MSP430 DriverLib for MSP430FR57xx Devices

User's Guide

DOCNUM-2.10.00.09

Copyright

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 10:35:07.

Table of Contents

1 Introduction

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430 FR5xx/FR6xx 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™

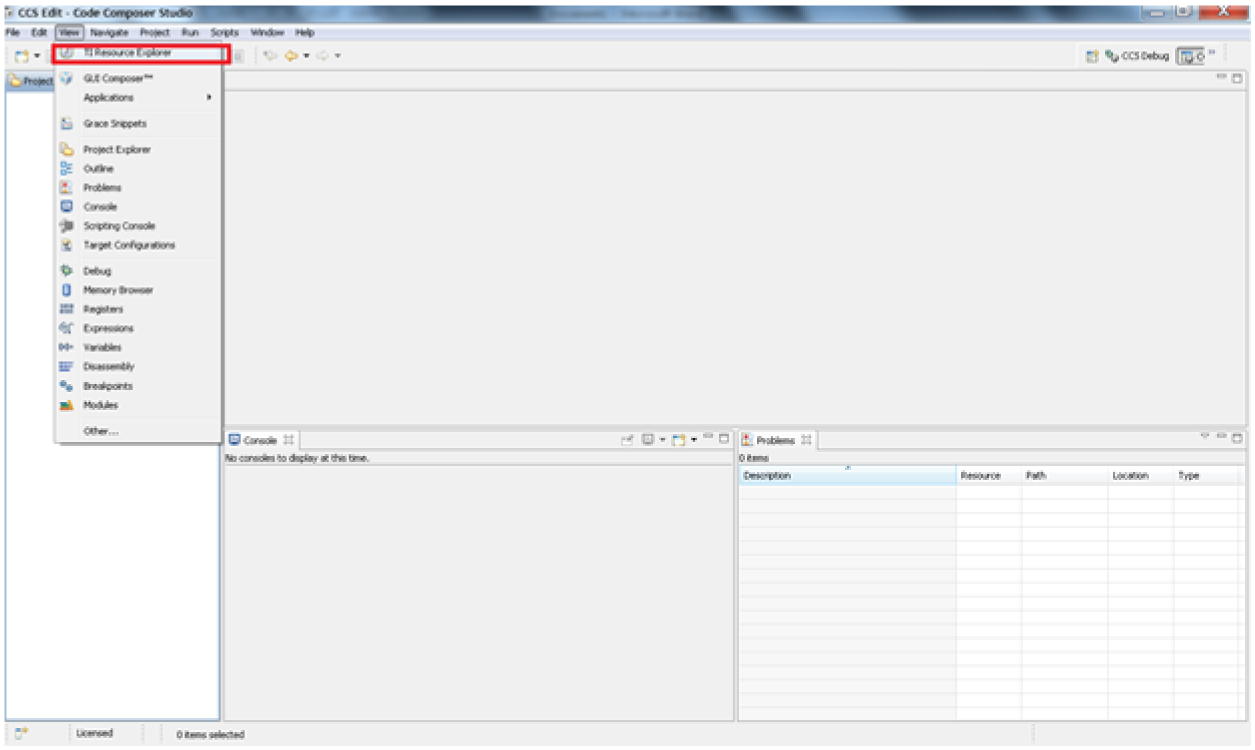
Using assert statements to debug

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

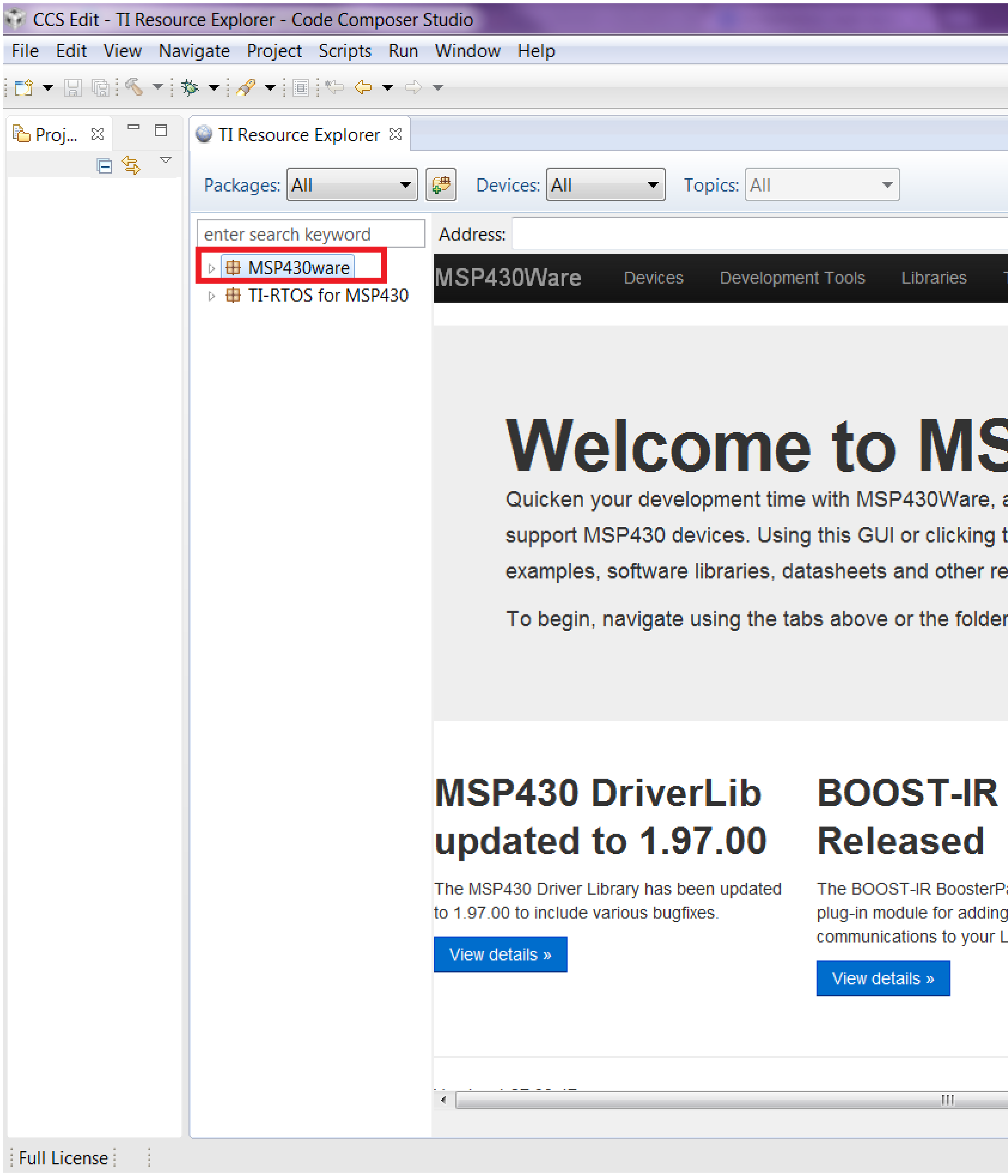
2.1 Introduction

In CCS, click View->TI Resource Explorer



No consoles to display at this time.

In Resource Explorer View, click on MSP430ware



Welcome to MS

Quicken your development time with MSP430Ware, a
support MSP430 devices. Using this GUI or clicking t
examples, software libraries, datasheets and other re

To begin, navigate using the tabs above or the folder

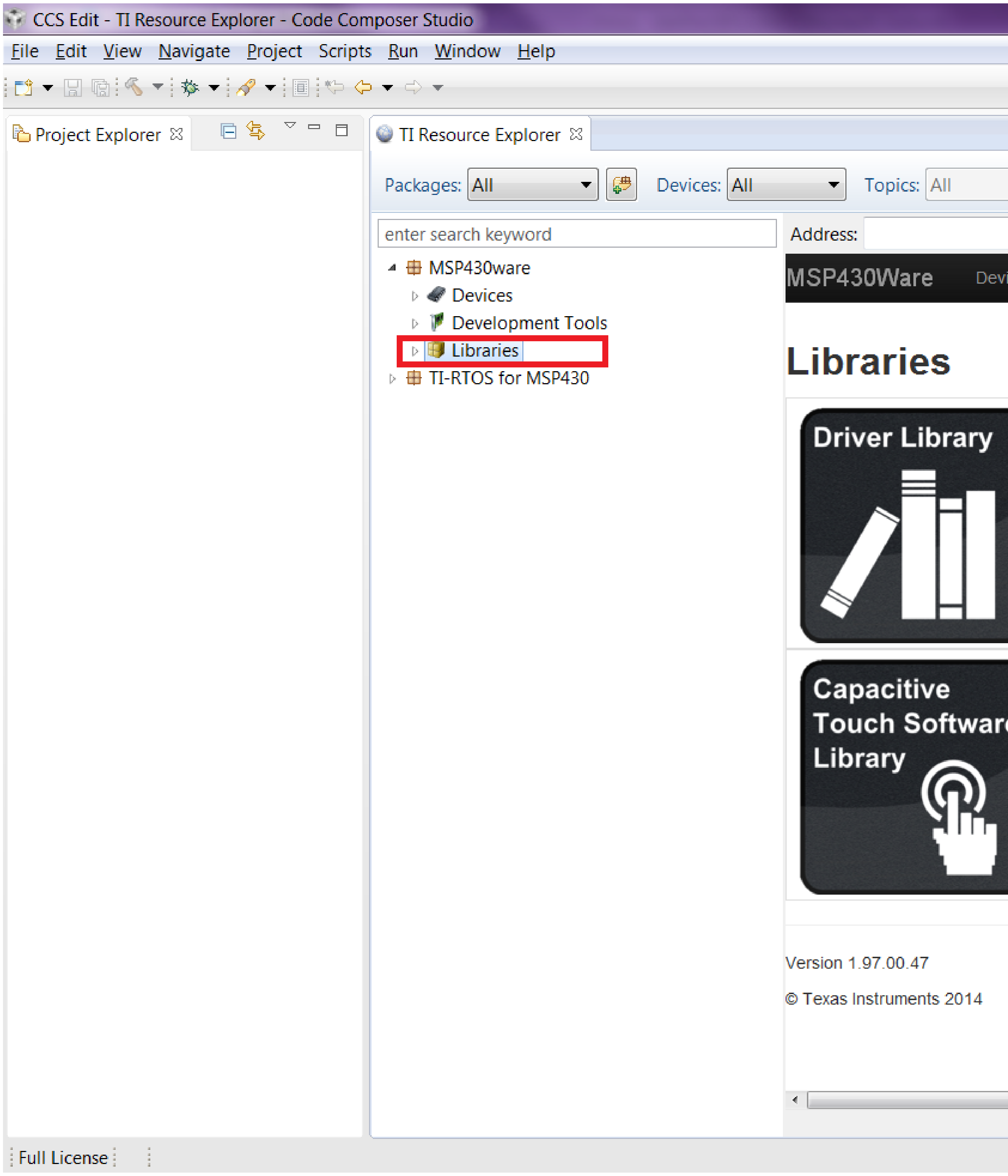
MSP430 DriverLib updated to 1.97.00

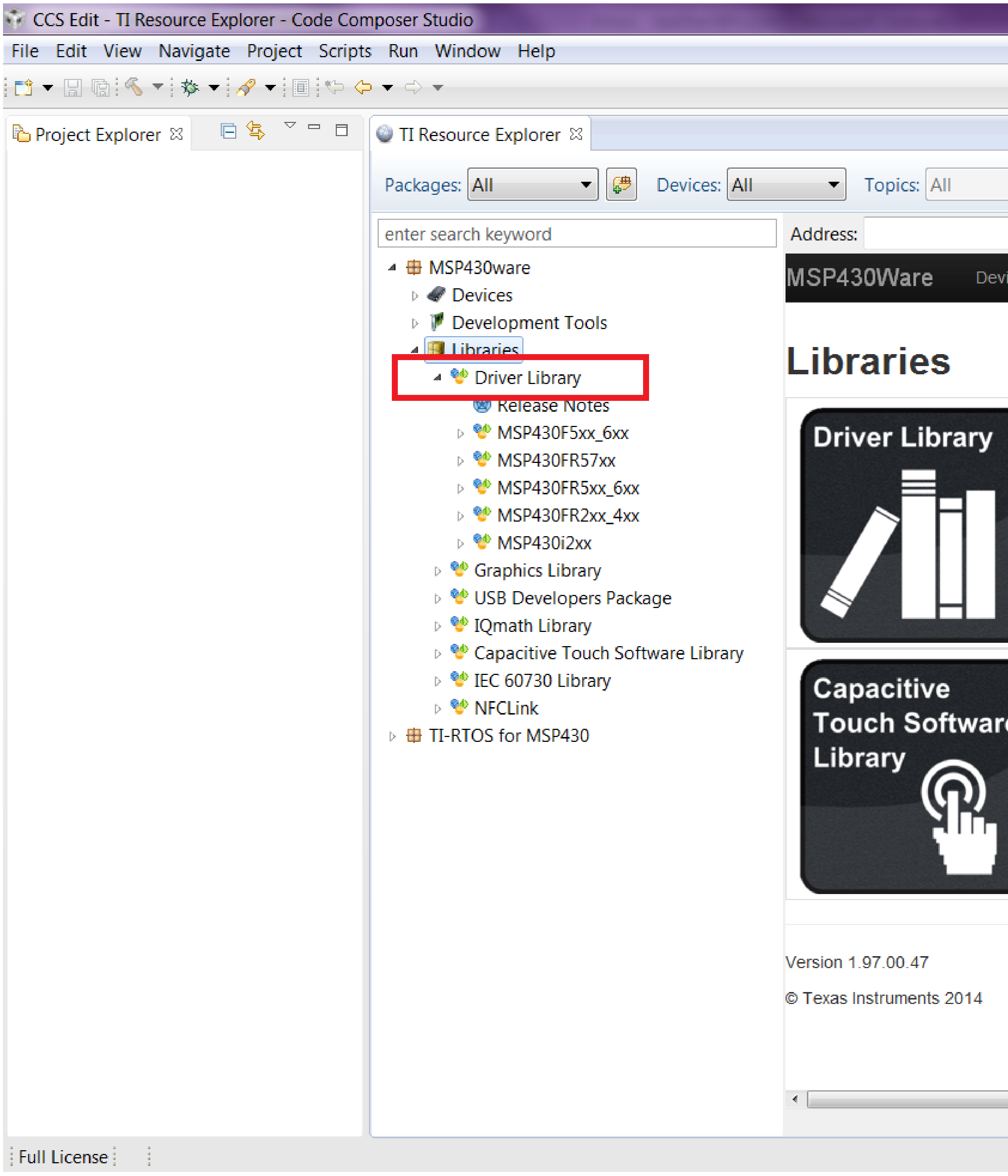
The MSP430 Driver Library has been updated to 1.97.00 to include various bugfixes.

BOOST-IR Released

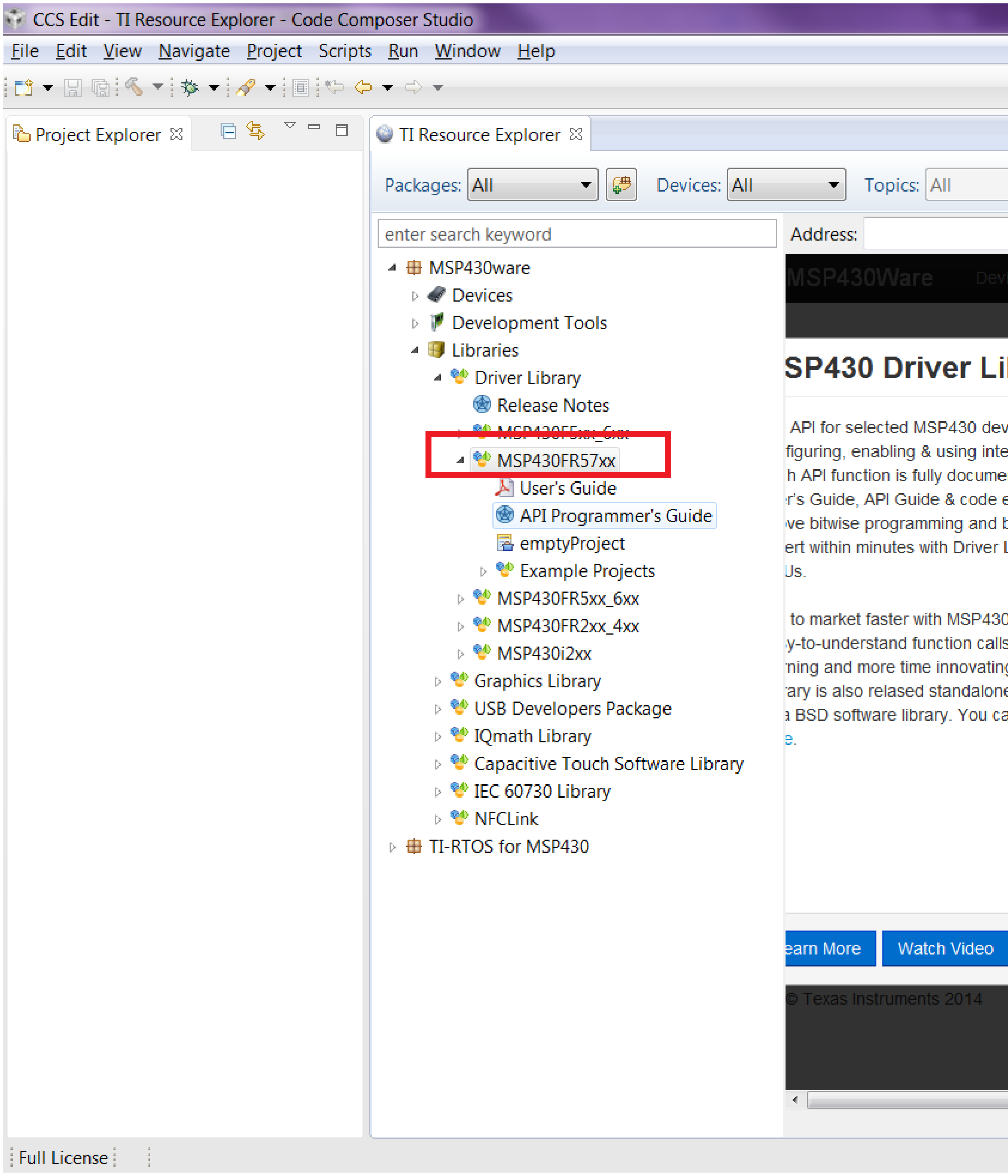
The BOOST-IR BoosterPa
plug-in module for adding
communications to your L

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


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



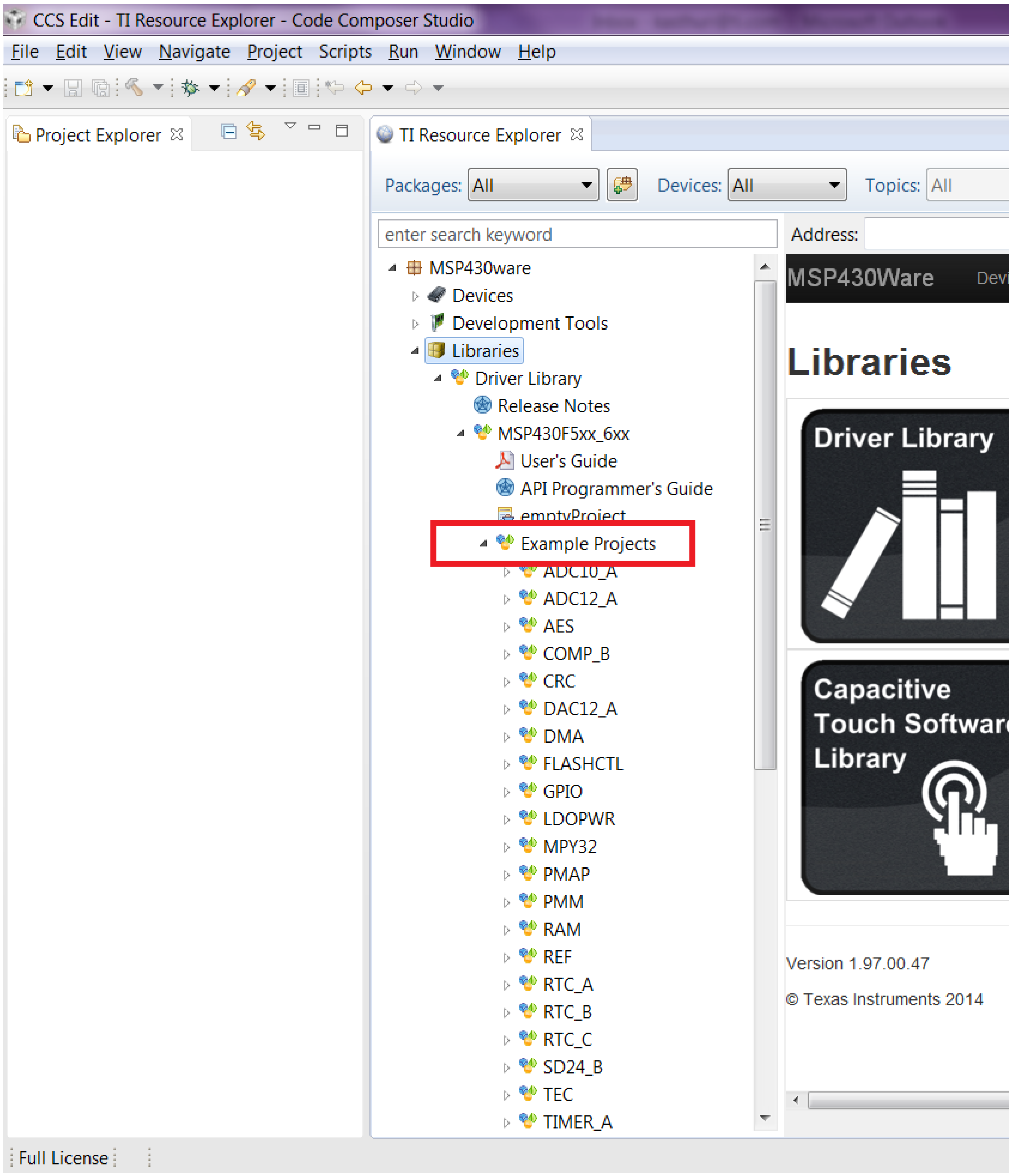
Driverlib is designed per Family. If a common device family user's guide exists for a group of devices, these devices belong to the same 'family'. Currently driverlib is available for the following family of devices. MSP430F5xx_6xx MSP430FR57xx MSP430FR5xx_6xx MSP430i2xx



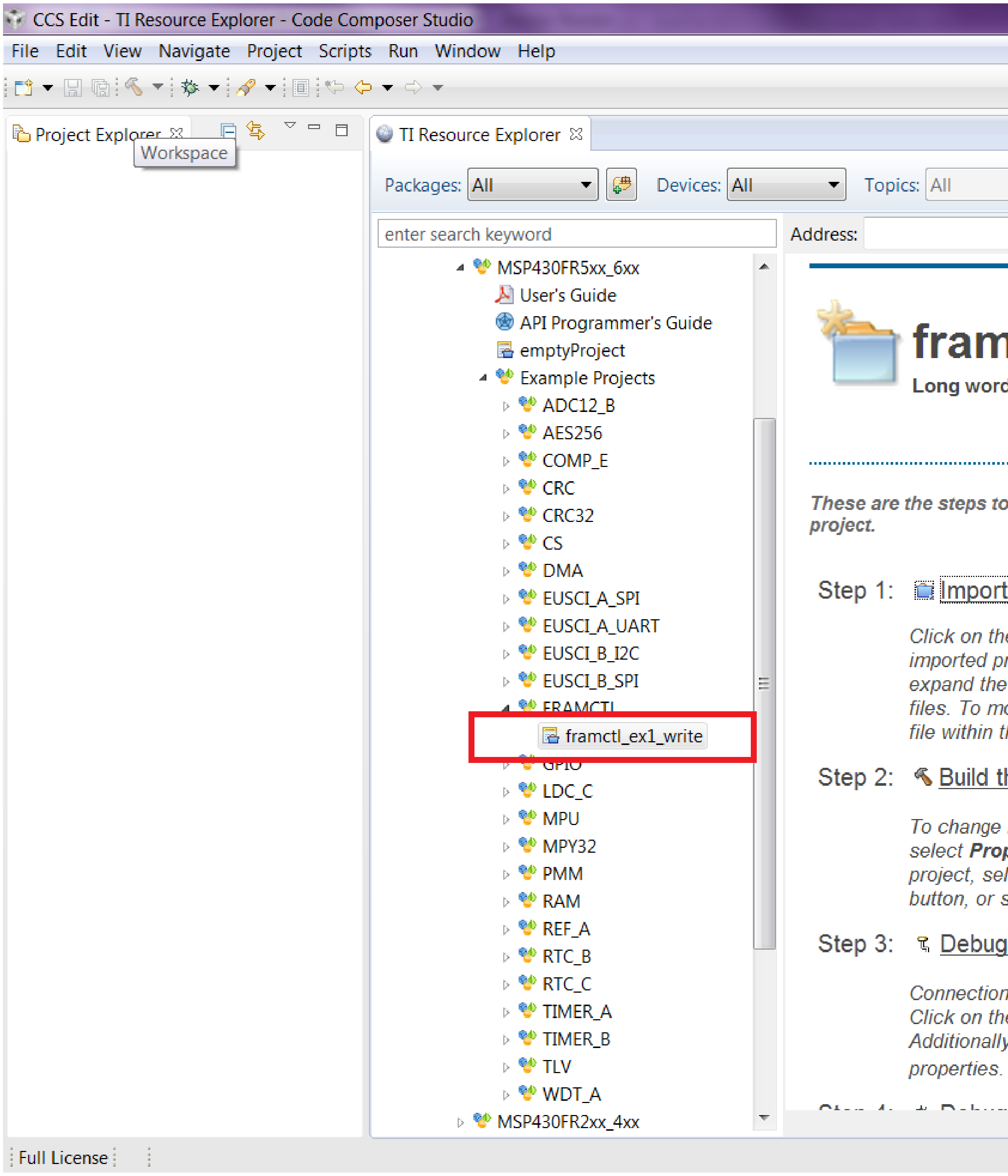
API for selected MSP430 dev
figuring, enabling & using inte
h API function is fully docume
r's Guide, API Guide & code e
ve bitwise programming and b
ert within minutes with Driver L
Us.

to market faster with MSP430
y-to-understand function calls
ning and more time innovating
ary is also relased standalone
a BSD software library. You ca
e.

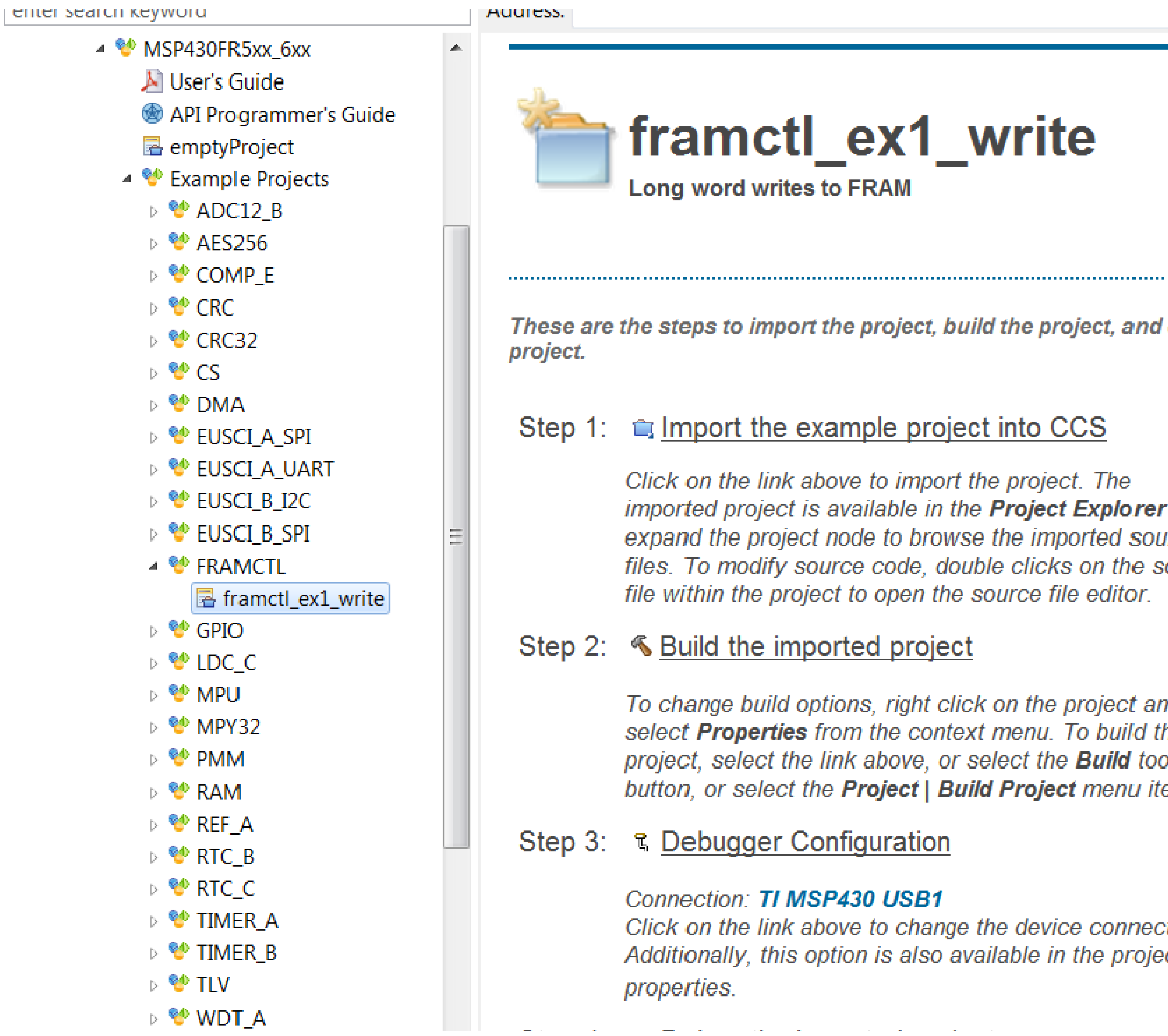
Click on the MSP430FR5xx_6xx to navigate to the driverlib based example code for that family.


CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Scripts Run Window Help
Project Explorer
TI Resource Explorer
Packages: All
Devices: All
Topics: All
enter search keyword
Address:
MSP430ware
Devices
Development Tools
Libraries
Driver Library
Release Notes
MSP430F5xx_6xx
User's Guide
API Programmer's Guide
emptyProject
Example Projects
ADC10_A
ADC12_A
AES
COMP_B
CRC
DAC12_A
DMA
FLASHCTL
GPIO
LDOPWR
MPY32
PMAP
PMM
RAM
REF
RTC_A
RTC_B
RTC_C
SD24_B
TEC
TIMER_A
MSP430Ware
Libraries
Driver Library
Capacitive Touch Software Library
Version 1.97.00.47
© Texas Instruments 2014
Full License

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"



framctl_ex1_write

Long word writes to FRAM

These are the steps to import the project, build the project, and
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 sou files. To modify source code, double clicks on the so file within the project to open the source file editor.*

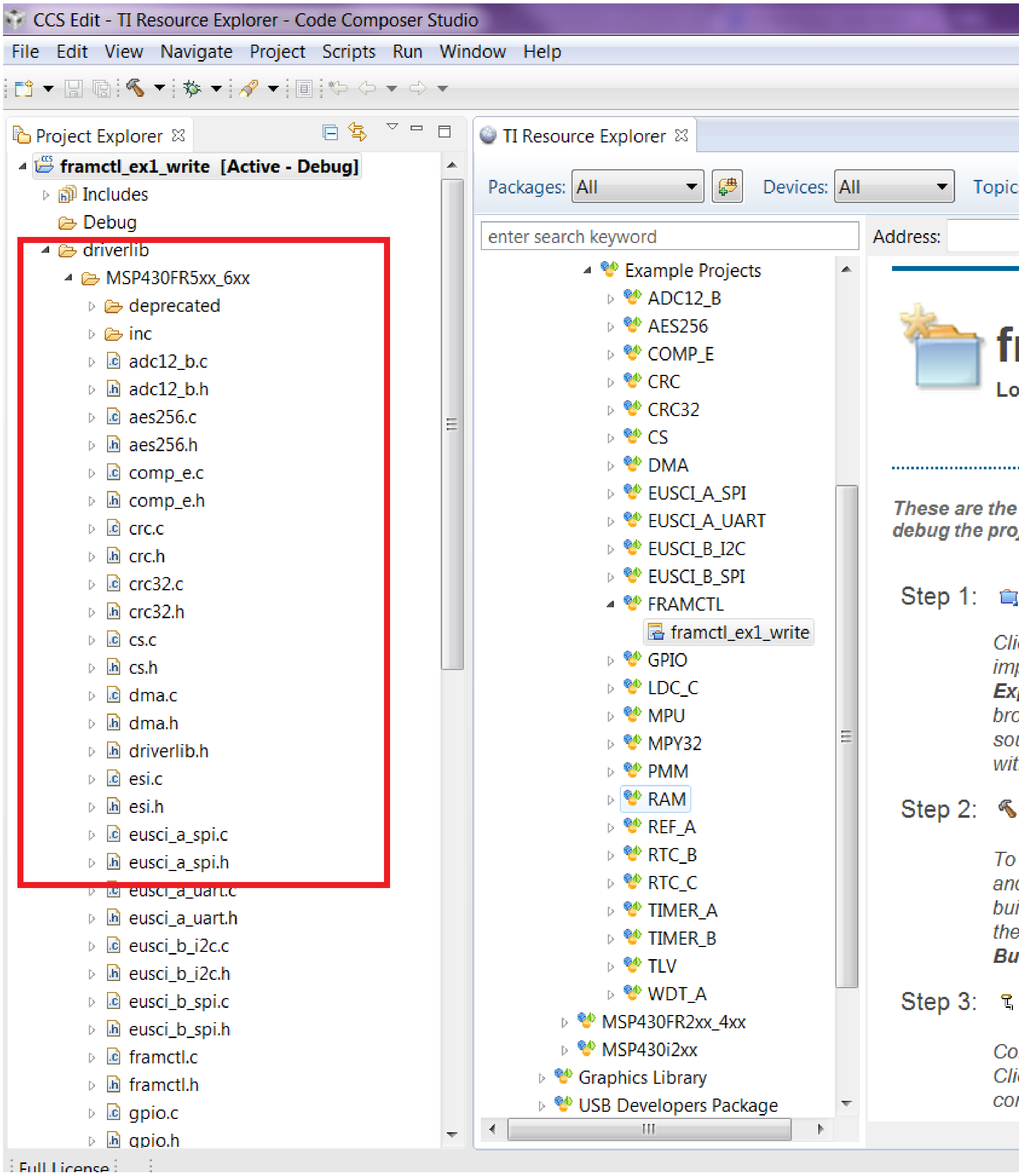
Step 2: Build the imported project

*To change build options, right click on the project an select **Properties** from the context menu. To build th project, select the link above, or select the **Build** too button, or select the **Project | Build Project** menu ite*

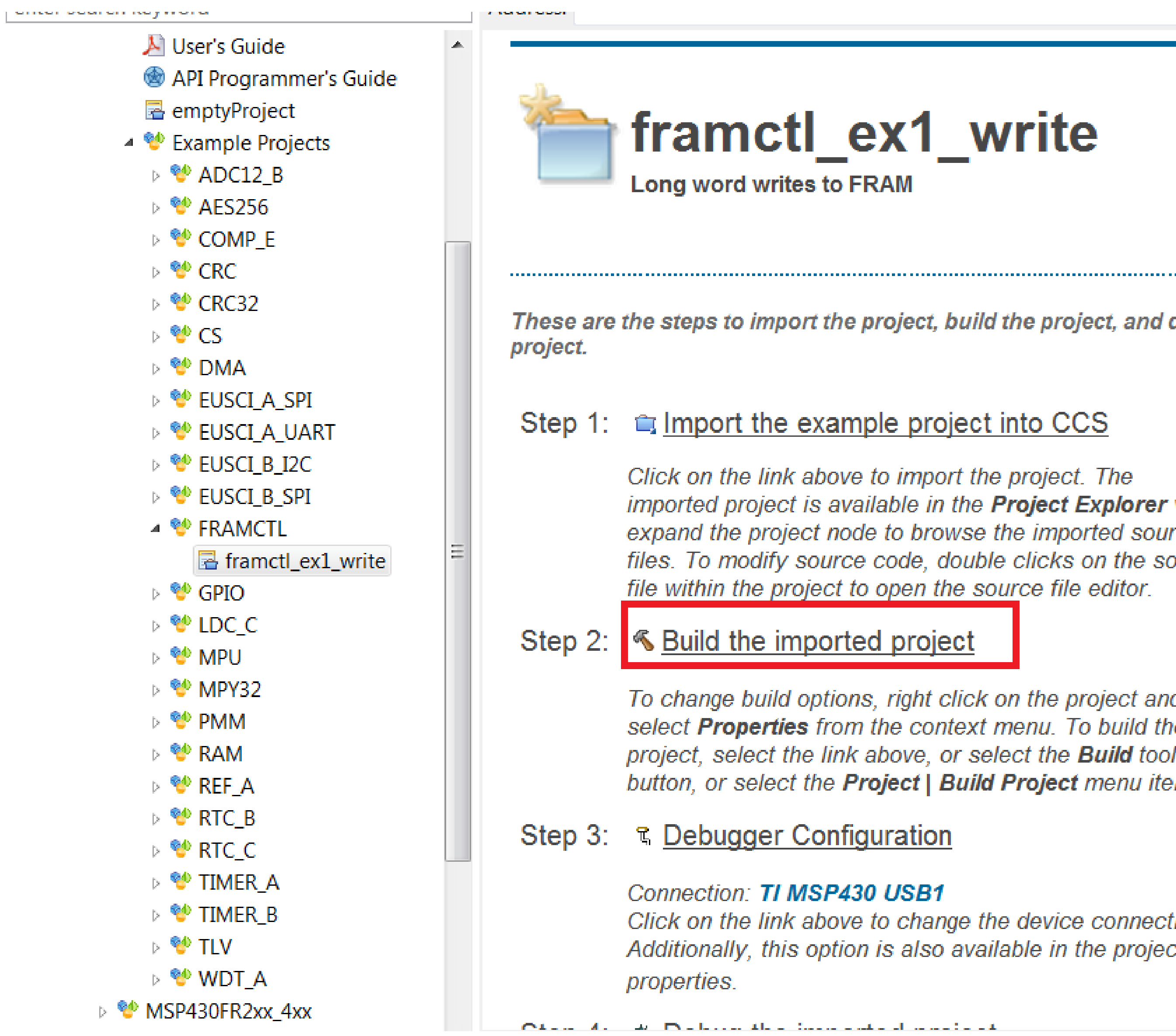
Step 3: Debugger Configuration

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



framctl_ex1_write

Long word writes to FRAM

These are the steps to import the project, build the project, and d 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 sour files. To modify source code, double clicks on the so file within the project to open the source file editor.*

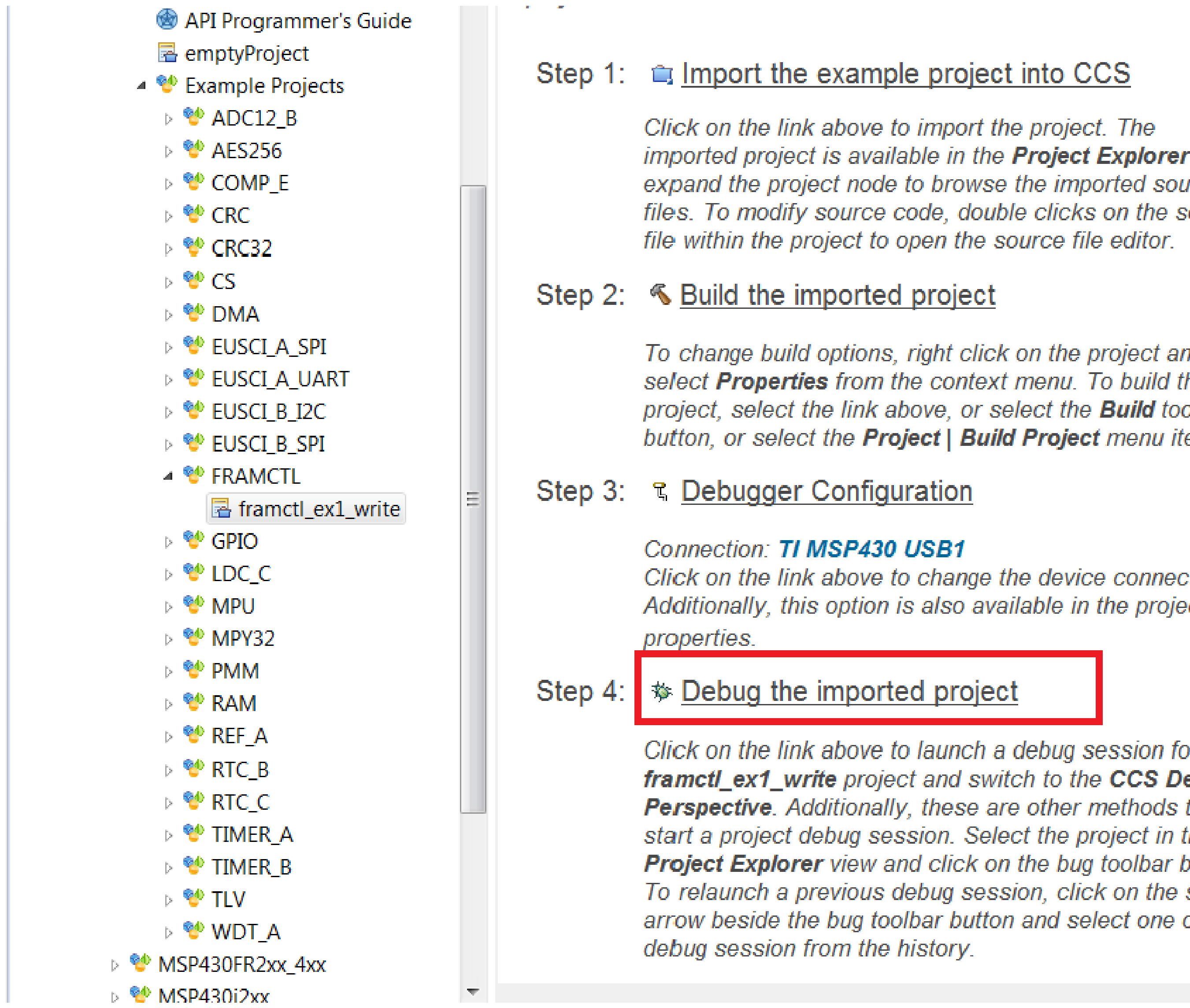
Step 2: Build the imported project

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

Step 3: Debugger Configuration

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

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



- API Programmer's Guide
- emptyProject
- Example Projects
  - ADC12_B
  - AES256
  - COMP_E
  - CRC
  - CRC32
  - CS
  - DMA
  - EUSCI_A_SPI
  - EUSCI_A_UART
  - EUSCI_B_I2C
  - EUSCI_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
- 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 sou files. To modify source code, double clicks on the s 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 an select **Properties** from the context menu. To build th project, select the link above, or select the **Build** too button, or select the **Project | Build Project** menu ite*

Step 3: Debugger Configuration

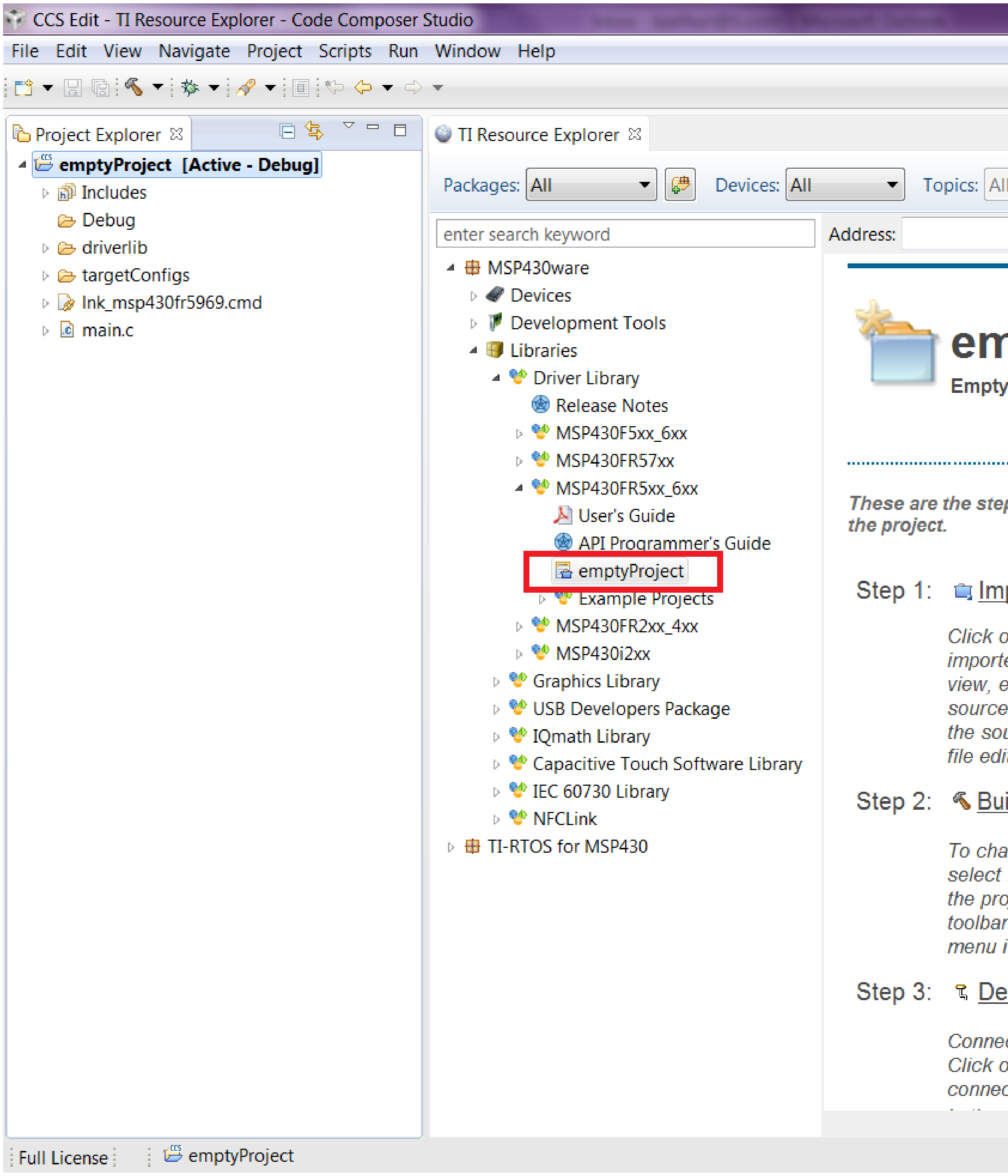
*Connection: **TI MSP430 USB1***
Click on the link above to change the device connec Additionally, this option is also available in the proje properties.

Step 4: Debug the imported project

*Click on the link above to launch a debug session fo **framctl_ex1_write** project and switch to the **CCS De Perspective**. Additionally, these are other methods t start a project debug session. Select the project in t **Project Explorer** view and click on the bug toolbar b To relaunch a previous debug session, click on the s arrow beside the bug toolbar button and select one o debug session from the history.*

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

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

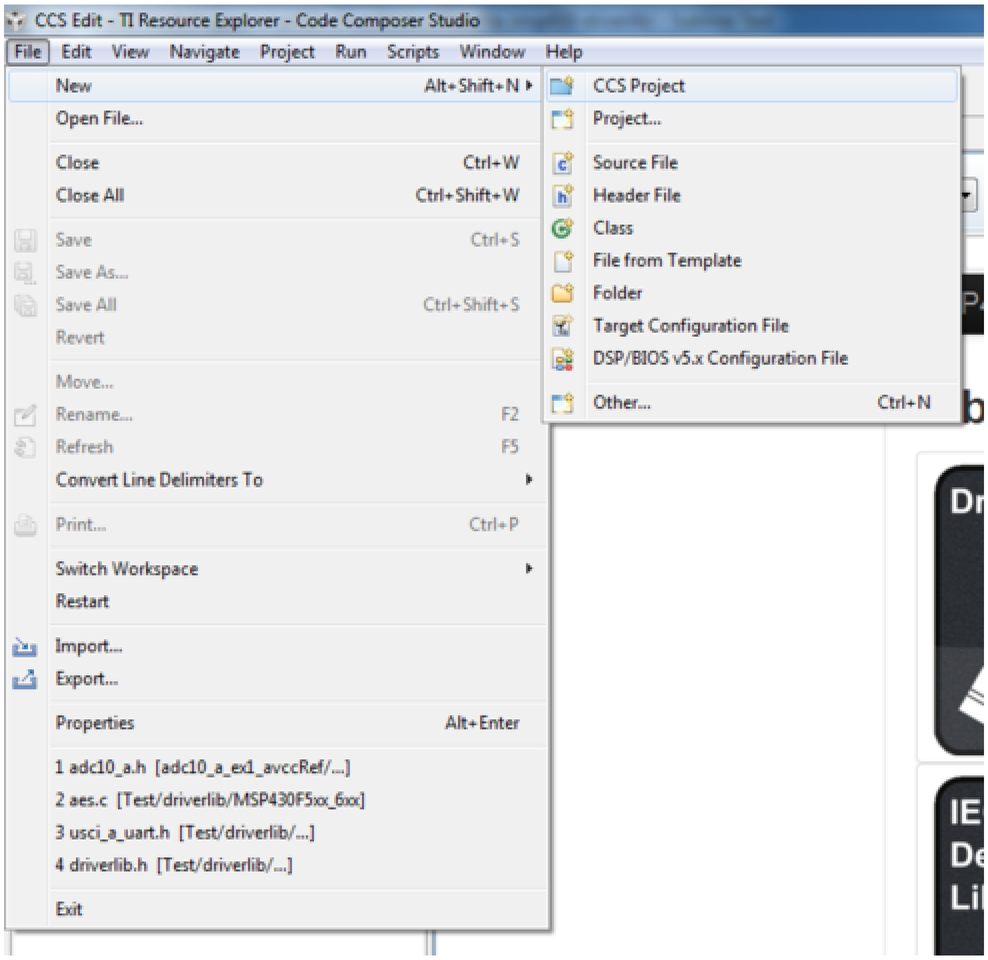


The main.c included with the empty project can be modified to include user code.

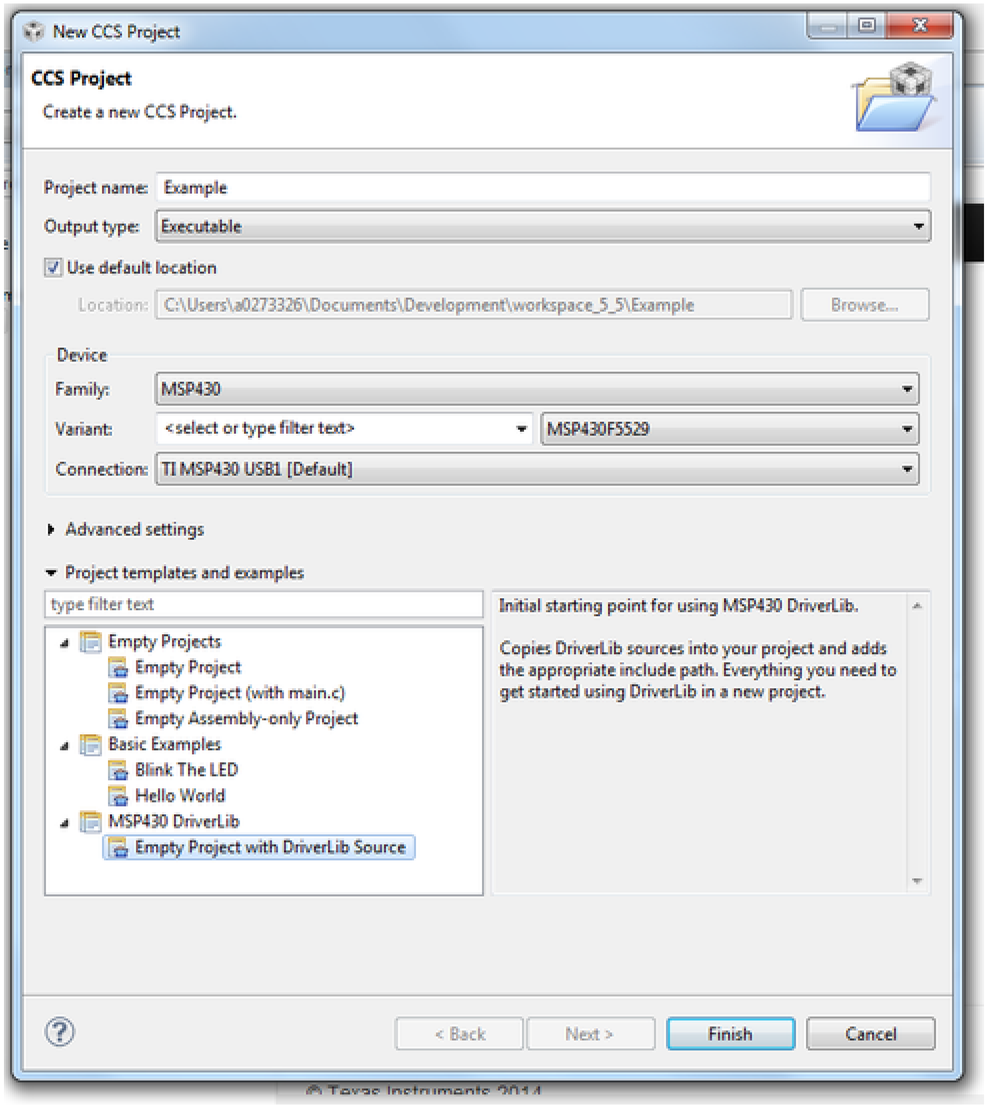
3 How to create a new CCS project that uses Driverlib

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.


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

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



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

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

Advanced settings
Project templates and examples

- Empty Projects
  - Empty Project
  - Empty Project (with main.c)
  - Empty Assembly-only Project
- Basic Examples
  - Blink The LED
  - Hello World
- MSP430 DriverLib
  - Empty Project with DriverLib Source

Initial starting point for using MSP430 DriverLib.

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

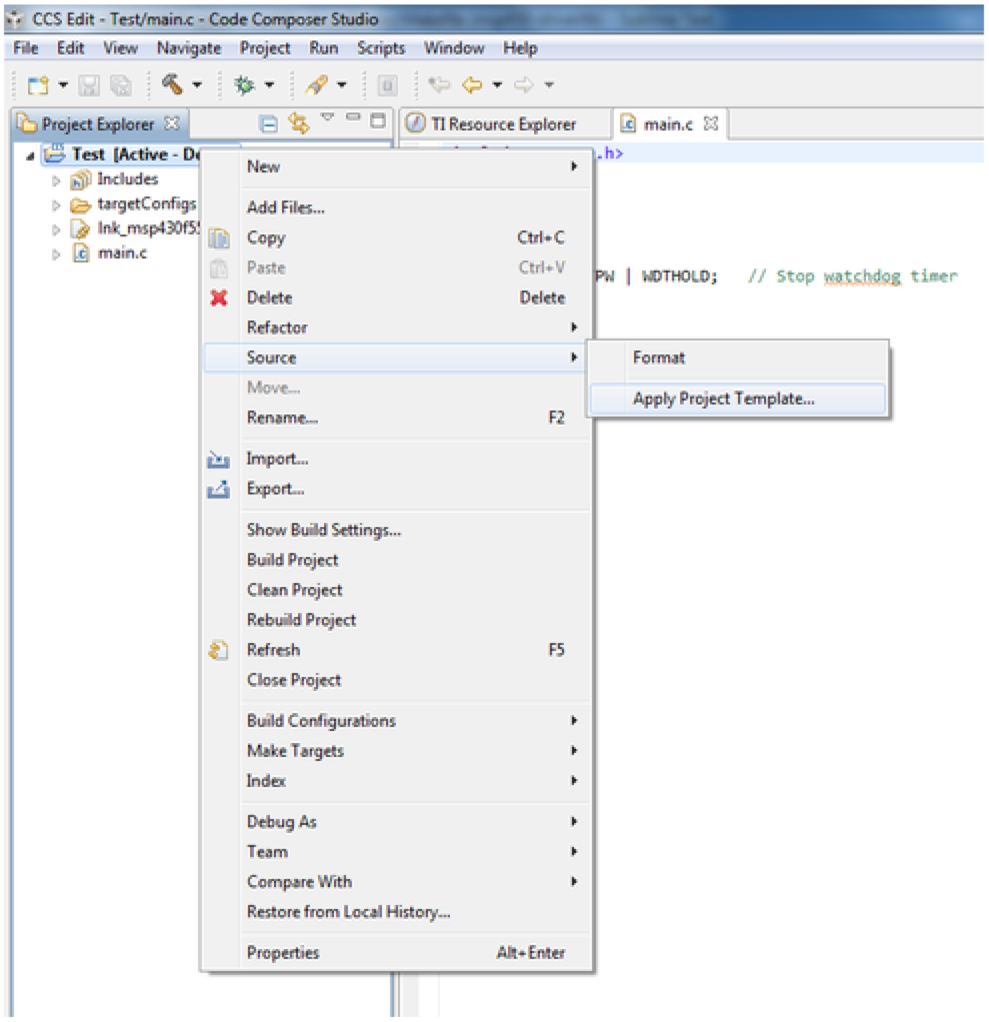
Finally click "Finish" and begin developing with your Driver Library enabled project.

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

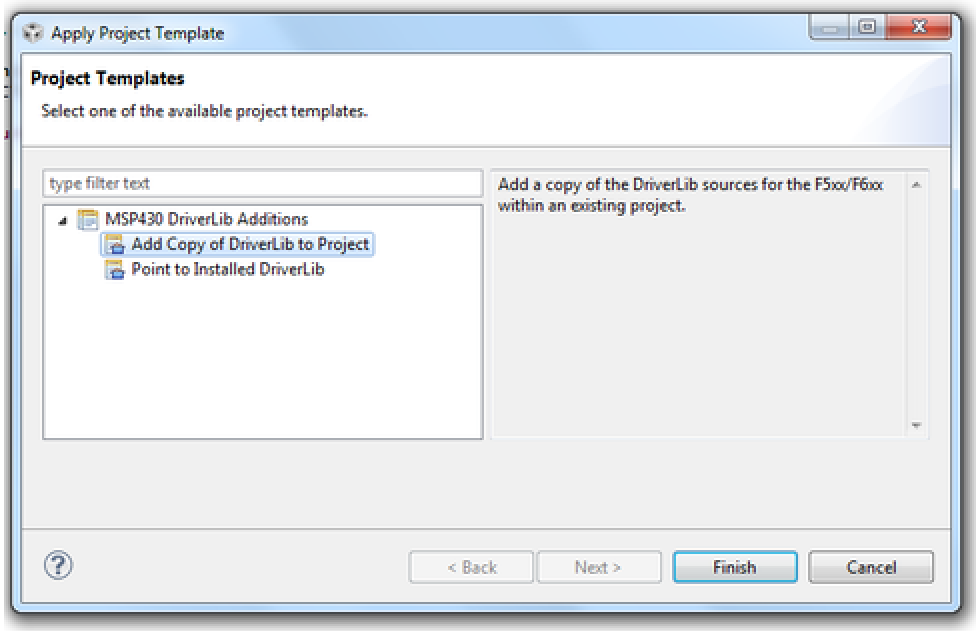
To add driver library to an existing project we recommend using CCS project templates. For driver library to work with project templates CCS must have discovered the driver library RTSC product. For more information refer to the installation steps of the release notes. CCS project templates adds the 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.



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

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

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


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

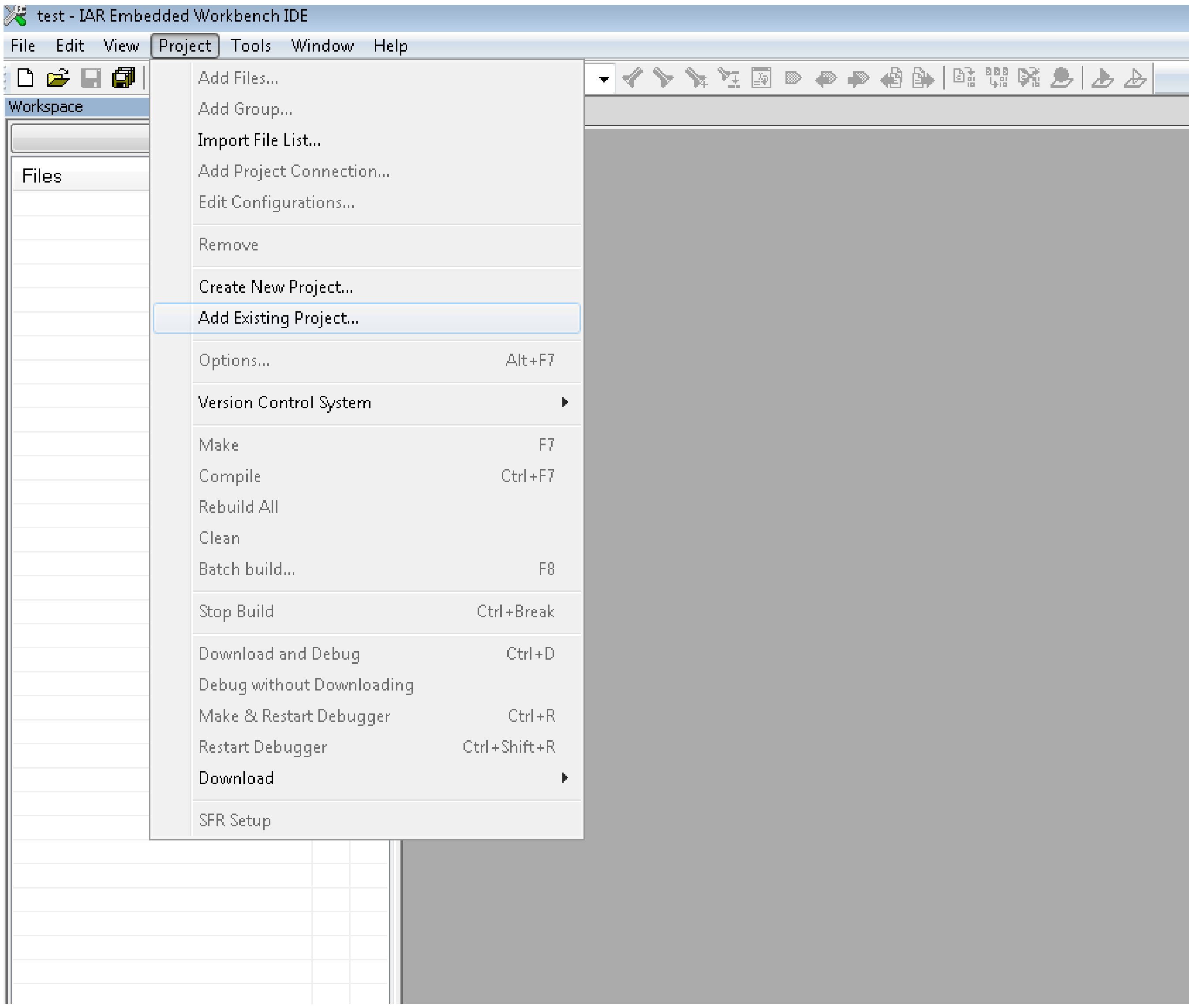
Add a copy of the DriverLib sources for the F5xx/F6xx within an existing project.

