllOscFlagsWithTimeout` (uint16_t timeout)
Clears all the Oscillator Flags.

38.2.1 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.initClockSignal()`,
- `UCS.initFLLSettle()`,
- `UCS.enableClockRequest()`,
- `UCS.disableClockRequest()`,
- `UCS.turnOffSMCLK()`,
- `UCS.turnOnSMCLK()`

External crystal specific configuration and initialization is handled by

- `UCS.setExternalClockSource()`,
- `UCS.turnOnLFXT1()`,
- `UCS.turnOnHFXT1()`,
- `UCS.bypassXT1()`,

- `UCS_turnOnLFXT1WithTimeout()`,
- `UCS_turnOnHFXT1WithTimeout()`,
- `UCS_bypassXT1WithTimeout()`,
- `UCS_turnOffXT1()`,
- `UCS_turnOnXT2()`,
- `UCS_turnOffXT2()`,
- `UCS_bypassXT2()`,
- `UCS_turnOnXT2WithTimeout()`,
- `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_initClockSignal()` 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_getFaultFlagStatus()`,
- `UCS_clearFaultFlag()`,
- `UCS_getACLK()`,
- `UCS_getSMCLK()`,
- `UCS_getMCLK()`

38.2.2 Macro Definition Documentation

`#define CC430_DEVICE`

Value:

```
(defined (__CC430F5133__) || defined(__CC430F5135__) || defined(__CC430F5137__) || \
 defined(__CC430F6125__) || defined(__CC430F6126__) || defined(__CC430F6127__) || \
 defined(__CC430F6135__) || defined(__CC430F6137__) || defined(__CC430F5123__) || \
 defined(__CC430F5125__) || defined(__CC430F5143__) || defined(__CC430F5145__) || \
 defined(__CC430F5147__) || defined(__CC430F6143__) || defined(__CC430F6145__) || \
 defined(__CC430F6147__))
```

`#define NOT_CC430_DEVICE`

Value:

```
(!defined (__CC430F5133__) && !defined(__CC430F5135__) && !defined(__CC430F5137__) && \
 !defined(__CC430F6125__) && !defined(__CC430F6126__) && !defined(__CC430F6127__) && \
 !defined(__CC430F6135__) && !defined(__CC430F6137__) && !defined(__CC430F5123__) && \
 !defined(__CC430F5125__) && !defined(__CC430F5143__) && !defined(__CC430F5145__) && \
 !defined(__CC430F5147__) && !defined(__CC430F6143__) && !defined(__CC430F6145__) && \
 !defined(__CC430F6147__))
```

38.2.3 Function Documentation

```
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.

Parameters

<i>highOrLow↔ Frequency</i>	selects high frequency or low frequency mode for XT1. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>highOrLow↔ Frequency</i>	selects high frequency or low frequency mode for XT1. Valid values are: <ul style="list-style-type: none"> ■ UCS_XT1_HIGH_FREQUENCY ■ UCS_XT1_LOW_FREQUENCY [Default]
<i>timeout</i>	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

```
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.

Returns

None

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.

Parameters

<i>timeout</i>	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

uint16_t UCS_clearAllOscFlagsWithTimeout (uint16_t *timeout*)

Clears all the Oscillator Flags.

Parameters

<i>timeout</i>	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

void UCS_clearFaultFlag (uint8_t *mask*)

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

Parameters

<i>mask</i>	is the masked interrupt flag status to be returned. mask parameter can be any one of the following Valid values are: <ul style="list-style-type: none"> ■ 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

```
void UCS_disableClockRequest ( uint8_t selectClock )
```

Disables conditional module requests.

Parameters

<i>selectClock</i>	selects specific request disable Valid values are: <ul style="list-style-type: none"> ■ UCS_ACLK ■ UCS_SMCLK ■ UCS_MCLK ■ UCS_MODOSC
--------------------	--

Modified bits of **UCSCTL8** register.

Returns

None

```
void UCS_enableClockRequest ( uint8_t selectClock )
```

Enables conditional module requests.

Parameters

<i>selectClock</i>	selects specific request enables Valid values are: <ul style="list-style-type: none"> ■ UCS_ACLK ■ UCS_SMCLK ■ UCS_MCLK ■ UCS_MODOSC
--------------------	--

Modified bits of **UCSCTL8** register.

Returns

None

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.

Returns

Current ACLK frequency in Hz

uint8_t UCS_getFaultFlagStatus (uint8_t *mask*)

Gets the current UCS fault flag status.

Parameters

<i>mask</i>	<p>is the masked interrupt flag status to be returned. Mask parameter can be either any of the following selection. Valid values are:</p> <ul style="list-style-type: none"> ■ 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
-------------	--

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.

Returns

Current MCLK frequency in Hz

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.

Returns

Current SMCLK frequency in Hz

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

Initializes a clock signal.

This function initializes each of the clock signals. The user must ensure that this function is called for each clock signal. If not, the default state is assumed for the particular clock signal. Refer MSP430Ware documentation for UCS module or Device Family User's Guide for details of default clock signal states.

Parameters

<i>selectedClockSignal</i>	selected clock signal Valid values are: <ul style="list-style-type: none"> ■ UCS_ACLK ■ UCS_MCLK ■ UCS_SMCLK ■ UCS_FLLREF
<i>clockSource</i>	is clock source for the selectedClockSignal Valid values are: <ul style="list-style-type: none"> ■ UCS_XT1CLK_SELECT ■ UCS_VLOCLK_SELECT ■ UCS_REFOCLK_SELECT ■ UCS_DCOCLK_SELECT ■ UCS_DCOCLKDIV_SELECT ■ UCS_XT2CLK_SELECT
<i>clockSourceDivider</i>	selected the clock divider to calculate clocksignal from clock source. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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. The function [PMM_setVCore\(\)](#) is required to call first if the target frequency is beyond current Vcore supported frequency range.

Parameters

<i>fsystem</i>	is the target frequency for MCLK in kHz
<i>ratio</i>	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

Referenced by UCS_initFLLSettle().

```
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. The function [PMM_setVCore\(\)](#) is required to call first if the target frequency is beyond current Vcore supported frequency range.

Parameters

<i>fsystem</i>	is the target frequency for MCLK in kHz
<i>ratio</i>	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

References UCS_initFLL().

```
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.

Parameters

<i>XT1CLK_↔ frequency</i>	is the XT1 crystal frequencies in Hz
<i>XT2CLK_↔ frequency</i>	is the XT2 crystal frequencies in Hz

Returns

None

```
void UCS_turnOffSMCLK ( void )
```

Turns off SMCLK using the SMCLKOFF bit.

Modified bits of **UCSCTL6** register.

Returns

None

```
void UCS_turnOffXT1 ( void )
```

Stops the XT1 oscillator using the XT1OFF bit.

Returns

None

```
void UCS_turnOffXT2 ( void )
```

Stops the XT2 oscillator using the XT2OFF bit.

Modified bits of **UCSCTL6** register.

Returns

None

```
void UCS_turnOnHFXT1 ( uint16_t xt1drive )
```

Initializes the XT1 crystal oscillator in high frequency mode.

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

Parameters

<i>xt1drive</i>	is the target drive strength for the XT1 crystal oscillator. Valid values are: <ul style="list-style-type: none"> ■ UCS_XT1_DRIVE_0 ■ UCS_XT1_DRIVE_1 ■ UCS_XT1_DRIVE_2 ■ UCS_XT1_DRIVE_3 [Default]
-----------------	---

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns

None

```
bool UCS_turnOnHFXT1WithTimeout ( 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.

Parameters

<i>xt1drive</i>	is the target drive strength for the XT1 crystal oscillator. Valid values are: <ul style="list-style-type: none"> ■ UCS_XT1_DRIVE_0 ■ UCS_XT1_DRIVE_1 ■ UCS_XT1_DRIVE_2 ■ UCS_XT1_DRIVE_3 [Default]
<i>timeout</i>	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

```
void UCS_turnOnLFXT1 ( 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.

Parameters

<i>xt1drive</i>	<p>is the target drive strength for the XT1 crystal oscillator. Valid values are:</p> <ul style="list-style-type: none"> ■ UCS_XT1_DRIVE_0 ■ UCS_XT1_DRIVE_1 ■ UCS_XT1_DRIVE_2 ■ UCS_XT1_DRIVE_3 [Default] <p>Modified bits are XT1DRIVE of UCSCTL6 register.</p>
<i>xcap</i>	<p>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 $C_{eff} = (C_{XIN} + C_{XOUT})/2$. It is assumed that $C_{XIN} = C_{XOUT}$ 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:</p> <ul style="list-style-type: none"> ■ UCS_XCAP_0 ■ UCS_XCAP_1 ■ UCS_XCAP_2 ■ UCS_XCAP_3 [Default]

Modified bits are **XCAP** of **UCSCTL6** register.

Returns

None

```
bool UCS_turnOnLFXT1WithTimeout ( 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.

Parameters

<i>xt1drive</i>	<p>is the target drive strength for the XT1 crystal oscillator. Valid values are:</p> <ul style="list-style-type: none"> ■ UCS_XT1_DRIVE_0 ■ UCS_XT1_DRIVE_1 ■ UCS_XT1_DRIVE_2 ■ UCS_XT1_DRIVE_3 [Default]
-----------------	--

<i>xcap</i>	<p>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</p> <ul style="list-style-type: none"> ■ 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]
<i>timeout</i>	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

`void UCS_turnOnSMCLK (void)`

Turns ON SMCLK using the SMCLKOFF bit.

Modified bits of **UCSCTL6** register.

Returns

None

`void UCS_turnOnXT2 (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.

Parameters

<i>xt2drive</i>	<p>is the target drive strength for the XT2 crystal oscillator. Valid values are:</p> <ul style="list-style-type: none"> ■ UCS_XT2_DRIVE_4MHZ_8MHZ ■ UCS_XT2_DRIVE_8MHZ_16MHZ ■ UCS_XT2_DRIVE_16MHZ_24MHZ ■ UCS_XT2_DRIVE_24MHZ_32MHZ [Default]
-----------------	---

Modified bits of **UCSCTL7** register, bits of **UCSCTL6** register and bits of **SFRIFG** register.

Returns

None

```
bool UCS_turnOnXT2WithTimeout ( 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.

Parameters

<i>xt2drive</i>	is the target drive strength for the XT2 crystal oscillator. Valid values are: <ul style="list-style-type: none"> ■ UCS_XT2_DRIVE_4MHZ_8MHZ ■ UCS_XT2_DRIVE_8MHZ_16MHZ ■ UCS_XT2_DRIVE_16MHZ_24MHZ ■ UCS_XT2_DRIVE_24MHZ_32MHZ [Default]
<i>timeout</i>	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

38.3 Programming Example

The following example shows some UCS operations using the APIs

```
// Set DCO FLL reference = REFO
UCS_initClockSignal(UCS_BASE,
    UCS_FLLREF,
    UCS_REFOCLK_SELECT,
    UCS_CLOCK_DIVIDER_1
);

// Set ACLK = REFO
UCS_initClockSignal(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);
```

39 USCI Universal Asynchronous Receiver/Transmitter (USCI_A_UART)

Introduction	433
API Functions	433
Programming Example	441

39.1 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.

39.2 API Functions

Functions

- `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 clockPrescaler, firstModReg, secondModReg and overSampling parameters should be pre-computed and passed into the initialization function.
- `void USCI_A_UART_transmitData (uint16_t baseAddress, uint8_t transmitData)`
Transmits a byte from the UART Module.
- `uint8_t USCI_A_UART_receiveData (uint16_t baseAddress)`

- Receives a byte that has been sent to the UART Module.
- void [USCI_A_UART_enableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Enables individual UART interrupt sources.
- void [USCI_A_UART_disableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Disables individual UART interrupt sources.
- uint8_t [USCI_A_UART_getInterruptStatus](#) (uint16_t baseAddress, uint8_t mask)
 - Gets the current UART interrupt status.
- void [USCI_A_UART_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)
 - Clears UART interrupt sources.
- void [USCI_A_UART_enable](#) (uint16_t baseAddress)
 - Enables the UART block.
- void [USCI_A_UART_disable](#) (uint16_t baseAddress)
 - Disables the UART block.
- uint8_t [USCI_A_UART_queryStatusFlags](#) (uint16_t baseAddress, uint8_t mask)
 - Gets the current UART status flags.
- void [USCI_A_UART_setDormant](#) (uint16_t baseAddress)
 - Sets the UART module in dormant mode.
- void [USCI_A_UART_resetDormant](#) (uint16_t baseAddress)
 - Re-enables UART module from dormant mode.
- 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.
- void [USCI_A_UART_transmitBreak](#) (uint16_t baseAddress)
 - Transmit break.
- uint32_t [USCI_A_UART_getReceiveBufferAddressForDMA](#) (uint16_t baseAddress)
 - Returns the address of the RX Buffer of the UART for the DMA module.
- uint32_t [USCI_A_UART_getTransmitBufferAddressForDMA](#) (uint16_t baseAddress)
 - Returns the address of the TX Buffer of the UART for the DMA module.

39.2.1 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_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_clearInterrupt()`
- `USCI_A_UART_queryStatusFlags()`

DMA related

- `USCI_A_UART_getReceiveBufferAddressForDMA()`
- `USCI_A_UART_getTransmitBufferAddressForDMA()`

39.2.2 Function Documentation

```
void USCI_A_UART_clearInterrupt ( 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

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>mask</i>	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void USCI_A_UART_disable ( uint16_t baseAddress )
```

Disables the UART block.

This will disable operation of the UART block.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void USCI_A_UART_enable ( uint16_t baseAddress )
```

Enables the UART block.

This will enable operation of the UART block.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

Returns

Address of RX Buffer

`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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

Returns

Address of TX Buffer

`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

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>param</i>	is the pointer to struct for initialization.

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

Returns

STATUS_SUCCESS or STATUS_FAIL of the initialization process

References USCI_A_UART_initParam::clockPrescalar, USCI_A_UART_initParam::firstModReg, USCI_A_UART_initParam::msborLsbFirst, USCI_A_UART_initParam::numberOfStopBits, USCI_A_UART_initParam::overSampling, USCI_A_UART_initParam::parity, USCI_A_UART_initParam::secondModReg, USCI_A_UART_initParam::selectClockSource, and USCI_A_UART_initParam::uartMode.

```
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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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.

Parameters

<i>baseAddress</i>	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`.

`void USCI_A_UART_resetDormant (uint16_t baseAddress)`

Re-enables UART module from dormant mode.

Not dormant. All received characters set UCRXIFG.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

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

Returns

None

`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.

Parameters

<i>baseAddress</i>	is the base address of the USCI_A_UART module.
--------------------	--

Modified bits of **UCAxCTL1** register.

Returns

None

`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.


```

        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;
    }
}

```

40 USCI Synchronous Peripheral Interface (USCI_A_SPI)

Introduction	443
API Functions	443
Programming Example	451

40.1 Introduction

The Serial Peripheral Interface Bus or USCLA_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 USCLA_SPI module can be configured as either a master or a slave device.

The USCLA_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.

40.2 API Functions

Functions

- bool [USCI_A_SPI_initMaster](#) (uint16_t baseAddress, [USCI_A_SPI_initMasterParam](#) *param)
Initializes the SPI Master block.
- 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.
- bool [USCI_A_SPI_initSlave](#) (uint16_t baseAddress, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
Initializes the SPI Slave block.
- 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.
- void [USCI_A_SPI_transmitData](#) (uint16_t baseAddress, uint8_t transmitData)
Transmits a byte from the SPI Module.
- uint8_t [USCI_A_SPI_receiveData](#) (uint16_t baseAddress)
Receives a byte that has been sent to the SPI Module.
- void [USCI_A_SPI_enableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Enables individual SPI interrupt sources.
- void [USCI_A_SPI_disableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Disables individual SPI interrupt sources.
- uint8_t [USCI_A_SPI_getInterruptStatus](#) (uint16_t baseAddress, uint8_t mask)
Gets the current SPI interrupt status.
- void [USCI_A_SPI_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Clears the selected SPI interrupt status flag.

- void [USCI_A_SPI_enable](#) (uint16_t baseAddress)
Enables the SPI block.
- void [USCI_A_SPI_disable](#) (uint16_t baseAddress)
Disables the SPI block.
- uint32_t [USCI_A_SPI_getReceiveBufferAddressForDMA](#) (uint16_t baseAddress)
Returns the address of the RX Buffer of the SPI for the DMA module.
- uint32_t [USCI_A_SPI_getTransmitBufferAddressForDMA](#) (uint16_t baseAddress)
Returns the address of the TX Buffer of the SPI for the DMA module.
- uint8_t [USCI_A_SPI_isBusy](#) (uint16_t baseAddress)
Indicates whether or not the SPI bus is busy.