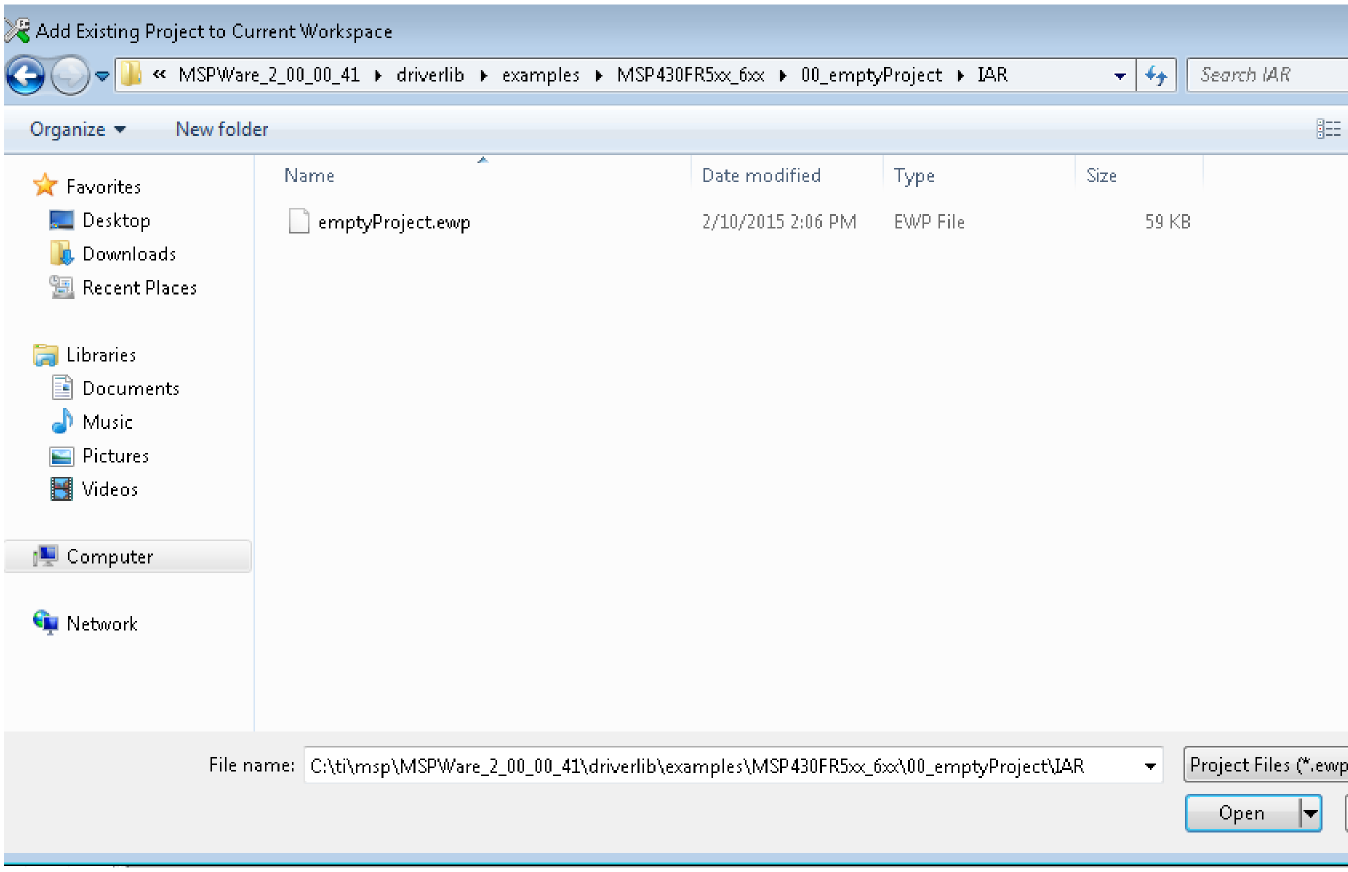
Click ”Finish” and start developing with driver library in your project.

5 How to create a new IAR project that uses Driverlib

5.1 Introduction

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

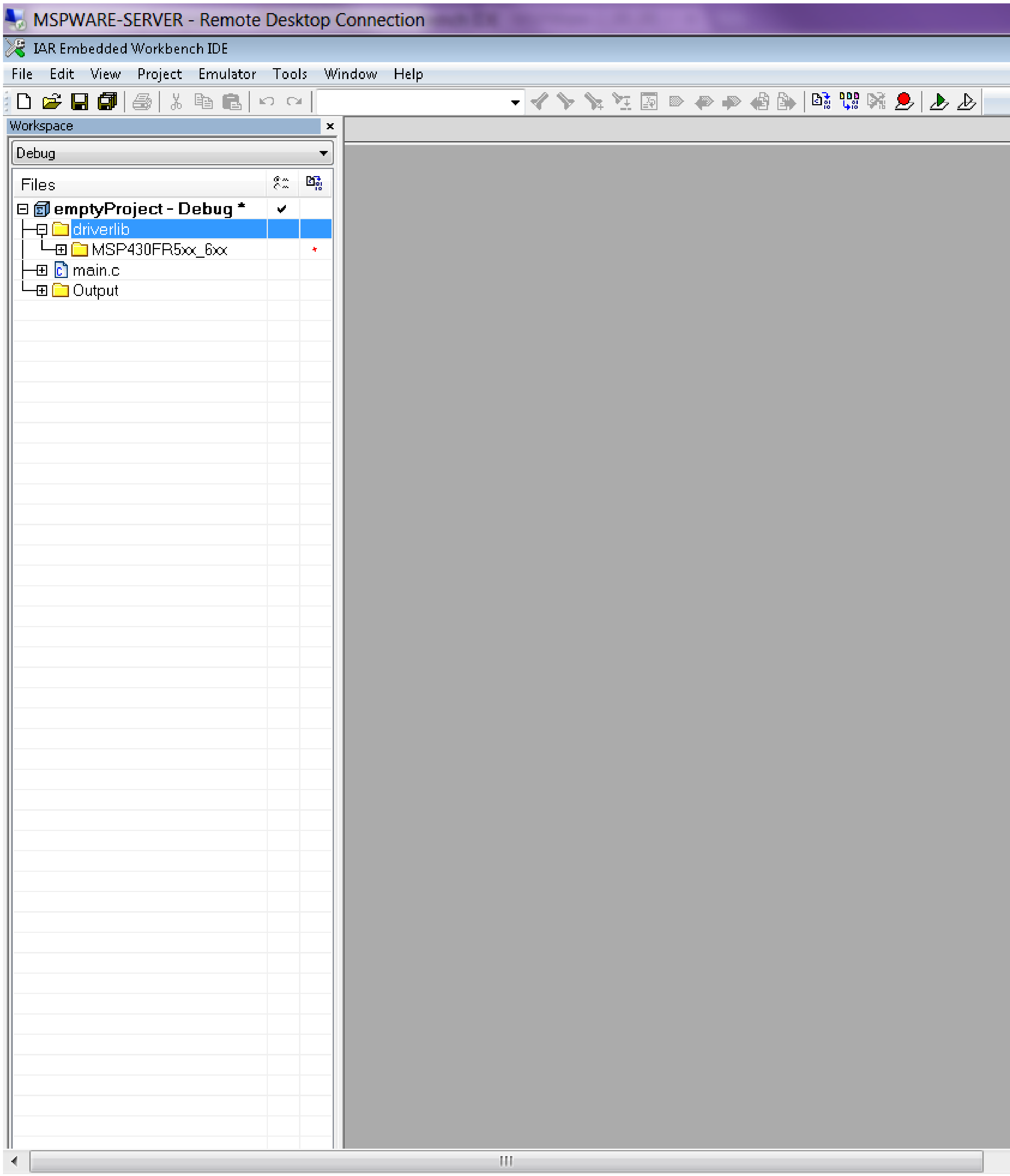



Add Existing Project to Current Workspace
« MSPWare_2_00_00_41 ▸ driverlib ▸ examples ▸ MSP430FR5xx_6xx ▸ 00_emptyProject ▸ IAR
Search IAR
Organize
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Network
Open

Name	Date modified	Type	Size
emptyProject.ewp	2/10/2015 2:06 PM	EWP File	59 KB

File name: C:\ti\msp\MSPWare_2_00_00_41\driverlib\examples\MSP430FR5xx_6xx\00_emptyProject\IAR

Project Files (*.ewp

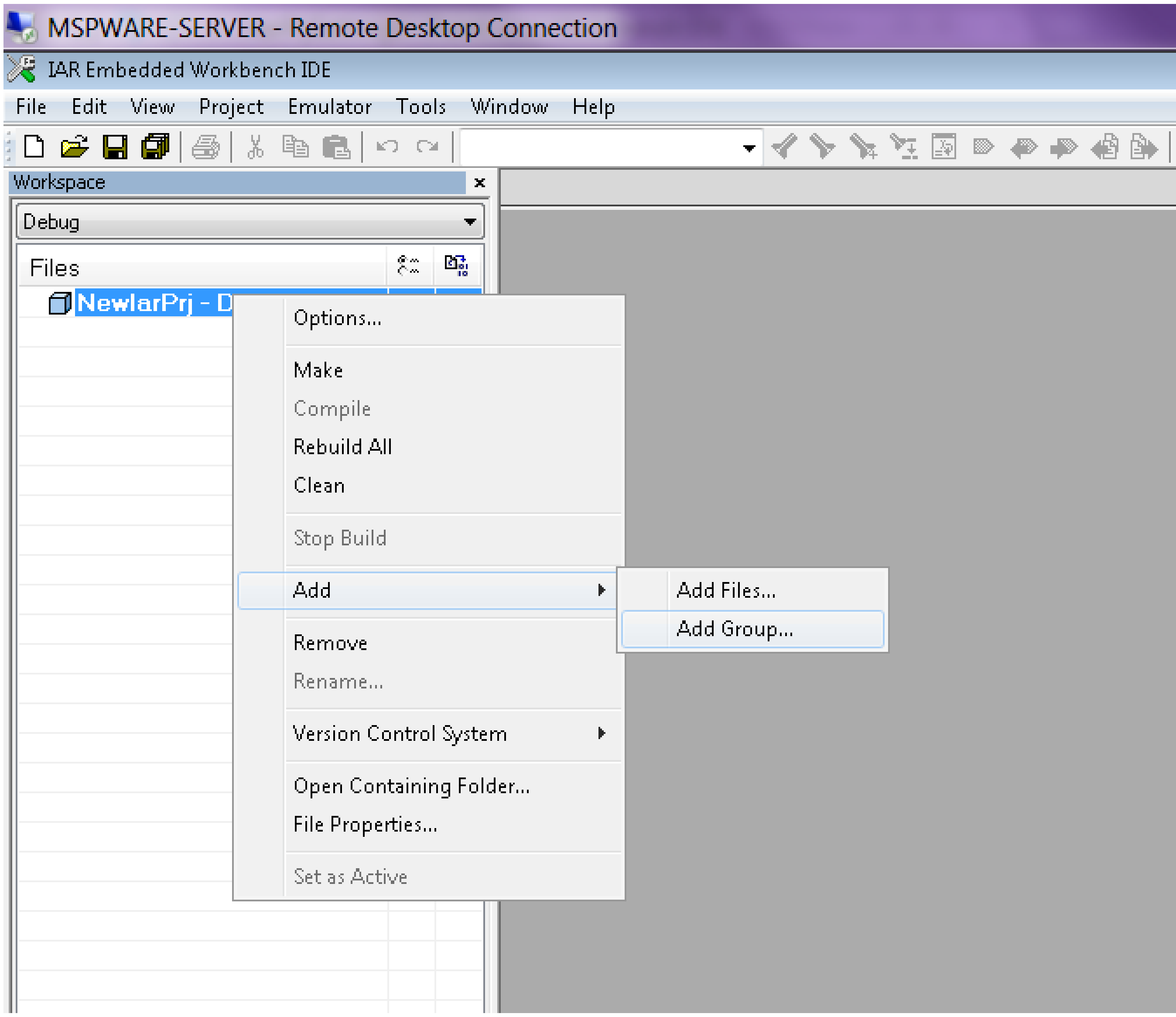

MSPWARE-SERVER - Remote Desktop Connection
IAR Embedded Workbench IDE
File
Edit
View
Project
Emulator
Tools
Window
Help
Workspace
Debug
Files

emptyProject - Debug *
driverlib
MSP430FR5xx_6xx
main.c
Output

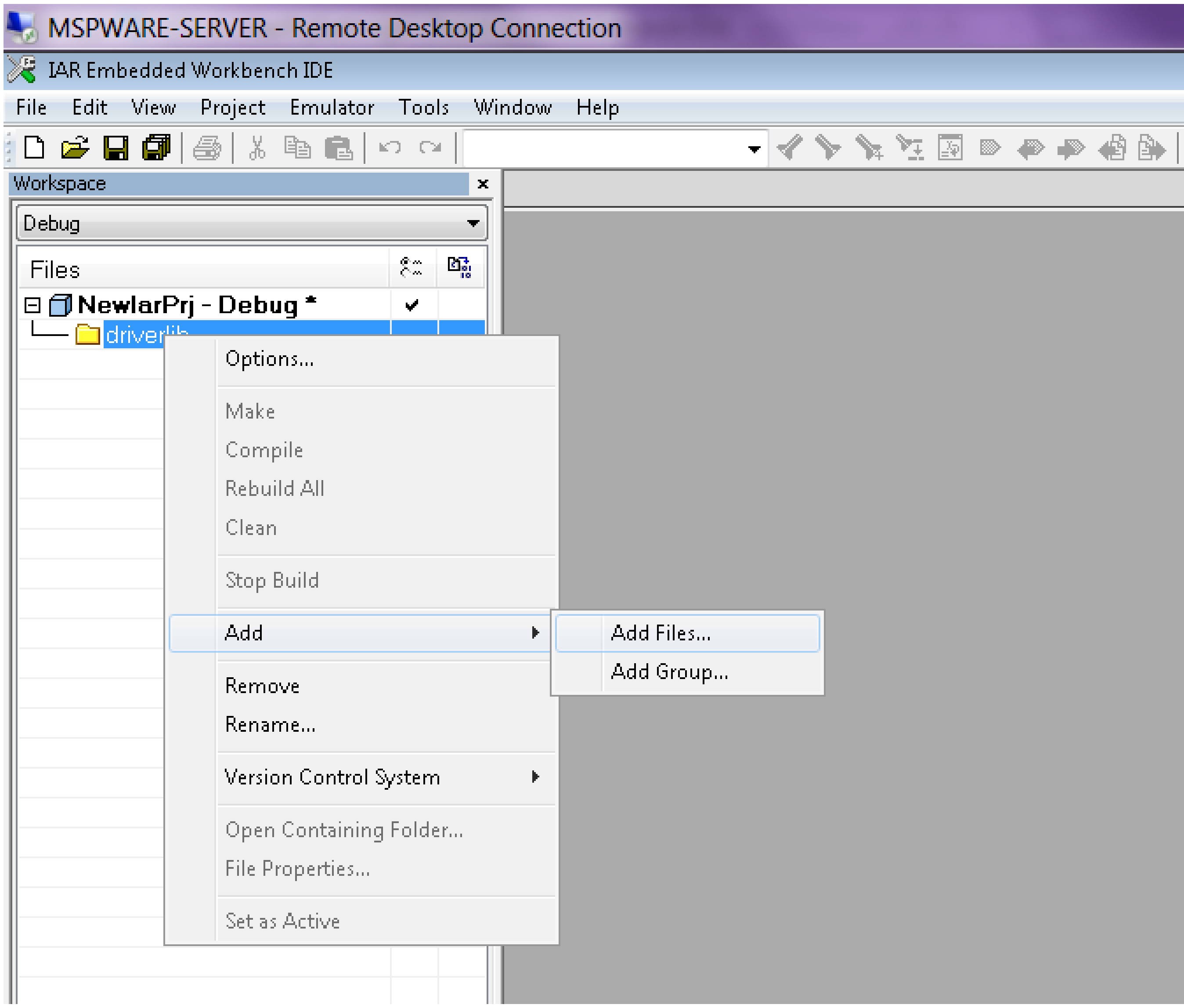
6 How to include driverlib into your existing IAR project

6.1 Introduction

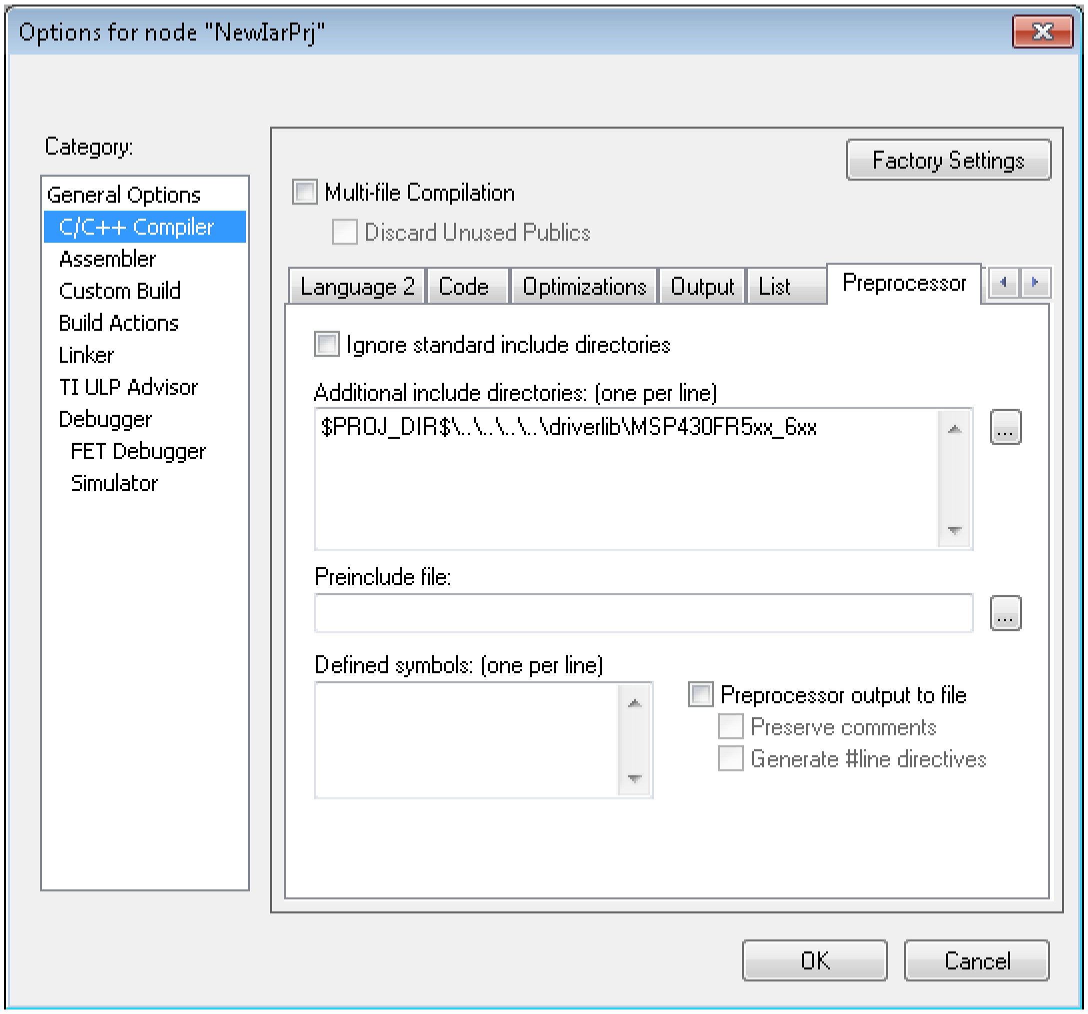
To add driver library to an existing project, right click project click on Add Group - ”driverlib”



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



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



Additional include directories: (one per line)

```
$PROJ_DIR$\..\..\..\..\driverlib\MSP430FR5xx_6xx
```

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

7 10-Bit Analog-to-Digital Converter (ADC10_B)

7.1 Introduction

The 10-Bit Analog-to-Digital (ADC10_B) API provides a set of functions for using the MSP430Ware ADC10_B modules. Functions are provided to initialize the ADC10_B modules, setup signal sources and reference voltages, and manage interrupts for the ADC10_B modules.

The ADC10_B module supports fast 10-bit analog-to-digital conversions. The module implements a 10-bit SAR core together, sample select control and a window comparator.

ADC10_B features include:

- Greater than 200-ksps maximum conversion rate
- Monotonic 10-bit converter with no missing codes
- Sample-and-hold with programmable sampling periods controlled by software or timers
- Conversion initiation by software or different timers
- Software-selectable on chip reference using the REF module or external reference
- Twelve individually configurable external input channels
- Conversion channel for temperature sensor of the REF module
- Selectable conversion clock source
- Single-channel, repeat-single-channel, sequence, and repeat-sequence conversion modes
- Window comparator for low-power monitoring of input signals
- Interrupt vector register for fast decoding of six ADC interrupts (ADC10IFG0, ADC10TOVIFG, ADC10OVIFG, ADC10LOIFG, ADC10INIFG, ADC10HIIFG)

7.2 API Functions

Functions

- bool ADC10_B_init (uint16_t baseAddress, uint16_t sampleHoldSignalSourceSelect, uint8_t clockSourceSelect, uint16_t clockSourceDivider)
  Initializes the ADC10B Module.
- void ADC10_B_enable (uint16_t baseAddress)
  Enables the ADC10B block.
- void ADC10_B_disable (uint16_t baseAddress)
  Disables the ADC10B block.
- void ADC10_B_setupSamplingTimer (uint16_t baseAddress, uint16_t clockCycleHoldCount, uint16_t multipleSamplesEnabled)

Sets up and enables the Sampling Timer Pulse Mode.

- void ADC10_B_disableSamplingTimer (uint16_t baseAddress)
  Disables Sampling Timer Pulse Mode.
- void ADC10_B_configureMemory (uint16_t baseAddress, uint8_t inputSourceSelect, uint8_t positiveRefVoltageSourceSelect, uint8_t negativeRefVoltageSourceSelect)
  Configures the controls of the selected memory buffer.
- void ADC10_B_enableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
  Enables selected ADC10B interrupt sources.
- void ADC10_B_disableInterrupt (uint16_t baseAddress, uint8_t interruptMask)
  Disables selected ADC10B interrupt sources.
- void ADC10_B_clearInterrupt (uint16_t baseAddress, uint8_t interruptFlagMask)
  Clears ADC10B selected interrupt flags.
- uint8_t ADC10_B_getInterruptStatus (uint16_t baseAddress, uint8_t interruptFlagMask)
  Returns the status of the selected memory interrupt flags.
- void ADC10_B_startConversion (uint16_t baseAddress, uint8_t conversionSequenceModeSelect)
  Enables/Starts an Analog-to-Digital Conversion.
- void ADC10_B_disableConversions (uint16_t baseAddress, bool preempt)
  Disables the ADC from converting any more signals.
- uint16_t ADC10_B_getResults (uint16_t baseAddress)
  Returns the raw contents of the specified memory buffer.
- void ADC10_B_setResolution (uint16_t baseAddress, uint8_t resolutionSelect)
  Use to change the resolution of the converted data.
- void ADC10_B_setSampleHoldSignalInversion (uint16_t baseAddress, uint16_t invertedSignal)
  Use to invert or un-invert the sample/hold signal.
- void ADC10_B_setDataReadBackFormat (uint16_t baseAddress, uint16_t readBackFormat)
  Use to set the read-back format of the converted data.
- void ADC10_B_enableReferenceBurst (uint16_t baseAddress)
  Enables the reference buffer's burst ability.
- void ADC10_B_disableReferenceBurst (uint16_t baseAddress)
  Disables the reference buffer's burst ability.
- void ADC10_B_setReferenceBufferSamplingRate (uint16_t baseAddress, uint16_t samplingRateSelect)
  Use to set the reference buffer's sampling rate.
- void ADC10_B_setWindowComp (uint16_t baseAddress, uint16_t highThreshold, uint16_t lowThreshold)
  Sets the high and low threshold for the window comparator feature.
- uint32_t ADC10_B_getMemoryAddressForDMA (uint16_t baseAddress)
  Returns the address of the memory buffer for the DMA module.
- uint8_t ADC10_B_isBusy (uint16_t baseAddress)
  Returns the busy status of the ADC10B core.

7.2.1 Detailed Description

The ADC10_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 ADC10.

The ADC10_B initialization and conversion functions are

- ADC10_B_init()
- ADC10_B_configureMemory()

- ADC10_B_setupSamplingTimer()
- ADC10_B_disableSamplingTimer()
- ADC10_B_setWindowComp()
- ADC10_B_startConversion()
- ADC10_B_disableConversions()
- ADC10_B_getResults()
- ADC10_B_isBusy()

The ADC10_B interrupts are handled by

- ADC10_B_enableInterrupt()
- ADC10_B_disableInterrupt()
- ADC10_B_clearInterrupt()
- ADC10_B_getInterruptStatus()

Auxiliary features of the ADC10_B are handled by

- ADC10_B_setResolution()
- ADC10_B_setSampleHoldSignalInversion()
- ADC10_B_setDataReadBackFormat()
- ADC10_B_enableReferenceBurst()
- ADC10_B_disableReferenceBurst()
- ADC10_B_setReferenceBufferSamplingRate()
- ADC10_B_getMemoryAddressForDMA()
- ADC10_B_enable()
- ADC10_B_disable()

7.2.2 Function Documentation

void ADC10_B_clearInterrupt (uint16_t *baseAddress,* uint8_t *interruptFlagMask*)

Clears ADC10B selected interrupt flags.

The selected ADC10B interrupt flags are cleared, so that it no longer asserts. The memory buffer interrupt flags are only cleared when the memory buffer is accessed.

Parameters

baseAddress	is the base address of the ADC10B module.
interruptFlag←Mask	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: ■ **ADC10_B_OVIFG** - Interrupt flag for when a new conversion is about to overwrite the previous one ■ **ADC10_B_TOVIFG** - Interrupt flag for when a new conversion is starting before the previous one has finished ■ **ADC10_B_HIIFG** - Interrupt flag for when the input signal has gone above the high threshold of the window comparator ■ **ADC10_B_LOIFG** - Interrupt flag for when the input signal has gone below the low threshold of the window comparator ■ **ADC10_B_INIFG** - Interrupt flag for when the input signal is in between the high and low thresholds of the window comparator ■ **ADC10_B_IFG0** - Interrupt flag for new conversion data in the memory buffer

Modified bits of **ADC10IFG** register.

Returns

None

void ADC10_B_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

baseAddress	is the base address of the ADC10B module.
inputSource←Select	is the input that will store the converted data into the specified memory buffer. Valid values are: ■ **ADC10_B_INPUT_A0** [Default] ■ **ADC10_B_INPUT_A1** ■ **ADC10_B_INPUT_A2** ■ **ADC10_B_INPUT_A3** ■ **ADC10_B_INPUT_A4** ■ **ADC10_B_INPUT_A5** ■ **ADC10_B_INPUT_A6** ■ **ADC10_B_INPUT_A7** ■ **ADC10_B_INPUT_VEREF_P** ■ **ADC10_B_INPUT_VEREF_N** ■ **ADC10_B_INPUT_TEMPSENSOR** ■ **ADC10_B_INPUT_BATTERYMONITOR** ■ **ADC10_B_INPUT_A12** ■ **ADC10_B_INPUT_A13** ■ **ADC10_B_INPUT_A14** ■ **ADC10_B_INPUT_A15** Modified bits are **ADC10INCHx** of **ADC10MCTL0** register.
positiveRef←Voltage←SourceSelect	is the reference voltage source to set as the upper limit for the conversion that is to be stored in the specified memory buffer. Valid values are: ■ **ADC10_B_VREFPOS_AVCC** [Default] ■ **ADC10_B_VREFPOS_EXT** ■ **ADC10_B_VREFPOS_INT** Modified bits are **ADC10SREF** of **ADC10MCTL0** register.
negativeRef←Voltage←SourceSelect	is the reference voltage source to set as the lower limit for the conversion that is to be stored in the specified memory buffer. Valid values are: ■ **ADC10_B_VREFNEG_AVSS** [Default] ■ **ADC10_B_VREFNEG_EXT** Modified bits are **ADC10SREF** of **ADC10MCTL0** register.

Returns

None

void ADC10_B_disable (uint16_t *baseAddress*)

Disables the ADC10B block.

This will disable operation of the ADC10B block.

Parameters

baseAddress	is the base address of the ADC10B module.

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

Returns

None

void ADC10_B_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_B_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 ADC10CTL1 and ADC10CTL0

Parameters

baseAddress	is the base address of the ADC10B module.
preempt	specifies if the current conversion should be pre-empted stopped before the end of the conversion Valid values are: ■ **ADC10_B_COMPLETECONVERSION** - Allows the ADC10B to end the current conversion before disabling conversions. ■ **ADC10_B_PREEMPTCONVERSION** - Stops the ADC10B immediately, with unpredicatble 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_B_disableInterrupt (uint16_t *baseAddress,* uint8_t *interruptMask*)

Disables selected ADC10B interrupt sources.

Disables the indicated ADC10B interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Parameters

baseAddress	is the base address of the ADC10B module.
interruptMask	is the bit mask of the memory buffer interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **ADC10_B_OVIE** - Interrupts when a new conversion is about to overwrite the previous one ■ **ADC10_B_TOVIE** - Interrupts when a new conversion is starting before the previous one has finished ■ **ADC10_B_HIIE** - Interrupts when the input signal has gone above the high threshold of the window comparator ■ **ADC10_B_LOIE** - Interrupts when the input signal has gone below the low threshold of the low window comparator ■ **ADC10_B_INIE** - Interrupts when the input signal is in between the high and low thresholds of the window comparator ■ **ADC10_B_IE0** - Interrupt for new conversion data in the memory buffer

Modified bits of **ADC10IE** register.

Returns

None

void ADC10_B_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

baseAddress	is the base address of the ADC10B module.

Modified bits are **ADC10REFBURST** of **ADC10CTL2** register.

Returns

None

void ADC10_B_disableSamplingTimer (uint16_t *baseAddress*)

Disables Sampling Timer Pulse Mode.

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

Parameters

baseAddress	is the base address of the ADC10B module.

Modified bits are **ADC10SHP** of **ADC10CTL1** register.

Returns

None

void ADC10_B_enable (uint16_t *baseAddress*)

Enables the ADC10B block.

This will enable operation of the ADC10B block.

Parameters

baseAddress	is the base address of the ADC10B module.

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

Returns

None

void ADC10_B_enableInterrupt (uint16_t *baseAddress,* uint8_t *interruptMask*)

Enables selected ADC10B interrupt sources.

Enables the indicated ADC10B interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the ADC10B module.
interruptMask	is the bit mask of the memory buffer interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **ADC10_B_OVIE** - Interrupts when a new conversion is about to overwrite the previous one ■ **ADC10_B_TOVIE** - Interrupts when a new conversion is starting before the previous one has finished ■ **ADC10_B_HIIE** - Interrupts when the input signal has gone above the high threshold of the window comparator ■ **ADC10_B_LOIE** - Interrupts when the input signal has gone below the low threshold of the low window comparator ■ **ADC10_B_INIE** - Interrupts when the input signal is in between the high and low thresholds of the window comparator ■ **ADC10_B_IE0** - Interrupt for new conversion data in the memory buffer

Modified bits of **ADC10IE** register.

Returns

None

void ADC10_B_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

baseAddress	is the base address of the ADC10B module.

Modified bits are **ADC10REFBURST** of **ADC10CTL2** register.

Returns

None

uint8_t ADC10_B_getInterruptStatus (uint16_t *baseAddress,* uint8_t *interruptFlagMask*)

Returns the status of the selected memory interrupt flags.

Returns the status of the selected interrupt flags.

Parameters

baseAddress	is the base address of the ADC10B module.
interruptFlag←Mask	is a bit mask of the interrupt flags status to be returned. Mask value is the logical OR of any of the following: ■ **ADC10_B_OVIFG** - Interrupt flag for when a new conversion is about to overwrite the previous one ■ **ADC10_B_TOVIFG** - Interrupt flag for when a new conversion is starting before the previous one has finished ■ **ADC10_B_HIIFG** - Interrupt flag for when the input signal has gone above the high threshold of the window comparator ■ **ADC10_B_LOIFG** - Interrupt flag for when the input signal has gone below the low threshold of the window comparator ■ **ADC10_B_INIFG** - Interrupt flag for when the input signal is in between the high and low thresholds of the window comparator ■ **ADC10_B_IFG0** - Interrupt flag for new conversion data in the memory buffer

Modified bits of **ADC10IFG** register.

Returns

The current interrupt flag status for the corresponding mask.

uint32_t ADC10_B_getMemoryAddressForDMA (uint16_t *baseAddress*)

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

Parameters

baseAddress	is the base address of the ADC10B module.