40.2.1 Detailed Description

To use the module as a master, the user must call [USCI_A_SPI_initMaster\(\)](#) 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_initSlave\(\)](#) 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_initMaster\(\)](#)
- [USCI_A_SPI_initSlave\(\)](#)
- [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_clearInterrupt\(\)](#)

DMA related

- [USCI_A_SPI_getReceiveBufferAddressForDMA\(\)](#)
- [USCI_A_SPI_getTransmitBufferAddressForDMA\(\)](#)

40.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ USCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ONNEXT [Default] ■ USCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ONNEXT
<i>clockPolarity</i>	Valid values are: <ul style="list-style-type: none"> ■ USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH ■ USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW [Default]

Modified bits are **UCCKPL** and **UCCKPH** of **UCAxCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>param</i>	is the pointer to struct for master clock setting.

Modified bits of **UCAxBRW** register.

Returns

None

References **USCI_A_SPI_changeMasterClockParam::clockSourceFrequency**, and **USCI_A_SPI_changeMasterClockParam::desiredSpiClock**.

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

Clears the selected SPI interrupt status flag.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ USCI_A_SPI_TRANSMIT_INTERRUPT ■ USCI_A_SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIFG** register.

Returns

None

```
void USCI_A_SPI_disable ( uint16_t baseAddress )
```

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the USCI SPI module.
--------------------	---

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ USCI_A_SPI_TRANSMIT_INTERRUPT ■ USCI_A_SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIE** register.

Returns

None

```
void USCI_A_SPI_enable ( uint16_t baseAddress )
```

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the USCI SPI module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ USCI_A_SPI_TRANSMIT_INTERRUPT ■ USCI_A_SPI_RECEIVE_INTERRUPT

Modified bits of **UCAxIE** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

the address of the RX Buffer

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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

the address of the TX Buffer

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\(\)](#)

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>param</i>	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

References USCI_A_SPI_initMasterParam::clockPhase, USCI_A_SPI_initMasterParam::clockPolarity, USCI_A_SPI_initMasterParam::clockSourceFrequency, USCI_A_SPI_initMasterParam::desiredSpiClock, USCI_A_SPI_initMasterParam::msbFirst, and USCI_A_SPI_initMasterParam::selectClockSource.

```
bool USCI_A_SPI_initSlave ( 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\(\)](#)

Parameters

<i>baseAddress</i>	is the base address of the SPI Slave module.
<i>msbFirst</i>	controls the direction of the receive and transmit shift register. Valid values are: <ul style="list-style-type: none"> ■ USCI_A_SPI_MSB_FIRST ■ USCI_A_SPI_LSB_FIRST [Default]
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ USCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ONNEXT [Default] ■ USCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ONNEXT
<i>clockPolarity</i>	Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

Parameters

<i>baseAddress</i>	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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

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

```
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

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>transmitData</i>	data to be transmitted from the SPI module

Returns

None

40.3 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
USCI_B_SPI_initMasterParam param = {0};
param.selectClockSource = USCI_B_SPI_CLOCKSOURCE_SMCLK;
param.clockSourceFrequency = UCS_getSMCLK();
param.desiredSpiClock = SPICLK;
param.msbFirst = USCI_B_SPI_MSB_FIRST;
param.clockPhase = USCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT;
param.clockPolarity = USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH;
returnValue = USCI_B_SPI_initMaster(USCI_B0_BASE, &param);

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;
    }
}

```

41 USCI Synchronous Peripheral Interface (USCI_B_SPI)

Introduction	453
API Functions	453
Programming Example	461

41.1 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.

41.2 API Functions

Functions

- bool [USCI_B_SPI_initMaster](#) (uint16_t baseAddress, [USCI_B_SPI_initMasterParam](#) *param)
Initializes the SPI Master block.
- 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.
- bool [USCI_B_SPI_initSlave](#) (uint16_t baseAddress, uint8_t msbFirst, uint8_t clockPhase, uint8_t clockPolarity)
Initializes the SPI Slave block.
- 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.
- void [USCI_B_SPI_transmitData](#) (uint16_t baseAddress, uint8_t transmitData)
Transmits a byte from the SPI Module.
- uint8_t [USCI_B_SPI_receiveData](#) (uint16_t baseAddress)
Receives a byte that has been sent to the SPI Module.
- void [USCI_B_SPI_enableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Enables individual SPI interrupt sources.
- void [USCI_B_SPI_disableInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Disables individual SPI interrupt sources.
- uint8_t [USCI_B_SPI_getInterruptStatus](#) (uint16_t baseAddress, uint8_t mask)
Gets the current SPI interrupt status.
- void [USCI_B_SPI_clearInterrupt](#) (uint16_t baseAddress, uint8_t mask)
Clears the selected SPI interrupt status flag.

- void `USCI_B_SPI_enable` (uint16_t baseAddress)
Enables the SPI block.
- void `USCI_B_SPI_disable` (uint16_t baseAddress)
Disables the SPI block.
- uint32_t `USCI_B_SPI_getReceiveBufferAddressForDMA` (uint16_t baseAddress)
Returns the address of the RX Buffer of the SPI for the DMA module.
- uint32_t `USCI_B_SPI_getTransmitBufferAddressForDMA` (uint16_t baseAddress)
Returns the address of the TX Buffer of the SPI for the DMA module.
- uint8_t `USCI_B_SPI_isBusy` (uint16_t baseAddress)
Indicates whether or not the SPI bus is busy.

41.2.1 Detailed Description

To use the module as a master, the user must call `USCI_B_SPI_initMaster()` 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_initSlave()` 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_initMaster()`
- `USCI_B_SPI_initSlave()`
- `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_clearInterrupt()`

DMA related

- `USCI_B_SPI_getReceiveBufferAddressForDMA()`
- `USCI_B_SPI_getTransmitBufferAddressForDMA()`

41.2.2 Function Documentation

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ USCI.B.SPI.PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT [Default] ■ USCI.B.SPI.PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT
<i>clockPolarity</i>	Valid values are: <ul style="list-style-type: none"> ■ USCI.B.SPI.CLOCKPOLARITY_INACTIVITY_HIGH ■ USCI.B.SPI.CLOCKPOLARITY_INACTIVITY_LOW [Default]

Modified bits are **UCCKPL** and **UCCKPH** of **UCAxCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>param</i>	is the pointer to struct for master clock setting.

Modified bits of **UCAxBRW** register.

Returns

None

References **USCI.B.SPI.changeMasterClockParam::clockSourceFrequency**, and **USCI.B.SPI.changeMasterClockParam::desiredSpiClock**.

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

Clears the selected SPI interrupt status flag.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the masked interrupt flag to be cleared. Valid values are: <ul style="list-style-type: none"> ■ USCI.B.SPI.TRANSMIT_INTERRUPT ■ USCI.B.SPI.RECEIVE_INTERRUPT

Modified bits of **UCBxIFG** register.

Returns

None

```
void USCI_B_SPI_disable ( uint16_t baseAddress )
```

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the USCI SPI module.
--------------------	---

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Valid values are: <ul style="list-style-type: none"> ■ USCI_B_SPI_TRANSMIT_INTERRUPT ■ USCI_B_SPI_RECEIVE_INTERRUPT

Modified bits of **UCBxIE** register.

Returns

None

```
void USCI_B_SPI_enable ( uint16_t baseAddress )
```

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

<i>baseAddress</i>	is the base address of the USCI SPI module.
--------------------	---

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

Returns

None

```
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

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Valid values are: <ul style="list-style-type: none"> ■ USCI_B_SPI_TRANSMIT_INTERRUPT ■ USCI_B_SPI_RECEIVE_INTERRUPT

Modified bits of UCBxIE register.

Returns**None**

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>mask</i>	is the masked interrupt flag status to be returned. Valid values are: <ul style="list-style-type: none"> ■ 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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

The address of the SPI RX buffer

`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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

The address of the SPI TX buffer

`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\(\)](#)

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>param</i>	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

References `USCI_B_SPI_initMasterParam::clockPhase`, `USCI_B_SPI_initMasterParam::clockPolarity`, `USCI_B_SPI_initMasterParam::clockSourceFrequency`, `USCI_B_SPI_initMasterParam::desiredSpiClock`, `USCI_B_SPI_initMasterParam::msbFirst`, and `USCI_B_SPI_initMasterParam::selectClockSource`.

```
bool USCI_B_SPI_initSlave ( 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\(\)](#)

Parameters

<i>baseAddress</i>	is the base address of the SPI Slave module.
<i>msbFirst</i>	controls the direction of the receive and transmit shift register. Valid values are: <ul style="list-style-type: none"> ■ USCI.B.SPI.MSB_FIRST ■ USCI.B.SPI.LSB_FIRST [Default]
<i>clockPhase</i>	is clock phase select. Valid values are: <ul style="list-style-type: none"> ■ USCI.B.SPI.PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ONNEXT [Default] ■ USCI.B.SPI.PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ONNEXT
<i>clockPolarity</i>	Valid values are: <ul style="list-style-type: none"> ■ 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

```
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

Parameters

<i>baseAddress</i>	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

```
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.

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
--------------------	--

Returns

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

```
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

Parameters

<i>baseAddress</i>	is the base address of the SPI module.
<i>transmitData</i>	data to be transmitted from the SPI module

Returns

None

41.3 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
USCI_B_SPI_initMasterParam param = {0};
param.selectClockSource = USCI_B_SPI_CLOCKSOURCE_SMCLK;
param.clockSourceFrequency = UCS_getSMCLK();
param.desiredSpiClock = SPICLK;
param.msbFirst = USCI_B_SPI_MSB_FIRST;
param.clockPhase = USCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT;
param.clockPolarity = USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH;
returnValue = USCI_B_SPI_initMaster(USCI_B0_BASE, &param);

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;
    }
}

```

42 USCI Inter-Integrated Circuit (USCI_B_I2C)

Introduction	463
API Functions	465
Programming Example	481

42.1 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.

42.2 Master Operations

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

- [USCI_B_I2C.initMaster\(\)](#)
- [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.initMaster\(\)](#). That function will set the clock and data rates. This is followed by a call to set the slave address with which the master intends to communicate with using [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_masterSendMultiByteStart()`
- `USCI_B_I2C_masterSendMultiByteNext()`
- `USCI_B_I2C_masterSendMultiByteFinish()`
- `USCI_B_I2C_masterSendMultiByteStop()`

Master Single Byte Reception

- `USCI_B_I2C_masterReceiveSingleStart()`
- `USCI_B_I2C_masterReceiveSingle()`

Master Multiple Byte Reception

- `USCI_B_I2C_masterReceiveMultiByteStart()`
- `USCI_B_I2C_masterReceiveMultiByteNext()`
- `USCI_B_I2C_masterReceiveMultiByteFinish()`
- `USCI_B_I2C_masterReceiveMultiByteStop()`

Master Single Byte Transmission with Time-out

- `USCI_B_I2C_masterSendSingleByteWithTimeout()`

Master Multiple Byte Transmission with Time-out

- `USCI_B_I2C_masterSendMultiByteStartWithTimeout()`
- `USCI_B_I2C_masterSendMultiByteNextWithTimeout()`
- `USCI_B_I2C_masterReceiveMultiByteFinishWithTimeout()`
- `USCI_B_I2C_masterSendMultiByteStopWithTimeout()`

Master Single Byte Reception with Time-out `USCI_B_I2C_masterReceiveSingleStartWithTimeout()`

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.

42.3 Slave Operations

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

- `USCI_B_I2C_initSlave()`
- `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.initSlave` 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_slavePutData()`

Slave Reception API

- `USCI_B_I2C_slaveGetData()`

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.

42.4 API Functions

Functions

- void `USCI_B_I2C.initMaster` (uint16_t baseAddress, `USCI_B_I2C_initMasterParam` *param)
Initializes the I2C Master block.
- void `USCI_B_I2C.initSlave` (uint16_t baseAddress, uint8_t slaveAddress)
Initializes the I2C Slave block.
- void `USCI_B_I2C.enable` (uint16_t baseAddress)
Enables the I2C block.
- void `USCI_B_I2C.disable` (uint16_t baseAddress)
Disables the I2C block.
- void `USCI_B_I2C.setSlaveAddress` (uint16_t baseAddress, uint8_t slaveAddress)
Sets the address that the I2C Master will place on the bus.
- void `USCI_B_I2C.setMode` (uint16_t baseAddress, uint8_t mode)
Sets the mode of the I2C device.
- void `USCI_B_I2C_slavePutData` (uint16_t baseAddress, uint8_t transmitData)
Transmits a byte from the I2C Module.
- uint8_t `USCI_B_I2C_slaveGetData` (uint16_t baseAddress)
Receives a byte that has been sent to the I2C Module.
- uint8_t `USCI_B_I2C.isBusBusy` (uint16_t baseAddress)
Indicates whether or not the I2C bus is busy.
- 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.
- uint8_t `USCI_B_I2C.masterIsStopSent` (uint16_t baseAddress)
Indicates whether STOP got sent.
- uint8_t `USCI_B_I2C.masterIsStartSent` (uint16_t baseAddress)
Indicates whether START got sent.
- void `USCI_B_I2C.masterSendStart` (uint16_t baseAddress)
This function is used by the Master module to initiate START.
- void `USCI_B_I2C.enableInterrupt` (uint16_t baseAddress, uint8_t mask)

- Enables individual I2C interrupt sources.*

 - void `USCI_B_I2C_disableInterrupt` (uint16_t baseAddress, uint8_t mask)
- Disables individual I2C interrupt sources.*

 - void `USCI_B_I2C_clearInterrupt` (uint16_t baseAddress, uint8_t mask)
- Clears I2C interrupt sources.*

 - uint8_t `USCI_B_I2C_getInterruptStatus` (uint16_t baseAddress, uint8_t mask)
- Gets the current I2C interrupt status.*

 - void `USCI_B_I2C_masterSendSingleByte` (uint16_t baseAddress, uint8_t txData)
- Does single byte transmission from Master to Slave.*

 - bool `USCI_B_I2C_masterSendSingleByteWithTimeout` (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- Does single byte transmission from Master to Slave with timeout.*

 - void `USCI_B_I2C_masterSendMultiByteStart` (uint16_t baseAddress, uint8_t txData)
- Starts multi-byte transmission from Master to Slave.*

 - bool `USCI_B_I2C_masterSendMultiByteStartWithTimeout` (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- Starts multi-byte transmission from Master to Slave with timeout.*

 - void `USCI_B_I2C_masterSendMultiByteNext` (uint16_t baseAddress, uint8_t txData)
- Continues multi-byte transmission from Master to Slave.*

 - bool `USCI_B_I2C_masterSendMultiByteNextWithTimeout` (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- Continues multi-byte transmission from Master to Slave with timeout.*

 - void `USCI_B_I2C_masterSendMultiByteFinish` (uint16_t baseAddress, uint8_t txData)
- Finishes multi-byte transmission from Master to Slave.*

 - bool `USCI_B_I2C_masterSendMultiByteFinishWithTimeout` (uint16_t baseAddress, uint8_t txData, uint32_t timeout)
- Finishes multi-byte transmission from Master to Slave with timeout.*

 - void `USCI_B_I2C_masterSendMultiByteStop` (uint16_t baseAddress)
- Sends STOP byte at the end of a multi-byte transmission from Master to Slave.*

 - bool `USCI_B_I2C_masterSendMultiByteStopWithTimeout` (uint16_t baseAddress, uint32_t timeout)
- Sends STOP byte at the end of a multi-byte transmission from Master to Slave with timeout.*

 - void `USCI_B_I2C_masterReceiveMultiByteStart` (uint16_t baseAddress)
- Starts multi-byte reception at the Master end.*

 - uint8_t `USCI_B_I2C_masterReceiveMultiByteNext` (uint16_t baseAddress)
- Starts multi-byte reception at the Master end one byte at a time.*

 - uint8_t `USCI_B_I2C_masterReceiveMultiByteFinish` (uint16_t baseAddress)
- Finishes multi-byte reception at the Master end.*

 - bool `USCI_B_I2C_masterReceiveMultiByteFinishWithTimeout` (uint16_t baseAddress, uint8_t *rxData, uint32_t timeout)
- Finishes multi-byte reception at the Master end with timeout.*

 - void `USCI_B_I2C_masterReceiveMultiByteStop` (uint16_t baseAddress)
- Sends the STOP at the end of a multi-byte reception at the Master end.*

 - void `USCI_B_I2C_masterReceiveSingleStart` (uint16_t baseAddress)
- Initiates a single byte Reception at the Master End.*

 - bool `USCI_B_I2C_masterReceiveSingleStartWithTimeout` (uint16_t baseAddress, uint32_t timeout)
- Initiates a single byte Reception at the Master End with timeout.*

 - uint8_t `USCI_B_I2C_masterReceiveSingle` (uint16_t baseAddress)
- Receives a byte that has been sent to the I2C Master Module.*

 - uint32_t `USCI_B_I2C_getReceiveBufferAddressForDMA` (uint16_t baseAddress)
- Returns the address of the RX Buffer of the I2C for the DMA module.*

 - uint32_t `USCI_B_I2C_getTransmitBufferAddressForDMA` (uint16_t baseAddress)
- Returns the address of the TX Buffer of the I2C for the DMA module.*

42.4.1 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_clearInterrupt()`
- `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_initMaster()`
- `USCI_B_I2C_enable()`
- `USCI_B_I2C_disable()`
- `USCI_B_I2C_isBusBusy()`
- `USCI_B_I2C_isBusy()`
- `USCI_B_I2C_initSlave()`
- `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_slavePutData()`
- `USCI_B_I2C_slaveGetData()`

Sending and receiving data from the USCI_B_I2C slave module is handled by

- `USCI_B_I2C_masterSendSingleByte()`
- `USCI_B_I2C_masterSendMultiByteStart()`
- `USCI_B_I2C_masterSendMultiByteNext()`
- `USCI_B_I2C_masterSendMultiByteFinish()`
- `USCI_B_I2C_masterSendMultiByteStop()`
- `USCI_B_I2C_masterReceiveMultiByteStart()`
- `USCI_B_I2C_masterReceiveMultiByteNext()`
- `USCI_B_I2C_masterReceiveMultiByteFinish()`
- `USCI_B_I2C_masterReceiveMultiByteStop()`
- `USCI_B_I2C_masterReceiveSingleStart()`
- `USCI_B_I2C_masterReceiveSingle()`
- `USCI_B_I2C_getReceiveBufferAddressForDMA()`
- `USCI_B_I2C_getTransmitBufferAddressForDMA()`

DMA related

- [USCI_B_I2C_getReceiveBufferAddressForDMA\(\)](#)
- [USCI_B_I2C_getTransmitBufferAddressForDMA\(\)](#)

42.4.2 Function Documentation

`void USCI_B_I2C_clearInterrupt (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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Slave module.
<i>mask</i>	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`void USCI_B_I2C_disable (uint16_t baseAddress)`

Disables the I2C block.

This will disable operation of the I2C block.

Parameters

<i>baseAddress</i>	is the base address of the USCI I2C module.
--------------------	---

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