Returns

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

uint16_t ADC10_B_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

baseAddress	is the base address of the ADC10B module.

Returns

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

bool ADC10_B_init (uint16_t *baseAddress,* uint16_t *sampleHoldSignalSourceSelect,* uint8_t *clockSourceSelect,* uint16_t *clockSourceDivider*)

Initializes the ADC10B Module.

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

Parameters

baseAddress	is the base address of the ADC10B module.
sampleHold←↩ SignalSource←↩ Select	is the signal that will trigger a sample-and-hold for an input signal to be converted. This parameter is device specific and sources should be found in the device's datasheet. Valid values are: ■ **ADC10_B_SAMPLEHOLDSOURCE_SC** [Default] ■ **ADC10_B_SAMPLEHOLDSOURCE_1** ■ **ADC10_B_SAMPLEHOLDSOURCE_2** ■ **ADC10_B_SAMPLEHOLDSOURCE_3** Modified bits are **ADC10SHSx** of **ADC10CTL1** register.
clockSource←↩ Select	selects the clock that will be used by the ADC10B core and the sampling timer if a sampling pulse mode is enabled. Valid values are: ■ **ADC10_B_CLOCKSOURCE_ADC10OSC** [Default] - MODOSC 5 MHz oscillator from the clock system ■ **ADC10_B_CLOCKSOURCE_ACLK** - The Auxiliary Clock ■ **ADC10_B_CLOCKSOURCE_MCLK** - The Master Clock ■ **ADC10_B_CLOCKSOURCE_SMCLK** - The Sub-Master Clock Modified bits are **ADC10SSELx** of **ADC10CTL1** register.
clockSource←↩ Divider	selects the amount that the clock will be divided. Valid values are: ■ **ADC10_B_CLOCKDIVIDER_1** [Default] ■ **ADC10_B_CLOCKDIVIDER_2** ■ **ADC10_B_CLOCKDIVIDER_3** ■ **ADC10_B_CLOCKDIVIDER_4** ■ **ADC10_B_CLOCKDIVIDER_5** ■ **ADC10_B_CLOCKDIVIDER_6** ■ **ADC10_B_CLOCKDIVIDER_7** ■ **ADC10_B_CLOCKDIVIDER_8** ■ **ADC10_B_CLOCKDIVIDER_12** ■ **ADC10_B_CLOCKDIVIDER_16** ■ **ADC10_B_CLOCKDIVIDER_20** ■ **ADC10_B_CLOCKDIVIDER_24** ■ **ADC10_B_CLOCKDIVIDER_28** ■ **ADC10_B_CLOCKDIVIDER_32** ■ **ADC10_B_CLOCKDIVIDER_64** ■ **ADC10_B_CLOCKDIVIDER_128** ■ **ADC10_B_CLOCKDIVIDER_192** ■ **ADC10_B_CLOCKDIVIDER_256** ■ **ADC10_B_CLOCKDIVIDER_320** ■ **ADC10_B_CLOCKDIVIDER_384** ■ **ADC10_B_CLOCKDIVIDER_448** ■ **ADC10_B_CLOCKDIVIDER_512** Modified bits are **ADC10DIVx** of **ADC10CTL1** register; bits **ADC10PDIVx** of **AD←↩ C10CTL2** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process.

uint8_t ADC10_B_isBusy (uint16_t *baseAddress*)

Returns the busy status of the ADC10B core.

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

Parameters

baseAddress	is the base address of the ADC10B module.

Returns

ADC10_B_BUSY or ADC10_B_NOTBUSY dependent if there is a conversion currently taking place. Return one of the following:

- **ADC10_B_NOTBUSY**
- **ADC10_B_BUSY**

void ADC10_B_setDataReadBackFormat (uint16_t *baseAddress,* uint16_t *readBackFormat*)

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

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

Parameters

baseAddress	is the base address of the ADC10B module.
readBackFormat	is the specified format to store the conversions in the memory buffer. Valid values are: ■ **ADC10_B_UNSIGNED_BINARY** [Default] ■ **ADC10_B_SIGNED_2SCOMPLEMENT** Modified bits are **ADC10DF** of **ADC10CTL2** register.

Returns

None

void ADC10_B_setReferenceBufferSamplingRate (uint16_t *baseAddress,* uint16_t *samplingRateSelect*)

Use to set the reference buffer's sampling rate.

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

Parameters

baseAddress	is the base address of the ADC10B module.
samplingRate←Select	is the specified maximum sampling rate. Valid values are: ■ **ADC10_B_MAXSAMPLINGRATE_200KSPS** [Default] ■ **ADC10_B_MAXSAMPLINGRATE_50KSPS** Modified bits are **ADC10SR** of **ADC10CTL2** register.

Modified bits of **ADC10CTL2** register.

Returns

None

void ADC10_B_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

baseAddress	is the base address of the ADC10B module.
resolutionSelect	determines the resolution of the converted data. Valid values are: ■ **ADC10_B_RESOLUTION_8BIT** ■ **ADC10_B_RESOLUTION_10BIT** [Default] Modified bits are **ADC10RES** of **ADC10CTL2** register.

Returns

None

void ADC10_B_setSampleHoldSignalInversion (uint16_t *baseAddress,* uint16_t *invertedSignal*)

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

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

Parameters

baseAddress	is the base address of the ADC10B module.
invertedSignal	set if the sample/hold signal should be inverted Valid values are: ■ **ADC10_B_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_B_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_B_setupSamplingTimer (uint16_t *baseAddress,* uint16_t *clockCycleHoldCount,* uint16_t *multipleSamplesEnabled*)

Sets up and enables the Sampling Timer Pulse Mode.

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

Parameters

baseAddress	is the base address of the ADC10B module.
clockCycle←HoldCount	sets the amount of clock cycles to sample-and- hold for the memory buffer. Valid values are: ■ **ADC10_B_CYCLEHOLD_4_CYCLES** [Default] ■ **ADC10_B_CYCLEHOLD_8_CYCLES** ■ **ADC10_B_CYCLEHOLD_16_CYCLES** ■ **ADC10_B_CYCLEHOLD_32_CYCLES** ■ **ADC10_B_CYCLEHOLD_64_CYCLES** ■ **ADC10_B_CYCLEHOLD_96_CYCLES** ■ **ADC10_B_CYCLEHOLD_128_CYCLES** ■ **ADC10_B_CYCLEHOLD_192_CYCLES** ■ **ADC10_B_CYCLEHOLD_256_CYCLES** ■ **ADC10_B_CYCLEHOLD_384_CYCLES** ■ **ADC10_B_CYCLEHOLD_512_CYCLES** ■ **ADC10_B_CYCLEHOLD_768_CYCLES** ■ **ADC10_B_CYCLEHOLD_1024_CYCLES** Modified bits are **ADC10SHTx** of **ADC10CTL0** register.
multiple←Samples←Enabled	allows multiple conversions to start without a trigger signal from the sample/hold signal Valid values are: ■ **ADC10_B_MULTIPLESAMPLESDISABLE** - a timer trigger will be needed to start every ADC conversion. ■ **ADC10_B_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_B_setWindowComp (uint16_t *baseAddress,* uint16_t *highThreshold,* uint16_t *lowThreshold*)

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

Sets the high and low threshold for the window comparator feature. Use the ADC10HIIE, ADC10INIE, ADC10LOIE interrupts to utilize this feature.

Parameters

baseAddress	is the base address of the ADC10B module.
highThreshold	is the upper bound that could trip an interrupt for the window comparator.
lowThreshold	is the lower bound that could trip on interrupt for the window comparator.

Modified bits of **ADC10LO** register and bits of **ADC10HI** register.

Returns

None

void ADC10_B_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_B_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_B_stopConversions() has to be called to re-initialize the ADC, reconfigure a memory buffer control, enable/disable the sampling timer, or to change the internal reference voltage.

Parameters

baseAddress	is the base address of the ADC10B module.
conversion↩ Sequence↩ ModeSelect	determines the ADC operating mode. Valid values are: ■ **ADC10_B_SINGLECHANNEL** [Default] - one-time conversion of a single channel into a single memory buffer ■ **ADC10_B_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_B_REPEATED_SINGLECHANNEL** - repeated conversions of one channel into a single memory buffer ■ **ADC10_B_REPEATED_SEQOFCHANNELS** - repeated conversions of multiple channels into the specified starting memory buffer and each subsequent memory buffer up until the conversion is stored in a memory buffer dedicated as the end-of-sequence by the memory's control register Modified bits are **ADC10CONSEQx** of **ADC10CTL1** register.

Returns

None

7.3 Programming Example

The following example shows how to initialize and use the ADC10_B API to start a single channel, single conversion.

```
// Initialize ADC10_B with ADC10_B's built-in oscillator
ADC10_B_init (ADC10_B_BASE,
            ADC10_B_SAMPLEHOLDSOURCE_SC,
            ADC10_B_CLOCKSOURCE_ADC10OSC,
            ADC10_B_CLOCKDIVIDEBY_1);

//Switch ON ADC10_B
ADC10_B_enable(ADC10_B_BASE);

// Setup sampling timer to sample-and-hold for 16 clock cycles
ADC10_B_setupSamplingTimer (ADC10_B_BASE,
                          ADC10_B_CYCLEHOLD_16_CYCLES,
                          FALSE);

// Configure the Input to the Memory Buffer with the specified Reference Voltages
ADC10_B_configureMemory(ADC10_B_BASE,
                       ADC10_B_INPUT_A0,
                       ADC10_B_VREFPOS_AVCC, // Vref+ = AVcc
                       ADC10_B_VREFNEG_AVSS  // Vref- = AVss
                       );
while (1)
{
    // Start a single conversion, no repeating or sequences.
    ADC10_B_startConversion (ADC10_B_BASE,
                           ADC10_B_SINGLECHANNEL);

    // Wait for the Interrupt Flag to assert
    while( !(ADC10_B_getInterruptStatus(ADC10_B_BASE,ADC10IFG0)) );

    // Clear the Interrupt Flag and start another conversion
    ADC10_B_clearInterrupt(ADC10_B_BASE,ADC10IFG0);
}
```

8 Comparator (COMP_D)

8.1 Introduction

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

The Comp_D 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_D 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_D module also has control over the REF module to generate a reference voltage as an input.

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

8.2 API Functions

Functions

- bool Comp_D_init (uint16_t baseAddress, Comp_D_initParam *param)
  Initializes the Comp_D Module.
- void Comp_D_setReferenceVoltage (uint16_t baseAddress, uint16_t supplyVoltageReferenceBase, uint16_t lowerLimitSupplyVoltageFractionOf32, uint16_t upperLimitSupplyVoltageFractionOf32)
  Generates a Reference Voltage to the terminal selected during initialization.
- void Comp_D_setReferenceAccuracy (uint16_t baseAddress, uint16_t referenceAccuracy)
  Sets the reference accuracy.
- void Comp_D_enableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Enables selected Comparator interrupt sources.
- void Comp_D_disableInterrupt (uint16_t baseAddress, uint16_t interruptMask)
  Disables selected Comparator interrupt sources.
- void Comp_D_clearInterrupt (uint16_t baseAddress, uint16_t interruptFlagMask)
  Clears Comparator interrupt flags.
- uint8_t Comp_D_getInterruptStatus (uint16_t baseAddress, uint16_t interruptFlagMask)
  Gets the current Comparator interrupt status.
- void Comp_D_setInterruptEdgeDirection (uint16_t baseAddress, uint16_t edgeDirection)
  Explicitly sets the edge direction that would trigger an interrupt.
- void Comp_D_toggleInterruptEdgeDirection (uint16_t baseAddress)
  Toggles the edge direction that would trigger an interrupt.
- void Comp_D_enable (uint16_t baseAddress)
  Turns on the Comparator module.

- void Comp_D_disable (uint16_t baseAddress)
  Turns off the Comparator module.
- void Comp_D_shortInputs (uint16_t baseAddress)
  Shorts the two input pins chosen during initialization.
- void Comp_D_unshortInputs (uint16_t baseAddress)
  Disables the short of the two input pins chosen during initialization.
- void Comp_D_disableInputBuffer (uint16_t baseAddress, uint8_t inputPort)
  Disables the input buffer of the selected input port to effectively allow for analog signals.
- void Comp_D_enableInputBuffer (uint16_t baseAddress, uint8_t inputPort)
  Enables the input buffer of the selected input port to allow for digital signals.
- void Comp_D_swapIO (uint16_t baseAddress)
  Toggles the bit that swaps which terminals the inputs go to, while also inverting the output of the comparator.
- uint16_t Comp_D_outputValue (uint16_t baseAddress)
  Returns the output value of the Comp_D module.

8.2.1 Detailed Description

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

The Comp_D initialization and output functions are

- Comp_D_init()
- Comp_D_setReferenceVoltage()
- Comp_D_enable()
- Comp_D_disable()
- Comp_D_outputValue()

The Comp_D interrupts are handled by

- Comp_D_enableInterrupt()
- Comp_D_disableInterrupt()
- Comp_D_clearInterrupt()
- Comp_D_getInterruptStatus()
- Comp_D_setInterruptEdgeDirection()
- Comp_D_toggleInterruptEdgeDirection()

Auxiliary features of the Comp_D are handled by

- Comp_D_enableShortOfInputs()
- Comp_D_disableShortOfInputs()
- Comp_D_disableInputBuffer()
- Comp_D_enableInputBuffer()
- Comp_D_swapIO()
- Comp_D_setReferenceAccuracy()

8.2.2 Function Documentation

void Comp_D_clearInterrupt (uint16_t *baseAddress,* uint16_t *interruptFlagMask*)

Clears Comparator interrupt flags.

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

Parameters

baseAddress	is the base address of the COMP_D module.
interruptFlag←Mask	Mask value is the logical OR of any of the following: ■ **COMP_D_INTERRUPT_FLAG** - Output interrupt flag ■ **COMP_D_INTERRUPT_FLAG_INVERTED_POLARITY** - Output interrupt flag inverted polarity

void Comp_D_disable (uint16_t *baseAddress*)

Turns off the Comparator module.

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

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

None

void Comp_D_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 comparator input pins. This bit is automatically set when the input is initialized to be used with the comparator module. This function should be used whenever an analog input is connected to one of these pins to prevent parasitic voltage from causing unexpected results.

Parameters

baseAddress	is the base address of the COMP_D module.
inputPort	is the port in which the input buffer will be disabled. Valid values are: ■ **COMP_D_INPUT0** [Default] ■ **COMP_D_INPUT1** ■ **COMP_D_INPUT2** ■ **COMP_D_INPUT3** ■ **COMP_D_INPUT4** ■ **COMP_D_INPUT5** ■ **COMP_D_INPUT6** ■ **COMP_D_INPUT7** ■ **COMP_D_INPUT8** ■ **COMP_D_INPUT9** ■ **COMP_D_INPUT10** ■ **COMP_D_INPUT11** ■ **COMP_D_INPUT12** ■ **COMP_D_INPUT13** ■ **COMP_D_INPUT14** ■ **COMP_D_INPUT15** ■ **COMP_D_VREF** Modified bits are **CDPDx** of **CDCTL3** register.

Returns

None

void Comp_D_disableInterrupt (uint16_t *baseAddress,* uint16_t *interruptMask*)

Disables selected Comparator interrupt sources.

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

Parameters

baseAddress	is the base address of the COMP_D module.
interruptMask	Mask value is the logical OR of any of the following: ■ **COMP_D_INTERRUPT** - Output interrupt ■ **COMP_D_INTERRUPT_INVERTED_POLARITY** - Output interrupt inverted polarity

void Comp_D_enable (uint16_t *baseAddress*)

Turns on the Comparator module.

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

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

None

void Comp_D_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 comparator input pins. This should not be reset if there is an analog signal connected to the specified input pin to prevent from unexpected results.

Parameters

baseAddress	is the base address of the COMP_D module.
inputPort	is the port in which the input buffer will be enabled. Valid values are: ■ **COMP_D_INPUT0** [Default] ■ **COMP_D_INPUT1** ■ **COMP_D_INPUT2** ■ **COMP_D_INPUT3** ■ **COMP_D_INPUT4** ■ **COMP_D_INPUT5** ■ **COMP_D_INPUT6** ■ **COMP_D_INPUT7** ■ **COMP_D_INPUT8** ■ **COMP_D_INPUT9** ■ **COMP_D_INPUT10** ■ **COMP_D_INPUT11** ■ **COMP_D_INPUT12** ■ **COMP_D_INPUT13** ■ **COMP_D_INPUT14** ■ **COMP_D_INPUT15** ■ **COMP_D_VREF** Modified bits are **CDPDx** of **CDCTL3** register.

Returns

None

void Comp_D_enableInterrupt (uint16_t *baseAddress,* uint16_t *interruptMask*)

Enables selected Comparator interrupt sources.

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

Parameters

baseAddress	is the base address of the COMP_D module.
interruptMask	Mask value is the logical OR of any of the following: ■ **COMP_D_INTERRUPT** - Output interrupt ■ **COMP_D_INTERRUPT_INVERTED_POLARITY** - Output interrupt inverted polarity

uint8_t Comp_D_getInterruptStatus (uint16_t *baseAddress,* uint16_t *interruptFlagMask*)

Gets the current Comparator interrupt status.

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

Parameters

baseAddress	is the base address of the COMP_D module.
interruptFlag←Mask	Mask value is the logical OR of any of the following: ■ **COMP_D_INTERRUPT_FLAG** - Output interrupt flag ■ **COMP_D_INTERRUPT_FLAG_INVERTED_POLARITY** - Output interrupt flag inverted polarity

bool Comp_D_init (uint16_t *baseAddress,* **Comp_D_initParam** ∗ *param*)

Initializes the Comp_D Module.

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

Parameters

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the initialization process

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

uint16_t Comp_D_outputValue (uint16_t *baseAddress*)

Returns the output value of the Comp_D module.

Returns the output value of the Comp_D module.

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

Comp_D_HIGH or Comp_D_LOW as the output value of the Comparator module. Return one of the following:

- **Comp_D_HIGH**
- **Comp_D_LOW**
  indicates the output should be normal

void Comp_D_setInterruptEdgeDirection (uint16_t *baseAddress,* uint16_t *edgeDirection*)

Explicitly sets the edge direction that would trigger an interrupt.

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

Parameters

baseAddress	is the base address of the COMP_D module.
edgeDirection	determines which direction the edge would have to go to generate an interrupt based on the non-inverted interrupt flag. Valid values are: ■ **COMP_D_FALLINGEDGE** [Default] - sets the bit to generate an interrupt when the output of the comparator falls from HIGH to LOW if the normal interrupt bit is set(and LOW to HIGH if the inverted interrupt enable bit is set). ■ **COMP_D_RISINGEDGE** - sets the bit to generate an interrupt when the output of the comparator rises from LOW to HIGH if the normal interrupt bit is set(and HIGH to LOW if the inverted interrupt enable bit is set). Modified bits are **CDIES** of **CDCTL1** register.

Returns

None

void Comp_D_setReferenceAccuracy (uint16_t *baseAddress,* uint16_t *referenceAccuracy*)

Sets the reference accuracy.

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

Parameters

baseAddress	is the base address of the COMP_D module.
reference←↩ Accuracy	is the reference accuracy setting of the comparator. Clocked is for low power/low accuracy. Valid values are: ■ **COMP_D_ACCURACY_STATIC** ■ **COMP_D_ACCURACY_CLOCKED** Modified bits are **CDREFACC** of **CDCTL2** register.

Returns

None

void Comp_D_setReferenceVoltage (uint16_t *baseAddress,* uint16_t *supplyVoltage←↩ ReferenceBase,* uint16_t *lowerLimitSupplyVoltageFractionOf32,* uint16_t *upperLimitSupplyVoltageFractionOf32*)

Generates a Reference Voltage to the terminal selected during initialization.

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

Parameters

baseAddress	is the base address of the COMP_D module.
supplyVoltage↩ ReferenceBase	decides the source and max amount of Voltage that can be used as a reference. Valid values are: ■ **COMP_D_REFERENCE_AMPLIFIER_DISABLED** ■ **COMP_D_VREFBASE1_5V** ■ **COMP_D_VREFBASE2_0V** ■ **COMP_D_VREFBASE2_5V** Modified bits are **CDREFL** of **CDCTL2** register.
lowerLimit↩ SupplyVoltage↩ FractionOf32	is the numerator of the equation to generate the reference voltage for the lower limit reference voltage. Modified bits are **CDREF0** of **CDCTL2** register.
upperLimit↩ SupplyVoltage↩ FractionOf32	is the numerator of the equation to generate the reference voltage for the upper limit reference voltage. Modified bits are **CDREF1** of **CDCTL2** register.

Returns

None

void Comp_D_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 comparator.

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

None

void Comp_D_swapIO (uint16_t *baseAddress*)

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

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

baseAddress	is the base address of the COMP_D module.

Returns

None

void Comp_D_toggleInterruptEdgeDirection (uint16_t *baseAddress*)

Toggles the edge direction that would trigger an interrupt.

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

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

None

void Comp_D_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 comparator.

Parameters

baseAddress	is the base address of the COMP_D module.

Returns

None

8.3 Programming Example

The following example shows how to initialize and use the Comp_D 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 D module
/* Base Address of Comparator D,
  Pin CD2 to Positive(+) Terminal,
  Reference Voltage to Negative(-) Terminal,
  Normal Power Mode,
  Output Filter On with minimal delay,
  Non-Inverted Output Polarity
*/
Comp_D_initParam param = {0};
```

```
param.positiveTerminalInput = COMP_D_INPUT2;
param.negativeTerminalInput = COMP_D_VREF;
param.outputFilterEnableAndDelayLevel = COMP_D_FILTEROUTPUT_OFF;
param.invertedOutputPolarity = COMP_D_NORMALOUTPUTPOLARITY;
Comp_D_init(COMP_D_BASE, &param);

// Set the reference voltage that is being supplied to the (-) terminal
/* Base Address of Comparator D,
 Reference Voltage of 2.0 V,
 Upper Limit of 2.0*(32/32) = 2.0V,
 Lower Limit of 2.0*(32/32) = 2.0V
*/
Comp_D_setReferenceVoltage(COMP_D_BASE,
    COMP_D_VREFBASE2_0V,
    32,
    32,
    COMP_D_ACCURACY_STATIC
    );

  //Disable Input Buffer on P1.2/CD2
  /* Base Address of Comparator D,
    * Input Buffer port
    * Selecting the CDx input pin to the comparator
    * multiplexer with the CDx bits automatically
    * disables output driver and input buffer for
    * that pin, regardless of the state of the
    * associated CDPD.x bit
    */
    Comp_D_disableInputBuffer(COMP_D_BASE,
        COMP_D_INPUT2);
// Allow power to Comparator module
Comp_D_enable(COMP_D_BASE);

__delay_cycles(400);           // delay for the reference to settle
```

9 Cyclical Redundancy Check (CRC)

9.1 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).

9.2 API Functions

Functions

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

9.2.1 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()

9.2.2 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

baseAddress	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 pf the Signature Result.

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

Parameters

baseAddress	is the base address of the CRC module.

Returns

The value currently in the data register

uint16_t CRC_getResultBitsReversed (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the CRC module.

Returns

The bit-wise reversed format of the Signature Result

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

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

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

Parameters

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

Returns

None

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

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

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

Parameters

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

Returns

None

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

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

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

Parameters

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

Returns

None

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

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

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

Parameters

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

Returns

None

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

Sets the seed for the CRC.

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

Parameters

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

Returns

None

9.3 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);
```

10 Clock System (CS)

10.1 Introduction

The clock system module supports low system cost and low power consumption. Using three internal clock signals, the user can select the best balance of performance and low power consumption. The clock module can be configured to operate without any external components, with one or two external crystals,or with resonators, under full software control.

The clock system module includes up to five clock sources:

- XT1CLK - Low-frequency/high-frequency oscillator that can be used either with low-frequency 32768-Hz watch crystals, standard crystals, resonators, or external clock sources in the 4 MHz to 24 MHz range. When optional XT2 is present, the XT1 high-frequency mode may or may not be available, depending on the device configuration. See the device-specific data sheet for supported functions.
- VLOCLK - Internal very-low-power low-frequency oscillator with 10-kHz typical frequency
- DCOCLK - Internal digitally controlled oscillator (DCO) with three selectable fixed frequencies
- XT2CLK - Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 4 MHz to 24 MHz range. See device-specific data sheet for availability.

Four system clock signals are available from the clock module:

- ACLK - Auxiliary clock. The ACLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. ACLK can be divided by 1, 2, 4, 8, 16, or 32. ACLK is software selectable by individual peripheral modules.
- MCLK - Master clock. MCLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. MCLK can be divided by 1, 2, 4, 8, 16, or 32. MCLK is used by the CPU and system.
- SMCLK - Subsystem master clock. SMCLK is software selectable as XT1CLK, VLOCLK, DCOCLK, and when available, XT2CLK. SMCLK is software selectable by individual peripheral modules.
- MODCLK - Module clock. MODCLK is used by various peripheral modules and is sourced by MODOSC.

Fail-Safe logic The crystal oscillator faults are set if the corresponding crystal oscillator is turned on and not operating properly. Once set, the fault bits remain set until reset in software, regardless if the fault condition no longer exists. If the user clears the fault bits and the fault condition still exists, the fault bits are automatically set, otherwise they remain cleared.

The OFIFG oscillator-fault interrupt flag is set and latched at POR or when any oscillator fault is detected. When OFIFG is set and OFIE is set, the OFIFG requests a user NMI. When the interrupt is granted, the OFIE is not reset automatically as it is in previous MSP430 families. It is no longer required to reset the OFIE. NMI entry/exit circuitry removes this requirement. The OFIFG flag must be cleared by software. The source of the fault can be identified by checking the individual fault bits.

If XT1 in LF mode is sourcing any system clock (ACLK, MCLK, or SMCLK), and a fault is detected, the system clock is automatically switched to the VLO for its clock source (VLOCLK). Similarly, if XT1 in HF mode is sourcing any system clock and a fault is detected, the system clock is automatically switched to MODOSC for its clock source (MODCLK).

When XT2 (if available) is sourcing any system clock and a fault is detected, the system clock is automatically switched to MODOSC for its clock source (MODCLK).

The fail-safe logic does not change the respective SELA, SELM, and SELS bit settings. The fail-safe mechanism behaves the same in normal and bypass modes.

10.2 API Functions

Macros

- #define **CS_DCO_FREQ_1** 5330000
- #define **CS_DCO_FREQ_2** 6670000
- #define **CS_DCO_FREQ_3** 8000000
- #define **CS_DCO_FREQ_4** 16000000
- #define **CS_DCO_FREQ_5** 20000000
- #define **CS_DCO_FREQ_6** 24000000
- #define **CS_VLOCLK_FREQUENCY** 10000
- #define **CS_MODCLK_FREQUENCY** 5000000
- #define **CS_LFMODCLK_FREQUENCY** 39062
- #define **XT1_FREQUENCY_THRESHOLD** 50000

Functions

- void CS_setExternalClockSource (uint32_t XT1CLK_frequency, uint32_t XT2CLK_frequency)
  Sets the external clock source.
- void CS_initClockSignal (uint8_t selectedClockSignal, uint16_t clockSource, uint16_t clockSourceDivider)
  Initializes clock signal.
- void CS_turnOnXT1 (uint16_t xt1drive)
  Initializes the XT1 crystal oscillator in low frequency mode.
- void CS_bypassXT1 (void)
  Bypasses the XT1 crystal oscillator.
- bool CS_turnOnXT1WithTimeout (uint16_t xt1drive, uint32_t timeout)
  Initializes the XT1 crystal oscillator in low frequency mode with timeout.
- bool CS_bypassXT1WithTimeout (uint32_t timeout)
  Bypasses the XT1 crystal oscillator with timeout.
- void CS_turnOffXT1 (void)
  Stops the XT1 oscillator using the XT1OFF bit.
- void CS_turnOnXT2 (uint16_t xt2drive)
  Starts the XT2 crystal.
- void CS_bypassXT2 (void)
  Bypasses the XT2 crystal oscillator.
- bool CS_turnOnXT2WithTimeout (uint16_t xt2drive, uint32_t timeout)

Initializes the XT2 crystal oscillator with timeout.
- bool CS_bypassXT2WithTimeout (uint32_t timeout)
  Bypasses the XT2 crystal oscillator with timeout.
- void CS_turnOffXT2 (void)
  Stops the XT2 oscillator using the XT2OFF bit.
- void CS_enableClockRequest (uint8_t selectClock)
  Enables conditional module requests.
- void CS_disableClockRequest (uint8_t selectClock)
  Disables conditional module requests.
- uint8_t CS_getFaultFlagStatus (uint8_t mask)
  Gets the current CS fault flag status.
- void CS_clearFaultFlag (uint8_t mask)
  Clears the current CS fault flag status for the masked bit.
- uint32_t CS_getACLK (void)
  Get the current ACLK frequency.
- uint32_t CS_getSMCLK (void)
  Get the current SMCLK frequency.
- uint32_t CS_getMCLK (void)
  Get the current MCLK frequency.
- uint16_t CS_clearAllOscFlagsWithTimeout (uint32_t timeout)
  Clears all the Oscillator Flags.
- void CS_setDCOFreq (uint16_t dcorsel, uint16_t dcofsel)
  Set DCO frequency.

10.2.1 Detailed Description

The CS API is broken into four groups of functions: an API that initializes the clock module, those that deal with clock configuration and control, and external crystal and bypass specific configuration and initialization, and those that handle interrupts.

General CS configuration and initialization are handled by the following API

- CS_initClockSignal()
- CS_enableClockRequest()
- CS_disableClockRequest()
- CS_getACLK()
- CS_getSMCLK()
- CS_getMCLK()
- CS_setDCOFreq()

The following external crystal and bypass specific configuration and initialization functions are available for FR57xx devices:

- CS_turnOnXT1()
- CS_bypassXT1()
- CS_bypassXT1WithTimeout()
- CS_turnOnXT1WithTimeout()
- CS_turnOffXT1()
- CS_turnOnXT2()

- CS_bypassXT2()
- CS_turnOnXT2WithTimeout()
- CS_bypassXT2WithTimeout()
- CS_turnOffXT2()

The CS interrupts are handled by

- CS_enableClockRequest()
- CS_disableClockRequest()
- CS_getFaultFlagStatus()
- CS_clearFaultFlag()
- CS_clearAllOscFlagsWithTimeout()

CS_setExternalClockSource must be called if an external crystal XT1 or XT2 is used and the user intends to call CS_getMCLK, CS_getSMCLK or CS_getACLK APIs and turnOnXT1, XT1ByPass, turnOnXT1WithTimeout, XT1ByPassWithTimeout. If not any of the previous API are going to be called, it is not necessary to invoke this API.

10.2.2 Function Documentation

void CS_bypassXT1 (void)

Bypasses the XT1 crystal oscillator.

Loops until all oscillator fault flags are cleared, with no timeout. IMPORTANT: User must call CS_setExternalClockSource function to set frequency of external clocks before calling this function.

Modified bits of **CSCTL0** register, bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG** register.

Returns

None

bool CS_bypassXT1WithTimeout (uint32_t *timeout*)

Bypasses the XT1 crystal oscillator with timeout.

Loops until all oscillator fault flags are cleared or until a timeout counter is decremented and equals to zero.IMPORTANT: User must call CS_setExternalClockSource to set frequency of external clocks before calling this function

Parameters

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

Modified bits of **CSCTL0** register, bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

void CS_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. NOTE: User must call CS_setExternalClockSource to set frequency of external clocks before calling this function.

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

Returns

None

bool CS_bypassXT2WithTimeout (uint32_t *timeout*)

Bypasses the XT2 crystal oscillator with timeout.

Bypasses the XT2 crystal oscillator with timeout, 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. NOTE: User must call CS_setExternalClockSource to set frequency of external clocks before calling this function.

Parameters

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

Modified bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG1** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

uint16_t CS_clearAllOscFlagsWithTimeout (uint32_t *timeout*)

Clears all the Oscillator Flags.

Parameters

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

Modified bits of **CSCTL5** register and bits of **SFRIFG1** register.

Returns

the mask of the oscillator flag status

void CS_clearFaultFlag (uint8_t *mask*)

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

Parameters

mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **CS_XT2OFFG** - XT2 oscillator fault flag ■ **CS_XT1OFFG** - XT2 oscillator fault flag (HF mode)

Modified bits of **CSCTL5** register.