```
void USCI_B_I2C_enable ( uint16_t baseAddress )
```

Enables the I2C block.

This will enable operation of the I2C block.

Parameters

<i>baseAddress</i>	is the base address of the USCI I2C module.
--------------------	---

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

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

`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.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
<i>mask</i>	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: <ul style="list-style-type: none"> ■ 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

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.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

the address of the RX Buffer

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.

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

the address of the TX Buffer

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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>param</i>	is the pointer 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

References USCI_B_I2C_initMasterParam::dataRate, USCI_B_I2C_initMasterParam::i2cClk, and USCI_B_I2C_initMasterParam::selectClockSource.


```
void USCI_B_I2C_initSlave ( 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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Slave module.
<i>slaveAddress</i>	7-bit slave address

Modified bits of **UCBxI2COA** register; bits **UCSWRST** of **UCBxCTL1** register; bits **UCMODE_3** and **UCSYNC** of **UCBxCTL0** register.

Returns

None

```
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.

Parameters

<i>baseAddress</i>	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

```
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.

Parameters

<i>baseAddress</i>	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

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.

Parameters

<i>baseAddress</i>	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**

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.

Parameters

<i>baseAddress</i>	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**

uint8_t USCI_B_I2C_masterReceiveMultiByteFinish (uint16_t *baseAddress*)

Finishes multi-byte reception at the Master end.

This function is used by the Master module to initiate completion of a multi-byte reception. This function does the following: - Receives the current byte and initiates the STOP from Master to Slave

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

Received byte at Master end.

```
bool USCI_B_I2C_masterReceiveMultiByteFinishWithTimeout ( 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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>rxData</i>	is a pointer to the location to store the received byte at master end
<i>timeout</i>	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.

```
uint8_t USCI_B_I2C_masterReceiveMultiByteNext ( uint16_t baseAddress )
```

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

Returns

Received byte at Master end.

```
void USCI_B_I2C_masterReceiveMultiByteStart ( 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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

```
void USCI_B_I2C_masterReceiveMultiByteStop ( uint16_t baseAddress )
```

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

This function is used by the Master module to initiate STOP

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

```
uint8_t USCI_B_I2C_masterReceiveSingle ( uint16_t baseAddress )
```

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

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

```
void USCI_B_I2C_masterReceiveSingleStart ( 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

Parameters

<i>baseAddress</i>	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

```
bool USCI_B_I2C_masterReceiveSingleStartWithTimeout ( 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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>timeout</i>	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.

```
void USCI_B_I2C_masterSendMultiByteFinish ( uint16_t baseAddress, uint8_t txData )
```

Finishes multi-byte transmission from Master to Slave.

This function is used by the Master module to send the last byte and STOP. This function does the following: - Transmits the last data byte of a multi-byte transmission to the Slave; - Sends STOP

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

```
bool USCI_B_I2C_masterSendMultiByteFinishWithTimeout ( uint16_t baseAddress, uint8_t
txData, uint32_t timeout )
```

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

This function is used by the Master module to send the last byte and STOP. This function does the following: - Transmits the last data byte of a multi-byte transmission to the Slave; - Sends STOP

Parameters

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

```
void USCI_B_I2C_masterSendMultiByteNext ( uint16_t baseAddress, uint8_t txData )
```

Continues multi-byte transmission from Master to Slave.

This function is used by the Master module continue each byte of a multi- byte transmission. This function does the following: -Transmits each data byte of a multi-byte transmission to the Slave

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

None

```
bool USCI_B_I2C_masterSendMultiByteNextWithTimeout ( uint16_t baseAddress, uint8_t txData, uint32_t timeout )
```

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

This function is used by the Master module continue each byte of a multi- byte transmission. This function does the following: -Transmits each data byte of a multi-byte transmission to the Slave

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is the next data byte to be transmitted
<i>timeout</i>	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.

```
void USCI_B_I2C_masterSendMultiByteStart ( uint16_t baseAddress, uint8_t txData )
```

Starts multi-byte transmission from Master to Slave.

This function is used by the Master module to send a single byte. This function does the following: - Sends START; - Transmits the first data byte of a multi-byte transmission to the Slave

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

```
bool USCI_B_I2C_masterSendMultiByteStartWithTimeout ( uint16_t baseAddress, uint8_t
    txData, uint32_t timeout )
```

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

This function is used by the Master module to send a single byte. This function does the following:
 - Sends START; - Transmits the first data byte of a multi-byte transmission to the Slave

Parameters

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

```
void USCI_B_I2C_masterSendMultiByteStop ( uint16_t baseAddress )
```

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

This function is used by the Master module send STOP at the end of a multi- byte transmission.
 This function does the following: - Sends a STOP after current transmission is complete

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

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

Returns

None

```
bool USCI_B_I2C_masterSendMultiByteStopWithTimeout ( uint16_t baseAddress, uint32_t
    timeout )
```

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

This function is used by the Master module send STOP at the end of a multi- byte transmission.
 This function does the following: - Sends a STOP after current transmission is complete

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>timeout</i>	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.

```
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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	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

```
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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>txData</i>	is the data byte to be transmitted
<i>timeout</i>	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.

```
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

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
--------------------	---

Returns

None


```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>mode</i>	indicates whether module is in transmit/receive mode Valid values are: <ul style="list-style-type: none"> ■ USCI_B_I2C_TRANSMIT_MODE ■ USCI_B_I2C_RECEIVE_MODE [Default]

Returns

None

```
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.

Parameters

<i>baseAddress</i>	is the base address of the I2C Master module.
<i>slaveAddress</i>	7-bit slave address

Modified bits of **UCBxI2CSA** register; bits **UCSWRST** of **UCBxCTL1** register.

Returns

None

```
uint8_t USCI_B_I2C_slaveGetData ( uint16_t baseAddress )
```

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

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

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
--------------------	--

Returns

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

```
void USCI_B_I2C_slavePutData ( uint16_t baseAddress, uint8_t transmitData )
```

Transmits a byte from the I2C Module.

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

Modified bit is UCBxTXBUF register

Parameters

<i>baseAddress</i>	is the base address of the I2C module.
<i>transmitData</i>	data to be transmitted from the I2C module

Modified bits of **UCBxTXBUF** register.

Returns

None

42.5 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_initMasterParam param = {0};
param.selectClockSource = USCI_B_I2C_CLOCKSOURCE_SMCLK;
param.i2cClk = UCS_getSMCLK();
param.dataRate = USCI_B_I2C_SET_DATA_RATE_400KBPS;
USCI_B_I2C_initMaster(USCI_B0_BASE, &param);

// 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++;
}
```

43 WatchDog Timer (WDT_A)

Introduction	482
API Functions	482
Programming Example	485

43.1 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.

43.2 API Functions

Functions

- void [WDT_A_hold](#) (uint16_t baseAddress)
Holds the Watchdog Timer.
- void [WDT_A_start](#) (uint16_t baseAddress)
Starts the Watchdog Timer.
- void [WDT_A_resetTimer](#) (uint16_t baseAddress)
Resets the timer counter of the Watchdog Timer.
- void [WDT_A_initWatchdogTimer](#) (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
Sets the clock source for the Watchdog Timer in watchdog mode.
- void [WDT_A_initIntervalTimer](#) (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
Sets the clock source for the Watchdog Timer in timer interval mode.

43.2.1 Detailed Description

The WDT_A API is one group that controls the WDT_A module.

- [WDT_A_hold\(\)](#)
- [WDT_A_start\(\)](#)
- [WDT_A_clearCounter\(\)](#)
- [WDT_A_initWatchdogTimer\(\)](#)
- [WDT_A_initIntervalTimer\(\)](#)

43.2.2 Function Documentation

void WDT_A_hold (uint16_t *baseAddress*)

Holds the Watchdog Timer.

This function stops the watchdog timer from running, that way no interrupt or PUC is asserted.

Parameters

<i>baseAddress</i>	is the base address of the WDT_A module.
--------------------	--

Returns

None

void WDT_A_initIntervalTimer (uint16_t *baseAddress*, uint8_t *clockSelect*, uint8_t *clockDivider*)

Sets the clock source for the Watchdog Timer in timer interval mode.

This function sets the watchdog timer as timer interval mode, which will assert an interrupt without causing a PUC.

Parameters

<i>baseAddress</i>	is the base address of the WDT_A module.
<i>clockSelect</i>	is the clock source that the watchdog timer will use. Valid values are: <ul style="list-style-type: none"> ■ WDT_A_CLOCKSOURCE_SMCLK [Default] ■ WDT_A_CLOCKSOURCE_ACLK ■ WDT_A_CLOCKSOURCE_VLOCLK ■ WDT_A_CLOCKSOURCE_XCLK Modified bits are WDTSSSEL of WDTCTL register.
<i>clockDivider</i>	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void WDT_A_initWatchdogTimer ( uint16_t baseAddress, uint8_t clockSelect, uint8_t
clockDivider )
```

Sets the clock source for the Watchdog Timer in watchdog mode.

This function sets the watchdog timer in watchdog mode, which will cause a PUC when the timer overflows. When in the mode, a PUC can be avoided with a call to [WDT_A_resetTimer\(\)](#) before the timer runs out.

Parameters

<i>baseAddress</i>	is the base address of the WDT_A module.
<i>clockSelect</i>	is the clock source that the watchdog timer will use. Valid values are: <ul style="list-style-type: none"> ■ WDT_A_CLOCKSOURCE_SMCLK [Default] ■ WDT_A_CLOCKSOURCE_ACLK ■ WDT_A_CLOCKSOURCE_VLOCLK ■ WDT_A_CLOCKSOURCE_XCLK Modified bits are WDTSSSEL of WDTCTL register.
<i>clockDivider</i>	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: <ul style="list-style-type: none"> ■ 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

```
void WDT_A_resetTimer ( uint16_t baseAddress )
```

Resets the timer counter of the Watchdog Timer.

This function resets the watchdog timer to 0x0000h.

Parameters

<i>baseAddress</i>	is the base address of the WDT_A module.
--------------------	--

Returns

None

```
void WDT_A_start ( uint16_t baseAddress )
```

Starts the Watchdog Timer.

This function starts the watchdog timer functionality to start counting again.

Parameters

<i>baseAddress</i>	is the base address of the WDT_A module.
--------------------	--

Returns

None

43.3 Programming Example

The following example shows how to initialize and use the WDT_A API to interrupt about every 32 ms, toggling the LED in the ISR.

```
//Initialize WDT_A module in timer interval mode,
//with SMCLK as source at an interval of 32 ms.
WDT_A_initIntervalTimer(WDT_A.BASE,
    WDT_A.CLOCKSOURCE_SMCLK,
    WDT_A.CLOCKDIVIDER_32K);

//Enable Watchdog Interrupt
SFR_enableInterrupt (SFR.WATCHDOG_INTERVAL_TIMER_INTERRUPT);

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

//Enter LPM0, enable interrupts
__bis_SR_register(LPM0_bits + GIE);
//For debugger
__no_operation();
```

44 Data Structure Documentation

44.1 Data Structures

Here are the data structures with brief descriptions:

ADC12_A_configureMemoryParam	Used in the ADC12_A_configureMemory() function as the param parameter	488
Calendar	Used in the RTC_A_initCalendar() function as the CalendarTime parameter	490
Comp_B_configureReferenceVoltageParam	Used in the Comp_B_configureReferenceVoltage() function as the param parameter	491
Comp_B_initParam	Used in the Comp_B_init() function as the param parameter	492
DAC12_A_initParam	Used in the DAC12_A_init() function as the param parameter	495
DMA_initParam	Used in the DMA_init() function as the param parameter	497
EUSCI_A_SPI_changeMasterClockParam	Used in the EUSCI_A_SPI_changeMasterClock() function as the param parameter .	499
EUSCI_A_SPI_initMasterParam	Used in the EUSCI_A_SPI_initMaster() function as the param parameter	500
EUSCI_A_SPI_initSlaveParam	Used in the EUSCI_A_SPI_initSlave() function as the param parameter	502
EUSCI_A_UART_initParam	Used in the EUSCI_A_UART_init() function as the param parameter	503
EUSCI_B_I2C_initMasterParam	Used in the EUSCI_B_I2C_initMaster() function as the param parameter	505
EUSCI_B_I2C_initSlaveParam	Used in the EUSCI_B_I2C_initSlave() function as the param parameter	507
EUSCI_B_SPI_changeMasterClockParam	Used in the EUSCI_B_SPI_changeMasterClock() function as the param parameter .	508
EUSCI_B_SPI_initMasterParam	Used in the EUSCI_B_SPI_initMaster() function as the param parameter	508
EUSCI_B_SPI_initSlaveParam	Used in the EUSCI_B_SPI_initSlave() function as the param parameter	510
PMAP_initPortsParam	Used in the PMAP_initPorts() function as the param parameter	511
RTC_A_configureCalendarAlarmParam	Used in the RTC_A_configureCalendarAlarm() function as the param parameter . .	512
RTC_B_configureCalendarAlarmParam	Used in the RTC_B_configureCalendarAlarm() function as the param parameter . .	513
RTC_C_configureCalendarAlarmParam	Used in the RTC_C_configureCalendarAlarm() function as the param parameter . .	514
s_Peripheral_Memory_Data	??
s_TLV_ADC_Cal_Data	??
s_TLV_Die_Record	??
s_TLV_REF_Cal_Data	??
s_TLV_Timer_D_Cal_Data	??
SD24_B_initConverterAdvancedParam	Used in the SD24_B_initConverterAdvanced() function as the param parameter . .	516

SD24_B_initConverterParam	Used in the SD24_B_initConverter() function as the param parameter	519
SD24_B_initParam	Used in the SD24_B_init() function as the param parameter	520
TEC_initExternalFaultInputParam	Used in the TEC_initExternalFaultInput() function as the param parameter	565
Timer_A_initCaptureModeParam	Used in the Timer_A_initCaptureMode() function as the param parameter	522
Timer_A_initCompareModeParam	Used in the Timer_A_initCompareMode() function as the param parameter	525
Timer_A_initContinuousModeParam	Used in the Timer_A_initContinuousMode() function as the param parameter	526
Timer_A_initUpDownModeParam	Used in the Timer_A_initUpDownMode() function as the param parameter	528
Timer_A_initUpModeParam	Used in the Timer_A_initUpMode() function as the param parameter	530
Timer_A_outputPWMParam	Used in the Timer_A_outputPWM() function as the param parameter	532
Timer_B_initCaptureModeParam	Used in the Timer_B_initCaptureMode() function as the param parameter	535
Timer_B_initCompareModeParam	Used in the Timer_B_initCompareMode() function as the param parameter	537
Timer_B_initContinuousModeParam	Used in the Timer_B_initContinuousMode() function as the param parameter	538
Timer_B_initUpDownModeParam	Used in the Timer_B_initUpDownMode() function as the param parameter	540
Timer_B_initUpModeParam	Used in the Timer_B_initUpMode() function as the param parameter	542
Timer_B_outputPWMParam	Used in the Timer_B_outputPWM() function as the param parameter	545
Timer_D_combineTDCCRToOutputPWMParam	Used in the Timer_D_combineTDCCRToOutputPWM() function as the param parameter	547
Timer_D_initCaptureModeParam	Used in the Timer_D_initCaptureMode() function as the param parameter	549
Timer_D_initCompareModeParam	Used in the Timer_D_initCompareMode() function as the param parameter	552
Timer_D_initContinuousModeParam	Used in the Timer_D_initContinuousMode() function as the param parameter	553
Timer_D_initHighResGeneratorInRegulatedModeParam	Used in the Timer_D_initHighResGeneratorInRegulatedMode() function as the param parameter	555
Timer_D_initUpDownModeParam	Used in the Timer_D_initUpDownMode() function as the param parameter	558
Timer_D_initUpModeParam	Used in the Timer_D_initUpMode() function as the param parameter	560
Timer_D_outputPWMParam	Used in the Timer_D_outputPWM() function as the param parameter	563
USCI_A_SPI_changeMasterClockParam	Used in the USCI_A_SPI_changeMasterClock() function as the param parameter	566
USCI_A_SPI_initMasterParam	Used in the USCI_A_SPI_initMaster() function as the param parameter	567

USCI_A_UART_initParam	
Used in the USCI_A_UART_init() function as the param parameter	568
USCI_B_I2C_initMasterParam	
Used in the USCI_B_I2C_initMaster() function as the param parameter	570
USCI_B_SPI_changeMasterClockParam	
Used in the USCI_B_SPI_changeMasterClock() function as the param parameter	571
USCI_B_SPI_initMasterParam	
Used in the USCI_B_SPI_initMaster() function as the param parameter	572

44.2 ADC12_A_configureMemoryParam Struct Reference

Used in the [ADC12_A_configureMemory\(\)](#) function as the param parameter.