Returns

None

void CS_disableClockRequest (uint8_t *selectClock*)

Disables conditional module requests.

Parameters

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

Modified bits of **CSCTL6** register.

Returns

None

void CS_enableClockRequest (uint8_t *selectClock*)

Enables conditional module requests.

Parameters

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

Modified bits of **CSCTL6** register.

Returns

None

uint32_t CS_getACLK (void)

Get the current ACLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_externalClockSourceInit API was invoked before in case XT1 or XT2 is being used.

Returns

Current ACLK frequency in Hz

uint8_t CS_getFaultFlagStatus (uint8_t *mask*)

Gets the current CS fault flag status.

Parameters

mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **CS_XT2OFFG** - XT2 oscillator fault flag ■ **CS_XT1OFFG** - XT2 oscillator fault flag (HF mode)

Returns

Logical OR of any of the following:
- **CS_XT2OFFG** XT2 oscillator fault flag
- **CS_XT1OFFG** XT2 oscillator fault flag (HF mode)
  indicating the status of the masked interrupts

uint32_t CS_getMCLK (void)

Get the current MCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_externalClockSourceInit API was invoked before in case XT1 or XT2 is being used.

Returns

Current MCLK frequency in Hz

uint32_t CS_getSMCLK (void)

Get the current SMCLK frequency.

If a oscillator fault is set, the frequency returned will be based on the fail safe mechanism of CS module. The user of this API must ensure that CS_externalClockSourceInit API was invoked before in case XT1 or XT2 is being used.

Returns

Current SMCLK frequency in Hz

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

Initializes 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 to MSP430ware documentation for CS module or Device Family User's Guide for details of default clock signal states.

Parameters

selectedClock←Signal	is the selected clock signal Valid values are: ■ **CS_ACLK** ■ **CS_MCLK** ■ **CS_SMCLK** ■ **CS_MODOSC**
clockSource	is Clock source for the selectedClock Valid values are: ■ **CS_XT1CLK_SELECT** ■ **CS_VLOCLK_SELECT** ■ **CS_DCOCLK_SELECT** ■ **CS_XT2CLK_SELECT**
clockSource←Divider	selects the clock divider to calculate clock signal from clock source. Valid values are: ■ **CS_CLOCK_DIVIDER_1** - [Default for ACLK] ■ **CS_CLOCK_DIVIDER_2** ■ **CS_CLOCK_DIVIDER_4** ■ **CS_CLOCK_DIVIDER_8** - [Default for SMCLK and MCLK] ■ **CS_CLOCK_DIVIDER_16** ■ **CS_CLOCK_DIVIDER_32**

Modified bits of **CSCTL0** register, bits of **CSCTL3** register and bits of **CSCTL2** register.

Returns

None

void CS_setDCOFreq (uint16_t *dcorsel,* uint16_t *dcofsel*)

Set DCO frequency.

Parameters

dcorsel	selects frequency range option. Valid options are: CS_DCORSEL_0 [Default] CS_DCO↩ RSEL_1 Valid values are: ■ **CS_DCORSEL_0** ■ **CS_DCORSEL_1**
dcofsel	selects valid frequency options based on dco frequency range selection (dcorsel). Valid values are: ■ **CS_DCOFSEL_0** - Low frequency option 5.33MHZ. High frequency option 16MHz. ■ **CS_DCOFSEL_1** - Low frequency option 6.67MHZ. High frequency option 20MHz. ■ **CS_DCOFSEL_2** - Low frequency option 5.33MHZ. High frequency option 16MHz. ■ **CS_DCOFSEL_3** [Default] - Low frequency option 8MHZ. High frequency option 24↩ MHz.

Returns

None

void CS_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 CS_getMCLK, CS_getSMCLK, CS_getACLK and turnOnXT1, XT1ByPass, turnOnXT1WithTimeout, XT1ByPassWithTimeout.

Parameters

XT1CLK_↩ frequency	is the XT1 crystal frequencies in Hz
XT2CLK_↩ frequency	is the XT2 crystal frequencies in Hz

Returns

None

void CS_turnOffXT1 (void)

Stops the XT1 oscillator using the XT1OFF bit.

Modified bits of **CSCTL4** register.

Returns

None

void CS_turnOffXT2 (void)

Stops the XT2 oscillator using the XT2OFF bit.

Modified bits of **CSCTL4** register.

Returns

None

void CS_turnOnXT1 (uint16_t *xt1drive*)

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. IMPORTANT: User must call CS_setExternalClockSource function to set frequency of external clocks before calling this function.

Parameters

xt1drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default]

Modified bits of **CSCTL0** register, bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG1** register.

Returns

None

bool CS_turnOnXT1WithTimeout (uint16_t *xt1drive,* uint32_t *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. IMPORTANT: User must call CS_setExternalClockSource function to set frequency of external clocks before calling this function.

Parameters

xt1drive	is the target drive strength for the XT1 crystal oscillator. Valid values are: ■ **CS_XT1_DRIVE_0** ■ **CS_XT1_DRIVE_1** ■ **CS_XT1_DRIVE_2** ■ **CS_XT1_DRIVE_3** [Default]
timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Modified bits of **CSCTL0** register, bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG1** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

void CS_turnOnXT2 (uint16_t *xt2drive*)

Starts the XT2 crystal.

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. NOTE: User must call CS_setExternalClockSource to set frequency of external clocks before calling this function.

Parameters

xt2drive	is the target drive strength for the XT2 crystal oscillator. Valid values are: ■ **CS_XT2_DRIVE_4MHZ_8MHZ** ■ **CS_XT2_DRIVE_8MHZ_16MHZ** ■ **CS_XT2_DRIVE_16MHZ_24MHZ** ■ **CS_XT2_DRIVE_24MHZ_32MHZ** [Default]

Modified bits of **CSCTL0** register, bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG1** register.

Returns

None

bool CS_turnOnXT2WithTimeout (uint16_t *xt2drive,* uint32_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. NOTE: User must call CS_setExternalClockSource to set frequency of external clocks before calling this function.

Parameters

xt2drive	is the target drive strength for the XT2 crystal oscillator. Valid values are: ■ **CS_XT2_DRIVE_4MHZ_8MHZ** ■ **CS_XT2_DRIVE_8MHZ_16MHZ** ■ **CS_XT2_DRIVE_16MHZ_24MHZ** ■ **CS_XT2_DRIVE_24MHZ_32MHZ** [Default]
timeout	is the count value that gets decremented every time the loop that clears oscillator fault flags gets executed.

Modified bits of **CSCTL5** register, bits of **CSCTL4** register and bits of **SFRIFG1** register.

Returns

STATUS_SUCCESS or STATUS_FAIL

10.3 Programming Example

The following example shows the configuration of the CS module that sets
ACLK=SMCLK=MCLK=DCOCLK

```
//Set DCO Frequency to 8MHz
CS_setDCOFreq(CS_BASE,CS_DCORSEL_0,CS_DCOFSEL_3);

//configure MCLK, SMCLK and ACLK to be source by DCOCLK
CS_initClockSignal(CS_BASE,CS_ACLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
CS_initClockSignal(CS_BASE,CS_SMCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
CS_initClockSignal(CS_BASE,CS_MCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1);
```

11 Direct Memory Access (DMA)

11.1 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.

11.2 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.

11.2.1 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()

11.2.2 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

channelSelect	is the specified channel to clear the interrupt flag for. Valid values are: ■ **DMA_CHANNEL_0** ■ **DMA_CHANNEL_1** ■ **DMA_CHANNEL_2** ■ **DMA_CHANNEL_3** ■ **DMA_CHANNEL_4** ■ **DMA_CHANNEL_5** ■ **DMA_CHANNEL_6** ■ **DMA_CHANNEL_7**

Returns

None

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

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

Returns

None

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

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

Returns

None

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

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

Returns

None

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

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

Returns

None

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

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

Returns

None

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

channelSelect	is the specified channel to return the interrupt flag status from. Valid values are: ■ **DMA_CHANNEL_0** ■ **DMA_CHANNEL_1** ■ **DMA_CHANNEL_2** ■ **DMA_CHANNEL_3** ■ **DMA_CHANNEL_4** ■ **DMA_CHANNEL_5** ■ **DMA_CHANNEL_6** ■ **DMA_CHANNEL_7**

Returns

One of the following:
- **DMA_INT_INACTIVE**
- **DMA_INT_ACTIVE**
  indicating the status of the current interrupt flag

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

channelSelect	is the specified channel to return the status of the NMI Abort flag for. Valid values are: ■ **DMA_CHANNEL_0** ■ **DMA_CHANNEL_1** ■ **DMA_CHANNEL_2** ■ **DMA_CHANNEL_3** ■ **DMA_CHANNEL_4** ■ **DMA_CHANNEL_5** ■ **DMA_CHANNEL_6** ■ **DMA_CHANNEL_7**

Returns

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

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

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

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

param	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

channelSelect	is the specified channel to set the destination address direction for. Valid values are: ■ **DMA_CHANNEL_0** ■ **DMA_CHANNEL_1** ■ **DMA_CHANNEL_2** ■ **DMA_CHANNEL_3** ■ **DMA_CHANNEL_4** ■ **DMA_CHANNEL_5** ■ **DMA_CHANNEL_6** ■ **DMA_CHANNEL_7**
dstAddress	is the address of where the data will be transferred to. Modified bits are **DMAxDA** of **DMAxDA** register.
directionSelect	is the specified direction of the destination address after a transfer. Valid values are: ■ **DMA_DIRECTION_UNCHANGED** ■ **DMA_DIRECTION_DECREMENT** ■ **DMA_DIRECTION_INCREMENT** Modified bits are **DMADSTINCR** of **DMAxCTL** register.

Returns

None

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

channelSelect	is the specified channel to set source address direction for. Valid values are: ■ **DMA_CHANNEL_0** ■ **DMA_CHANNEL_1** ■ **DMA_CHANNEL_2** ■ **DMA_CHANNEL_3** ■ **DMA_CHANNEL_4** ■ **DMA_CHANNEL_5** ■ **DMA_CHANNEL_6** ■ **DMA_CHANNEL_7**
srcAddress	is the address of where the data will be transferred from. Modified bits are **DMAxSA** of **DMAxSA** register.
directionSelect	is the specified direction of the source address after a transfer. Valid values are: ■ **DMA_DIRECTION_UNCHANGED** ■ **DMA_DIRECTION_DECREMENT** ■ **DMA_DIRECTION_INCREMENT** Modified bits are **DMASRCINCR** of **DMAxCTL** register.

Returns

None

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

channelSelect	is the specified channel to set source address direction for. Valid values are: ▪ **DMA_CHANNEL_0** ▪ **DMA_CHANNEL_1** ▪ **DMA_CHANNEL_2** ▪ **DMA_CHANNEL_3** ▪ **DMA_CHANNEL_4** ▪ **DMA_CHANNEL_5** ▪ **DMA_CHANNEL_6** ▪ **DMA_CHANNEL_7**
transferSize	is the amount of transfers to complete in a block transfer mode, as well as how many transfers to complete before the interrupt flag is set. Valid value is between 1-65535, if 0, no transfers will occur. Modified bits are **DMAxSZ** of **DMAxSZ** register.

Returns

None

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

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

Returns

None

11.3 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_TRIGGERSOURCE_0;
param.transferUnitSelect = DMA_SIZE_SRCWORD_DSTWORD;
param.triggerTypeSelect = DMA_TRIGGER_RISINGEDGE;
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);
}
```

12 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)

12.1 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.

12.2 API Functions

Functions

- bool EUSCI_A_UART_init (uint16_t baseAddress, EUSCI_A_UART_initParam *param)
  Advanced initialization routine for the UART block. The values to be written into the clockPrescalar, firstModReg, secondModReg and overSampling parameters should be pre-computed and passed into the initialization function.
- void EUSCI_A_UART_transmitData (uint16_t baseAddress, uint8_t transmitData)
  Transmits a byte from the UART Module.
- 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.

12.2.1 Detailed Description

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

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

Configuration and control of the EUSI_UART are handled by the

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

Sending and receiving data via the EUSI_UART is handled by the

- EUSCI_A_UART_transmitData()
- EUSCI_A_UART_receiveData()

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

Managing the EUSI_UART interrupts and status are handled by the

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

12.2.2 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

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_STARTBIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT_FLAG**

Modified bits of **UCAxIFG** register.

Returns

None

void EUSCI_A_UART_disable (uint16_t *baseAddress*)

Disables the UART block.

This will disable operation of the UART block.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

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

Returns

None

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

Disables individual UART interrupt sources.

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_RECEIVE_INTERRUPT** - Receive interrupt ■ **EUSCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt ■ **EUSCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable ■ **EUSCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable ■ **EUSCI_A_UART_STARTBIT_INTERRUPT** - Start bit received interrupt enable ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT** - Transmit complete interrupt enable

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

Returns

None

void EUSCI_A_UART_enable (uint16_t *baseAddress*)

Enables the UART block.

This will enable operation of the UART block.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

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

Returns

None

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

Enables individual UART interrupt sources.

Enables the indicated UART interrupt sources. The interrupt flag is first and then the corresponding interrupt is enabled. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_RECEIVE_INTERRUPT** - Receive interrupt ■ **EUSCI_A_UART_TRANSMIT_INTERRUPT** - Transmit interrupt ■ **EUSCI_A_UART_RECEIVE_ERRONEOUSCHAR_INTERRUPT** - Receive erroneous-character interrupt enable ■ **EUSCI_A_UART_BREAKCHAR_INTERRUPT** - Receive break character interrupt enable ■ **EUSCI_A_UART_STARTBIT_INTERRUPT** - Start bit received interrupt enable ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT** - Transmit complete interrupt enable

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

Returns

None

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

Gets the current UART interrupt status.

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_STARTBIT_INTERRUPT_FLAG** ■ **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT_FLAG**

Modified bits of **UCAxIFG** register.

Returns

Logical OR of any of the following:
- **EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG**
- **EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG**
- **EUSCI_A_UART_STARTBIT_INTERRUPT_FLAG**
- **EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT_FLAG**
  indicating the status of the masked flags

uint32_t EUSCI_A_UART_getReceiveBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Returns

Address of RX Buffer

uint32_t EUSCI_A_UART_getTransmitBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Returns

Address of TX Buffer

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/MSP430Baud↩`
`RateConverter/index.html`

Parameters

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

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

Returns

STATUS_SUCCESS or STATUS_FAIL of the initialization process

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

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

Gets the current UART status flags.

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

Parameters

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

Modified bits of **UCAxSTAT** register.

Returns

Logical OR of any of the following:

- **EUSCI_A_UART_LISTEN_ENABLE**
- **EUSCI_A_UART_FRAMING_ERROR**
- **EUSCI_A_UART_OVERRUN_ERROR**
- **EUSCI_A_UART_PARITY_ERROR**
- **EUSCI_A_UART_BREAK_DETECT**
- **EUSCI_A_UART_RECEIVE_ERROR**
- **EUSCI_A_UART_ADDRESS_RECEIVED**

- **EUSCI_A_UART_IDLELINE**
- **EUSCI_A_UART_BUSY**
  indicating the status of the masked interrupt flags

uint8_t EUSCI_A_UART_receiveData (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits of **UCAxRXBUF** register.

Returns

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

void EUSCI_A_UART_resetDormant (uint16_t *baseAddress*)

Re-enables UART module from dormant mode.

Not dormant. All received characters set UCRXIFG.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

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

Returns

None

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

Sets the deglitch time.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.
deglitchTime	is the selected deglitch time Valid values are: ■ **EUSCI_A_UART_DEGLITCH_TIME_2ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_50ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_100ns** ■ **EUSCI_A_UART_DEGLITCH_TIME_200ns**

Returns

None

void EUSCI_A_UART_setDormant (uint16_t *baseAddress*)

Sets the UART module in dormant mode.

Puts USCI in sleep mode Only characters that are preceded by an idle-line or with address bit set UCRXIFG. In UART mode with automatic baud-rate detection, only the combination of a break and sync field sets UCRXIFG.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

Modified bits of **UCAxCTL1** register.

Returns

None

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

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

Parameters

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

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

Returns

None

void EUSCI_A_UART_transmitBreak (uint16_t *baseAddress*)

Transmit break.

Transmits a break with the next write to the transmit buffer. In UART mode with automatic baud-rate detection, EUSCI_A_UART_AUTOMATICBAUDRATE_SYNC(0x55) must be written into UCAxTXBUF to generate the required break/sync fields. Otherwise, DEFAULT_SYNC(0x00) must be written into the transmit buffer. Also ensures module is ready for transmitting the next data.

Parameters

baseAddress	is the base address of the EUSCI_A_UART module.

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

Returns

None

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

Transmits a byte from the UART Module.

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

Parameters

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

Modified bits of **UCAxTXBUF** register.

Returns

None

12.3 Programming Example

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

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

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

  EUSCI_A_UART_enable(EUSCI_A0_BASE);

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

13 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)

13.1 Introduction

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame.

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

The SPI module can be configured as either a master or a slave device.

The SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the module's input clock.

13.2 Functions

Functions

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

13.2.1 Detailed Description

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

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

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

The status and initialization of the SPI module are managed by

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

Data handling is done by

- EUSCI_A_SPI_transmitData()
- EUSCI_A_SPI_receiveData()

Interrupts from the SPI module are managed using

- EUSCI_A_SPI_disableInterrupt()

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

DMA related

- EUSCI_A_SPI_getReceiveBufferAddressForDMA()
- EUSCI_A_SPI_getTransmitBufferAddressForDMA()

13.2.2 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

baseAddress	is the base address of the EUSCI_A_SPI module.
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_A_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

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

Returns

None

void EUSCI_A_SPI_changeMasterClock (uint16_t *baseAddress,* **EUSCI_A_SPI_changeMasterClockParam** ∗ *param*)

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

Parameters

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

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

Returns

None

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

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

Clears the selected SPI interrupt status flag.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the masked interrupt flag to be cleared. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Modified bits of **UCAxIFG** register.

Returns

None

void EUSCI_A_SPI_disable (uint16_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

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

Returns

None

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

Disables individual SPI interrupt sources.

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

Parameters

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

Modified bits of **UCAxIE** register.

Returns

None

void EUSCI_A_SPI_enable (uint16_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

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

Returns

None

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

Enables individual SPI interrupt sources.

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

Parameters

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

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

Returns

None

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

Gets the current SPI interrupt status.

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

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_A_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_A_SPI_RECEIVE_INTERRUPT**

Returns

Logical OR of any of the following:
- **EUSCI_A_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_A_SPI_RECEIVE_INTERRUPT**
indicating the status of the masked interrupts

uint32_t EUSCI_A_SPI_getReceiveBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

the address of the RX Buffer

uint32_t EUSCI_A_SPI_getTransmitBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

the address of the TX Buffer

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

Initializes the SPI Master block.

Upon successful initialization of the SPI master block, this function will have set the bus speed for the master, but the SPI Master block still remains disabled and must be enabled with EUSCI_A_SPI_enable()

Parameters

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

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

Returns

STATUS_SUCCESS

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

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

Initializes the SPI Slave block.

Upon successful initialization of the SPI slave block, this function will have initialized the slave block, but the SPI Slave block still remains disabled and must be enabled with EUSCI_A_SPI_enable()

Parameters

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

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

Returns

STATUS_SUCCESS

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

uint16_t EUSCI_A_SPI_isBusy (uint16_t *baseAddress*)

Indicates whether or not the SPI bus is busy.

This function returns an indication of whether or not the SPI bus is busy.This function checks the status of the bus via UCBBUSY bit

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

One of the following:
- **EUSCI_A_SPI_BUSY**
- **EUSCI_A_SPI_NOT_BUSY**
  indicating if the EUSCI_A_SPI is busy

uint8_t EUSCI_A_SPI_receiveData (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.

Returns

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

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

Selects 4Pin Functionality.

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

Parameters

baseAddress	is the base address of the EUSCI_A_SPI module.
select4Pin↩ Functionality	selects 4 pin functionality Valid values are: ■ **EUSCI_A_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_A_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

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

Returns

None

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

Transmits a byte from the SPI Module.

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

Parameters

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

Returns

None

13.3 Programming Example

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

```
//Initialize slave to MSB first, inactive high clock polarity and 3 wire SPI
EUSCI_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
        );
```

14 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)

14.1 Introduction

The Serial Peripheral Interface Bus or SPI bus is a synchronous serial data link standard named by Motorola that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame.

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

The SPI module can be configured as either a master or a slave device.

The SPI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the module's input clock.

14.2 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.

14.2.1 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()

14.2.2 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

baseAddress	is the base address of the EUSCI_B_SPI module.
clockPhase	is clock phase select. Valid values are: ■ **EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT** [Default] ■ **EUSCI_B_SPI_PHASE_DATA_CAPTURED_ONFIRST_CHANGED_ON_NEXT**
clockPolarity	is clock polarity select Valid values are: ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH** ■ **EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_LOW** [Default]

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

Returns

None

void EUSCI_B_SPI_changeMasterClock (uint16_t *baseAddress,* **EUSCI_B_SPI_changeMasterClockParam** ∗ *param*)

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

Parameters

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

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

Returns

None

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

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

Clears the selected SPI interrupt status flag.

Parameters

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

Modified bits of **UCAxIFG** register.

Returns

None

void EUSCI_B_SPI_disable (uint16_t *baseAddress*)

Disables the SPI block.

This will disable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

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

Returns

None

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

Disables individual SPI interrupt sources.

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

Parameters

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

Modified bits of **UCAxIE** register.

Returns

None

void EUSCI_B_SPI_enable (uint16_t *baseAddress*)

Enables the SPI block.

This will enable operation of the SPI block.

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

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

Returns

None

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

Enables individual SPI interrupt sources.

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

Parameters

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

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

Returns

None

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

Gets the current SPI interrupt status.

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_B_SPI_TRANSMIT_INTERRUPT** ■ **EUSCI_B_SPI_RECEIVE_INTERRUPT**

Returns

Logical OR of any of the following:
- **EUSCI_B_SPI_TRANSMIT_INTERRUPT**
- **EUSCI_B_SPI_RECEIVE_INTERRUPT**
  indicating the status of the masked interrupts

uint32_t EUSCI_B_SPI_getReceiveBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns

the address of the RX Buffer

uint32_t EUSCI_B_SPI_getTransmitBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns

the address of the TX Buffer

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

Initializes the SPI Master block.

Upon successful initialization of the SPI master block, this function will have set the bus speed for the master, but the SPI Master block still remains disabled and must be enabled with EUSCI_B_SPI_enable()

Parameters

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

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

Returns

STATUS_SUCCESS

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

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

Initializes the SPI Slave block.

Upon successful initialization of the SPI slave block, this function will have initialized the slave block, but the SPI Slave block still remains disabled and must be enabled with EUSCI_B_SPI_enable()

Parameters

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

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

Returns

STATUS_SUCCESS

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

uint16_t EUSCI_B_SPI_isBusy (uint16_t *baseAddress*)

Indicates whether or not the SPI bus is busy.

This function returns an indication of whether or not the SPI bus is busy.This function checks the status of the bus via UCBBUSY bit

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns

One of the following:
- **EUSCI_B_SPI_BUSY**
- **EUSCI_B_SPI_NOT_BUSY**
  indicating if the EUSCI_B_SPI is busy

uint8_t EUSCI_B_SPI_receiveData (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.

Returns

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

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

Selects 4Pin Functionality.

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
select4Pin←Functionality	selects 4 pin functionality Valid values are: ■ **EUSCI_B_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_B_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

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

Returns

None

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

Transmits a byte from the SPI Module.

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

Parameters

baseAddress	is the base address of the EUSCI_B_SPI module.
transmitData	data to be transmitted from the SPI module

Returns

None

14.3 Programming Example

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

```
//Initialize slave to MSB first, inactive high clock polarity and 3 wire SPI
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
        );
```

15 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)

15.1 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.

15.2 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.

15.3 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.

15.4 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.

15.4.1 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

15.4.2 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

baseAddress	is the base address of the I2C module.
mask	is a bit mask of the interrupt sources to be cleared. Mask value is the logical OR of any of the following: ■ **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt ■ **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt ■ **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt ■ **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3 ■ **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt ■ **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable ■ **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Modified bits of **UCBxIFG** register.

Returns

None

void EUSCI_B_I2C_disable (uint16_t *baseAddress*)

Disables the I2C block.

This will disable operation of the I2C block.

Parameters

baseAddress	is the base address of the USCI I2C module.

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

Returns

None

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

Disables individual I2C interrupt sources.

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

Parameters

baseAddress	is the base address of the I2C module.
mask	is the bit mask of the interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt ■ **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt ■ **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt ■ **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3 ■ **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt ■ **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable ■ **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Modified bits of **UCBxIE** register.

Returns

None

void EUSCI_B_I2C_disableMultiMasterMode (uint16_t *baseAddress*)

Disables Multi Master Mode.

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

Parameters

baseAddress	is the base address of the I2C module.

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

Returns

None

void EUSCI_B_I2C_enable (uint16_t *baseAddress*)

Enables the I2C block.

This will enable operation of the I2C block.

Parameters

baseAddress	is the base address of the USCI I2C module.

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

Returns

None

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

Enables individual I2C interrupt sources.

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

Parameters

baseAddress	is the base address of the I2C module.
mask	is the bit mask of the interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt ■ **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt ■ **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt ■ **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3 ■ **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt ■ **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable ■ **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Modified bits of **UCBxIE** register.

Returns

None

void EUSCI_B_I2C_enableMultiMasterMode (uint16_t *baseAddress*)

Enables Multi Master Mode.

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

Parameters

baseAddress	is the base address of the I2C module.

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

Returns

None

uint16_t EUSCI_B_I2C_getInterruptStatus (uint16_t *baseAddress,* uint16_t *mask*)

Gets the current I2C interrupt status.

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

Parameters

baseAddress	is the base address of the I2C module.
mask	is the masked interrupt flag status to be returned. Mask value is the logical OR of any of the following: ■ **EUSCI_B_I2C_NAK_INTERRUPT** - Not-acknowledge interrupt ■ **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** - Arbitration lost interrupt ■ **EUSCI_B_I2C_STOP_INTERRUPT** - STOP condition interrupt ■ **EUSCI_B_I2C_START_INTERRUPT** - START condition interrupt ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** - Transmit interrupt0 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** - Transmit interrupt1 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** - Transmit interrupt2 ■ **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** - Transmit interrupt3 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT0** - Receive interrupt0 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT1** - Receive interrupt1 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT2** - Receive interrupt2 ■ **EUSCI_B_I2C_RECEIVE_INTERRUPT3** - Receive interrupt3 ■ **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** - Bit position 9 interrupt ■ **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** - Clock low timeout interrupt enable ■ **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** - Byte counter interrupt enable

Returns

Logical OR of any of the following:

- **EUSCI_B_I2C_NAK_INTERRUPT** Not-acknowledge interrupt
- **EUSCI_B_I2C_ARBITRATIONLOST_INTERRUPT** Arbitration lost interrupt
- **EUSCI_B_I2C_STOP_INTERRUPT** STOP condition interrupt
- **EUSCI_B_I2C_START_INTERRUPT** START condition interrupt
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT0** Transmit interrupt0
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT1** Transmit interrupt1

- **EUSCI_B_I2C_TRANSMIT_INTERRUPT2** Transmit interrupt2
- **EUSCI_B_I2C_TRANSMIT_INTERRUPT3** Transmit interrupt3
- **EUSCI_B_I2C_RECEIVE_INTERRUPT0** Receive interrupt0
- **EUSCI_B_I2C_RECEIVE_INTERRUPT1** Receive interrupt1
- **EUSCI_B_I2C_RECEIVE_INTERRUPT2** Receive interrupt2
- **EUSCI_B_I2C_RECEIVE_INTERRUPT3** Receive interrupt3
- **EUSCI_B_I2C_BIT9_POSITION_INTERRUPT** Bit position 9 interrupt
- **EUSCI_B_I2C_CLOCK_LOW_TIMEOUT_INTERRUPT** Clock low timeout interrupt enable
- **EUSCI_B_I2C_BYTE_COUNTER_INTERRUPT** Byte counter interrupt enable
  indicating the status of the masked interrupts

uint8_t EUSCI_B_I2C_getMode (uint16_t *baseAddress*)

Gets the mode of the I2C device.

Current I2C transmit/receive mode.

Parameters

baseAddress	is the base address of the I2C module.

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

Returns

One of the following:
- **EUSCI_B_I2C_TRANSMIT_MODE**
- **EUSCI_B_I2C_RECEIVE_MODE**
  indicating the current mode

uint32_t EUSCI_B_I2C_getReceiveBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C module.

Returns

The address of the I2C RX Buffer

uint32_t EUSCI_B_I2C_getTransmitBufferAddress (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C module.

Returns

The address of the I2C TX Buffer

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

Initializes the I2C Master block.

This function initializes operation of the I2C Master block. Upon successful initialization of the I2C block, this function will have set the bus speed for the master; however I2C module is still disabled till EUSCI_B_I2C_enable is invoked.

Parameters

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

Returns

None

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

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

Initializes the I2C Slave block.

This function initializes operation of the I2C as a Slave mode. Upon successful initialization of the I2C blocks, this function will have set the slave address but the I2C module is still disabled till EUSCI_B_I2C_enable is invoked.

Parameters

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

Returns

None

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

uint16_t EUSCI_B_I2C_isBusBusy (uint16_t *baseAddress*)

Indicates whether or not the I2C bus is busy.

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

Parameters

baseAddress	is the base address of the I2C module.

Returns

One of the following:
- **EUSCI_B_I2C_BUS_BUSY**
- **EUSCI_B_I2C_BUS_NOT_BUSY**
  indicating whether the bus is busy

uint16_t EUSCI_B_I2C_masterIsStartSent (uint16_t *baseAddress*)

Indicates whether Start got sent.

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

One of the following:
- **EUSCI_B_I2C_START_SEND_COMPLETE**
- **EUSCI_B_I2C_SENDING_START**
  indicating whether the start was sent

uint16_t EUSCI_B_I2C_masterIsStopSent (uint16_t *baseAddress*)

Indicates whether STOP got sent.

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

One of the following:
- **EUSCI_B_I2C_STOP_SEND_COMPLETE**
- **EUSCI_B_I2C_SENDING_STOP**
  indicating whether the stop was sent

uint8_t EUSCI_B_I2C_masterReceiveMultiByteFinish (uint16_t *baseAddress*)

Finishes multi-byte reception at the Master end.

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

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

Received byte at Master end.

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

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

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

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is a pointer to the location to store the received byte at master end
timeout	is the amount of time to wait until giving up

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the reception process

uint8_t EUSCI_B_I2C_masterReceiveMultiByteNext (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

Received byte at Master end.

void EUSCI_B_I2C_masterReceiveMultiByteStop (uint16_t *baseAddress*)

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

This function is used by the Master module to initiate STOP

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

None

uint8_t EUSCI_B_I2C_masterReceiveSingle (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C Master module.

Returns

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

uint8_t EUSCI_B_I2C_masterReceiveSingleByte (uint16_t *baseAddress*)

Does single byte reception from Slave.

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

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

void EUSCI_B_I2C_masterReceiveStart (uint16_t *baseAddress*)

Starts reception at the Master end.

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

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

None

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

Finishes multi-byte transmission from Master to Slave.

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

Parameters

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

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

Returns

None

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

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

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

Parameters

baseAddress	is the base address of the I2C Master module.
txData	is the last data byte to be transmitted in a multi-byte transmission
timeout	is the amount of time to wait until giving up

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

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

Continues multi-byte transmission from Master to Slave.

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

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

None

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

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

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

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

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

Starts multi-byte transmission from Master to Slave.

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

Parameters

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

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

Returns

None

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

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

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

Parameters

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

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

void EUSCI_B_I2C_masterSendMultiByteStop (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

None

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

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

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

Parameters

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

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

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

Does single byte transmission from Master to Slave.

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

Parameters

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

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

Returns

None

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

Does single byte transmission from Master to Slave with timeout.

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

Parameters

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

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

Returns

STATUS_SUCCESS or STATUS_FAILURE of the transmission process.

void EUSCI_B_I2C_masterSendStart (uint16_t *baseAddress*)

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

This function is used by the Master module to initiate START

Parameters

baseAddress	is the base address of the I2C Master module.

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

Returns

None

void EUSCI_B_I2C_setMode (uint16_t *baseAddress,* uint8_t *mode*)

Sets the mode of the I2C device.

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

Parameters

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

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

Returns

None

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

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

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

Parameters

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

Modified bits of **UCBxI2CSA** register.

Returns

None

uint8_t EUSCI_B_I2C_slaveGetData (uint16_t *baseAddress*)

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

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

Parameters

baseAddress	is the base address of the I2C Slave module.

Returns

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

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

Transmits a byte from the I2C Module.

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

Parameters

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

Modified bits of **UCBxTXBUF** register.

Returns

None

15.5 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);
```

16 FRAMCtl - FRAM Controller

16.1 Introduction

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

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

16.2 API Functions

FRAMCtl_enableInterrupt enables selected FRAM interrupt sources.

FRAMCtl_getInterruptStatus returns the status of the selected FRAM interrupt flags.

FRAMCtl_disableInterrupt disables selected FRAM interrupt sources.

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

FRAMCtl_write8 facilitates writing into the FRAM memory in byte format. FRAMCtl_write16 facilitates writing into the FRAM memory in word format. FRAMCtl_write32 facilitates writing into the FRAM memory in long format, pass by reference. FRAMCtl_memoryFill32 facilitates writing into the FRAM memory in long format, pass by value.

The FRAM API is broken into 3 groups of functions: those that write into FRAM, those that handle interrupts, and those that configure the wait state.

FRAM writes are managed by

- FRAMCtl_write8()
- FRAMCtl_write16()
- FRAMCtl_write32()
- FRAMCtl_memoryFill32()

The FRAM interrupts are handled by

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

The FRAM wait state is handled by

- FRAMCtl_configureWaitStateControl()

16.3 Programming Example

The following example shows some FRAM operations using the APIs

```
//Writes the value of "data", 128 times to FRAM
FRAMCtl_memoryFill32(FRAM_BASE,data,
                     (unsigned long *)FRAMCTL_TEST_START,128);
```

17 GPIO

17.1 Introduction

The Digital I/O (GPIO) API provides a set of functions for using the MSP430Ware GPIO modules. Functions are provided to setup and enable use of input/output pins, setting them up with or without interrupts and those that access the pin value.

The digital I/O features include:

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

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

Ports P1 and P2 always have interrupt capability. Each interrupt for the P1 and P2 I/O lines can be individually enabled and configured to provide an interrupt on a rising or falling edge of an input signal. All P1 I/O lines source a single interrupt vector P1IV, and all P2 I/O lines source a different, single interrupt vector P2IV. On some devices, additional ports with interrupt capability may be available (see the device-specific data sheet for details) and contain their own respective interrupt vectors. Individual ports can be accessed as byte-wide ports or can be combined into word-wide ports and accessed via word formats. Port pairs P1/P2, P3/P4, P5/P6, P7/P8, etc., are associated with the names PA, PB, PC, PD, etc., respectively. All port registers are handled in this manner with this naming convention except for the interrupt vector registers, P1IV and P2IV; that is, PAIV does not exist. When writing to port PA with word operations, all 16 bits are written to the port. When writing to the lower byte of the PA port using byte operations, the upper byte remains unchanged. Similarly, writing to the upper byte of the PA port using byte instructions leaves the lower byte unchanged. When writing to a port that contains less than the maximum number of bits possible, the unused bits are a ”don’t care”. Ports PB, PC, PD, PE, and PF behave similarly.

Reading of the PA port using word operations causes all 16 bits to be transferred to the destination. Reading the lower or upper byte of the PA port (P1 or P2) and storing to memory using byte operations causes only the lower or upper byte to be transferred to the destination, respectively. Reading of the PA port and storing to a general-purpose register using byte operations causes the byte transferred to be written to the least significant byte of the register. The upper significant byte of the destination register is cleared automatically. Ports PB, PC, PD, PE, and PF behave similarly. When reading from ports that contain less than the maximum bits possible, unused bits are read as zeros (similarly for port PJ).

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

17.2 API Functions

Functions

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

17.2.1 Detailed Description

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

The GPIO pins are configured with

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

The GPIO interrupts are handled with

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

The GPIO pin state is accessed with

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

17.2.2 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

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

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

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

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

Modified bits of **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

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

Modified bits of **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

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

selectedPins	is the specified pin in the selected port. Valid values are: ■ **GPIO_PIN0** ■ **GPIO_PIN1** ■ **GPIO_PIN2** ■ **GPIO_PIN3** ■ **GPIO_PIN4** ■ **GPIO_PIN5** ■ **GPIO_PIN6** ■ **GPIO_PIN7** ■ **GPIO_PIN8** ■ **GPIO_PIN9** ■ **GPIO_PIN10** ■ **GPIO_PIN11** ■ **GPIO_PIN12** ■ **GPIO_PIN13** ■ **GPIO_PIN14** ■ **GPIO_PIN15**

Returns

One of the following:

- **GPIO_INPUT_PIN_HIGH**
- **GPIO_INPUT_PIN_LOW**
  indicating the status of the pin

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

This function gets the interrupt status of the selected pin.

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

Parameters

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

Returns

Logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**
- **GPIO_PIN6**
- **GPIO_PIN7**
- **GPIO_PIN8**
- **GPIO_PIN9**
- **GPIO_PIN10**
- **GPIO_PIN11**
- **GPIO_PIN12**
- **GPIO_PIN13**
- **GPIO_PIN14**
- **GPIO_PIN15**
  indicating the interrupt status of the selected pins [Default: 0]

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

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

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

Parameters

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

edgeSelect	specifies what transition sets the interrupt flag Valid values are: ■ **GPIO_HIGH_TO_LOW_TRANSITION** ■ **GPIO_LOW_TO_HIGH_TRANSITION**

Modified bits of **PxIES** register.

Returns

None

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

This function configures the selected Pin as input pin.

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

Parameters

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

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

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

Returns

None

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

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

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

Parameters

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

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

Returns

None

void GPIO_setAsInputPinWithPullUpResistor (uint8_t *selectedPort,* uint16_t *selectedPins*)

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

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

Parameters

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

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

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

Returns

None

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

This function configures the selected Pin as output pin.

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

Parameters

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

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

Returns

None

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

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

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

Parameters

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

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

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

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

Returns

None

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

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

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

Parameters

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

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

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

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

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

Modified bits of **PxOUT** register.

Returns

None

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

This function sets output LOW on the selected Pin.

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

Parameters

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

Modified bits of **PxOUT** register.

Returns

None

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

This function toggles the output on the selected Pin.

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

Parameters

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

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

Modified bits of **PxOUT** register.

Returns

None

17.3 Programming Example

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

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


//Enable P1.4 internal resistance as pull-Up resistance
GPIO_setAsInputPinWithPullUpresistor(

    GPIO_PORT_P1,
    GPIO_PIN4
    );

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


    //P1.4 IFG cleared
    GPIO_clearInterrupt(

        GPIO_PORT_P1,
        GPIO_PIN4
        );

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

    //For debugger
    __no_operation();
}

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


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

18 Memory Protection Unit (MPU)

18.1 Introduction

The MPU protects against accidental writes to designated read-only memory segments or execution of code from a constant memory segment memory. Clearing the MPUENA bit disables the MPU, making the complete memory accessible for read, write, and execute operations. After a BOR, the complete memory is accessible without restrictions for read, write, and execute operations.

MPU features include:

- Main memory can be configured up to three segments of variable size
- Access rights for each segment can be set independently
- Information memory can have its access rights set independently
- All MPU registers are protected from access by password

18.2 API Functions

Macros

- #define **MPU_MAX_SEG_VALUE** 20

Functions

- void MPU_initTwoSegments (uint16_t baseAddress, uint16_t seg1boundary, uint8_t seg1accmask, uint8_t seg2accmask)
  Initializes MPU with two memory segments.
- void MPU_initThreeSegments (uint16_t baseAddress, MPU_initThreeSegmentsParam ∗param)
  Initializes MPU with three memory segments.
- void MPU_initInfoSegment (uint16_t baseAddress, uint8_t accmask)
  Initializes user information memory segment.
- void MPU_start (uint16_t baseAddress)
  The following function enables the MPU module in the device.
- void MPU_enablePUCOnViolation (uint16_t baseAddress, uint16_t segment)
  The following function enables PUC generation when an access violation has occurred on the memory segment selected by the user.
- void MPU_disablePUCOnViolation (uint16_t baseAddress, uint16_t segment)
  The following function disables PUC generation when an access violation has occurred on the memory segment selected by the user.
- uint16_t MPU_getInterruptStatus (uint16_t baseAddress, uint16_t memAccFlag)

Returns the memory segment violation flag status requested by the user.

- uint16_t MPU_clearInterrupt (uint16_t baseAddress, uint16_t memAccFlag)
  Clears the masked interrupt flags.
- uint16_t MPU_clearAllInterrupts (uint16_t baseAddress)
  Clears all Memory Segment Access Violation Interrupt Flags.

18.2.1 Detailed Description

The MPU API is broken into three groups of functions: those that handle initialization, those that deal with memory segmentation and access rights definition, and those that handle interrupts. Please note that write access to all MPU registers is disabled after calling any MPU API.

The MPU initialization function is

- MPU_start()

The MPU memory segmentation and access right definition functions are

- MPU_initTwoSegments()
- MPU_initThreeSegments()
- MPU_initInfoSegment()

The MPU interrupt handler functions

- MPU_enablePUCOnViolation()
- MPU_getInterruptStatus()
- MPU_clearInterrupt()
- MPU_clearAllInterrupts()

18.2.2 Function Documentation

uint16_t MPU_clearAllInterrupts (uint16_t *baseAddress*)

Clears all Memory Segment Access Violation Interrupt Flags.

Parameters

baseAddress	is the base address of the MPU module.

Modified bits of **MPUCTL1** register.

Returns

Logical OR of any of the following:

- **MPU_SEG_1_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 1 is detected
- **MPU_SEG_2_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 2 is detected
- **MPU_SEG_3_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 3 is detected

- **MPU_SEG_INFO_ACCESS_VIOLATION** is set if an access violation in User Information Memory Segment is detected
  indicating the status of the interrupt flags.

uint16_t MPU_clearInterrupt (uint16_t *baseAddress,* uint16_t *memAccFlag*)

Clears the masked interrupt flags.

Returns the memory segment violation flag status requested by the user or if user is providing a bit mask value, the function will return a value indicating if all flags were cleared.

Parameters

baseAddress	is the base address of the MPU module.
memAccFlag	is the is the memory access violation flag. Mask value is the logical OR of any of the following: ■ **MPU_SEG_1_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 1 is detected ■ **MPU_SEG_2_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 2 is detected ■ **MPU_SEG_3_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 3 is detected ■ **MPU_SEG_INFO_ACCESS_VIOLATION** - is set if an access violation in User Information Memory Segment is detected

Returns

Logical OR of any of the following:

- **MPU_SEG_1_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 1 is detected
- **MPU_SEG_2_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 2 is detected
- **MPU_SEG_3_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 3 is detected
- **MPU_SEG_INFO_ACCESS_VIOLATION** is set if an access violation in User Information Memory Segment is detected
  indicating the status of the masked flags.

void MPU_disablePUCOnViolation (uint16_t *baseAddress,* uint16_t *segment*)

The following function disables PUC generation when an access violation has occurred on the memory segment selected by the user.

Note that only specified segments for PUC generation are disabled. Other segments for PUC generation are left untouched. Users may call MPU_enablePUCOnViolation() and MPU_disablePUCOnViolation() to assure that all the bits will be set and/or cleared.

Parameters

baseAddress	is the base address of the MPU module.
segment	is the bit mask of memory segment that will NOT generate a PUC when an access violation occurs. Mask value is the logical OR of any of the following: ■ **MPU_FIRST_SEG** - PUC generation on first memory segment ■ **MPU_SECOND_SEG** - PUC generation on second memory segment ■ **MPU_THIRD_SEG** - PUC generation on third memory segment ■ **MPU_INFO_SEG** - PUC generation on user information memory segment

Modified bits of **MPUSAM** register and bits of **MPUCTL0** register.

Returns

None

void MPU_enablePUCOnViolation (uint16_t *baseAddress,* uint16_t *segment*)

The following function enables PUC generation when an access violation has occurred on the memory segment selected by the user.

Note that only specified segments for PUC generation are enabled. Other segments for PUC generation are left untouched. Users may call MPU_enablePUCOnViolation() and MPU_disablePUCOnViolation() to assure that all the bits will be set and/or cleared.

Parameters

baseAddress	is the base address of the MPU module.
segment	is the bit mask of memory segment that will generate a PUC when an access violation occurs. Mask value is the logical OR of any of the following: ■ **MPU_FIRST_SEG** - PUC generation on first memory segment ■ **MPU_SECOND_SEG** - PUC generation on second memory segment ■ **MPU_THIRD_SEG** - PUC generation on third memory segment ■ **MPU_INFO_SEG** - PUC generation on user information memory segment

Modified bits of **MPUSAM** register and bits of **MPUCTL0** register.

Returns

None

uint16_t MPU_getInterruptStatus (uint16_t *baseAddress,* uint16_t *memAccFlag*)

Returns the memory segment violation flag status requested by the user.

Parameters

baseAddress	is the base address of the MPU module.
memAccFlag	is the is the memory access violation flag. Mask value is the logical OR of any of the following: ■ **MPU_SEG_1_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 1 is detected ■ **MPU_SEG_2_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 2 is detected ■ **MPU_SEG_3_ACCESS_VIOLATION** - is set if an access violation in Main Memory Segment 3 is detected ■ **MPU_SEG_INFO_ACCESS_VIOLATION** - is set if an access violation in User Information Memory Segment is detected

Returns

Logical OR of any of the following:
- **MPU_SEG_1_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 1 is detected
- **MPU_SEG_2_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 2 is detected
- **MPU_SEG_3_ACCESS_VIOLATION** is set if an access violation in Main Memory Segment 3 is detected
- **MPU_SEG_INFO_ACCESS_VIOLATION** is set if an access violation in User Information Memory Segment is detected
indicating the status of the masked flags.

void MPU_initInfoSegment (uint16_t *baseAddress,* uint8_t *accmask*)

Initializes user information memory segment.

This function initializes user information memory segment with specified access rights.

Parameters

baseAddress	is the base address of the MPU module.
accmask	is the bit mask of access right for user information memory segment. Mask value is the logical OR of any of the following: ■ **MPU_READ** - Read rights ■ **MPU_WRITE** - Write rights ■ **MPU_EXEC** - Execute rights ■ **MPU_NO_READ_WRITE_EXEC** - no read/write/execute rights

Modified bits of **MPUSAM** register and bits of **MPUCTL0** register.

Returns

None

void MPU_initThreeSegments (uint16_t *baseAddress,* **MPU_initThreeSegmentsParam** ∗ *param*)

Initializes MPU with three memory segments.

This function creates three memory segments in FRAM allowing the user to set access right to each segment. To set the correct value for seg1boundary, the user must consult the Device Family User's Guide and provide the MPUSBx value corresponding to the memory address where the user wants to create the partition. Consult the "Segment Border Setting" section in the User's Guide to find the options available for MPUSBx.

Parameters

baseAddress	is the base address of the MPU module.
param	is the pointer to struct for initializing three segments.

Modified bits of **MPUSAM** register, bits of **MPUSEG** register and bits of **MPUCTL0** register.

Returns

None

References MPU_initThreeSegmentsParam::seg1accmask, MPU_initThreeSegmentsParam::seg1boundary, MPU_initThreeSegmentsParam::seg2accmask, MPU_initThreeSegmentsParam::seg2boundary, and MPU_initThreeSegmentsParam::seg3accmask.

void MPU_initTwoSegments (uint16_t *baseAddress,* uint16_t *seg1boundary,* uint8_t *seg1accmask,* uint8_t *seg2accmask*)

Initializes MPU with two memory segments.

This function creates two memory segments in FRAM allowing the user to set access right to each segment. To set the correct value for seg1boundary, the user must consult the Device Family User's Guide and provide the MPUSBx value corresponding to the memory address where the user wants to create the partition. Consult the "Segment Border Setting" section in the User's Guide to find the options available for MPUSBx.

Parameters

baseAddress	is the base address of the MPU module.
seg1boundary	Valid values can be found in the Family User's Guide
seg1accmask	is the bit mask of access right for memory segment 1. Mask value is the logical OR of any of the following: ■ **MPU_READ** - Read rights ■ **MPU_WRITE** - Write rights ■ **MPU_EXEC** - Execute rights ■ **MPU_NO_READ_WRITE_EXEC** - no read/write/execute rights

seg2accmask	is the bit mask of access right for memory segment 2 Mask value is the logical OR of any of the following: ■ **MPU_READ** - Read rights ■ **MPU_WRITE** - Write rights ■ **MPU_EXEC** - Execute rights ■ **MPU_NO_READ_WRITE_EXEC** - no read/write/execute rights

Modified bits of **MPUSAM** register, bits of **MPUSEG** register and bits of **MPUCTL0** register.

Returns

None

void MPU_start (uint16_t *baseAddress*)

The following function enables the MPU module in the device.

This function needs to be called once all memory segmentation has been done. If this function is not called the MPU module will not be activated.

Parameters

baseAddress	is the base address of the MPU module.

Modified bits of **MPUCTL0** register.

Returns

None

18.3 Programming Example

The following example shows some MPU operations using the APIs

```
//Initialize struct for three segments configuration
MPU_initThreeSegmentsParam threeSegParam;
threeSegParam.seg1boundary = 0x04;
threeSegParam.seg1boundary = 0x08;
threeSegParam.seg1accmask = MPU_READ|MPU_WRITE|MPU_EXEC;
threeSegParam.seg2accmask = MPU_READ;
threeSegParam.seg3accmask = MPU_READ|MPU_WRITE|MPU_EXEC;

//Define memory segment boundaries and set access right for each memory segment
MPU_initThreeSegments(MPU_BASE, &threeSegParam);

// Configures MPU to generate a PUC on access violation on the second segment
MPU_enablePUCOnViolation(MPU_BASE,MPU_SECOND_SEG);

//Enables the MPU module
MPU_start(MPU_BASE);
```

19 32-Bit Hardware Multiplier (MPY32)

19.1 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.

19.2 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.

19.2.1 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()

19.2.2 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

result	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

multiplication↩ Type	is the type of multiplication to perform once the second operand is set. Valid values are: ■ **MPY32_MULTIPLY_UNSIGNED** ■ **MPY32_MULTIPLY_SIGNED** ■ **MPY32_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand	is the 16-bit value to load into the 1st operand.

Returns

None

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

multiplication↩ Type	is the type of multiplication to perform once the second operand is set. Valid values are: ■ **MPY32_MULTIPLY_UNSIGNED** ■ **MPY32_MULTIPLY_SIGNED** ■ **MPY32_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY32_MULTIPLYACCUMULATE_SIGNED**
operand	is the 24-bit value to load into the 1st operand.

Returns

None

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

multiplication↩ Type	is the type of multiplication to perform once the second operand is set. Valid values are: ■ **MPY32_MULTIPLY_UNSIGNED** ■ **MPY32_MULTIPLY_SIGNED** ■ **MPY32_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY32_MULTIPLYACCUMULATE_SIGNED**

operand	is the 32-bit value to load into the 1st operand.

Returns

None

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

multiplication↩ Type	is the type of multiplication to perform once the second operand is set. Valid values are: ■ **MPY32_MULTIPLY_UNSIGNED** ■ **MPY32_MULTIPLY_SIGNED** ■ **MPY32_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY32_MULTIPLYACCUMULATE_SIGNED**
operand	is the 8-bit value to load into the 1st operand.

Returns

None

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

operand	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

operand	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

operand	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

operand	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 unpredicatble results if registers are changed before the results are ready.

Parameters

writeDelay←→ Select	delays the write to any MPY32 register until the selected bit size of result has been written. Valid values are: ■ **MPY32_WRITEDELAY_OFF** [Default] - writes are not delayed ■ **MPY32_WRITEDELAY_32BIT** - writes are delayed until a 32-bit result is available in the result registers ■ **MPY32_WRITEDELAY_64BIT** - writes are delayed until a 64-bit result is available in the result registers Modified bits are **MPYDLY32** and **MPYDLYWRTEN** of **MPY32CTL0** register.

Returns

None

19.3 Programming Example

The following example shows how to initialize and use the MPY32 API to calculate a 16-bit by 16-bit unsigned multiplication operation.

```
WDT_hold(WDT_A_BASE);   // Stop WDT

// Set a 16-bit Operand into the specific Operand 1 register to specify
 // unsigned multiplication
MPY32_setOperandOne16Bit(MPY32_MULTIPLY_UNSIGNED,
                         0x1234);
// Set Operand 2 to begin the multiplication operation
MPY32_setOperandTwo16Bit(0x5678);

__bis_SR_register(LPM4_bits);            // Enter LPM4
__no_operation();                         // BREAKPOINT HERE to verify the
                                            // correct result in the registers
```

20 Power Management Module (PMM)

20.1 Introduction

The PMM manages all functions related to the power supply and its supervision for the device. Its primary functions are first to generate a supply voltage for the core logic, and second, provide several mechanisms for the supervision of the voltage applied to the device (DVCC).

The PMM uses an integrated low-dropout voltage regulator (LDO) to produce a secondary core voltage (VCORE) from the primary one applied to the device (DVCC). In general, VCORE supplies the CPU, memories, and the digital modules, while DVCC supplies the I/Os and analog modules. The VCORE output is maintained using a dedicated voltage reference. The input or primary side of the regulator is referred to as its high side. The output or secondary side is referred to as its low side.

20.2 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_enableSVSH (void)
  Enables the high-side SVS circuitry.
- void PMM_disableSVSH (void)
  Disables the high-side SVS circuitry.
- void PMM_turnOnRegulator (void)
  Makes the low-dropout voltage regulator (LDO) remain ON when going into LPM 3/4.
- void PMM_turnOffRegulator (void)
  Turns OFF the low-dropout voltage regulator (LDO) when going into LPM3/4, thus the system will enter LPM3.5 or LPM4.5 respectively.
- void PMM_trigPOR (void)
  Calling this function will trigger a software Power On Reset (POR).
- void PMM_trigBOR (void)
  Calling this function will trigger a software Brown Out Rest (BOR).
- void PMM_clearInterrupt (uint16_t mask)
  Clears interrupt flags for the PMM.
- uint16_t PMM_getInterruptStatus (uint16_t mask)
  Returns interrupt status.
- void PMM_unlockLPM5 (void)
  Unlock LPM5.

20.2.1 Detailed Description

PMM_enableSVSH() / **PMM_disableSVSH()** If disabled on FR57xx, High-side SVS (SVSH) is disabled in LPM4.5. SVSH is always enabled in active mode and LPM0/1/2/3/4 and LPM3.5. If enabled, SVSH is always enabled. Note: this API has different functionality depending on the part.

PMM_enableSVSL() / **PMM_disableSVSL()** If disabled, Low-side SVS (SVSL) is disabled in low power modes. SVSL is always enabled in active mode and LPM0. If enabled, SVSL is enabled in LPM0/1/2. SVSL is always enabled in AM and always disabled in LPM3/4 and LPM3.5/4.5.

PMM_turnOffRegulator() / **PMM_turnOnRegulator()** If off, Regulator is turned off when going to LPM3/4. System enters LPM3.5 or LPM4.5, respectively. If on, Regulator remains on when going into LPM3/4

PMM_clearInterrupt() Clear selected or all interrupt flags for the PMM

PMM_getInterruptStatus() Returns interrupt status of the selected flag in the PMM module

PMM_lockLPM5() / **PMM_unlockLPM5()** If unlocked, LPMx.5 configuration is not locked and defaults to its reset condition. if locked, LPMx.5 configuration remains locked. Pin state is held during LPMx.5 entry and exit.

20.2.2 Function Documentation

void PMM_clearInterrupt (uint16_t *mask*)

Clears interrupt flags for the PMM.

Parameters

mask	is the mask for specifying the required flag Mask value is the logical OR of any of the following: ■ **PMM_BOR_INTERRUPT** - Software BOR interrupt ■ **PMM_RST_INTERRUPT** - RESET pin interrupt ■ **PMM_POR_INTERRUPT** - Software POR interrupt ■ **PMM_SVSH_INTERRUPT** - SVS high side interrupt ■ **PMM_SVSL_INTERRUPT** - SVS low side interrupt, not available for FR58xx/59xx ■ **PMM_LPM5_INTERRUPT** - LPM5 indication ■ **PMM_ALL** - All interrupts

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

Returns

None

void PMM_disableSVSH (void)

Disables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_disableSVSL (void)

Disables the low-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_enableSVSH (void)

Enables the high-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_enableSVSL (void)

Enables the low-side SVS circuitry.

Modified bits of **PMMCTL0** register.

Returns

None

uint16_t PMM_getInterruptStatus (uint16_t *mask*)

Returns interrupt status.

Parameters

mask	is the mask for specifying the required flag Mask value is the logical OR of any of the following: ■ **PMM_BOR_INTERRUPT** - Software BOR interrupt ■ **PMM_RST_INTERRUPT** - RESET pin interrupt ■ **PMM_POR_INTERRUPT** - Software POR interrupt ■ **PMM_SVSH_INTERRUPT** - SVS high side interrupt ■ **PMM_SVSL_INTERRUPT** - SVS low side interrupt, not available for FR58xx/59xx ■ **PMM_LPM5_INTERRUPT** - LPM5 indication ■ **PMM_ALL** - All interrupts

Returns

Logical OR of any of the following:

- **PMM_BOR_INTERRUPT** Software BOR interrupt
- **PMM_RST_INTERRUPT** RESET pin interrupt
- **PMM_POR_INTERRUPT** Software POR interrupt
- **PMM_SVSH_INTERRUPT** SVS high side interrupt
- **PMM_SVSL_INTERRUPT** SVS low side interrupt, not available for FR58xx/59xx
- **PMM_LPM5_INTERRUPT** LPM5 indication
- **PMM_ALL** All interrupts
  indicating the status of the selected interrupt flags

void PMM_trigBOR (void)

Calling this function will trigger a software Brown Out Rest (BOR).

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_trigPOR (void)

Calling this function will trigger a software Power On Reset (POR).

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_turnOffRegulator (void)

Turns OFF the low-dropout voltage regulator (LDO) when going into LPM3/4, thus the system will enter LPM3.5 or LPM4.5 respectively.

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_turnOnRegulator (void)

Makes the low-dropout voltage regulator (LDO) remain ON when going into LPM 3/4.

Modified bits of **PMMCTL0** register.

Returns

None

void PMM_unlockLPM5 (void)

Unlock LPM5.

LPMx.5 configuration is not locked and defaults to its reset condition. Disable the GPIO power-on default high-impedance mode to activate previously configured port settings.

Returns

None

20.3 Programming Example

The following example shows some pmm operations using the APIs

```
 //Unlock the GPIO pins.
/*
* By default, the pins are unlocked unless waking
* up from an LPMx.5 state in which case all GPIO
* are previously locked.

*/
PMM_unlockLPM5();

//Get Interrupt Status from the PMMIFG register.
/* mask:
 *  PMM_BOR_INTERRUPT
 *  PMM_RST_INTERRUPT,
 *  PMM_POR_INTERRUPT,
 *  PMM_SVSL_INTERRUPT,
 *  PMM_SVSH_INTERRUPT
 *  PMM_LPM5_INTERRUPT,
 * return STATUS_SUCCESS (0x01) or STATUS_FAIL (0x00)
 */
if (PMM_getInterruptStatus(PMM_LPM5_INTERRUPT)) // Was this device in LPMx.5 mode
       before the reset was triggered?
{
    //Clear Interrupt Flag from the PMMIFG register.
    /* mask:
     *  PMM_BOR_INTERRUPT
     *  PMM_RST_INTERRUPT,
     *  PMM_POR_INTERRUPT,
     *  PMM_SVSL_INTERRUPT,
     *  PMM_SVSH_INTERRUPT
     *  PMM_LPM5_INTERRUPT,
     *  PMM_ALL
     */
    PMM_clearInterrupt(PMM_LPM5_INTERRUPT);         // Clear the LPMx.5 flag
}

if (PMM_getInterruptStatus(PMM_RST_INTERRUPT))  // Was this reset triggered by the
       Reset flag?
{
    PMM_clearInterrupt(PMM_RST_INTERRUPT);          // Clear reset flag

    //Trigger a software Brown Out Reset (BOR)
    /*
     * Base Address of PMM,
     * Forces the devices to perform a BOR.
     */
    PMM_trigBOR();                              // Software trigger a BOR.
}
```

```
if (PMM_getInterruptStatus(PMM_BOR_INTERRUPT))    // Was this reset triggered by the
       BOR flag?
{
    PMM_clearInterrupt(PMM_BOR_INTERRUPT);          // Clear BOR flag

    //Disable SVSH
    /*
     * High-side SVS (SVSH) is disabled in LPM4.5. SVSH is
     * always enabled in active mode and LPM0/1/2/3/4 and LPM3.5.
     */
    PMM_disableSVSH();
    //Disable SVSL
    /*
     * Low-side SVS (SVSL) is disabled in low power modes.
     * SVSL is always enabled in active mode and LPM0.
     */
    PMM_disableSVSL();
    //Disable Regulator
    /*
     * Regulator is turned off when going to LPM3/4.
     * System enters LPM3.5 or LPM4.5, respectively.
     */
    PMM_turnOffRegulator();
    __bis_SR_register(LPM4_bits); // Enter LPM4.5, This automatically locks
                                  //(if not locked already) all GPIO pins.
                                  //  and will set the LPM5 flag and set the LOCKLPM5 bit
                                  //  in the PM5CTL0 register upon wake up.
}

while (1)
{
    __no_operation();             // Don't sleep
}
```

21 Internal Reference (REF)

21.1 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 limited to the ADC12_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.

21.2 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_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.

21.2.1 Detailed Description

The REF 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_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()

21.2.2 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

Parameters

baseAddress	is the base address of the REF module.

Modified bits are **REFON** 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

baseAddress	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

baseAddress	is the base address of the REF module.

Modified bits are **REFON** 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

baseAddress	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

baseAddress	is the base address of the REF module.

Returns

One of the following:
- **Ref_STATICMODE** if the bandgap is operating in static mode
- **Ref_SAMPLEMODE** if the bandgap is operating in sample mode
  bandgap mode of the Ref 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

baseAddress	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

baseAddress	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

baseAddress	is the base address of the REF module.

Returns

One of the following:

- **Ref_NOTBUSY** if the reference generator is not being used
- **Ref_BUSY** if the reference generator is being used, disallowing 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

baseAddress	is the base address of the REF module.
reference←VoltageSelect	is the desired voltage to generate for a reference voltage. Valid values are: ■ **REF_VREF1_5V** [Default] ■ **REF_VREF2_0V** ■ **REF_VREF2_5V** Modified bits are **REFVSEL** of **REFCTL0** register.