```
#include <adc12_a.h>
```

Data Fields

- `uint8_t` [memoryBufferControlIndex](#)
- `uint8_t` [inputSourceSelect](#)
- `uint8_t` [positiveRefVoltageSourceSelect](#)
- `uint8_t` [negativeRefVoltageSourceSelect](#)
- `uint8_t` [endOfSequence](#)

44.2.1 Detailed Description

Used in the [ADC12_A_configureMemory\(\)](#) function as the param parameter.

44.2.2 Field Documentation

`uint8_t` [ADC12_A_configureMemoryParam::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]
- **ADC12_A_ENDOFSEQUENCE**

Referenced by [ADC12_A_configureMemory\(\)](#).

`uint8_t` [ADC12_A_configureMemoryParam::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**

Referenced by `ADC12_A.configureMemory()`.

`uint8_t ADC12_A.configureMemoryParam::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**

Referenced by `ADC12_A.configureMemory()`.

uint8_t ADC12_A_configureMemoryParam::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**

Referenced by ADC12_A_configureMemory().

uint8_t ADC12_A_configureMemoryParam::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**

Referenced by ADC12_A_configureMemory().

The documentation for this struct was generated from the following file:

- `adc12_a.h`

44.3 Calendar Struct Reference

Used in the [RTC_A_initCalendar\(\)](#) function as the CalendarTime parameter.

```
#include <rtc_a.h>
```

Data Fields

- `uint8_t Seconds`
Seconds of minute between 0-59.
- `uint8_t Minutes`
Minutes of hour between 0-59.
- `uint8_t Hours`
Hour of day between 0-23.
- `uint8_t DayOfWeek`
Day of week between 0-6.
- `uint8_t DayOfMonth`
Day of month between 1-31.
- `uint8_t Month`
Month between 0-11.
- `uint16_t Year`
Year between 0-4095.

44.3.1 Detailed Description

Used in the [RTC_A_initCalendar\(\)](#) function as the CalendarTime parameter.

Used in the [RTC_C_initCalendar\(\)](#) function as the CalendarTime parameter.

Used in the [RTC_B_initCalendar\(\)](#) function as the CalendarTime parameter.

The documentation for this struct was generated from the following files:

- [rtc_a.h](#)
- [rtc_b.h](#)
- [rtc_c.h](#)

44.4 Comp_B_configureReferenceVoltageParam Struct Reference

Used in the [Comp_B_configureReferenceVoltage\(\)](#) function as the param parameter.

```
#include <comp_b.h>
```

Data Fields

- [uint16_t supplyVoltageReferenceBase](#)
- [uint16_t lowerLimitSupplyVoltageFractionOf32](#)
- [uint16_t upperLimitSupplyVoltageFractionOf32](#)
- [uint16_t referenceAccuracy](#)

44.4.1 Detailed Description

Used in the [Comp_B_configureReferenceVoltage\(\)](#) function as the param parameter.

44.4.2 Field Documentation

`uint16_t Comp_B_configureReferenceVoltageParam::lowerLimitSupplyVoltageFractionOf32`

Is the numerator of the equation to generate the reference voltage for the lower limit reference voltage.

Referenced by [Comp_B_configureReferenceVoltage\(\)](#).

`uint16_t Comp_B_configureReferenceVoltageParam::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**

Referenced by `Comp_B_configureReferenceVoltage()`.

uint16_t `Comp_B_configureReferenceVoltageParam::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**

Referenced by `Comp_B_configureReferenceVoltage()`.

uint16_t `Comp_B_configureReferenceVoltageParam::upperLimitSupplyVoltageFractionOf32`

Is the numerator of the equation to generate the reference voltage for the upper limit reference voltage.

Referenced by `Comp_B_configureReferenceVoltage()`.

The documentation for this struct was generated from the following file:

- `comp.b.h`

44.5 `Comp_B_initParam` Struct Reference

Used in the `Comp_B_init()` function as the param parameter.

```
#include <comp_b.h>
```

Data Fields

- uint8_t [positiveTerminalInput](#)
- uint8_t [negativeTerminalInput](#)
- uint16_t [powerModeSelect](#)
- uint8_t [outputFilterEnableAndDelayLevel](#)
- uint16_t [invertedOutputPolarity](#)

44.5.1 Detailed Description

Used in the `Comp_B_init()` function as the param parameter.

44.5.2 Field Documentation

uint16_t Comp_B_initParam::invertedOutputPolarity

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

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

Referenced by Comp_B_init().

uint8_t Comp_B_initParam::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**

Referenced by Comp_B_init().

uint8_t Comp_B_initParam::outputFilterEnableAndDelayLevel

Controls the output filter delay state, which is either off or enabled with a specified delay level. This parameter is device specific and delay levels should be found in the device's datasheet.
Valid values are:

- **COMP_B_FILTEROUTPUT_OFF** [Default]
- **COMP_B_FILTEROUTPUT_DLYLVL1**

- **COMP_B_FILTEROUTPUT_DLYLVL2**
- **COMP_B_FILTEROUTPUT_DLYLVL3**
- **COMP_B_FILTEROUTPUT_DLYLVL4**

Referenced by `Comp_B_init()`.

`uint8_t Comp_B_initParam::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**

Referenced by `Comp_B_init()`.

`uint16_t Comp_B_initParam::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**

Referenced by `Comp_B_init()`.

The documentation for this struct was generated from the following file:

- `comp_b.h`

44.6 DAC12_A_initParam Struct Reference

Used in the [DAC12_A_init\(\)](#) function as the param parameter.

```
#include <dac12_a.h>
```

Data Fields

- [uint8_t submoduleSelect](#)
- [uint16_t outputSelect](#)
- [uint16_t positiveReferenceVoltage](#)
- [uint16_t outputVoltageMultiplier](#)
- [uint8_t amplifierSetting](#)
- [uint16_t conversionTriggerSelect](#)

44.6.1 Detailed Description

Used in the [DAC12_A_init\(\)](#) function as the param parameter.

44.6.2 Field Documentation

uint8_t DAC12_A_initParam::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]
- **DAC12_A_AMP_OFF_PINOUTLOW**
- **DAC12_A_AMP_LOWIN_LOWOUT**
- **DAC12_A_AMP_LOWIN_MEDOUT**
- **DAC12_A_AMP_LOWIN_HIGHOUT**
- **DAC12_A_AMP_MEDIN_MEDOUT**
- **DAC12_A_AMP_MEDIN_HIGHOUT**
- **DAC12_A_AMP_HIGHIN_HIGHOUT**

Referenced by [DAC12_A_init\(\)](#).

uint16_t DAC12_A_initParam::conversionTriggerSelect

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

- **DAC12_A_TRIGGER_ENCBYPASS** [Default]
- **DAC12_A_TRIGGER_ENC**
- **DAC12_A_TRIGGER_TA**

- **DAC12_A_TRIGGER_TB**

Referenced by DAC12_A_init().

uint16_t DAC12_A_initParam::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**

Referenced by DAC12_A_init().

uint16_t DAC12_A_initParam::outputVoltageMultiplier

Is the multiplier of the Vout voltage.
Valid values are:

- **DAC12_A_VREFx1** [Default]
- **DAC12_A_VREFx2**
- **DAC12_A_VREFx3**

Referenced by DAC12_A_init().

uint16_t DAC12_A_initParam::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**

Referenced by DAC12_A_init().

uint8_t DAC12_A_initParam::submoduleSelect

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

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

Referenced by DAC12_A_init().

The documentation for this struct was generated from the following file:

- dac12_a.h

44.7 DMA_initParam Struct Reference

Used in the [DMA_init\(\)](#) function as the param parameter.

```
#include <dma.h>
```

Data Fields

- [uint8_t channelSelect](#)
- [uint16_t transferModeSelect](#)
- [uint16_t transferSize](#)
- [uint8_t triggerSourceSelect](#)
- [uint8_t transferUnitSelect](#)
- [uint8_t triggerTypeSelect](#)

44.7.1 Detailed Description

Used in the [DMA_init\(\)](#) function as the param parameter.

44.7.2 Field Documentation

uint8_t DMA_initParam::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**

Referenced by [DMA_init\(\)](#).

uint16_t DMA_initParam::transferModeSelect

Is the transfer mode of the selected channel.

Valid values are:

- **DMA_TRANSFER_SINGLE** [Default]
- **DMA_TRANSFER_BLOCK**
- **DMA_TRANSFER_BURSTBLOCK**

- **DMA_TRANSFER_REPEATED_SINGLE**
- **DMA_TRANSFER_REPEATED_BLOCK**
- **DMA_TRANSFER_REPEATED_BURSTBLOCK**

Referenced by DMA_init().

uint16_t DMA_initParam::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.

Referenced by DMA_init().

uint8_t DMA_initParam::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**

Referenced by DMA_init().

uint8_t DMA_initParam::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`

Referenced by `DMA_init()`.

`uint8_t DMA_initParam::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`

Referenced by `DMA_init()`.

The documentation for this struct was generated from the following file:

- `dma.h`

44.8 EUSCI_A_SPI_changeMasterClockParam Struct Reference

Used in the [EUSCI_A_SPI_changeMasterClock\(\)](#) function as the `param` parameter.

```
#include <eusci_a_spi.h>
```

Data Fields

- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.

44.8.1 Detailed Description

Used in the [EUSCI_A_SPI.changeMasterClock\(\)](#) function as the param parameter.

The documentation for this struct was generated from the following file:

- `eusci_a_spi.h`

44.9 EUSCI_A_SPI_initMasterParam Struct Reference

Used in the [EUSCI_A_SPI.initMaster\(\)](#) function as the param parameter.

```
#include <eusci_a_spi.h>
```

Data Fields

- `uint8_t` [selectClockSource](#)
- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.
- `uint16_t` [msbFirst](#)
- `uint16_t` [clockPhase](#)
- `uint16_t` [clockPolarity](#)
- `uint16_t` [spiMode](#)

44.9.1 Detailed Description

Used in the [EUSCI_A_SPI.initMaster\(\)](#) function as the param parameter.

44.9.2 Field Documentation

`uint16_t` `EUSCI_A_SPI_initMasterParam::clockPhase`

Is clock phase select.
Valid values are:

- `EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT` [Default]

- **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

Referenced by EUSCI_A_SPI_initMaster().

uint16_t EUSCI_A_SPI_initMasterParam::clockPolarity

Is clock polarity select

Valid values are:

- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by EUSCI_A_SPI_initMaster().

uint16_t EUSCI_A_SPI_initMasterParam::msbFirst

Controls the direction of the receive and transmit shift register.

Valid values are:

- **EUSCI_A_SPI_MSB_FIRST**
- **EUSCI_A_SPI_LSB_FIRST** [Default]

Referenced by EUSCI_A_SPI_initMaster().

uint8_t EUSCI_A_SPI_initMasterParam::selectClockSource

Selects Clock source.

Valid values are:

- **EUSCI_A_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_A_SPI_CLOCKSOURCE_SMCLK**

Referenced by EUSCI_A_SPI_initMaster().

uint16_t EUSCI_A_SPI_initMasterParam::spiMode

Is SPI mode select

Valid values are:

- **EUSCI_A_SPI_3PIN**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Referenced by EUSCI_A_SPI_initMaster().

The documentation for this struct was generated from the following file:

- eusci_a_spi.h

44.10 EUSCI_A_SPI_initSlaveParam Struct Reference

Used in the [EUSCI_A_SPI_initSlave\(\)](#) function as the param parameter.

```
#include <eusci_a_spi.h>
```

Data Fields

- uint16_t [msbFirst](#)
- uint16_t [clockPhase](#)
- uint16_t [clockPolarity](#)
- uint16_t [spiMode](#)

44.10.1 Detailed Description

Used in the [EUSCI_A_SPI_initSlave\(\)](#) function as the param parameter.

44.10.2 Field Documentation

uint16_t EUSCI_A_SPI_initSlaveParam::clockPhase

Is clock phase select.

Valid values are:

- **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

Referenced by [EUSCI_A_SPI_initSlave\(\)](#).

uint16_t EUSCI_A_SPI_initSlaveParam::clockPolarity

Is clock polarity select

Valid values are:

- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by [EUSCI_A_SPI_initSlave\(\)](#).

uint16_t EUSCI_A_SPI_initSlaveParam::msbFirst

Controls the direction of the receive and transmit shift register.

Valid values are:

- **EUSCI_A_SPI_MSB_FIRST**

- **EUSCI_A_SPI_LSB_FIRST** [Default]

Referenced by `EUSCI_A_SPI_initSlave()`.

`uint16_t EUSCI_A_SPI_initSlaveParam::spiMode`

Is SPI mode select

Valid values are:

- **EUSCI_A_SPI_3PIN**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_A_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Referenced by `EUSCI_A_SPI_initSlave()`.

The documentation for this struct was generated from the following file:

- `eusci_a_spi.h`

44.11 EUSCI_A_UART_initParam Struct Reference

Used in the `EUSCI_A_UART_init()` function as the param parameter.

```
#include <eusci_a_uart.h>
```

Data Fields

- `uint8_t selectClockSource`
- `uint16_t clockPrescalar`
Is the value to be written into UCBRx bits.
- `uint8_t firstModReg`
- `uint8_t secondModReg`
- `uint8_t parity`
- `uint16_t msborLsbFirst`
- `uint16_t numberOfStopBits`
- `uint16_t uartMode`
- `uint8_t overSampling`

44.11.1 Detailed Description

Used in the `EUSCI_A_UART_init()` function as the param parameter.

44.11.2 Field Documentation

`uint8_t EUSCI_A_UART_initParam::firstModReg`

Is First modulation stage register setting. This value is a pre- calculated value which can be obtained from the Device Users Guide. This value is written into UCBRFx bits of UCxMCTLW.

Referenced by EUSCI_A_UART_init().

uint16_t EUSCI_A_UART_initParam::msborLsbFirst

Controls direction of receive and transmit shift register.
Valid values are:

- **EUSCI_A_UART_MSB_FIRST**
- **EUSCI_A_UART_LSB_FIRST** [Default]

Referenced by EUSCI_A_UART_init().

uint16_t EUSCI_A_UART_initParam::numberOfStopBits

Indicates one/two STOP bits
Valid values are:

- **EUSCI_A_UART_ONE_STOP_BIT** [Default]
- **EUSCI_A_UART_TWO_STOP_BITS**

Referenced by EUSCI_A_UART_init().

uint8_t EUSCI_A_UART_initParam::overSampling

Indicates low frequency or oversampling baud generation
Valid values are:

- **EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION**
- **EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

Referenced by EUSCI_A_UART_init().

uint8_t EUSCI_A_UART_initParam::parity

Is the desired parity.
Valid values are:

- **EUSCI_A_UART_NO_PARITY** [Default]
- **EUSCI_A_UART_ODD_PARITY**
- **EUSCI_A_UART_EVEN_PARITY**

Referenced by EUSCI_A_UART_init().

uint8_t EUSCI_A_UART_initParam::secondModReg

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

Referenced by EUSCI_A_UART_init().

uint8_t EUSCI_A_UART_initParam::selectClockSource

Selects Clock source.

Valid values are:

- **EUSCI_A_UART_CLOCKSOURCE_SMCLK**
- **EUSCI_A_UART_CLOCKSOURCE_ACLK**

Referenced by EUSCI_A_UART_init().

uint16_t EUSCI_A_UART_initParam::uartMode

Selects the mode of operation

Valid values are:

- **EUSCI_A_UART_MODE** [Default]
- **EUSCI_A_UART_IDLE_LINE_MULTI_PROCESSOR_MODE**
- **EUSCI_A_UART_ADDRESS_BIT_MULTI_PROCESSOR_MODE**
- **EUSCI_A_UART_AUTOMATIC_BAUDRATE_DETECTION_MODE**

Referenced by EUSCI_A_UART_init().

The documentation for this struct was generated from the following file:

- `eusci_a_uart.h`

44.12 EUSCI_B_I2C_initMasterParam Struct Reference

Used in the [EUSCI_B_I2C_initMaster\(\)](#) function as the param parameter.

```
#include <eusci_b_i2c.h>
```

Data Fields

- `uint8_t` [selectClockSource](#)
- `uint32_t` [i2cClk](#)
- `uint32_t` [dataRate](#)
- `uint8_t` [byteCounterThreshold](#)
 - Sets threshold for automatic STOP or UCSTPIFG.*
- `uint8_t` [autoSTOPGeneration](#)

44.12.1 Detailed Description

Used in the `EUSCI_B_I2C_initMaster()` function as the `param` parameter.

44.12.2 Field Documentation

`uint8_t EUSCI_B_I2C_initMasterParam::autoSTOPGeneration`

Sets up the STOP condition generation.

Valid values are:

- `EUSCI_B_I2C_NO_AUTO_STOP`
- `EUSCI_B_I2C_SET_BYTECOUNT_THRESHOLD_FLAG`
- `EUSCI_B_I2C_SEND_STOP_AUTOMATICALLY_ON_BYTECOUNT_THRESHOLD`

Referenced by `EUSCI_B_I2C_initMaster()`.

`uint32_t EUSCI_B_I2C_initMasterParam::dataRate`

Setup for selecting data transfer rate.

Valid values are:

- `EUSCI_B_I2C_SET_DATA_RATE_400KBPS`
- `EUSCI_B_I2C_SET_DATA_RATE_100KBPS`

Referenced by `EUSCI_B_I2C_initMaster()`.

`uint32_t EUSCI_B_I2C_initMasterParam::i2cClk`

Is the rate of the clock supplied to the I2C module (the frequency in Hz of the clock source specified in `selectClockSource`).

Referenced by `EUSCI_B_I2C_initMaster()`.

`uint8_t EUSCI_B_I2C_initMasterParam::selectClockSource`

Is the clocksource.

Valid values are:

- `EUSCI_B_I2C_CLOCKSOURCE_ACLK`
- `EUSCI_B_I2C_CLOCKSOURCE_SMCLK`

Referenced by `EUSCI_B_I2C_initMaster()`.

The documentation for this struct was generated from the following file:

- `eusci_b_i2c.h`

44.13 EUSCI_B_I2C_initSlaveParam Struct Reference

Used in the [EUSCI_B_I2C_initSlave\(\)](#) function as the param parameter.

```
#include <eusci_b_i2c.h>
```

Data Fields

- `uint8_t slaveAddress`
7-bit slave address
- `uint8_t slaveAddressOffset`
- `uint32_t slaveOwnAddressEnable`

44.13.1 Detailed Description

Used in the [EUSCI_B_I2C_initSlave\(\)](#) function as the param parameter.

44.13.2 Field Documentation

`uint8_t EUSCI_B_I2C_initSlaveParam::slaveAddressOffset`

Own address Offset referred to- 'x' value of UCBxI2COAx.
Valid values are:

- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET0**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET1**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET2**
- **EUSCI_B_I2C_OWN_ADDRESS_OFFSET3**

Referenced by [EUSCI_B_I2C_initSlave\(\)](#).

`uint32_t EUSCI_B_I2C_initSlaveParam::slaveOwnAddressEnable`

Selects if the specified address is enabled or disabled.
Valid values are:

- **EUSCI_B_I2C_OWN_ADDRESS_DISABLE**
- **EUSCI_B_I2C_OWN_ADDRESS_ENABLE**

Referenced by [EUSCI_B_I2C_initSlave\(\)](#).

The documentation for this struct was generated from the following file:

- `eusci_b_i2c.h`

44.14 EUSCI_B_SPI_changeMasterClockParam Struct Reference

Used in the [EUSCI_B_SPI_changeMasterClock\(\)](#) function as the param parameter.

```
#include <eusci_b_spi.h>
```

Data Fields

- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.

44.14.1 Detailed Description

Used in the [EUSCI_B_SPI_changeMasterClock\(\)](#) function as the param parameter.

The documentation for this struct was generated from the following file:

- `eusci_b_spi.h`

44.15 EUSCI_B_SPI_initMasterParam Struct Reference

Used in the [EUSCI_B_SPI_initMaster\(\)](#) function as the param parameter.

```
#include <eusci_b_spi.h>
```

Data Fields

- `uint8_t` [selectClockSource](#)
- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.
- `uint16_t` [msbFirst](#)
- `uint16_t` [clockPhase](#)
- `uint16_t` [clockPolarity](#)
- `uint16_t` [spiMode](#)

44.15.1 Detailed Description

Used in the [EUSCI_B_SPI_initMaster\(\)](#) function as the param parameter.

44.15.2 Field Documentation

uint16_t EUSCI_B_SPI_initMasterParam::clockPhase

Is clock phase select.
Valid values are:

- **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

Referenced by EUSCI_B_SPI_initMaster().

uint16_t EUSCI_B_SPI_initMasterParam::clockPolarity

Is clock polarity select
Valid values are:

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

Referenced by EUSCI_B_SPI_initMaster().

uint16_t EUSCI_B_SPI_initMasterParam::msbFirst

Controls the direction of the receive and transmit shift register.
Valid values are:

- **EUSCI_B_SPI_MSB_FIRST**
- **EUSCI_B_SPI_LSB_FIRST** [Default]

Referenced by EUSCI_B_SPI_initMaster().

uint8_t EUSCI_B_SPI_initMasterParam::selectClockSource

Selects Clock source.
Valid values are:

- **EUSCI_B_SPI_CLOCKSOURCE_ACLK**
- **EUSCI_B_SPI_CLOCKSOURCE_SMCLK**

Referenced by EUSCI_B_SPI_initMaster().

uint16_t EUSCI_B_SPI_initMasterParam::spiMode

Is SPI mode select
Valid values are:

- **EUSCI_B_SPI_3PIN**

- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Referenced by `EUSCI_B_SPI_initMaster()`.

The documentation for this struct was generated from the following file:

- `eusci_b_spi.h`

44.16 EUSCI_B_SPI_initSlaveParam Struct Reference

Used in the `EUSCI_B_SPI_initSlave()` function as the param parameter.

```
#include <eusci_b_spi.h>
```

Data Fields

- `uint16_t` [msbFirst](#)
- `uint16_t` [clockPhase](#)
- `uint16_t` [clockPolarity](#)
- `uint16_t` [spiMode](#)

44.16.1 Detailed Description

Used in the `EUSCI_B_SPI_initSlave()` function as the param parameter.

44.16.2 Field Documentation

`uint16_t` `EUSCI_B_SPI_initSlaveParam::clockPhase`

Is clock phase select.
Valid values are:

- **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default]
- **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**

Referenced by `EUSCI_B_SPI_initSlave()`.

`uint16_t` `EUSCI_B_SPI_initSlaveParam::clockPolarity`

Is clock polarity select
Valid values are:

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

Referenced by `EUSCI_B_SPI_initSlave()`.

uint16_t EUSCI_B_SPI_initSlaveParam::msbFirst

Controls the direction of the receive and transmit shift register.
Valid values are:

- **EUSCI_B_SPI_MSB_FIRST**
- **EUSCI_B_SPI_LSB_FIRST** [Default]

Referenced by EUSCI_B_SPI_initSlave().

uint16_t EUSCI_B_SPI_initSlaveParam::spiMode

Is SPI mode select
Valid values are:

- **EUSCI_B_SPI_3PIN**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_HIGH**
- **EUSCI_B_SPI_4PIN_UCxSTE_ACTIVE_LOW**

Referenced by EUSCI_B_SPI_initSlave().

The documentation for this struct was generated from the following file:

- eusci_b_spi.h

44.17 PMAP_initPortsParam Struct Reference

Used in the [PMAP_initPorts\(\)](#) function as the param parameter.

```
#include <pmap.h>
```

Data Fields

- const uint8_t * [portMapping](#)
Is the pointer to init Data.
- uint8_t * [PxMAPy](#)
Is the pointer start of first PMAP to initialize.
- uint8_t [numberOfPorts](#)
Is the number of Ports to initialize.
- uint8_t [portMapReconfigure](#)

44.17.1 Detailed Description

Used in the [PMAP_initPorts\(\)](#) function as the param parameter.

44.17.2 Field Documentation

uint8_t PMAP_initPortsParam::portMapReconfigure

Is used to enable/disable reconfiguration
Valid values are:

- **PMAP_ENABLE_RECONFIGURATION**
- **PMAP_DISABLE_RECONFIGURATION** [Default]

Referenced by PMAP_initPorts().

The documentation for this struct was generated from the following file:

- pmap.h

44.18 RTC_A_configureCalendarAlarmParam Struct Reference

Used in the [RTC_A_configureCalendarAlarm\(\)](#) function as the param parameter.

```
#include <rtc_a.h>
```

Data Fields

- uint8_t [minutesAlarm](#)
- uint8_t [hoursAlarm](#)
- uint8_t [dayOfWeekAlarm](#)
- uint8_t [dayOfMonthAlarm](#)

44.18.1 Detailed Description

Used in the [RTC_A_configureCalendarAlarm\(\)](#) function as the param parameter.

44.18.2 Field Documentation

uint8_t RTC_A_configureCalendarAlarmParam::dayOfMonthAlarm

Is the alarm condition for the day of the month.
Valid values are:

- **RTC_A_ALARMCONDITION_OFF** [Default]

Referenced by RTC_A_configureCalendarAlarm().

uint8_t RTC_A_configureCalendarAlarmParam::dayOfWeekAlarm

Is the alarm condition for the day of week.

Valid values are:

- **RTC_A_ALARMCONDITION_OFF** [Default]

Referenced by `RTC_A_configureCalendarAlarm()`.

uint8_t RTC_A_configureCalendarAlarmParam::hoursAlarm

Is the alarm condition for the hours.

Valid values are:

- **RTC_A_ALARMCONDITION_OFF** [Default]

Referenced by `RTC_A_configureCalendarAlarm()`.

uint8_t RTC_A_configureCalendarAlarmParam::minutesAlarm

Is the alarm condition for the minutes.

Valid values are:

- **RTC_A_ALARMCONDITION_OFF** [Default]

Referenced by `RTC_A_configureCalendarAlarm()`.

The documentation for this struct was generated from the following file:

- `rtc.a.h`

44.19 RTC_B_configureCalendarAlarmParam Struct Reference

Used in the [RTC_B_configureCalendarAlarm\(\)](#) function as the param parameter.

```
#include <rtc.b.h>
```

Data Fields

- uint8_t [minutesAlarm](#)
- uint8_t [hoursAlarm](#)
- uint8_t [dayOfWeekAlarm](#)
- uint8_t [dayOfMonthAlarm](#)

44.19.1 Detailed Description

Used in the [RTC_B_configureCalendarAlarm\(\)](#) function as the param parameter.

44.19.2 Field Documentation

uint8_t RTC_B_configureCalendarAlarmParam::dayOfMonthAlarm

Is the alarm condition for the day of the month.
Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]

Referenced by RTC_B_configureCalendarAlarm().

uint8_t RTC_B_configureCalendarAlarmParam::dayOfWeekAlarm

Is the alarm condition for the day of week.
Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]

Referenced by RTC_B_configureCalendarAlarm().

uint8_t RTC_B_configureCalendarAlarmParam::hoursAlarm

Is the alarm condition for the hours.
Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]

Referenced by RTC_B_configureCalendarAlarm().

uint8_t RTC_B_configureCalendarAlarmParam::minutesAlarm

Is the alarm condition for the minutes.
Valid values are:

- **RTC_B_ALARMCONDITION_OFF** [Default]

Referenced by RTC_B_configureCalendarAlarm().

The documentation for this struct was generated from the following file:

- rtc.b.h

44.20 RTC_C_configureCalendarAlarmParam Struct Reference

Used in the [RTC_C_configureCalendarAlarm\(\)](#) function as the param parameter.

```
#include <rtc.c.h>
```

Data Fields

- uint8_t [minutesAlarm](#)
- uint8_t [hoursAlarm](#)
- uint8_t [dayOfWeekAlarm](#)
- uint8_t [dayOfMonthAlarm](#)

44.20.1 Detailed Description

Used in the [RTC_C_configureCalendarAlarm\(\)](#) function as the param parameter.

44.20.2 Field Documentation

uint8_t RTC_C_configureCalendarAlarmParam::dayOfMonthAlarm

Is the alarm condition for the day of the month.
Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]

Referenced by [RTC_C_configureCalendarAlarm\(\)](#).

uint8_t RTC_C_configureCalendarAlarmParam::dayOfWeekAlarm

Is the alarm condition for the day of week.
Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]

Referenced by [RTC_C_configureCalendarAlarm\(\)](#).

uint8_t RTC_C_configureCalendarAlarmParam::hoursAlarm

Is the alarm condition for the hours.
Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]

Referenced by [RTC_C_configureCalendarAlarm\(\)](#).

uint8_t RTC_C_configureCalendarAlarmParam::minutesAlarm

Is the alarm condition for the minutes.
Valid values are:

- **RTC_C_ALARMCONDITION_OFF** [Default]

Referenced by `RTC_C_configureCalendarAlarm()`.

The documentation for this struct was generated from the following file:

- `rtc.c.h`

44.21 SD24_B_initConverterAdvancedParam Struct Reference

Used in the `SD24_B_initConverterAdvanced()` function as the param parameter.

```
#include <sd24_b.h>
```

Data Fields

- `uint8_t converter`
- `uint8_t alignment`
- `uint8_t startSelect`
- `uint8_t conversionMode`
- `uint8_t dataFormat`
- `uint8_t sampleDelay`
- `uint16_t oversampleRatio`
- `uint8_t gain`

44.21.1 Detailed Description

Used in the `SD24_B_initConverterAdvanced()` function as the param parameter.

44.21.2 Field Documentation

`uint8_t SD24_B_initConverterAdvancedParam::alignment`

Selects how the data will be aligned in result
Valid values are:

- `SD24_B_ALIGN_RIGHT` [Default]
- `SD24_B_ALIGN_LEFT`

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::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**

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::converter`

Selects the converter that will be configured. 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**

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::dataFormat`

Selects how the data format of the results
Valid values are:

- **SD24_B_DATA_FORMAT_BINARY** [Default]
- **SD24_B_DATA_FORMAT_2COMPLEMENT**

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::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**

Referenced by `SD24_B_initConverterAdvanced()`.

`uint16_t SD24_B_initConverterAdvancedParam::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`

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::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`

Referenced by `SD24_B_initConverterAdvanced()`.

`uint8_t SD24_B_initConverterAdvancedParam::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`

Referenced by `SD24_B_initConverterAdvanced()`.

The documentation for this struct was generated from the following file:

- `sd24_b.h`

44.22 SD24_B_initConverterParam Struct Reference

Used in the [SD24_B_initConverter\(\)](#) function as the param parameter.

```
#include <sd24_b.h>
```

Data Fields

- uint8_t [converter](#)
- uint8_t [alignment](#)
- uint8_t [startSelect](#)
- uint8_t [conversionMode](#)

44.22.1 Detailed Description

Used in the [SD24_B_initConverter\(\)](#) function as the param parameter.

44.22.2 Field Documentation

uint8_t SD24_B_initConverterParam::alignment

Selects how the data will be aligned in result
Valid values are:

- **SD24_B_ALIGN_RIGHT** [Default]
- **SD24_B_ALIGN_LEFT**

Referenced by [SD24_B_initConverter\(\)](#).

uint8_t SD24_B_initConverterParam::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**

Referenced by [SD24_B_initConverter\(\)](#).

uint8_t SD24_B_initConverterParam::converter

Selects the converter that will be configured. 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**

Referenced by `SD24_B_initConverter()`.

uint8_t SD24_B_initConverterParam::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**

Referenced by `SD24_B_initConverter()`.

The documentation for this struct was generated from the following file:

- `sd24_b.h`

44.23 SD24_B_initParam Struct Reference

Used in the [SD24_B.init\(\)](#) function as the param parameter.