Returns

None

21.3 Programming Example

The following example shows how to initialize and use the REF API with the ADC12 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
```

22 Real-Time Clock (RTC_B)

22.1 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.

22.2 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.

22.2.1 Detailed Description

The RTC_B API is broken into 5 groups of functions: clock settings, calender mode, prescale counter, interrupt condition setup/enable functions and data conversion.

The RTC_B clock settings are handled by

- RTC_B_startClock()
- RTC_B_holdClock()
- RTC_B_setCalibrationFrequency()
- RTC_B_setCalibrationData()

The RTC_B calender mode is initialized and handled by

- RTC_B_initCalendar()
- RTC_B_configureCalendarAlarm()
- RTC_B_getCalendarTime()

The RTC_B prescale counter is handled by

- RTC_B_getPrescaleValue()
- RTC_B_setPrescaleValue()

The RTC_B interrupts are handled by

- RTC_B_definePrescaleEvent()
- RTC_B_setCalendarEvent()
- RTC_B_enableInterrupt()
- RTC_B_disableInterrupt()
- RTC_B_getInterruptStatus()
- RTC_B_clearInterrupt()

The RTC_B conversions are handled by

- RTC_B_convertBCDToBinary()
- RTC_B_convertBinaryToBCD()

22.2.2 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

baseAddress	is the base address of the RTC_B module.
interruptFlag←Mask	is a bit mask of the interrupt flags to be cleared. Mask value is the logical OR of any of the following: ■ **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met. ■ **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns

None

void RTC_B_configureCalendarAlarm (uint16_t *baseAddress,* **RTC_B_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_B_ALARM_OFF for any alarm settings that should not be apart of the alarm condition.

Parameters

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

Returns

None

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

baseAddress	is the base address of the RTC_B module.
valueToConvert	is the raw value in BCD format to convert to Binary. Modified bits are **BCD2BIN** of **BCD2BIN** register.

Returns

The binary version of the input parameter

uint16_t RTC_B_convertBinaryToBCD (uint16_t *baseAddress,* uint16_t *valueToConvert*)

Convert the given binary value to BCD format.

This function converts binary values to BCD format. This API uses the hardware registers to perform the conversion rather than a software method.

Parameters

baseAddress	is the base address of the RTC_B module.
valueToConvert	is the raw value in Binary format to convert to BCD. Modified bits are **BIN2BCD** of **BIN2BCD** register.

Returns

The BCD version of the valueToConvert parameter

void RTC_B_definePrescaleEvent (uint16_t *baseAddress,* uint8_t *prescaleSelect,* uint8_t *prescaleEventDivider*)

Sets up an interrupt condition for the selected Prescaler.

This function sets the condition for an interrupt to assert based on the individual prescalers.

Parameters

baseAddress	is the base address of the RTC_B module.
prescaleSelect	is the prescaler to define an interrupt for. Valid values are: ■ **RTC_B_PRESCALE_0** ■ **RTC_B_PRESCALE_1**
prescaleEvent←↩ Divider	is a divider to specify when an interrupt can occur based on the clock source of the selected prescaler. (Does not affect timer of the selected prescaler). Valid values are: ■ **RTC_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

baseAddress	is the base address of the RTC_B module.
interruptMask	is a bit mask of the interrupts to disable. Mask value is the logical OR of any of the following: ■ **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met. ■ **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns

None

void RTC_B_enableInterrupt (uint16_t *baseAddress,* uint8_t *interruptMask*)

Enables selected RTC interrupt sources.

This function enables the selected RTC interrupt source. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the RTC_B module.
interruptMask	is a bit mask of the interrupts to enable. Mask value is the logical OR of any of the following: ■ **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met. ■ **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns

None

Calendar RTC_B_getCalendarTime (uint16_t *baseAddress*)

Returns the Calendar Time stored in the Calendar registers of the RTC.

This function returns the current Calendar time in the form of a Calendar structure. The RTCRDY polling is used in this function to prevent reading invalid time.

Parameters

baseAddress	is the base address of the RTC_B module.

Returns

A Calendar structure containing the current time.

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

baseAddress	is the base address of the RTC_B module.
interruptFlag←Mask	is a bit mask of the interrupt flags to return the status of. Mask value is the logical OR of any of the following: ■ **RTC_B_TIME_EVENT_INTERRUPT** - asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met. ■ **RTC_B_CLOCK_ALARM_INTERRUPT** - asserts when alarm condition in Calendar mode is met. ■ **RTC_B_CLOCK_READ_READY_INTERRUPT** - asserts when Calendar registers are settled. ■ **RTC_B_PRESCALE_TIMER0_INTERRUPT** - asserts when Prescaler 0 event condition is met. ■ **RTC_B_PRESCALE_TIMER1_INTERRUPT** - asserts when Prescaler 1 event condition is met. ■ **RTC_B_OSCILLATOR_FAULT_INTERRUPT** - asserts if there is a problem with the 32kHz oscillator, while the RTC is running.

Returns

Logical OR of any of the following:

- **RTC_B_TIME_EVENT_INTERRUPT** asserts when counter overflows in counter mode or when Calendar event condition defined by defineCalendarEvent() is met.
- **RTC_B_CLOCK_ALARM_INTERRUPT** asserts when alarm condition in Calendar mode is met.
- **RTC_B_CLOCK_READ_READY_INTERRUPT** asserts when Calendar registers are settled.
- **RTC_B_PRESCALE_TIMER0_INTERRUPT** asserts when Prescaler 0 event condition is met.
- **RTC_B_PRESCALE_TIMER1_INTERRUPT** asserts when Prescaler 1 event condition is met.
- **RTC_B_OSCILLATOR_FAULT_INTERRUPT** asserts if there is a problem with the 32kHz oscillator, while the RTC is running.
indicating the status of the masked interrupts

uint8_t RTC_B_getPrescaleValue (uint16_t *baseAddress*, uint8_t *prescaleSelect*)

Returns the selected prescaler value.

This function returns the value of the selected prescale counter register. Note that the counter value should be held by calling RTC_B_holdClock() before calling this API.

Parameters

baseAddress	is the base address of the RTC_B module.
prescaleSelect	is the prescaler to obtain the value of. Valid values are: ■ **RTC_B_PRESCALE_0** ■ **RTC_B_PRESCALE_1**

Returns

The value of the specified prescaler count register

void RTC_B_holdClock (uint16_t *baseAddress*)

Holds the RTC.

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

Parameters

baseAddress	is the base address of the RTC_B module.

Returns

None

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

baseAddress	is the base address of the RTC_B module.
CalendarTime	is the pointer to the structure containing the values for the Calendar to be initialized to. Valid values should be of type pointer to Calendar and should contain the following members and corresponding values: **Seconds** between 0-59 **Minutes** between 0-59 **Hours** between 0-23 **DayOfWeek** between 0-6 **DayOfMonth** between 1-31 **Year** between 0-4095 NOTE: Values beyond the ones specified may result in erratic behavior.
formatSelect	is the format for the Calendar registers to use. Valid values are: ■ **RTC_B_FORMAT_BINARY** [Default] ■ **RTC_B_FORMAT_BCD** Modified bits are **RTCBCD** of **RTCCTL1** register.

Returns

None

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

baseAddress	is the base address of the RTC_B module.
eventSelect	is the condition selected. Valid values are: ■ **RTC_B_CALENDAREVENT_MINUTECHANGE** - assert interrupt on every minute ■ **RTC_B_CALENDAREVENT_HOURCHANGE** - assert interrupt on every hour ■ **RTC_B_CALENDAREVENT_NOON** - assert interrupt when hour is 12 ■ **RTC_B_CALENDAREVENT_MIDNIGHT** - assert interrupt when hour is 0 Modified bits are **RTCTEV** of **RTCCTL** register.

Returns

None

void RTC_B_setCalibrationData (uint16_t *baseAddress,* uint8_t *offsetDirection,* uint8_t *offsetValue*)

Sets the specified calibration for the RTC.

This function sets the calibration offset to make the RTC as accurate as possible. The offsetDirection can be either +4-ppm or -2-ppm, and the offsetValue should be from 1-63 and is multiplied by the direction setting (i.e. +4-ppm ∗ 8 (offsetValue) = +32-ppm). Please note, when measuring the frequency after setting the calibration, you will only see a change on the 1Hz frequency.

Parameters

baseAddress	is the base address of the RTC_B module.
offsetDirection	is the direction that the calibration offset will go. Valid values are: ■ **RTC_B_CALIBRATION_DOWN2PPM** - calibrate at steps of -2 ■ **RTC_B_CALIBRATION_UP4PPM** - calibrate at steps of +4 Modified bits are **RTCCALS** of **RTCCTL2** register.
offsetValue	is the value that the offset will be a factor of; a valid value is any integer from 1-63. Modified bits are **RTCCAL** of **RTCCTL2** register.

Returns

None

void RTC_B_setCalibrationFrequency (uint16_t *baseAddress,* uint16_t *frequencySelect*)

Allows and Sets the frequency output to RTCCLK pin for calibration measurement.

This function sets a frequency to measure at the RTCCLK output pin. After testing the set frequency, the calibration could be set accordingly.

Parameters

baseAddress	is the base address of the RTC_B module.
frequencySelect	is the frequency output to RTCCLK. Valid values are: ■ **RTC_B_CALIBRATIONFREQ_OFF** [Default] - turn off calibration output ■ **RTC_B_CALIBRATIONFREQ_512HZ** - output signal at 512Hz for calibration ■ **RTC_B_CALIBRATIONFREQ_256HZ** - output signal at 256Hz for calibration ■ **RTC_B_CALIBRATIONFREQ_1HZ** - output signal at 1Hz for calibration Modified bits are **RTCCALF** of **RTCCTL3** register.

Returns

None

void RTC_B_setPrescaleValue (uint16_t *baseAddress,* uint8_t *prescaleSelect,* uint8_t *prescaleCounterValue*)

Sets the selected prescaler value.

This function sets the prescale counter value. Before setting the prescale counter, it should be held by calling RTC_B_holdClock().

Parameters

baseAddress	is the base address of the RTC_B module.
prescaleSelect	is the prescaler to set the value for. Valid values are: ■ **RTC_B_PRESCALE_0** ■ **RTC_B_PRESCALE_1**
prescale↩ CounterValue	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_B_startClock (uint16_t *baseAddress*)

Starts the RTC.

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

Parameters

baseAddress	is the base address of the RTC_B module.

Returns

None

22.3 Programming Example

The following example shows how to initialize and use the RTC API to setup Calender Mode with the current time and various interrupts.

```
//Initialize calendar struct
Calendar currentTime;
currentTime.Seconds    = 0x00;
currentTime.Minutes    = 0x26;
currentTime.Hours      = 0x13;
currentTime.DayOfWeek  = 0x03;
currentTime.DayOfMonth = 0x20;
currentTime.Month      = 0x07;
currentTime.Year       = 0x2011;

//Initialize alarm struct
RTC_B_configureCalendarAlarmParam alarmParam;
alarmParam.minutesAlarm = 0x00;
alarmParam.hoursAlarm = 0x17;
alarmParam.dayOfWeekAlarm = RTC_B_ALARMCONDITION_OFF;
alarmParam.dayOfMonthAlarm = 0x05;

//Initialize Calendar Mode of RTC_B
/*
 * Base Address of the RTC_B
 * Pass in current time, initialized above
 * Use BCD as Calendar Register Format
 */
RTC_B_initCalendar(RTC_B_BASE,
    &currentTime,
    RTC_B_FORMAT_BCD);

//Setup Calendar Alarm for 5:00pm on the 5th day of the month.
//Note: Does not specify day of the week.
RTC_B_setCalendarAlarm(RTC_B_BASE, &alarmParam);

//Specify an interrupt to assert every minute
RTC_B_setCalendarEvent(RTC_B_BASE,
    RTC_B_CALENDAREVENT_MINUTECHANGE);

//Enable interrupt for RTC_B Ready Status, which asserts when the RTC_B
//Calendar registers are ready to read.
//Also, enable interrupts for the Calendar alarm and Calendar event.
RTC_B_enableInterrupt(RTC_B_BASE,
    RTC_B_CLOCK_READ_READY_INTERRUPT +
    RTC_B_TIME_EVENT_INTERRUPT +
    RTC_B_CLOCK_ALARM_INTERRUPT);

//Start RTC_B Clock
RTC_B_startClock(RTC_B_BASE);

//Enter LPM3 mode with interrupts enabled
__bis_SR_register(LPM3_bits + GIE);
__no_operation();
```

23 SFR Module

23.1 Introduction

The Special Function Registers API provides a set of functions for using the MSP430Ware SFR module. Functions are provided to enable and disable interrupts and control the ∼RST/NMI pin

The SFR module can enable interrupts to be generated from other peripherals of the device.

23.2 API Functions

Functions

- void SFR_enableInterrupt (uint8_t interruptMask)
  Enables selected SFR interrupt sources.
- void SFR_disableInterrupt (uint8_t interruptMask)
  Disables selected SFR interrupt sources.
- uint8_t SFR_getInterruptStatus (uint8_t interruptFlagMask)
  Returns the status of the selected SFR interrupt flags.
- void SFR_clearInterrupt (uint8_t interruptFlagMask)
  Clears the selected SFR interrupt flags.
- void SFR_setResetPinPullResistor (uint16_t pullResistorSetup)
  Sets the pull-up/down resistor on the ∼RST/NMI pin.
- void SFR_setNMIEdge (uint16_t edgeDirection)
  Sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if NMI function is active.
- void SFR_setResetNMIPinFunction (uint8_t resetPinFunction)
  Sets the function of the ∼RST/NMI pin.

23.2.1 Detailed Description

The SFR API is broken into 2 groups: the SFR interrupts and the SFR ∼RST/NMI pin control

The SFR interrupts are handled by

- SFR_enableInterrupt()
- SFR_disableInterrupt()
- SFR_getInterruptStatus()
- SFR_clearInterrupt()

The SFR ∼RST/NMI pin is controlled by

- SFR_setResetPinPullResistor()
- SFR_setNMIEdge()
- SFR_setResetNMIPinFunction()

23.2.2 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

interruptFlag←Mask	is the bit mask of interrupt flags that will be cleared. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

void SFR_disableInterrupt (uint8_t *interruptMask*)

Disables selected SFR interrupt sources.

This function disables the selected SFR interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Parameters

interruptMask	is the bit mask of interrupts that will be disabled. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

void SFR_enableInterrupt (uint8_t *interruptMask*)

Enables selected SFR interrupt sources.

This function enables the selected SFR interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. Does not clear interrupt flags.

Parameters

interruptMask	is the bit mask of interrupts that will be enabled. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

None

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

interruptFlag← Mask	is the bit mask of interrupt flags that the status of should be returned. Mask value is the logical OR of any of the following: ■ **SFR_JTAG_OUTBOX_INTERRUPT** - JTAG outbox interrupt ■ **SFR_JTAG_INBOX_INTERRUPT** - JTAG inbox interrupt ■ **SFR_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** - Vacant memory access interrupt ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** - Watchdog interval timer interrupt

Returns

A bit mask of the status of the selected interrupt flags. Return Logical OR of any of the following:

- **SFR_JTAG_OUTBOX_INTERRUPT** JTAG outbox interrupt
- **SFR_JTAG_INBOX_INTERRUPT** JTAG inbox interrupt
- **SFR_NMI_PIN_INTERRUPT** NMI pin interrupt, if NMI function is chosen
- **SFR_VACANT_MEMORY_ACCESS_INTERRUPT** Vacant memory access interrupt
- **SFR_OSCILLATOR_FAULT_INTERRUPT** Oscillator fault interrupt
- **SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT** Watchdog interval timer interrupt indicating the status of the masked interrupts

void SFR_setNMIEdge (uint16_t *edgeDirection*)

Sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if NMI function is active.

This function sets the edge direction that will assert an NMI from a signal on the ∼RST/NMI pin if the NMI function is active. To activate the NMI function of the ∼RST/NMI use the SFR_setResetNMIPinFunction() passing SFR_RESETPINFUNC_NMI into the resetPinFunction parameter.

Parameters

edgeDirection	is the direction that the signal on the ∼RST/NMI pin should go to signal an interrupt, if enabled. Valid values are: ■ **SFR_NMI_RISINGEDGE** [Default] ■ **SFR_NMI_FALLINGEDGE** Modified bits are **SYSNMIIES** of **SFRRPCR** register.

Returns

None

void SFR_setResetNMIPinFunction (uint8_t *resetPinFunction*)

Sets the function of the ∼RST/NMI pin.

This function sets the functionality of the ∼RST/NMI pin, whether in reset mode which will assert a reset if a low signal is observed on that pin, or an NMI which will assert an interrupt from an edge of the signal dependent on the setting of the edgeDirection parameter in SFR_setNMIEdge().

Parameters

resetPin←↩ Function	is the function that the ∼RST/NMI pin should take on. Valid values are: ■ **SFR_RESETPINFUNC_RESET** [Default] ■ **SFR_RESETPINFUNC_NMI** Modified bits are **SYSNMI** of **SFRRPCR** register.

Returns

None

void SFR_setResetPinPullResistor (uint16_t *pullResistorSetup*)

Sets the pull-up/down resistor on the ∼RST/NMI pin.

This function sets the pull-up/down resistors on the ∼RST/NMI pin to the settings from the pullResistorSetup parameter.

Parameters

pullResistor↩ Setup	is the selection of how the pull-up/down resistor on the ∼RST/NMI pin should be setup or disabled. Valid values are: ■ **SFR_RESISTORDISABLE** ■ **SFR_RESISTORENABLE_PULLUP** [Default] ■ **SFR_RESISTORENABLE_PULLDOWN** Modified bits are **SYSRSTUP** and **SYSRSTRE** of **SFRRPCR** register.

Returns

None

23.3 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));
```

24 System Control Module

24.1 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.

24.2 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_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.

24.2.1 Detailed Description

The SYS API is broken into 2 groups: the various SYS controls and the JTAG mailbox controls.

The various SYS controls are handled by

- SysCtl_enableDedicatedJTAGPins()
- SysCtl_getBSLEntryIndication()
- SysCtl_enablePMMAccessProtect()
- SysCtl_enableRAMBasedInterruptVectors()
- SysCtl_disableRAMBasedInterruptVectors()

The JTAG Mailbox controls are handled by

- SysCtl_initJTAGMailbox()
- SysCtl_getJTAGMailboxFlagStatus()
- SysCtl_getJTAGInboxMessage16Bit()
- SysCtl_getJTAGInboxMessage32Bit()
- SysCtl_setJTAGOutgoingMessage16Bit()
- SysCtl_setJTAGOutgoingMessage32Bit()
- SysCtl_clearJTAGMailboxFlagStatus()

24.2.2 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

mailboxFlag↩ Mask	is the bit mask of JTAG mailbox flags that the status of should be cleared. Mask value is the logical OR of any of the following: - **SYSCTL_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0 - **SYSCTL_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1 - **SYSCTL_JTAGINBOX_FLAG0** - flag for JTAG inbox 0 - **SYSCTL_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns

None

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_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_BSLENTRY_INDICATED**
- **SysCtl_BSLENTRY_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

inboxSelect	is the chosen JTAG inbox that the contents of should be returned Valid values are: ■ **SYSCTL_JTAGINBOX_0** - return contents of JTAG inbox 0 ■ **SYSCTL_JTAGINBOX_1** - return contents of JTAG inbox 1

Returns

The contents of the selected JTAG inbox in a 16 bit format.

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

mailboxFlag↵ Mask	is the bit mask of JTAG mailbox flags that the status of should be returned. Mask value is the logical OR of any of the following: ■ **SYSCTL_JTAGOUTBOX_FLAG0** - flag for JTAG outbox 0 ■ **SYSCTL_JTAGOUTBOX_FLAG1** - flag for JTAG outbox 1 ■ **SYSCTL_JTAGINBOX_FLAG0** - flag for JTAG inbox 0 ■ **SYSCTL_JTAGINBOX_FLAG1** - flag for JTAG inbox 1

Returns

A bit mask of the status of the selected mailbox flags.

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

mailboxSize↩ Select	is the size of the JTAG Mailboxes, whether 16- or 32-bits. Valid values are: ■ **SYSCTL_JTAGMBSIZE_16BIT** [Default] - the JTAG messages will take up only one JTAG mailbox (i. e. an outgoing message will take up only 1 outbox of the JTAG mailboxes) ■ **SYSCTL_JTAGMBSIZE_32BIT** - the JTAG messages will be contained within both JTAG mailboxes (i. e. an outgoing message will take up both Outboxes of the JTAG mailboxes) Modified bits are **JMBMODE** of **SYSJMBC** register.
autoClear↩ InboxFlagSelect	decides how the JTAG inbox flags should be cleared, whether automatically after the corresponding outbox has been written to, or manually by software. Valid values are: ■ **SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1AUTO** [Default] - both JTAG inbox flags will be reset automatically when the corresponding inbox is read from. ■ **SYSCTL_JTAGINBOX0AUTO_JTAGINBOX1SW** - only JTAG inbox 0 flag is reset automatically, while JTAG inbox 1 is reset with the ■ **SYSCTL_JTAGINBOX0SW_JTAGINBOX1AUTO** - only JTAG inbox 1 flag is reset automatically, while JTAG inbox 0 is reset with the ■ **SYSCTL_JTAGINBOX0SW_JTAGINBOX1SW** - both JTAG inbox flags will need to be reset manually by the Modified bits are **JMBCLR0OFF** and **JMBCLR1OFF** of **SYSJMBC** register.

Returns

None

void SysCtl_setJTAGOutgoingMessage16Bit (uint8_t *outboxSelect,* uint16_t *outgoingMessage*)

Sets a 16 bit outgoing message in to the selected JTAG Outbox.

This function sets the outgoing message in the selected JTAG outbox. The corresponding JTAG outbox flag is cleared after this function, and set after the JTAG has read the message.

Parameters

outboxSelect	is the chosen JTAG outbox that the message should be set it. Valid values are: ■ **SYSCTL_JTAGOUTBOX_0** - set the contents of JTAG outbox 0 ■ **SYSCTL_JTAGOUTBOX_1** - set the contents of JTAG outbox 1

outgoing←Message	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

outgoing←Message	is the message to send to the JTAG. Modified bits are **MSGHI** and **MSGLO** of **SYSJMBOx** register.

Returns

None

24.3 Programming Example

The following example shows how to initialize and use the SYS API

```
SysCtl_enableRAMBasedInterruptVectors();
```

25 16-Bit Timer_A (TIMER_A)

25.1 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.

25.2 API Functions

Functions

- void Timer_A_startCounter (uint16_t baseAddress, uint16_t timerMode)
  Starts Timer_A counter.
- void Timer_A_initContinuousMode (uint16_t baseAddress, Timer_A_initContinuousModeParam ∗param)
  Configures Timer_A in continuous mode.
- void Timer_A_initUpMode (uint16_t baseAddress, Timer_A_initUpModeParam ∗param)
  Configures Timer_A in up mode.
- void Timer_A_initUpDownMode (uint16_t baseAddress, Timer_A_initUpDownModeParam ∗param)
  Configures Timer_A in up down mode.
- void Timer_A_initCaptureMode (uint16_t baseAddress, Timer_A_initCaptureModeParam ∗param)
  Initializes Capture Mode.
- void Timer_A_initCompareMode (uint16_t baseAddress, Timer_A_initCompareModeParam ∗param)
  Initializes Compare Mode.
- void Timer_A_enableInterrupt (uint16_t baseAddress)
  Enable timer interrupt.
- void Timer_A_disableInterrupt (uint16_t baseAddress)
  Disable timer interrupt.
- uint32_t Timer_A_getInterruptStatus (uint16_t baseAddress)
  Get timer interrupt status.
- void Timer_A_enableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Enable capture compare interrupt.
- void Timer_A_disableCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Disable capture compare interrupt.
- uint32_t Timer_A_getCaptureCompareInterruptStatus (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t mask)
  Return capture compare interrupt status.
- void Timer_A_clear (uint16_t baseAddress)
  Reset/Clear the timer clock divider, count direction, count.
- uint8_t Timer_A_getSynchronizedCaptureCompareInput (uint16_t baseAddress, uint16_t captureCompareRegister, uint16_t synchronized)
  Get synchronized capturecompare input.
- uint8_t Timer_A_getOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get output bit for output mode.
- uint16_t Timer_A_getCaptureCompareCount (uint16_t baseAddress, uint16_t captureCompareRegister)
  Get current capturecompare count.
- void Timer_A_setOutputForOutputModeOutBitValue (uint16_t baseAddress, uint16_t captureCompareRegister, uint8_t outputModeOutBitValue)
  Set output bit for output mode.
- void Timer_A_outputPWM (uint16_t baseAddress, Timer_A_outputPWMParam ∗param)
  Generate a PWM with timer running in up mode.
- void Timer_A_stop (uint16_t baseAddress)

Stops the timer.

- void Timer_A_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
  Sets the value of the capture-compare register.
- void Timer_A_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.

25.2.1 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()

25.2.2 Function Documentation

void Timer_A_clear (uint16_t *baseAddress*)

Reset/Clear the timer clock divider, count direction, count.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

void Timer_A_clearCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_A module.
capture←↩ Compare←↩ Register	selects the Capture-compare register being used. Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits are **CCIFG** of **TAxCCTLn** register.

Returns

None

void Timer_A_clearTimerInterrupt (uint16_t *baseAddress*)

Clears the Timer TAIFG interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits are **TAIFG** of **TAxCTL** register.

Returns

None

void Timer_A_disableCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Disable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_A module.
capture←↩ Compare←↩ Register	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TAxCCTLn** register.

Returns

None

void Timer_A_disableInterrupt (uint16_t *baseAddress*)

Disable timer interrupt.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

void Timer_A_enableCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Enable capture compare interrupt.

Does not clear interrupt flags

Parameters

baseAddress	is the base address of the TIMER_A module.
capture←Compare←Register	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TAxCCTLn** register.

Returns

None

void Timer_A_enableInterrupt (uint16_t *baseAddress*)

Enable timer interrupt.

Does not clear interrupt flags

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

uint16_t Timer_A_getCaptureCompareCount (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Get current capturecompare count.

Parameters

baseAddress	is the base address of the TIMER_A module.
capture← Compare← Register	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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

baseAddress	is the base address of the TIMER_A module.
capture← Compare← Register	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

mask	is the mask for the interrupt status Mask value is the logical OR of any of the following: ■ **TIMER_A_CAPTURE_OVERFLOW** ■ **TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG**

Returns

Logical OR of any of the following:
- **Timer_A_CAPTURE_OVERFLOW**
- **Timer_A_CAPTURECOMPARE_INTERRUPT_FLAG**
  indicating the status of the masked interrupts

uint16_t Timer_A_getCounterValue (uint16_t *baseAddress*)

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The TIMER_A_THRESHOLD #define in the corresponding header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

baseAddress	is the base address of the TIMER_A module.

Returns

Majority vote of timer count value

uint32_t Timer_A_getInterruptStatus (uint16_t *baseAddress*)

Get timer interrupt status.

Parameters

baseAddress	is the base address of the TIMER_A module.

Returns

One of the following:
- **Timer_A_INTERRUPT_NOT_PENDING**
- **Timer_A_INTERRUPT_PENDING**
  indicating the Timer_A interrupt status

uint8_t Timer_A_getOutputForOutputModeOutBitValue (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Get output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
capture←Compare←Register	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

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

baseAddress	is the base address of the TIMER_A module.
capture←Compare←Register	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

synchronized	Valid values are: ■ **TIMER_A_READ_SYNCHRONIZED_CAPTURECOMPAREINPUT** ■ **TIMER_A_READ_CAPTURE_COMPARE_INPUT**

Returns

One of the following:
- **Timer_A_CAPTURECOMPARE_INPUT_HIGH**
- **Timer_A_CAPTURECOMPARE_INPUT_LOW**

void Timer_A_initCaptureMode (uint16_t *baseAddress,* **Timer_A_initCaptureModeParam** ∗ *param*)

Initializes Capture Mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for capture mode initialization.

Modified bits of **TAxCCTLn** register.

Returns

None

References Timer_A_initCaptureModeParam::captureInputSelect,
Timer_A_initCaptureModeParam::captureInterruptEnable,
Timer_A_initCaptureModeParam::captureMode,
Timer_A_initCaptureModeParam::captureOutputMode,
Timer_A_initCaptureModeParam::captureRegister, and
Timer_A_initCaptureModeParam::synchronizeCaptureSource.

void Timer_A_initCompareMode (uint16_t *baseAddress,* **Timer_A_initCompareModeParam** ∗ *param*)

Initializes Compare Mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for compare mode initialization.

Modified bits of **TAxCCRn** register and bits of **TAxCCTLn** register.

Returns

None

References Timer_A_initCompareModeParam::compareInterruptEnable,
Timer_A_initCompareModeParam::compareOutputMode,
Timer_A_initCompareModeParam::compareRegister, and
Timer_A_initCompareModeParam::compareValue.

void Timer_A_initContinuousMode (uint16_t *baseAddress,* **Timer_A_initContinuous←ModeParam** $*$ *param*)

Configures Timer_A in continuous mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for continuous mode initialization.

Modified bits of **TAxCTL** register.

Returns

None

References Timer_A_initContinuousModeParam::clockSource,
Timer_A_initContinuousModeParam::clockSourceDivider,
Timer_A_initContinuousModeParam::startTimer, Timer_A_initContinuousModeParam::timerClear,
and Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE.

void Timer_A_initUpDownMode (uint16_t *baseAddress,* **Timer_A_initUpDownModeParam** ∗ *param*)

Configures Timer_A in up down mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for up-down mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE,
Timer_A_initUpDownModeParam::clockSource,
Timer_A_initUpDownModeParam::clockSourceDivider,
Timer_A_initUpDownModeParam::startTimer, Timer_A_initUpDownModeParam::timerClear,
Timer_A_initUpDownModeParam::timerInterruptEnable_TAIE, and
Timer_A_initUpDownModeParam::timerPeriod.

void Timer_A_initUpMode (uint16_t *baseAddress,* **Timer_A_initUpModeParam** ∗ *param*)

Configures Timer_A in up mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for up mode initialization.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register and bits of **TAxCCR0** register.

Returns

None

References Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE,
Timer_A_initUpModeParam::clockSource, Timer_A_initUpModeParam::clockSourceDivider,

Timer_A_initUpModeParam::startTimer, Timer_A_initUpModeParam::timerClear,
Timer_A_initUpModeParam::timerInterruptEnable_TAIE, and
Timer_A_initUpModeParam::timerPeriod.

void Timer_A_outputPWM (uint16_t *baseAddress,* **Timer_A_outputPWMParam** ∗ *param*)

Generate a PWM with timer running in up mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for PWM configuration.

Modified bits of **TAxCTL** register, bits of **TAxCCTL0** register, bits of **TAxCCR0** register and bits of **TAxCCTLn** register.

Returns

None

References Timer_A_outputPWMParam::clockSource,
Timer_A_outputPWMParam::clockSourceDivider,
Timer_A_outputPWMParam::compareOutputMode, Timer_A_outputPWMParam::compareRegister,
Timer_A_outputPWMParam::dutyCycle, and Timer_A_outputPWMParam::timerPeriod.

void Timer_A_setCompareValue (uint16_t *baseAddress,* uint16_t *compareRegister,* uint16_t *compareValue*)

Sets the value of the capture-compare register.

Parameters

baseAddress	is the base address of the TIMER_A module.
compare←↩ Register	selects the Capture register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**

compareValue	is the count to be compared with in compare mode

Modified bits of **TAxCCRn** register.

Returns

None

void Timer_A_setOutputForOutputModeOutBitValue (uint16_t *baseAddress,* uint16_t *captureCompareRegister,* uint8_t *outputModeOutBitValue*)

Set output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
capture←Compare←Register	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_6**
outputMode←OutBitValue	is the value to be set for out bit Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TAxCCTLn** register.

Returns

None

void Timer_A_startCounter (uint16_t *baseAddress,* uint16_t *timerMode*)

Starts Timer_A counter.

This function assumes that the timer has been previously configured using Timer_A_initContinuousMode, Timer_A_initUpMode or Timer_A_initUpDownMode.

Parameters

baseAddress	is the base address of the TIMER_A module.

timerMode	mode to put the timer in Valid values are: ■ **TIMER_A_STOP_MODE** ■ **TIMER_A_UP_MODE** ■ **TIMER_A_CONTINUOUS_MODE** [Default] ■ **TIMER_A_UPDOWN_MODE**

Modified bits of **TAxCTL** register.

Returns

None

void Timer_A_stop (uint16_t *baseAddress*)

Stops the timer.

Parameters

baseAddress	is the base address of the TIMER_A module.

Modified bits of **TAxCTL** register.