```
#include <sd24_b.h>
```

Data Fields

- uint16_t [clockSourceSelect](#)
- uint16_t [clockPreDivider](#)
- uint16_t [clockDivider](#)
- uint16_t [referenceSelect](#)

44.23.1 Detailed Description

Used in the [SD24_B.init\(\)](#) function as the param parameter.

44.23.2 Field Documentation

uint16_t SD24_B_initParam::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**

Referenced by SD24_B_init().

uint16_t SD24_B_initParam::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**

Referenced by SD24_B_init().

uint16_t SD24_B_initParam::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**

Referenced by SD24_B_init().

uint16_t SD24_B_initParam::referenceSelect

Selects the reference source for the SD24_B core
Valid values are:

- **SD24_B_REF_EXTERNAL** [Default]
- **SD24_B_REF_INTERNAL**

Referenced by SD24_B_init().

The documentation for this struct was generated from the following file:

- sd24_b.h

44.24 Timer_A_initCaptureModeParam Struct Reference

Used in the [Timer_A_initCaptureMode\(\)](#) function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- uint16_t [captureRegister](#)
- uint16_t [captureMode](#)
- uint16_t [captureInputSelect](#)
- uint16_t [synchronizeCaptureSource](#)
- uint16_t [captureInterruptEnable](#)
- uint16_t [captureOutputMode](#)

44.24.1 Detailed Description

Used in the [Timer_A.initCaptureMode\(\)](#) function as the param parameter.

44.24.2 Field Documentation

uint16_t Timer_A.initCaptureModeParam::captureInputSelect

Decides the Input Select
Valid values are:

- **TIMER_A_CAPTURE_INPUTSELECT_CC1xA**
- **TIMER_A_CAPTURE_INPUTSELECT_CC1xB**
- **TIMER_A_CAPTURE_INPUTSELECT_GND**
- **TIMER_A_CAPTURE_INPUTSELECT_Vcc**

Referenced by [Timer_A.initCaptureMode\(\)](#).

uint16_t Timer_A.initCaptureModeParam::captureInterruptEnable

Is to enable or disable timer captureCompare interrupt.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by [Timer_A.initCaptureMode\(\)](#).

uint16_t Timer_A.initCaptureModeParam::captureMode

Is the capture mode selected.
Valid values are:

- **TIMER_A_CAPTUREMODE_NO_CAPTURE** [Default]
- **TIMER_A_CAPTUREMODE_RISING_EDGE**
- **TIMER_A_CAPTUREMODE_FALLING_EDGE**
- **TIMER_A_CAPTUREMODE_RISING_AND_FALLING_EDGE**

Referenced by [Timer_A.initCaptureMode\(\)](#).

`uint16_t Timer_A_initCaptureModeParam::captureOutputMode`

Specifies the output mode.

Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_A_OUTPUTMODE_SET**
- **TIMER_A_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_A_OUTPUTMODE_SET_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE**
- **TIMER_A_OUTPUTMODE_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE_SET**
- **TIMER_A_OUTPUTMODE_RESET_SET**

Referenced by `Timer_A_initCaptureMode()`.

`uint16_t Timer_A_initCaptureModeParam::captureRegister`

Selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used.

Valid values are:

- **TIMER_A_CAPTURECOMPARE_REGISTER_0**
- **TIMER_A_CAPTURECOMPARE_REGISTER_1**
- **TIMER_A_CAPTURECOMPARE_REGISTER_2**
- **TIMER_A_CAPTURECOMPARE_REGISTER_3**
- **TIMER_A_CAPTURECOMPARE_REGISTER_4**
- **TIMER_A_CAPTURECOMPARE_REGISTER_5**
- **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Referenced by `Timer_A_initCaptureMode()`.

`uint16_t Timer_A_initCaptureModeParam::synchronizeCaptureSource`

Decides if capture source should be synchronized with timer clock

Valid values are:

- **TIMER_A_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_A_CAPTURE_SYNCHRONOUS**

Referenced by `Timer_A_initCaptureMode()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.25 Timer_A_initCompareModeParam Struct Reference

Used in the [Timer_A_initCompareMode\(\)](#) function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- `uint16_t compareRegister`
- `uint16_t compareInterruptEnable`
- `uint16_t compareOutputMode`
- `uint16_t compareValue`

Is the count to be compared with in compare mode.

44.25.1 Detailed Description

Used in the [Timer_A_initCompareMode\(\)](#) function as the param parameter.

44.25.2 Field Documentation

`uint16_t Timer_A_initCompareModeParam::compareInterruptEnable`

Is to enable or disable timer captureCompare interrupt.

Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by [Timer_A_initCompareMode\(\)](#).

`uint16_t Timer_A_initCompareModeParam::compareOutputMode`

Specifies the output mode.

Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_A_OUTPUTMODE_SET**
- **TIMER_A_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_A_OUTPUTMODE_SET_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE**
- **TIMER_A_OUTPUTMODE_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE_SET**
- **TIMER_A_OUTPUTMODE_RESET_SET**

Referenced by [Timer_A_initCompareMode\(\)](#).

uint16_t Timer_A_initCompareModeParam::compareRegister

Selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used.

Valid values are:

- `TIMER_A_CAPTURECOMPARE_REGISTER_0`
- `TIMER_A_CAPTURECOMPARE_REGISTER_1`
- `TIMER_A_CAPTURECOMPARE_REGISTER_2`
- `TIMER_A_CAPTURECOMPARE_REGISTER_3`
- `TIMER_A_CAPTURECOMPARE_REGISTER_4`
- `TIMER_A_CAPTURECOMPARE_REGISTER_5`
- `TIMER_A_CAPTURECOMPARE_REGISTER_6`

Referenced by `Timer_A_initCompareMode()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.26 Timer_A_initContinuousModeParam Struct Reference

Used in the `Timer_A_initContinuousMode()` function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- `uint16_t clockSource`
- `uint16_t clockSourceDivider`
- `uint16_t timerInterruptEnable_TAIE`
- `uint16_t timerClear`
- `bool startTimer`

Whether to start the timer immediately.

44.26.1 Detailed Description

Used in the `Timer_A_initContinuousMode()` function as the param parameter.

44.26.2 Field Documentation

uint16_t Timer_A_initContinuousModeParam::clockSource

Selects Clock source.

Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by `Timer_A_initContinuousMode()`.

`uint16_t Timer_A_initContinuousModeParam::clockSourceDivider`

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **TIMER_A_CLOCKSOURCE_DIVIDER_10**
- **TIMER_A_CLOCKSOURCE_DIVIDER_12**
- **TIMER_A_CLOCKSOURCE_DIVIDER_14**
- **TIMER_A_CLOCKSOURCE_DIVIDER_16**
- **TIMER_A_CLOCKSOURCE_DIVIDER_20**
- **TIMER_A_CLOCKSOURCE_DIVIDER_24**
- **TIMER_A_CLOCKSOURCE_DIVIDER_28**
- **TIMER_A_CLOCKSOURCE_DIVIDER_32**
- **TIMER_A_CLOCKSOURCE_DIVIDER_40**
- **TIMER_A_CLOCKSOURCE_DIVIDER_48**
- **TIMER_A_CLOCKSOURCE_DIVIDER_56**
- **TIMER_A_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_A_initContinuousMode()`.

`uint16_t Timer_A_initContinuousModeParam::timerClear`

Decides if `Timer_A` clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Referenced by `Timer_A_initContinuousMode()`.

uint16_t Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE

Is to enable or disable Timer_A interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_A_initContinuousMode()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.27 Timer_A_initUpDownModeParam Struct Reference

Used in the `Timer_A_initUpDownMode()` function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- uint16_t `clockSource`
- uint16_t `clockSourceDivider`
- uint16_t `timerPeriod`
Is the specified Timer_A period.
- uint16_t `timerInterruptEnable_TAIE`
- uint16_t `captureCompareInterruptEnable_CCR0_CCIE`
- uint16_t `timerClear`
- bool `startTimer`
Whether to start the timer immediately.

44.27.1 Detailed Description

Used in the `Timer_A_initUpDownMode()` function as the param parameter.

44.27.2 Field Documentation

uint16_t Timer_A_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE

Is to enable or disable Timer_A CCR0 captureComapre interrupt.
Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_A_initUpDownMode()`.

`uint16_t Timer_A_initUpDownModeParam::clockSource`

Selects Clock source.

Valid values are:

- `TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK` [Default]
- `TIMER_A_CLOCKSOURCE_ACLK`
- `TIMER_A_CLOCKSOURCE_SMCLK`
- `TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK`

Referenced by `Timer_A_initUpDownMode()`.

`uint16_t Timer_A_initUpDownModeParam::clockSourceDivider`

Is the desired divider for the clock source

Valid values are:

- `TIMER_A_CLOCKSOURCE_DIVIDER_1` [Default]
- `TIMER_A_CLOCKSOURCE_DIVIDER_2`
- `TIMER_A_CLOCKSOURCE_DIVIDER_3`
- `TIMER_A_CLOCKSOURCE_DIVIDER_4`
- `TIMER_A_CLOCKSOURCE_DIVIDER_5`
- `TIMER_A_CLOCKSOURCE_DIVIDER_6`
- `TIMER_A_CLOCKSOURCE_DIVIDER_7`
- `TIMER_A_CLOCKSOURCE_DIVIDER_8`
- `TIMER_A_CLOCKSOURCE_DIVIDER_10`
- `TIMER_A_CLOCKSOURCE_DIVIDER_12`
- `TIMER_A_CLOCKSOURCE_DIVIDER_14`
- `TIMER_A_CLOCKSOURCE_DIVIDER_16`
- `TIMER_A_CLOCKSOURCE_DIVIDER_20`
- `TIMER_A_CLOCKSOURCE_DIVIDER_24`
- `TIMER_A_CLOCKSOURCE_DIVIDER_28`
- `TIMER_A_CLOCKSOURCE_DIVIDER_32`
- `TIMER_A_CLOCKSOURCE_DIVIDER_40`
- `TIMER_A_CLOCKSOURCE_DIVIDER_48`
- `TIMER_A_CLOCKSOURCE_DIVIDER_56`
- `TIMER_A_CLOCKSOURCE_DIVIDER_64`

Referenced by `Timer_A_initUpDownMode()`.

uint16_t Timer_A_initUpDownModeParam::timerClear

Decides if Timer_A clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Referenced by `Timer_A_initUpDownMode()`.

uint16_t Timer_A_initUpDownModeParam::timerInterruptEnable_TAIE

Is to enable or disable Timer_A interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_A_initUpDownMode()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.28 Timer_A_initUpModeParam Struct Reference

Used in the [Timer_A_initUpMode\(\)](#) function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- `uint16_t` [clockSource](#)
- `uint16_t` [clockSourceDivider](#)
- `uint16_t` [timerPeriod](#)
- `uint16_t` [timerInterruptEnable_TAIE](#)
- `uint16_t` [captureCompareInterruptEnable_CCR0_CCIE](#)
- `uint16_t` [timerClear](#)
- `bool` [startTimer](#)

Whether to start the timer immediately.

44.28.1 Detailed Description

Used in the [Timer_A_initUpMode\(\)](#) function as the param parameter.

44.28.2 Field Documentation

uint16_t Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE

Is to enable or disable Timer_A CCR0 captureCompare interrupt.
Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpMode().

uint16_t Timer_A_initUpModeParam::clockSource

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_initUpMode().

uint16_t Timer_A_initUpModeParam::clockSourceDivider

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **TIMER_A_CLOCKSOURCE_DIVIDER_10**
- **TIMER_A_CLOCKSOURCE_DIVIDER_12**
- **TIMER_A_CLOCKSOURCE_DIVIDER_14**
- **TIMER_A_CLOCKSOURCE_DIVIDER_16**
- **TIMER_A_CLOCKSOURCE_DIVIDER_20**
- **TIMER_A_CLOCKSOURCE_DIVIDER_24**
- **TIMER_A_CLOCKSOURCE_DIVIDER_28**
- **TIMER_A_CLOCKSOURCE_DIVIDER_32**

- **TIMER_A_CLOCKSOURCE_DIVIDER_40**
- **TIMER_A_CLOCKSOURCE_DIVIDER_48**
- **TIMER_A_CLOCKSOURCE_DIVIDER_56**
- **TIMER_A_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_A_initUpMode()`.

`uint16_t Timer_A_initUpModeParam::timerClear`

Decides if `Timer_A` clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_A_DO_CLEAR**
- **TIMER_A_SKIP_CLEAR** [Default]

Referenced by `Timer_A_initUpMode()`.

`uint16_t Timer_A_initUpModeParam::timerInterruptEnable_TAIE`

Is to enable or disable `Timer_A` interrupt
Valid values are:

- **TIMER_A_TAIE_INTERRUPT_ENABLE**
- **TIMER_A_TAIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_A_initUpMode()`.

`uint16_t Timer_A_initUpModeParam::timerPeriod`

Is the specified `Timer_A` period. This is the value that gets written into the `CCR0`. Limited to 16 bits[`uint16_t`]

Referenced by `Timer_A_initUpMode()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.29 `Timer_A_outputPWMParm` Struct Reference

Used in the `Timer_A_outputPWM()` function as the `param` parameter.

```
#include <timer_a.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [timerPeriod](#)

Selects the desired timer period.

- uint16_t [compareRegister](#)
- uint16_t [compareOutputMode](#)
- uint16_t [dutyCycle](#)

Specifies the dutycycle for the generated waveform.

44.29.1 Detailed Description

Used in the [Timer_A_outputPWM\(\)](#) function as the param parameter.

44.29.2 Field Documentation

uint16_t Timer_A_outputPWMPParam::clockSource

Selects Clock source.

Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by [Timer_A_outputPWM\(\)](#).

uint16_t Timer_A_outputPWMPParam::clockSourceDivider

Is the desired divider for the clock source

Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **TIMER_A_CLOCKSOURCE_DIVIDER_10**
- **TIMER_A_CLOCKSOURCE_DIVIDER_12**
- **TIMER_A_CLOCKSOURCE_DIVIDER_14**

- **TIMER_A_CLOCKSOURCE_DIVIDER_16**
- **TIMER_A_CLOCKSOURCE_DIVIDER_20**
- **TIMER_A_CLOCKSOURCE_DIVIDER_24**
- **TIMER_A_CLOCKSOURCE_DIVIDER_28**
- **TIMER_A_CLOCKSOURCE_DIVIDER_32**
- **TIMER_A_CLOCKSOURCE_DIVIDER_40**
- **TIMER_A_CLOCKSOURCE_DIVIDER_48**
- **TIMER_A_CLOCKSOURCE_DIVIDER_56**
- **TIMER_A_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_A_outputPWM()`.

`uint16_t Timer_A_outputPWMPParam::compareOutputMode`

Specifies the output mode.

Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_A_OUTPUTMODE_SET**
- **TIMER_A_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_A_OUTPUTMODE_SET_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE**
- **TIMER_A_OUTPUTMODE_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE_SET**
- **TIMER_A_OUTPUTMODE_RESET_SET**

Referenced by `Timer_A_outputPWM()`.

`uint16_t Timer_A_outputPWMPParam::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**

Referenced by `Timer_A_outputPWM()`.

The documentation for this struct was generated from the following file:

- `timer_a.h`

44.30 Timer_B_initCaptureModeParam Struct Reference

Used in the [Timer_B_initCaptureMode\(\)](#) function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- [uint16_t captureRegister](#)
- [uint16_t captureMode](#)
- [uint16_t captureInputSelect](#)
- [uint16_t synchronizeCaptureSource](#)
- [uint16_t captureInterruptEnable](#)
- [uint16_t captureOutputMode](#)

44.30.1 Detailed Description

Used in the [Timer_B_initCaptureMode\(\)](#) function as the param parameter.

44.30.2 Field Documentation

`uint16_t Timer_B_initCaptureModeParam::captureInputSelect`

Decides the Input Select

Valid values are:

- **TIMER_B_CAPTURE_INPUTSELECT_CC1xA** [Default]
- **TIMER_B_CAPTURE_INPUTSELECT_CC1xB**
- **TIMER_B_CAPTURE_INPUTSELECT_GND**
- **TIMER_B_CAPTURE_INPUTSELECT_Vcc**

Referenced by [Timer_B_initCaptureMode\(\)](#).

`uint16_t Timer_B_initCaptureModeParam::captureInterruptEnable`

Is to enable or disable Timer_B capture compare interrupt.

Valid values are:

- **TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by [Timer_B_initCaptureMode\(\)](#).

`uint16_t Timer_B_initCaptureModeParam::captureMode`

Is the capture mode selected.

Valid values are:

- **TIMER_B_CAPTUREMODE_NO_CAPTURE** [Default]
- **TIMER_B_CAPTUREMODE_RISING_EDGE**
- **TIMER_B_CAPTUREMODE_FALLING_EDGE**
- **TIMER_B_CAPTUREMODE_RISING_AND_FALLING_EDGE**

Referenced by `Timer_B_initCaptureMode()`.

`uint16_t Timer_B_initCaptureModeParam::captureOutputMode`

Specifies the output mode.

Valid values are:

- **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_B_OUTPUTMODE_SET**
- **TIMER_B_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_B_OUTPUTMODE_SET_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE**
- **TIMER_B_OUTPUTMODE_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE_SET**
- **TIMER_B_OUTPUTMODE_RESET_SET**

Referenced by `Timer_B_initCaptureMode()`.

`uint16_t Timer_B_initCaptureModeParam::captureRegister`

Selects the capture register being used. Refer to datasheet to ensure the device has the capture register being used.

Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by `Timer_B_initCaptureMode()`.

uint16_t Timer_B_initCaptureModeParam::synchronizeCaptureSource

Decides if capture source should be synchronized with Timer_B clock
Valid values are:

- **TIMER_B_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_B_CAPTURE_SYNCHRONOUS**

Referenced by `Timer_B_initCaptureMode()`.

The documentation for this struct was generated from the following file:

- `timer_b.h`

44.31 Timer_B_initCompareModeParam Struct Reference

Used in the `Timer_B_initCompareMode()` function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- uint16_t [compareRegister](#)
- uint16_t [compareInterruptEnable](#)
- uint16_t [compareOutputMode](#)
- uint16_t [compareValue](#)

Is the count to be compared with in compare mode.

44.31.1 Detailed Description

Used in the `Timer_B_initCompareMode()` function as the param parameter.

44.31.2 Field Documentation

uint16_t Timer_B_initCompareModeParam::compareInterruptEnable

Is to enable or disable Timer_B capture compare interrupt.
Valid values are:

- **TIMER_B_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_B_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by `Timer_B_initCompareMode()`.

uint16_t Timer_B_initCompareModeParam::compareOutputMode

Specifies the output mode.

Valid values are:

- **TIMER_B_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_B_OUTPUTMODE_SET**
- **TIMER_B_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_B_OUTPUTMODE_SET_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE**
- **TIMER_B_OUTPUTMODE_RESET**
- **TIMER_B_OUTPUTMODE_TOGGLE_SET**
- **TIMER_B_OUTPUTMODE_RESET_SET**

Referenced by `Timer_B_initCompareMode()`.

uint16_t Timer_B_initCompareModeParam::compareRegister

Selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used.

Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by `Timer_B_initCompareMode()`.

The documentation for this struct was generated from the following file:

- `timer_b.h`

44.32 Timer_B_initContinuousModeParam Struct Reference

Used in the `Timer_B_initContinuousMode()` function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- `uint16_t clockSource`

- uint16_t [clockSourceDivider](#)
- uint16_t [timerInterruptEnable_TBIE](#)
- uint16_t [timerClear](#)
- bool [startTimer](#)

Whether to start the timer immediately.

44.32.1 Detailed Description

Used in the [Timer_B_initContinuousMode\(\)](#) function as the param parameter.

44.32.2 Field Documentation

uint16_t Timer_B_initContinuousModeParam::clockSource

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by [Timer_B_initContinuousMode\(\)](#).

uint16_t Timer_B_initContinuousModeParam::clockSourceDivider

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**

- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_B_initContinuousMode()`.

`uint16_t Timer_B_initContinuousModeParam::timerClear`

Decides if `Timer_B` clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by `Timer_B_initContinuousMode()`.

`uint16_t Timer_B_initContinuousModeParam::timerInterruptEnable_TBIE`

Is to enable or disable `Timer_B` interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_B_initContinuousMode()`.

The documentation for this struct was generated from the following file:

- `timer_b.h`

44.33 `Timer_B_initUpDownModeParam` Struct Reference

Used in the `Timer_B_initUpDownMode()` function as the `param` parameter.

```
#include <timer_b.h>
```

Data Fields

- `uint16_t clockSource`
- `uint16_t clockSourceDivider`
- `uint16_t timerPeriod`
Is the specified `Timer_B` period.
- `uint16_t timerInterruptEnable_TBIE`
- `uint16_t captureCompareInterruptEnable_CCR0_CCIE`

- uint16_t [timerClear](#)
- bool [startTimer](#)

Whether to start the timer immediately.

44.33.1 Detailed Description

Used in the [Timer_B.initUpDownMode\(\)](#) function as the param parameter.

44.33.2 Field Documentation

uint16_t [Timer_B.initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE](#)

Is to enable or disable Timer_B CCR0 capture compare interrupt.
Valid values are:

- **TIMER_B_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by [Timer_B.initUpDownMode\(\)](#).

uint16_t [Timer_B.initUpDownModeParam::clockSource](#)

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by [Timer_B.initUpDownMode\(\)](#).

uint16_t [Timer_B.initUpDownModeParam::clockSourceDivider](#)

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**
- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**

- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_B_initUpDownMode()`.

`uint16_t Timer_B_initUpDownModeParam::timerClear`

Decides if `Timer_B` clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by `Timer_B_initUpDownMode()`.

`uint16_t Timer_B_initUpDownModeParam::timerInterruptEnable_TBIE`

Is to enable or disable `Timer_B` interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_B_initUpDownMode()`.

The documentation for this struct was generated from the following file:

- `timer_b.h`

44.34 `Timer_B_initUpModeParam` Struct Reference

Used in the [`Timer_B_initUpMode\(\)`](#) function as the `param` parameter.

```
#include <timer_b.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [timerPeriod](#)
- uint16_t [timerInterruptEnable_TBIE](#)
- uint16_t [captureCompareInterruptEnable_CCR0_CCIE](#)
- uint16_t [timerClear](#)
- bool [startTimer](#)

Whether to start the timer immediately.

44.34.1 Detailed Description

Used in the [Timer_B.initUpMode\(\)](#) function as the param parameter.

44.34.2 Field Documentation

uint16_t Timer_B_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE

Is to enable or disable Timer_B CCR0 capture compare interrupt.
Valid values are:

- **TIMER_B_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_B_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by [Timer_B.initUpMode\(\)](#).

uint16_t Timer_B_initUpModeParam::clockSource

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by [Timer_B.initUpMode\(\)](#).

uint16_t Timer_B_initUpModeParam::clockSourceDivider

Is the divider for Clock source.
Valid values are:

- **TIMER_B_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_B_CLOCKSOURCE_DIVIDER_2**

- **TIMER_B_CLOCKSOURCE_DIVIDER_3**
- **TIMER_B_CLOCKSOURCE_DIVIDER_4**
- **TIMER_B_CLOCKSOURCE_DIVIDER_5**
- **TIMER_B_CLOCKSOURCE_DIVIDER_6**
- **TIMER_B_CLOCKSOURCE_DIVIDER_7**
- **TIMER_B_CLOCKSOURCE_DIVIDER_8**
- **TIMER_B_CLOCKSOURCE_DIVIDER_10**
- **TIMER_B_CLOCKSOURCE_DIVIDER_12**
- **TIMER_B_CLOCKSOURCE_DIVIDER_14**
- **TIMER_B_CLOCKSOURCE_DIVIDER_16**
- **TIMER_B_CLOCKSOURCE_DIVIDER_20**
- **TIMER_B_CLOCKSOURCE_DIVIDER_24**
- **TIMER_B_CLOCKSOURCE_DIVIDER_28**
- **TIMER_B_CLOCKSOURCE_DIVIDER_32**
- **TIMER_B_CLOCKSOURCE_DIVIDER_40**
- **TIMER_B_CLOCKSOURCE_DIVIDER_48**
- **TIMER_B_CLOCKSOURCE_DIVIDER_56**
- **TIMER_B_CLOCKSOURCE_DIVIDER_64**

Referenced by `Timer_B_initUpMode()`.

`uint16_t Timer_B_initUpModeParam::timerClear`

Decides if `Timer_B` clock divider, count direction, count need to be reset.
Valid values are:

- **TIMER_B_DO_CLEAR**
- **TIMER_B_SKIP_CLEAR** [Default]

Referenced by `Timer_B_initUpMode()`.

`uint16_t Timer_B_initUpModeParam::timerInterruptEnable_TBIE`

Is to enable or disable `Timer_B` interrupt
Valid values are:

- **TIMER_B_TBIE_INTERRUPT_ENABLE**
- **TIMER_B_TBIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_B_initUpMode()`.

uint16_t Timer_B_initUpModeParam::timerPeriod

Is the specified Timer_B period. This is the value that gets written into the CCR0. Limited to 16 bits[uint16_t]

Referenced by Timer_B_initUpMode().

The documentation for this struct was generated from the following file:

- timer_b.h

44.35 Timer_B_outputPWMPParam Struct Reference

Used in the [Timer_B_outputPWM\(\)](#) function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [timerPeriod](#)
Selects the desired Timer_B period.
- uint16_t [compareRegister](#)
- uint16_t [compareOutputMode](#)
- uint16_t [dutyCycle](#)
Specifies the dutycycle for the generated waveform.

44.35.1 Detailed Description

Used in the [Timer_B_outputPWM\(\)](#) function as the param parameter.

44.35.2 Field Documentation

uint16_t Timer_B_outputPWMPParam::clockSource

Selects the clock source

Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by [Timer_B_outputPWM\(\)](#).

`uint16_t Timer_B_outputPWMParam::clockSourceDivider`

Is the divider for Clock source.

Valid values are:

- `TIMER_B_CLOCKSOURCE_DIVIDER_1` [Default]
- `TIMER_B_CLOCKSOURCE_DIVIDER_2`
- `TIMER_B_CLOCKSOURCE_DIVIDER_3`
- `TIMER_B_CLOCKSOURCE_DIVIDER_4`
- `TIMER_B_CLOCKSOURCE_DIVIDER_5`
- `TIMER_B_CLOCKSOURCE_DIVIDER_6`
- `TIMER_B_CLOCKSOURCE_DIVIDER_7`
- `TIMER_B_CLOCKSOURCE_DIVIDER_8`
- `TIMER_B_CLOCKSOURCE_DIVIDER_10`
- `TIMER_B_CLOCKSOURCE_DIVIDER_12`
- `TIMER_B_CLOCKSOURCE_DIVIDER_14`
- `TIMER_B_CLOCKSOURCE_DIVIDER_16`
- `TIMER_B_CLOCKSOURCE_DIVIDER_20`
- `TIMER_B_CLOCKSOURCE_DIVIDER_24`
- `TIMER_B_CLOCKSOURCE_DIVIDER_28`
- `TIMER_B_CLOCKSOURCE_DIVIDER_32`
- `TIMER_B_CLOCKSOURCE_DIVIDER_40`
- `TIMER_B_CLOCKSOURCE_DIVIDER_48`
- `TIMER_B_CLOCKSOURCE_DIVIDER_56`
- `TIMER_B_CLOCKSOURCE_DIVIDER_64`

Referenced by `Timer_B_outputPWM()`.

`uint16_t Timer_B_outputPWMParam::compareOutputMode`

Specifies the output mode.

Valid values are:

- `TIMER_B_OUTPUTMODE_OUTBITVALUE` [Default]
- `TIMER_B_OUTPUTMODE_SET`
- `TIMER_B_OUTPUTMODE_TOGGLE_RESET`
- `TIMER_B_OUTPUTMODE_SET_RESET`
- `TIMER_B_OUTPUTMODE_TOGGLE`
- `TIMER_B_OUTPUTMODE_RESET`
- `TIMER_B_OUTPUTMODE_TOGGLE_SET`
- `TIMER_B_OUTPUTMODE_RESET_SET`

Referenced by `Timer_B_outputPWM()`.

uint16_t Timer_B_outputPWMParam::compareRegister

Selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used.

Valid values are:

- **TIMER_B_CAPTURECOMPARE_REGISTER_0**
- **TIMER_B_CAPTURECOMPARE_REGISTER_1**
- **TIMER_B_CAPTURECOMPARE_REGISTER_2**
- **TIMER_B_CAPTURECOMPARE_REGISTER_3**
- **TIMER_B_CAPTURECOMPARE_REGISTER_4**
- **TIMER_B_CAPTURECOMPARE_REGISTER_5**
- **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Referenced by `Timer_B_outputPWM()`.

The documentation for this struct was generated from the following file:

- `timer_b.h`

44.36 Timer_D_combineTDCCRToOutputPWMParam Struct Reference

Used in the [Timer_D_combineTDCCRToOutputPWM\(\)](#) function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [clockingMode](#)
- uint16_t [timerPeriod](#)
Is the specified timer period.
- uint16_t [combineCCRRegistersCombination](#)
- uint16_t [compareOutputMode](#)
- uint16_t [dutyCycle1](#)
Specifies the dutycycle for the generated waveform.
- uint16_t [dutyCycle2](#)
Specifies the dutycycle for the generated waveform.

44.36.1 Detailed Description

Used in the [Timer_D_combineTDCCRToOutputPWM\(\)](#) function as the param parameter.

44.36.2 Field Documentation

uint16_t Timer_D_combineTDCCRToOutputPWMParm::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**

Referenced by Timer_D_combineTDCCRToOutputPWM().

uint16_t Timer_D_combineTDCCRToOutputPWMParm::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**

Referenced by Timer_D_combineTDCCRToOutputPWM().

uint16_t Timer_D_combineTDCCRToOutputPWMParm::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**

Referenced by `Timer_D_combineTDCCRToOutputPWM()`.

`uint16_t Timer_D_combineTDCCRToOutputPWMParam::combineCCRRegistersCombination`

Selects desired CCR registers to combine
Valid values are:

- **TIMER_D_COMBINE_CCR1_CCR2**
- **TIMER_D_COMBINE_CCR3_CCR4**
- **TIMER_D_COMBINE_CCR5_CCR6**

Referenced by `Timer_D_combineTDCCRToOutputPWM()`.

`uint16_t Timer_D_combineTDCCRToOutputPWMParam::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**

Referenced by `Timer_D_combineTDCCRToOutputPWM()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.37 `Timer_D_initCaptureModeParam` Struct Reference

Used in the [`Timer_D_initCaptureMode\(\)`](#) function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- uint16_t [captureRegister](#)
- uint16_t [captureMode](#)
- uint16_t [captureInputSelect](#)
- uint16_t [synchronizeCaptureSource](#)
- uint16_t [captureInterruptEnable](#)
- uint16_t [captureOutputMode](#)
- uint8_t [channelCaptureMode](#)

44.37.1 Detailed Description

Used in the [Timer_D_initCaptureMode\(\)](#) function as the param parameter.

44.37.2 Field Documentation

uint16_t Timer_D_initCaptureModeParam::captureInputSelect

Decides the Input Select

Valid values are:

- **TIMER_D_CAPTURE_INPUTSELECT_CC1xA** [Default]
- **TIMER_D_CAPTURE_INPUTSELECT_CC1xB**
- **TIMER_D_CAPTURE_INPUTSELECT_GND**
- **TIMER_D_CAPTURE_INPUTSELECT_Vcc**

Referenced by [Timer_D_initCaptureMode\(\)](#).

uint16_t Timer_D_initCaptureModeParam::captureInterruptEnable

Is to enable or disabel capture interrupt

Valid values are:

- **TIMER_D_CAPTURE_INTERRUPT_ENABLE**
- **TIMER_D_CAPTURE_INTERRUPT_DISABLE** [Default]

Referenced by [Timer_D_initCaptureMode\(\)](#).

uint16_t Timer_D_initCaptureModeParam::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**

Referenced by `Timer_D_initCaptureMode()`.

`uint16_t Timer_D_initCaptureModeParam::captureOutputMode`

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**

Referenced by `Timer_D_initCaptureMode()`.

`uint16_t Timer_D_initCaptureModeParam::captureRegister`

Selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used

Valid values are:

- **TIMER_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**

Referenced by `Timer_D_initCaptureMode()`.

`uint8_t Timer_D_initCaptureModeParam::channelCaptureMode`

Specifies single/dual capture mode.

Valid values are:

- **TIMER_D_SINGLE_CAPTURE_MODE**
- **TIMER_D_DUAL_CAPTURE_MODE**

Referenced by `Timer_D_initCaptureMode()`.

uint16_t Timer_D_initCaptureModeParam::synchronizeCaptureSource

Decides if capture source should be synchronized with timer clock
Valid values are:

- **TIMER_D_CAPTURE_ASYNCHRONOUS** [Default]
- **TIMER_D_CAPTURE_SYNCHRONOUS**

Referenced by `Timer_D_initCaptureMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.38 Timer_D_initCompareModeParam Struct Reference

Used in the `Timer_D_initCompareMode()` function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- uint16_t [compareRegister](#)
- uint16_t [compareInterruptEnable](#)
- uint16_t [compareOutputMode](#)
- uint16_t [compareValue](#)

Is the count to be compared with in compare mode.

44.38.1 Detailed Description

Used in the `Timer_D_initCompareMode()` function as the param parameter.

44.38.2 Field Documentation

uint16_t Timer_D_initCompareModeParam::compareInterruptEnable

Is to enable or disable timer captureCompare interrupt.
Valid values are:

- **TIMER_D_CAPTURECOMPARE_INTERRUPT_ENABLE**
- **TIMER_D_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_D_initCompareMode()`.

uint16_t Timer_D_initCompareModeParam::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**

Referenced by `Timer_D_initCompareMode()`.

uint16_t Timer_D_initCompareModeParam::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**

Referenced by `Timer_D_initCompareMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.39 Timer_D_initContinuousModeParam Struct Reference

Used in the `Timer_D_initContinuousMode()` function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- `uint16_t clockSource`
- `uint16_t clockSourceDivider`
- `uint16_t clockingMode`
- `uint16_t timerInterruptEnable_TDIE`
- `uint16_t timerClear`

44.39.1 Detailed Description

Used in the [Timer_D.initContinuousMode\(\)](#) function as the param parameter.

44.39.2 Field Documentation

uint16_t Timer_D_initContinuousModeParam::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**

Referenced by [Timer_D.initContinuousMode\(\)](#).

uint16_t Timer_D_initContinuousModeParam::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**

Referenced by [Timer_D.initContinuousMode\(\)](#).

uint16_t Timer_D_initContinuousModeParam::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**

Referenced by `Timer_D_initContinuousMode()`.

`uint16_t Timer_D_initContinuousModeParam::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]

Referenced by `Timer_D_initContinuousMode()`.

`uint16_t Timer_D_initContinuousModeParam::timerInterruptEnable_TDIE`

Is to enable or disable timer interrupt
Valid values are:

- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_D_initContinuousMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.40 `Timer_D_initHighResGeneratorInRegulatedMode` `Param Struct` Reference

Used in the `Timer_D_initHighResGeneratorInRegulatedMode()` function as the `param` parameter.

```
#include <timer_d.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [clockingMode](#)
- uint8_t [highResClockMultiplyFactor](#)
- uint8_t [highResClockDivider](#)

44.40.1 Detailed Description

Used in the [Timer_D_initHighResGeneratorInRegulatedMode\(\)](#) function as the param parameter.

44.40.2 Field Documentation

uint16_t [Timer_D_initHighResGeneratorInRegulatedModeParam::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**

Referenced by [Timer_D_initHighResGeneratorInRegulatedMode\(\)](#).

uint16_t [Timer_D_initHighResGeneratorInRegulatedModeParam::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**

Referenced by [Timer_D_initHighResGeneratorInRegulatedMode\(\)](#).

uint16_t [Timer_D_initHighResGeneratorInRegulatedModeParam::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**

Referenced by `Timer_D_initHighResGeneratorInRegulatedMode()`.

`uint8_t Timer_D_initHighResGeneratorInRegulatedModeParam::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**

Referenced by `Timer_D_initHighResGeneratorInRegulatedMode()`.

`uint8_t Timer_D_initHighResGeneratorInRegulatedModeParam::highResClockMultiplyFactor`

Selects the high resolution multiply factor.
Valid values are:

- **TIMER_D_HIGHRES_CLK_MULTIPLY_FACTOR_8x**
- **TIMER_D_HIGHRES_CLK_MULTIPLY_FACTOR_16x**

Referenced by `Timer_D_initHighResGeneratorInRegulatedMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.41 Timer_D_initUpDownModeParam Struct Reference

Used in the [Timer_D_initUpDownMode\(\)](#) function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- uint16_t [clockSource](#)
- uint16_t [clockSourceDivider](#)
- uint16_t [clockingMode](#)
- uint16_t [timerPeriod](#)
Is the specified timer period.
- uint16_t [timerInterruptEnable_TDIE](#)
- uint16_t [captureCompareInterruptEnable_CCR0_CCIE](#)
- uint16_t [timerClear](#)

44.41.1 Detailed Description

Used in the [Timer_D_initUpDownMode\(\)](#) function as the param parameter.

44.41.2 Field Documentation

uint16_t [Timer_D_initUpDownModeParam::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]

Referenced by [Timer_D_initUpDownMode\(\)](#).

uint16_t [Timer_D_initUpDownModeParam::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**

Referenced by [Timer_D_initUpDownMode\(\)](#).

`uint16_t Timer_D_initUpDownModeParam::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`

Referenced by `Timer_D_initUpDownMode()`.

`uint16_t Timer_D_initUpDownModeParam::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`

Referenced by `Timer_D_initUpDownMode()`.

uint16_t Timer_D_initUpDownModeParam::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]

Referenced by `Timer_D_initUpDownMode()`.

uint16_t Timer_D_initUpDownModeParam::timerInterruptEnable_TDIE

Is to enable or disable timer interrupt
Valid values are:

- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_D_initUpDownMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.42 Timer_D_initUpModeParam Struct Reference

Used in the [Timer_D_initUpMode\(\)](#) function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- [uint16_t clockSource](#)
- [uint16_t clockSourceDivider](#)
- [uint16_t clockingMode](#)
- [uint16_t timerPeriod](#)
- [uint16_t timerInterruptEnable_TDIE](#)
- [uint16_t captureCompareInterruptEnable_CCR0_CCIE](#)
- [uint16_t timerClear](#)

44.42.1 Detailed Description

Used in the [Timer_D_initUpMode\(\)](#) function as the param parameter.

44.42.2 Field Documentation

uint16_t Timer_D_initUpModeParam::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]

Referenced by Timer_D_initUpMode().

uint16_t Timer_D_initUpModeParam::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**

Referenced by Timer_D_initUpMode().

uint16_t Timer_D_initUpModeParam::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**

Referenced by Timer_D_initUpMode().

uint16_t Timer_D_initUpModeParam::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**

Referenced by `Timer_D_initUpMode()`.

`uint16_t Timer_D_initUpModeParam::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]

Referenced by `Timer_D_initUpMode()`.

`uint16_t Timer_D_initUpModeParam::timerInterruptEnable_TDIE`

Is to enable or disable timer interrupt
Valid values are:

- **TIMER_D_TDIE_INTERRUPT_ENABLE**
- **TIMER_D_TDIE_INTERRUPT_DISABLE** [Default]

Referenced by `Timer_D_initUpMode()`.

`uint16_t Timer_D_initUpModeParam::timerPeriod`

Is the specified timer period. This is the value that gets written into the CCR0. Limited to 16 bits
[`uint16_t`]

Referenced by `Timer_D_initUpMode()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.43 Timer_D_outputPWMParm Struct Reference

Used in the [Timer_D_outputPWM\(\)](#) function as the param parameter.

```
#include <timer_d.h>
```

Data Fields

- [uint16_t clockSource](#)
 - [uint16_t clockSourceDivider](#)
 - [uint16_t clockingMode](#)
 - [uint16_t timerPeriod](#)
- Is the specified timer period.*
- [uint16_t compareRegister](#)
 - [uint16_t compareOutputMode](#)
 - [uint16_t dutyCycle](#)

Specifies the dutycycle for the generated waveform.

44.43.1 Detailed Description

Used in the [Timer_D_outputPWM\(\)](#) function as the param parameter.

44.43.2 Field Documentation

`uint16_t Timer_D_outputPWMParm::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**

Referenced by [Timer_D_outputPWM\(\)](#).

`uint16_t Timer_D_outputPWMParm::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**

Referenced by [Timer_D_outputPWM\(\)](#).

`uint16_t Timer_D_outputPWMParam::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`

Referenced by `Timer_D_outputPWM()`.

`uint16_t Timer_D_outputPWMParam::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`

Referenced by `Timer_D_outputPWM()`.

uint16_t Timer_D_outputPWMParm::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**

Referenced by `Timer_D_outputPWM()`.

The documentation for this struct was generated from the following file:

- `timer_d.h`

44.44 TEC_initExternalFaultInputParam Struct Reference

Used in the `TEC_initExternalFaultInput()` function as the param parameter.

```
#include <tec.h>
```

Data Fields

- `uint8_t` [selectedExternalFault](#)
- `uint16_t` [signalType](#)
- `uint8_t` [signalHold](#)
- `uint8_t` [polarityBit](#)

44.44.1 Detailed Description

Used in the `TEC_initExternalFaultInput()` function as the param parameter.

44.44.2 Field Documentation

uint8_t TEC_initExternalFaultInputParam::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**

Referenced by `TEC_initExternalFaultInput()`.

uint8_t TEC_initExternalFaultInputParam::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**

Referenced by TEC_initExternalFaultInput().

uint8_t TEC_initExternalFaultInputParam::signalHold

Is the selected signal hold
Valid values are:

- **TEC_EXTERNAL_FAULT_SIGNAL_NOT_HELD** [Default]
- **TEC_EXTERNAL_FAULT_SIGNAL_HELD**

Referenced by TEC_initExternalFaultInput().

uint16_t TEC_initExternalFaultInputParam::signalType

Is the selected signal type
Valid values are:

- **TEC_EXTERNAL_FAULT_SIGNALTYPE_EDGE_SENSITIVE** [Default]
- **TEC_EXTERNAL_FAULT_SIGNALTYPE_LEVEL_SENSITIVE**

Referenced by TEC_initExternalFaultInput().

The documentation for this struct was generated from the following file:

- `tec.h`

44.45 USCI_A_SPI_changeMasterClockParam Struct Reference

Used in the [USCI_A_SPI_changeMasterClock\(\)](#) function as the param parameter.

```
#include <usci_a_spi.h>
```

Data Fields

- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.

44.45.1 Detailed Description

Used in the [USCI_A_SPI.changeMasterClock\(\)](#) function as the param parameter.

The documentation for this struct was generated from the following file:

- `usci_a_spi.h`

44.46 USCI_A_SPI_initMasterParam Struct Reference

Used in the [USCI_A_SPI.initMaster\(\)](#) function as the param parameter.

```
#include <usci_a_spi.h>
```

Data Fields

- `uint8_t` [selectClockSource](#)
- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.
- `uint8_t` [msbFirst](#)
- `uint8_t` [clockPhase](#)
- `uint8_t` [clockPolarity](#)

44.46.1 Detailed Description

Used in the [USCI_A_SPI.initMaster\(\)](#) function as the param parameter.

44.46.2 Field Documentation

`uint8_t` [USCI_A_SPI_initMasterParam::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`

Referenced by [USCI_A_SPI.initMaster\(\)](#).

uint8_t USCI_A_SPI_initMasterParam::clockPolarity

Valid values are:

- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by USCI_A_SPI_initMaster().

uint8_t USCI_A_SPI_initMasterParam::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]

Referenced by USCI_A_SPI_initMaster().

uint8_t USCI_A_SPI_initMasterParam::selectClockSource

Selects Clock source.

Valid values are:

- **USCI_A_SPI_CLOCKSOURCE_ACLK**
- **USCI_A_SPI_CLOCKSOURCE_SMCLK**

Referenced by USCI_A_SPI_initMaster().

The documentation for this struct was generated from the following file:

- usci_a_spi.h

44.47 USCI_A_UART_initParam Struct Reference

Used in the [USCI_A_UART_init\(\)](#) function as the param parameter.

```
#include <usci_a_uart.h>
```

Data Fields

- uint8_t [selectClockSource](#)
- uint16_t [clockPrescalar](#)
Is the value to be written into UCBRx bits.
- uint8_t [firstModReg](#)
- uint8_t [secondModReg](#)
- uint8_t [parity](#)

- uint8_t [msborLsbFirst](#)
- uint8_t [numberOfStopBits](#)
- uint8_t [uartMode](#)
- uint8_t [overSampling](#)

44.47.1 Detailed Description

Used in the [USCI_A_UART_init\(\)](#) function as the param parameter.

44.47.2 Field Documentation

uint8_t USCI_A_UART_initParam::firstModReg

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

Referenced by [USCI_A_UART_init\(\)](#).

uint8_t USCI_A_UART_initParam::msborLsbFirst

Controls direction of receive and transmit shift register.

Valid values are:

- **USCI_A_UART_MSB_FIRST**
- **USCI_A_UART_LSB_FIRST** [Default]

Referenced by [USCI_A_UART_init\(\)](#).

uint8_t USCI_A_UART_initParam::numberOfStopBits

Indicates one/two STOP bits

Valid values are:

- **USCI_A_UART_ONE_STOP_BIT** [Default]
- **USCI_A_UART_TWO_STOP_BITS**

Referenced by [USCI_A_UART_init\(\)](#).

uint8_t USCI_A_UART_initParam::overSampling

Indicates low frequency or oversampling baud generation

Valid values are:

- **USCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION**
- **USCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION**

Referenced by [USCI_A_UART_init\(\)](#).

uint8_t USCI_A_UART_initParam::parity

Is the desired parity.

Valid values are:

- **USCI_A_UART_NO_PARITY** [Default]
- **USCI_A_UART_ODD_PARITY**
- **USCI_A_UART_EVEN_PARITY**

Referenced by USCI_A_UART_init().

uint8_t USCI_A_UART_initParam::secondModReg

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

Referenced by USCI_A_UART_init().

uint8_t USCI_A_UART_initParam::selectClockSource

Selects Clock source.

Valid values are:

- **USCI_A_UART_CLOCKSOURCE_SMCLK**
- **USCI_A_UART_CLOCKSOURCE_ACLK**

Referenced by USCI_A_UART_init().

uint8_t USCI_A_UART_initParam::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**

Referenced by USCI_A_UART_init().

The documentation for this struct was generated from the following file:

- usci_a_uart.h

44.48 USCI_B_I2C_initMasterParam Struct Reference

Used in the [USCI_B_I2C_initMaster\(\)](#) function as the param parameter.

```
#include <usci_b_i2c.h>
```

Data Fields

- uint8_t [selectClockSource](#)
- uint32_t [i2cClk](#)
Is the rate of the clock supplied to the I2C module.
- uint32_t [dataRate](#)

44.48.1 Detailed Description

Used in the [USCI_B_I2C_initMaster\(\)](#) function as the param parameter.

44.48.2 Field Documentation

uint32_t USCI_B_I2C_initMasterParam::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**

Referenced by [USCI_B_I2C_initMaster\(\)](#).

uint8_t USCI_B_I2C_initMasterParam::selectClockSource

Is the clocksource.
Valid values are:

- **USCI_B_I2C_CLOCKSOURCE_ACLK**
- **USCI_B_I2C_CLOCKSOURCE_SMCLK**

Referenced by [USCI_B_I2C_initMaster\(\)](#).

The documentation for this struct was generated from the following file:

- `usci_b_i2c.h`

44.49 USCI_B_SPI_changeMasterClockParam Struct Reference

Used in the [USCI_B_SPI_changeMasterClock\(\)](#) function as the param parameter.

```
#include <usci_b_spi.h>
```

Data Fields

- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.

44.49.1 Detailed Description

Used in the [USCI_B_SPI.changeMasterClock\(\)](#) function as the param parameter.

The documentation for this struct was generated from the following file:

- `usci_b_spi.h`

44.50 USCI_B_SPI_initMasterParam Struct Reference

Used in the [USCI_B_SPI.initMaster\(\)](#) function as the param parameter.

```
#include <usci_b_spi.h>
```

Data Fields

- `uint8_t` [selectClockSource](#)
- `uint32_t` [clockSourceFrequency](#)
Is the frequency of the selected clock source.
- `uint32_t` [desiredSpiClock](#)
Is the desired clock rate for SPI communication.
- `uint8_t` [msbFirst](#)
- `uint8_t` [clockPhase](#)
- `uint8_t` [clockPolarity](#)

44.50.1 Detailed Description

Used in the [USCI_B_SPI.initMaster\(\)](#) function as the param parameter.

44.50.2 Field Documentation

`uint8_t` [USCI_B_SPI_initMasterParam::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`

Referenced by [USCI_B_SPI.initMaster\(\)](#).

uint8_t USCI_B_SPI_initMasterParam::clockPolarity

Valid values are:

- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH**
- **USCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

Referenced by USCI_B_SPI_initMaster().

uint8_t USCI_B_SPI_initMasterParam::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]

Referenced by USCI_B_SPI_initMaster().

uint8_t USCI_B_SPI_initMasterParam::selectClockSource

Selects Clock source.

Valid values are:

- **USCI_B_SPI_CLOCKSOURCE_ACLK**
- **USCI_B_SPI_CLOCKSOURCE_SMCLK**

Referenced by USCI_B_SPI_initMaster().

The documentation for this struct was generated from the following file:

- usci_b_spi.h

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
 Copyright © 2015, Texas Instruments Incorporated