Returns

None

25.3 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();
}
```

26 16-Bit Timer_B (TIMER_B)

26.1 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.

26.2 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_outputPWMParam ∗param)
  Generate a PWM with Timer_B running in up mode.
- void Timer_B_stop (uint16_t baseAddress)
  Stops the Timer_B.
- void Timer_B_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
  Sets the value of the capture-compare register.
- void Timer_B_clearTimerInterrupt (uint16_t baseAddress)
  Clears the Timer_B TBIFG interrupt flag.
- void Timer_B_clearCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Clears the capture-compare interrupt flag.
- void Timer_B_selectCounterLength (uint16_t baseAddress, uint16_t counterLength)
  Selects Timer_B counter length.
- void Timer_B_selectLatchingGroup (uint16_t baseAddress, uint16_t groupLatch)
  Selects Timer_B Latching Group.
- void Timer_B_initCompareLatchLoadEvent (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareLatchLoadEvent)
  Selects Compare Latch Load Event.
- uint16_t Timer_B_getCounterValue (uint16_t baseAddress)
  Reads the current timer count value.

26.2.1 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()

26.2.2 Function Documentation

void Timer_B_clear (uint16_t *baseAddress*)

Reset/Clear the Timer_B clock divider, count direction, count.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

void Timer_B_clearCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Clears the capture-compare interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_B module.

capture←Compare←Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits are **CCIFG** of **TBxCCTLn** register.

Returns

None

void Timer_B_clearTimerInterrupt (uint16_t *baseAddress*)

Clears the Timer_B TBIFG interrupt flag.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits are **TBIFG** of **TBxCTL** register.

Returns

None

void Timer_B_disableCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Disable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.
capture←Compare←Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TBxCCTLn** register.

Returns

None

void Timer_B_disableInterrupt (uint16_t *baseAddress*)

Disable Timer_B interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

void Timer_B_enableCaptureCompareInterrupt (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Enable capture compare interrupt.

Parameters

baseAddress	is the base address of the TIMER_B module.
captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Modified bits of **TBxCCTLn** register.

Returns

None

void Timer_B_enableInterrupt (uint16_t *baseAddress*)

Enable Timer_B interrupt.

Enables Timer_B interrupt. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

uint16_t Timer_B_getCaptureCompareCount (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Get current capturecompare count.

Parameters

baseAddress	is the base address of the TIMER_B module.
capture↩ Compare↩ Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Returns

Current count as uint16_t

uint32_t Timer_B_getCaptureCompareInterruptStatus (uint16_t *baseAddress,* uint16_t *captureCompareRegister,* uint16_t *mask*)

Return capture compare interrupt status.

Parameters

baseAddress	is the base address of the TIMER_B module.
capture←→ Compare←→ Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
mask	is the mask for the interrupt status Mask value is the logical OR of any of the following: ■ **TIMER_B_CAPTURE_OVERFLOW** ■ **TIMER_B_CAPTURECOMPARE_INTERRUPT_FLAG**

Returns

Logical OR of any of the following:
- **Timer_B_CAPTURE_OVERFLOW**
- **Timer_B_CAPTURECOMPARE_INTERRUPT_FLAG**
  indicating the status of the masked interrupts

uint16_t Timer_B_getCounterValue (uint16_t *baseAddress*)

Reads the current timer count value.

Reads the current count value of the timer. There is a majority vote system in place to confirm an accurate value is returned. The Timer_B_THRESHOLD #define in the associated header file can be modified so that the votes must be closer together for a consensus to occur.

Parameters

baseAddress	is the base address of the Timer module.

Returns

Majority vote of timer count value

uint32_t Timer_B_getInterruptStatus (uint16_t *baseAddress*)

Get Timer_B interrupt status.

Parameters

baseAddress	is the base address of the TIMER_B module.

Returns

One of the following:

- **Timer_B_INTERRUPT_NOT_PENDING**
- **Timer_B_INTERRUPT_PENDING**
  indicating the status of the Timer_B interrupt

uint8_t Timer_B_getOutputForOutputModeOutBitValue (uint16_t *baseAddress,* uint16_t *captureCompareRegister*)

Get output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
capture←Compare←Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**

Returns

One of the following:

- **Timer_B_OUTPUTMODE_OUTBITVALUE_HIGH**
- **Timer_B_OUTPUTMODE_OUTBITVALUE_LOW**

uint8_t Timer_B_getSynchronizedCaptureCompareInput (uint16_t *baseAddress,* uint16_t *captureCompareRegister,* uint16_t *synchronized*)

Get synchronized capturecompare input.

Parameters

baseAddress	is the base address of the TIMER_B module.

captureCompareRegister	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
synchronized	selects the type of capture compare input Valid values are: ■ **TIMER_B_READ_SYNCHRONIZED_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

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for capture mode initialization.

Modified bits of **TBxCCTLn** register.

Returns

None

References Timer_B_initCaptureModeParam::captureInputSelect, Timer_B_initCaptureModeParam::captureInterruptEnable, Timer_B_initCaptureModeParam::captureMode, Timer_B_initCaptureModeParam::captureOutputMode, Timer_B_initCaptureModeParam::captureRegister, and Timer_B_initCaptureModeParam::synchronizeCaptureSource.

void Timer_B_initCompareLatchLoadEvent (uint16_t *baseAddress,* uint16_t *compareRegister,* uint16_t *compareLatchLoadEvent*)

Selects Compare Latch Load Event.

Parameters

baseAddress	is the base address of the TIMER_B module.
compare↩ Register	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**
compareLatch↩ LoadEvent	selects the latch load event Valid values are: ■ **TIMER_B_LATCH_ON_WRITE_TO_TBxCCRn_COMPARE_REGISTER** [Default] ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UP_OR_CONT_MODE** ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_0_IN_UPDOWN_MODE** ■ **TIMER_B_LATCH_WHEN_COUNTER_COUNTS_TO_CURRENT_COMPARE_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

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for compare mode initialization.

Modified bits of **TBxCCTLn** register and bits of **TBxCCRn** register.

Returns

None

References Timer_B_initCompareModeParam::compareInterruptEnable, Timer_B_initCompareModeParam::compareOutputMode, Timer_B_initCompareModeParam::compareRegister, and Timer_B_initCompareModeParam::compareValue.

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

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for continuous mode initialization.

Modified bits of **TBxCTL** register.

Returns

None

References Timer_B_initContinuousModeParam::clockSource, Timer_B_initContinuousModeParam::clockSourceDivider, Timer_B_initContinuousModeParam::startTimer, Timer_B_initContinuousModeParam::timerClear, and Timer_B_initContinuousModeParam::timerInterruptEnable_TBIE.

void Timer_B_initUpDownMode (uint16_t *baseAddress,* **Timer_B_initUpDownModeParam** ∗ *param*)

Configures Timer_B in up down mode.

This API does not start the timer. Timer needs to be started when required using the Timer_B_startCounter API.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for up-down mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_initUpDownModeParam::captureCompareInterruptEnable_CCR0_CCIE, Timer_B_initUpDownModeParam::clockSource, Timer_B_initUpDownModeParam::clockSourceDivider, Timer_B_initUpDownModeParam::startTimer, Timer_B_initUpDownModeParam::timerClear,

Timer_B_initUpDownModeParam::timerInterruptEnable_TBIE, and
Timer_B_initUpDownModeParam::timerPeriod.

void Timer_B_initUpMode (uint16_t *baseAddress,* **Timer_B_initUpModeParam** ∗ *param*)

Configures Timer_B in up mode.

This API does not start the timer. Timer needs to be started when required using the Timer_B_startCounter API.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for up mode initialization.

Modified bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE,
Timer_B_initUpModeParam::clockSource, Timer_B_initUpModeParam::clockSourceDivider,
Timer_B_initUpModeParam::startTimer, Timer_B_initUpModeParam::timerClear,
Timer_B_initUpModeParam::timerInterruptEnable_TBIE, and
Timer_B_initUpModeParam::timerPeriod.

void Timer_B_outputPWM (uint16_t *baseAddress,* **Timer_B_outputPWMParam** ∗ *param*)

Generate a PWM with Timer_B running in up mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
param	is the pointer to struct for PWM configuration.

Modified bits of **TBxCCTLn** register, bits of **TBxCTL** register, bits of **TBxCCTL0** register and bits of **TBxCCR0** register.

Returns

None

References Timer_B_outputPWMParam::clockSource,
Timer_B_outputPWMParam::clockSourceDivider,
Timer_B_outputPWMParam::compareOutputMode, Timer_B_outputPWMParam::compareRegister,
Timer_B_outputPWMParam::dutyCycle, and Timer_B_outputPWMParam::timerPeriod.

void Timer_B_selectCounterLength (uint16_t *baseAddress,* uint16_t *counterLength*)

Selects Timer_B counter length.

Parameters

baseAddress	is the base address of the TIMER_B module.
counterLength	selects the value of counter length. Valid values are: ■ **TIMER_B_COUNTER_16BIT** [Default] ■ **TIMER_B_COUNTER_12BIT** ■ **TIMER_B_COUNTER_10BIT** ■ **TIMER_B_COUNTER_8BIT**

Modified bits are **CNTL** of **TBxCTL** register.

Returns

None

void Timer_B_selectLatchingGroup (uint16_t *baseAddress,* uint16_t *groupLatch*)

Selects Timer_B Latching Group.

Parameters

baseAddress	is the base address of the TIMER_B module.
groupLatch	selects the latching group. Valid values are: ■ **TIMER_B_GROUP_NONE** [Default] ■ **TIMER_B_GROUP_CL12_CL23_CL56** ■ **TIMER_B_GROUP_CL123_CL456** ■ **TIMER_B_GROUP_ALL**

Modified bits are **TBCLGRP** of **TBxCTL** register.

Returns

None

void Timer_B_setCompareValue (uint16_t *baseAddress,* uint16_t *compareRegister,* uint16_t *compareValue*)

Sets the value of the capture-compare register.

Parameters

baseAddress	is the base address of the TIMER_B module.
compare←Register	selects the compare register being used. Refer to datasheet to ensure the device has the compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
compareValue	is the count to be compared with in compare mode

Modified bits of **TBxCCRn** register.

Returns

None

void Timer_B_setOutputForOutputModeOutBitValue (uint16_t *baseAddress,* uint16_t *captureCompareRegister,* uint8_t *outputModeOutBitValue*)

Set output bit for output mode.

Parameters

baseAddress	is the base address of the TIMER_B module.
capture←Compare←Register	selects the capture compare register being used. Refer to datasheet to ensure the device has the capture compare register being used. Valid values are: ■ **TIMER_B_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_2** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_3** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_4** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_5** ■ **TIMER_B_CAPTURECOMPARE_REGISTER_6**
outputMode←OutBitValue	the value to be set for out bit Valid values are: ■ **TIMER_B_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_B_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TBxCCTLn** register.

Returns

None

void Timer_B_startCounter (uint16_t *baseAddress,* uint16_t *timerMode*)

Starts Timer_B counter.

This function assumes that the timer has been previously configured using Timer_B_initContinuousMode, Timer_B_initUpMode or Timer_B_initUpDownMode.

Parameters

baseAddress	is the base address of the TIMER_B module.
timerMode	selects the mode of the timer Valid values are: ■ **TIMER_B_STOP_MODE** ■ **TIMER_B_UP_MODE** ■ **TIMER_B_CONTINUOUS_MODE** [Default] ■ **TIMER_B_UPDOWN_MODE**

Modified bits of **TBxCTL** register.

Returns

None

void Timer_B_stop (uint16_t *baseAddress*)

Stops the Timer_B.

Parameters

baseAddress	is the base address of the TIMER_B module.

Modified bits of **TBxCTL** register.

Returns

None

26.3 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
        );

}
```

27 Tag Length Value

27.1 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

27.2 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.

27.2.1 Detailed Description

The APIs that help in querying the information in the TLV structure are listed

- TLV_getInfo() This function retrieves the value of a tag and the length of the tag.
- TLV_getDeviceType() This function retrieves the unique device ID from the TLV structure.
- TLV_getMemory() The returned value is zero if the end of the memory list is reached.
- TLV_getPeripheral() The returned value is zero if the specified tag value (peripheral) is not available in the device.
- TLV_getInterrupt() The returned value is zero is the specified interrupt vector is not defined.

27.2.2 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

tag	represents the tag for which the information needs to be retrieved. Valid values are: ■ **TLV_TAG_LDTAG** ■ **TLV_TAG_PDTAG** ■ **TLV_TAG_Reserved3** ■ **TLV_TAG_Reserved4** ■ **TLV_TAG_BLANK** ■ **TLV_TAG_Reserved6** ■ **TLV_TAG_Reserved7** ■ **TLV_TAG_TAGEND** ■ **TLV_TAG_TAGEXT** ■ **TLV_TAG_TIMER_D_CAL** ■ **TLV_DEVICE_ID_0** ■ **TLV_DEVICE_ID_1** ■ **TLV_TAG_DIERECORD** ■ **TLV_TAG_ADCCAL** ■ **TLV_TAG_ADC12CAL** ■ **TLV_TAG_ADC10CAL** ■ **TLV_TAG_REFCAL**

instance	In some cases a specific tag may have more than one instance. For example there may be multiple instances of timer calibration data present under a single Timer Cal tag. This variable specifies the instance for which information is to be retrieved (0, 1, etc.). When only one instance exists; 0 is passed.
length	Acts as a return through indirect reference. The function retrieves the value of the TLV tag length. This value is pointed to by ∗length and can be used by the application level once the function is called. If the specified tag is not found then the pointer is null 0.
data_address	acts as a return through indirect reference. Once the function is called data_address points to the pointer that holds the value retrieved from the specified TLV tag. If the specified tag is not found then the pointer is null 0.

Returns

None

Referenced by 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

tag	represents the tag for the interrupt vector. Interrupt vector tags number from 0 to N depending on the number of available interrupts. Refer to the device datasheet for a list of available interrupts.

Returns

The returned value is zero is the specified interrupt vector is not defined.

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

instance	In some cases a specific tag may have more than one instance. This variable specifies the instance for which information is to be retrieved (0, 1 etc). When only one instance exists; 0 is passed.

Returns

The returned value is zero if the end of the memory list is reached.

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.

he Peripheral Descriptor tag is split into two portions a list of the available flash memory blocks followed by a list of available peripherals. This function is used to parse through the second portion and can be used to check if a specific peripheral is present in a device. The function calls TLV_getPeripheral() recursively until the end of the memory list and consequently the beginning of the peripheral list is reached. $<$

Parameters

tag	represents represents the tag for a specific peripheral for which the information needs to be retrieved. In the header file tlv. h specific peripheral tags are pre-defined, for example USCIA_B and TA0 are defined as TLV_PID_USCI_AB and TLV_PID_TA2 respectively. Valid values are: ■ **TLV_PID_NO_MODULE** - No Module ■ **TLV_PID_PORTMAPPING** - Port Mapping ■ **TLV_PID_MSP430CPUXV2** - MSP430CPUXV2 ■ **TLV_PID_JTAG** - JTAG ■ **TLV_PID_SBW** - SBW ■ **TLV_PID_EEM_XS** - EEM X-Small ■ **TLV_PID_EEM_S** - EEM Small ■ **TLV_PID_EEM_M** - EEM Medium ■ **TLV_PID_EEM_L** - EEM Large ■ **TLV_PID_PMM** - PMM ■ **TLV_PID_PMM_FR** - PMM FRAM ■ **TLV_PID_FCTL** - Flash ■ **TLV_PID_CRC16** - CRC16 ■ **TLV_PID_CRC16_RB** - CRC16 Reverse ■ **TLV_PID_WDT_A** - WDT_A ■ **TLV_PID_SFR** - SFR ■ **TLV_PID_SYS** - SYS ■ **TLV_PID_RAMCTL** - RAMCTL ■ **TLV_PID_DMA_1** - DMA 1 ■ **TLV_PID_DMA_3** - DMA 3 ■ **TLV_PID_UCS** - UCS ■ **TLV_PID_DMA_6** - DMA 6 ■ **TLV_PID_DMA_2** - DMA 2 ■ **TLV_PID_PORT1_2** - Port 1 + 2 / A ■ **TLV_PID_PORT3_4** - Port 3 + 4 / B ■ **TLV_PID_PORT5_6** - Port 5 + 6 / C ■ **TLV_PID_PORT7_8** - Port 7 + 8 / D ■ **TLV_PID_PORT9_10** - Port 9 + 10 / E ■ **TLV_PID_PORT11_12** - Port 11 + 12 / F ■ **TLV_PID_PORTU** - Port U ■ **TLV_PID_PORTJ** - Port J ■ **TLV_PID_TA2** - Timer A2 ■ **TLV_PID_TA3** - Timer A1 ■ **TLV_PID_TA5** - Timer A5 ■ **TLV_PID_TA7** - Timer A7 ■ **TLV_PID_TB3** - Timer B3 ■ **TLV_PID_TB5** - Timer B5 ■ **TLV_PID_TB7** - Timer B7 ■ **TLV_PID_RTC** - RTC ■ **TLV_PID_BT_RTC** - BT + RTC

Returns

The returned value is zero if the specified tag value (peripheral) is not available in the device.

References TLV_getInfo(), and TLV_getMemory().

27.3 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
            );
```

28 WatchDog Timer (WDT_A)

28.1 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.

28.2 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.

28.2.1 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()

28.2.2 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

baseAddress	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

baseAddress	is the base address of the WDT_A module.
clockSelect	is the clock source that the watchdog timer will use. Valid values are: ■ **WDT_A_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_A_CLOCKSOURCE_ACLK** ■ **WDT_A_CLOCKSOURCE_VLOCLK** ■ **WDT_A_CLOCKSOURCE_XCLK** Modified bits are **WDTSSEL** of **WDTCTL** register.
clockDivider	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: ■ **WDT_A_CLOCKDIVIDER_2G** ■ **WDT_A_CLOCKDIVIDER_128M** ■ **WDT_A_CLOCKDIVIDER_8192K** ■ **WDT_A_CLOCKDIVIDER_512K** ■ **WDT_A_CLOCKDIVIDER_32K** [Default] ■ **WDT_A_CLOCKDIVIDER_8192** ■ **WDT_A_CLOCKDIVIDER_512** ■ **WDT_A_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

void WDT_A_initWatchdogTimer (uint16_t *baseAddress,* uint8_t *clockSelect,* uint8_t *clockDivider*)

Sets the clock source for the Watchdog Timer in watchdog mode.

This function sets the watchdog timer in watchdog mode, which will cause a PUC when the timer overflows. When in the mode, a PUC can be avoided with a call to WDT_A_resetTimer() before the timer runs out.

Parameters

baseAddress	is the base address of the WDT_A module.
clockSelect	is the clock source that the watchdog timer will use. Valid values are: ■ **WDT_A_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_A_CLOCKSOURCE_ACLK** ■ **WDT_A_CLOCKSOURCE_VLOCLK** ■ **WDT_A_CLOCKSOURCE_XCLK** Modified bits are **WDTSSEL** of **WDTCTL** register.
clockDivider	is the divider of the clock source, in turn setting the watchdog timer interval. Valid values are: ■ **WDT_A_CLOCKDIVIDER_2G** ■ **WDT_A_CLOCKDIVIDER_128M** ■ **WDT_A_CLOCKDIVIDER_8192K** ■ **WDT_A_CLOCKDIVIDER_512K** ■ **WDT_A_CLOCKDIVIDER_32K** [Default] ■ **WDT_A_CLOCKDIVIDER_8192** ■ **WDT_A_CLOCKDIVIDER_512** ■ **WDT_A_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

void WDT_A_resetTimer (uint16_t *baseAddress*)

Resets the timer counter of the Watchdog Timer.

This function resets the watchdog timer to 0x0000h.

Parameters

baseAddress	is the base address of the WDT_A module.

Returns

None

void WDT_A_start (uint16_t *baseAddress*)

Starts the Watchdog Timer.

This function starts the watchdog timer functionality to start counting again.

Parameters

baseAddress	is the base address of the WDT_A module.

Returns

None

28.3 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();
```

29 Data Structure Documentation

29.1 Data Structures

Here are the data structures with brief descriptions:

29.2 Calendar Struct Reference

Used in the RTC_B_initCalendar() function as the CalendarTime parameter.

```
#include <rtc_b.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.

29.2.1 Detailed Description

Used in the RTC_B_initCalendar() function as the CalendarTime parameter.

The documentation for this struct was generated from the following file:

- rtc_b.h

29.3 Comp_D_initParam Struct Reference

Used in the Comp_D_init() function as the param parameter.

```
#include <comp_d.h>
```

Data Fields

- uint8_t positiveTerminalInput
- uint8_t negativeTerminalInput
- uint8_t outputFilterEnableAndDelayLevel
- uint16_t invertedOutputPolarity

29.3.1 Detailed Description

Used in the Comp_D_init() function as the param parameter.

29.3.2 Field Documentation

uint16_t Comp_D_initParam::invertedOutputPolarity

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

- **COMP_D_NORMALOUTPUTPOLARITY** [Default]
- **COMP_D_INVERTEDOUTPUTPOLARITY**

Referenced by Comp_D_init().

uint8_t Comp_D_initParam::negativeTerminalInput

Selects the input to the negative terminal.
Valid values are:

- **COMP_D_INPUT0** [Default]
- **COMP_D_INPUT1**
- **COMP_D_INPUT2**
- **COMP_D_INPUT3**
- **COMP_D_INPUT4**
- **COMP_D_INPUT5**
- **COMP_D_INPUT6**
- **COMP_D_INPUT7**
- **COMP_D_INPUT8**
- **COMP_D_INPUT9**
- **COMP_D_INPUT10**
- **COMP_D_INPUT11**
- **COMP_D_INPUT12**

- **COMP_D_INPUT13**
- **COMP_D_INPUT14**
- **COMP_D_INPUT15**
- **COMP_D_VREF**

Referenced by Comp_D_init().

uint8_t Comp_D_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_D_FILTEROUTPUT_OFF** [Default]
- **COMP_D_FILTEROUTPUT_DLYLVL1**
- **COMP_D_FILTEROUTPUT_DLYLVL2**
- **COMP_D_FILTEROUTPUT_DLYLVL3**
- **COMP_D_FILTEROUTPUT_DLYLVL4**

Referenced by Comp_D_init().

uint8_t Comp_D_initParam::positiveTerminalInput

Selects the input to the positive terminal.
Valid values are:

- **COMP_D_INPUT0** [Default]
- **COMP_D_INPUT1**
- **COMP_D_INPUT2**
- **COMP_D_INPUT3**
- **COMP_D_INPUT4**
- **COMP_D_INPUT5**
- **COMP_D_INPUT6**
- **COMP_D_INPUT7**
- **COMP_D_INPUT8**
- **COMP_D_INPUT9**
- **COMP_D_INPUT10**
- **COMP_D_INPUT11**
- **COMP_D_INPUT12**
- **COMP_D_INPUT13**
- **COMP_D_INPUT14**
- **COMP_D_INPUT15**
- **COMP_D_VREF**

Referenced by Comp_D_init().

The documentation for this struct was generated from the following file:

- comp_d.h

29.4 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

29.4.1 Detailed Description

Used in the DMA_init() function as the param parameter.

29.4.2 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

29.5 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.

29.5.1 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

29.6 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

29.6.1 Detailed Description

Used in the EUSCI_A_SPI_initMaster() function as the param parameter.

29.6.2 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

29.7 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

29.7.1 Detailed Description

Used in the EUSCI_A_SPI_initSlave() function as the param parameter.

29.7.2 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

29.8 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

29.8.1 Detailed Description

Used in the EUSCI_A_UART_init() function as the param parameter.

29.8.2 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 UCAxMCTLW.

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 UCBRSx 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

29.9 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

29.9.1 Detailed Description

Used in the EUSCI_B_I2C_initMaster() function as the param parameter.

29.9.2 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

29.10 EUSCI_B_I2C_initSlaveParam Struct Reference

Used in the EUSCI_B_I2C_initSlave() function as the param parameter.

`#include <eusci_b_i2c.h>`

Data Fields

- uint8_t slaveAddress
  7-bit slave address
- uint8_t slaveAddressOffset
- uint32_t slaveOwnAddressEnable

29.10.1 Detailed Description

Used in the EUSCI_B_I2C_initSlave() function as the param parameter.

29.10.2 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

29.11 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.

29.11.1 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

29.12 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

29.12.1 Detailed Description

Used in the EUSCI_B_SPI_initMaster() function as the param parameter.

29.12.2 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

29.13 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

29.13.1 Detailed Description

Used in the EUSCI_B_SPI_initSlave() function as the param parameter.

29.13.2 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

29.14 MPU_initThreeSegmentsParam Struct Reference

Used in the MPU_initThreeSegments() function as the param parameter.

```
#include <mpu.h>
```

Data Fields

- uint16_t seg1boundary

  Valid values can be found in the Family User's Guide.
- uint16_t seg2boundary

  Valid values can be found in the Family User's Guide.
- uint8_t seg1accmask
- uint8_t seg2accmask
- uint8_t seg3accmask

29.14.1 Detailed Description

Used in the MPU_initThreeSegments() function as the param parameter.

29.14.2 Field Documentation

uint8_t MPU_initThreeSegmentsParam::seg1accmask

Is the bit mask of access right for memory segment 1.
Logical OR of any of the following:

- **MPU_READ**
- **MPU_WRITE**
- **MPU_EXEC**
- **MPU_NO_READ_WRITE_EXEC**

Referenced by MPU_initThreeSegments().

uint8_t MPU_initThreeSegmentsParam::seg2accmask

Is the bit mask of access right for memory segment 2.
Logical OR of any of the following:

- **MPU_READ**
- **MPU_WRITE**
- **MPU_EXEC**
- **MPU_NO_READ_WRITE_EXEC**

Referenced by MPU_initThreeSegments().

uint8_t MPU_initThreeSegmentsParam::seg3accmask

Is the bit mask of access right for memory segment 3.
Logical OR of any of the following:

- **MPU_READ**
- **MPU_WRITE**
- **MPU_EXEC**
- **MPU_NO_READ_WRITE_EXEC**

Referenced by MPU_initThreeSegments().

The documentation for this struct was generated from the following file:

- mpu.h

29.15 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

29.15.1 Detailed Description

Used in the RTC_B_configureCalendarAlarm() function as the param parameter.

29.15.2 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

29.16 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

29.16.1 Detailed Description

Used in the Timer_A_initCaptureMode() function as the param parameter.

29.16.2 Field Documentation

uint16_t Timer_A_initCaptureModeParam::captureInputSelect

Decides the Input Select
Valid values are:

- **TIMER_A_CAPTURE_INPUTSELECT_CCIxA**
- **TIMER_A_CAPTURE_INPUTSELECT_CCIxB**
- **TIMER_A_CAPTURE_INPUTSELECT_GND**
- **TIMER_A_CAPTURE_INPUTSELECT_Vcc**

Referenced by Timer_A_initCaptureMode().

uint16_t Timer_A_initCaptureModeParam::captureInterruptEnable

Is to enable or disable timer captureComapre interrupt.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_A_initCaptureMode().

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

29.17 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.

29.17.1 Detailed Description

Used in the Timer_A_initCompareMode() function as the param parameter.

29.17.2 Field Documentation

uint16_t Timer_A_initCompareModeParam::compareInterruptEnable

Is to enable or disable timer captureComapre interrupt.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE** [Default]
- **TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE**

Referenced by Timer_A_initCompareMode().

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

29.18 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.

29.18.1 Detailed Description

Used in the Timer_A_initContinuousMode() function as the param parameter.

29.18.2 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

29.19 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.

29.19.1 Detailed Description

Used in the Timer_A_initUpDownMode() function as the param parameter.

29.19.2 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

29.20 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.

29.20.1 Detailed Description

Used in the Timer_A_initUpMode() function as the param parameter.

29.20.2 Field Documentation

uint16_t Timer_A_initUpModeParam::captureCompareInterruptEnable_CCR0_CCIE

Is to enable or disable Timer_A CCR0 captureComapre interrupt.
Valid values are:

- **TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE**
- **TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE** [Default]

Referenced by Timer_A_initUpMode().

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

29.21 Timer_A_outputPWMParam Struct Reference

Used in the Timer_A_outputPWM() function as the param parameter.

```
#include <timer_a.h>
```

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod

  Selects the desired timer period.
- uint16_t compareRegister
- uint16_t compareOutputMode
- uint16_t dutyCycle

  Specifies the dutycycle for the generated waveform.

29.21.1 Detailed Description

Used in the Timer_A_outputPWM() function as the param parameter.

29.21.2 Field Documentation

uint16_t Timer_A_outputPWMParam::clockSource

Selects Clock source.
Valid values are:

- **TIMER_A_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_A_CLOCKSOURCE_ACLK**
- **TIMER_A_CLOCKSOURCE_SMCLK**
- **TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_A_outputPWM().

uint16_t Timer_A_outputPWMParam::clockSourceDivider

Is the desired divider for the clock source
Valid values are:

- **TIMER_A_CLOCKSOURCE_DIVIDER_1** [Default]
- **TIMER_A_CLOCKSOURCE_DIVIDER_2**
- **TIMER_A_CLOCKSOURCE_DIVIDER_3**
- **TIMER_A_CLOCKSOURCE_DIVIDER_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_5**
- **TIMER_A_CLOCKSOURCE_DIVIDER_6**
- **TIMER_A_CLOCKSOURCE_DIVIDER_7**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**
- **TIMER_A_CLOCKSOURCE_DIVIDER_10**
- **TIMER_A_CLOCKSOURCE_DIVIDER_12**
- **TIMER_A_CLOCKSOURCE_DIVIDER_14**
- **TIMER_A_CLOCKSOURCE_DIVIDER_16**
- **TIMER_A_CLOCKSOURCE_DIVIDER_20**
- **TIMER_A_CLOCKSOURCE_DIVIDER_24**
- **TIMER_A_CLOCKSOURCE_DIVIDER_28**
- **TIMER_A_CLOCKSOURCE_DIVIDER_32**
- **TIMER_A_CLOCKSOURCE_DIVIDER_40**
- **TIMER_A_CLOCKSOURCE_DIVIDER_48**
- **TIMER_A_CLOCKSOURCE_DIVIDER_56**
- **TIMER_A_CLOCKSOURCE_DIVIDER_64**

Referenced by Timer_A_outputPWM().

uint16_t Timer_A_outputPWMParam::compareOutputMode

Specifies the output mode.
Valid values are:

- **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default]
- **TIMER_A_OUTPUTMODE_SET**
- **TIMER_A_OUTPUTMODE_TOGGLE_RESET**
- **TIMER_A_OUTPUTMODE_SET_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE**
- **TIMER_A_OUTPUTMODE_RESET**
- **TIMER_A_OUTPUTMODE_TOGGLE_SET**
- **TIMER_A_OUTPUTMODE_RESET_SET**

Referenced by Timer_A_outputPWM().

uint16_t Timer_A_outputPWMParam::compareRegister

Selects the compare register being used. Refer to datasheet to ensure the device has the capture compare register being used.
Valid values are:

- **TIMER_A_CAPTURECOMPARE_REGISTER_0**
- **TIMER_A_CAPTURECOMPARE_REGISTER_1**
- **TIMER_A_CAPTURECOMPARE_REGISTER_2**
- **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

29.22 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

29.22.1 Detailed Description

Used in the Timer_B_initCaptureMode() function as the param parameter.

29.22.2 Field Documentation

uint16_t Timer_B_initCaptureModeParam::captureInputSelect

Decides the Input Select
Valid values are:

- **TIMER_B_CAPTURE_INPUTSELECT_CCIxA** [Default]
- **TIMER_B_CAPTURE_INPUTSELECT_CCIxB**
- **TIMER_B_CAPTURE_INPUTSELECT_GND**
- **TIMER_B_CAPTURE_INPUTSELECT_Vcc**

Referenced by Timer_B_initCaptureMode().

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

29.23 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.

29.23.1 Detailed Description

Used in the Timer_B_initCompareMode() function as the param parameter.

29.23.2 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

29.24 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.

29.24.1 Detailed Description

Used in the Timer_B_initContinuousMode() function as the param parameter.

29.24.2 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

29.25 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.

29.25.1 Detailed Description

Used in the Timer_B_initUpDownMode() function as the param parameter.

29.25.2 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

29.26 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.

29.26.1 Detailed Description

Used in the Timer_B_initUpMode() function as the param parameter.

29.26.2 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

29.27 Timer_B_outputPWMParam Struct Reference

Used in the Timer_B_outputPWM() function as the param parameter.

```
#include <timer_b.h>
```

Data Fields

- uint16_t clockSource
- uint16_t clockSourceDivider
- uint16_t timerPeriod

  Selects the desired Timer_B period.
- uint16_t compareRegister
- uint16_t compareOutputMode
- uint16_t dutyCycle

  Specifies the dutycycle for the generated waveform.

29.27.1 Detailed Description

Used in the Timer_B_outputPWM() function as the param parameter.

29.27.2 Field Documentation

uint16_t Timer_B_outputPWMParam::clockSource

Selects the clock source
Valid values are:

- **TIMER_B_CLOCKSOURCE_EXTERNAL_TXCLK** [Default]
- **TIMER_B_CLOCKSOURCE_ACLK**
- **TIMER_B_CLOCKSOURCE_SMCLK**
- **TIMER_B_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK**

Referenced by Timer_B_outputPWM().

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

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless