MSP430 DriverLib for MSP430i2xx Devices

User's Guide

DOCNUM-2.80.00.01

Copyright

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

Revision Information

This is version 2.80.00.01 of this document, last updated on Fri Jul 29 2016 13:42:27.

Table of Contents

Copyright . . . 1
Revision Information . . . 1
1 Introduction . . . 3
2 Navigating to driverlib through CCS Resource Explorer . . . 4
2.1 Introduction . . . 4
3 How to create a new CCS project that uses Driverlib . . . 18
3.1 Introduction . . . 18
4 How to include driverlib into your existing CCS project . . . 20
4.1 Introduction . . . 20
5 How to create a new IAR project that uses Driverlib . . . 22
5.1 Introduction . . . 22
6 How to include driverlib into your existing IAR project . . . 25
6.1 Introduction . . . 25
7 Clock System (CS) . . . 28
7.1 Introduction . . . 28
7.2 API Functions . . . 28
7.3 Programming Example . . . 31
8 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART) . . . 33
8.1 Introduction . . . 33
8.2 API Functions . . . 33
8.3 Programming Example . . . 42
9 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI) . . . 44
9.1 Introduction . . . 44
9.2 Functions . . . 44
9.3 Programming Example . . . 53
10 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI) . . . 54
10.1 Introduction . . . 54
10.2 Functions . . . 54
10.3 Programming Example . . . 63
11 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C) . . . 64
11.1 Introduction . . . 64
11.2 Master Operations . . . 64
11.3 Slave Operations . . . 65
11.4 API Functions . . . 66
11.5 Programming Example . . . 86
12 FlashCtl - Flash Memory Controller . . . 87
12.1 Introduction . . . 87
12.2 API Functions . . . 87
12.3 Programming Example . . . 93
13 GPIO . . . 95
13.1 Introduction . . . 95
13.2 API Functions . . . 96
13.3 Programming Example . . . 123
14 16-Bit Hardware Multiplier (MPY) . . . 124
14.1 Introduction . . . 124

14.2 API Functions . . . 124
14.3 Programming Example . . . 127

15 Power Management Module (PMM) . . . 128
15.1 API Functions . . . 128
15.2 Programming Example . . . 132

16 24-Bit Sigma Delta Converter (SD24) . . . 133
16.1 Introduction . . . 133
16.2 API Functions . . . 133
16.3 Programming Example . . . 144
16.4 Programming Example . . . 144

17 Special Function Register (SFR) . . . 145
17.1 Introduction . . . 145
17.2 API Functions . . . 145
17.3 Programming Example . . . 147

18 16-Bit Timer_A (TIMER_A) . . . 149
18.1 Introduction . . . 149
18.2 API Functions . . . 150
18.3 Programming Example . . . 163

19 Tag Length Value (TLV) . . . 165
19.1 Introduction . . . 165
19.2 API Functions . . . 165
19.3 Programming Example . . . 167

20 WatchDog Timer (WDT) . . . 168
20.1 API Functions . . . 168
20.2 Programming Example . . . 171

21 Data Structure Documentation . . . 172
21.1 Data Structures . . . 172
21.2 EUSCI_A_SPI_changeMasterClockParam Struct Reference . . . 172
21.3 EUSCI_A_SPI_initMasterParam Struct Reference . . . 173
21.4 EUSCI_A_SPI_initSlaveParam Struct Reference . . . 175
21.5 EUSCI_A_UART_initParam Struct Reference . . . 176
21.6 EUSCI_B_I2C_initMasterParam Struct Reference . . . 178
21.7 EUSCI_B_I2C_initSlaveParam Struct Reference . . . 180
21.8 EUSCI_B_SPI_changeMasterClockParam Struct Reference . . . 181
21.9 EUSCI_B_SPI_initMasterParam Struct Reference . . . 181
21.10 EUSCI_B_SPI_initSlaveParam Struct Reference . . . 183
21.11 SD24_initConverterAdvancedParam Struct Reference . . . 184
21.12 Timer_A_initCaptureModeParam Struct Reference . . . 187
21.13 Timer_A_initCompareModeParam Struct Reference . . . 189
21.14 Timer_A_initContinuousModeParam Struct Reference . . . 190
21.15 Timer_A_initUpDownModeParam Struct Reference . . . 192
21.16 Timer_A_initUpModeParam Struct Reference . . . 193
21.17 Timer_A_outputPWMParam Struct Reference . . . 195

IMPORTANT NOTICE . . . 198

1 Introduction

The Texas Instruments® MSP430® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430i2xx 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.

Some driverlib APIs take 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.

The following tool chains are supported:

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

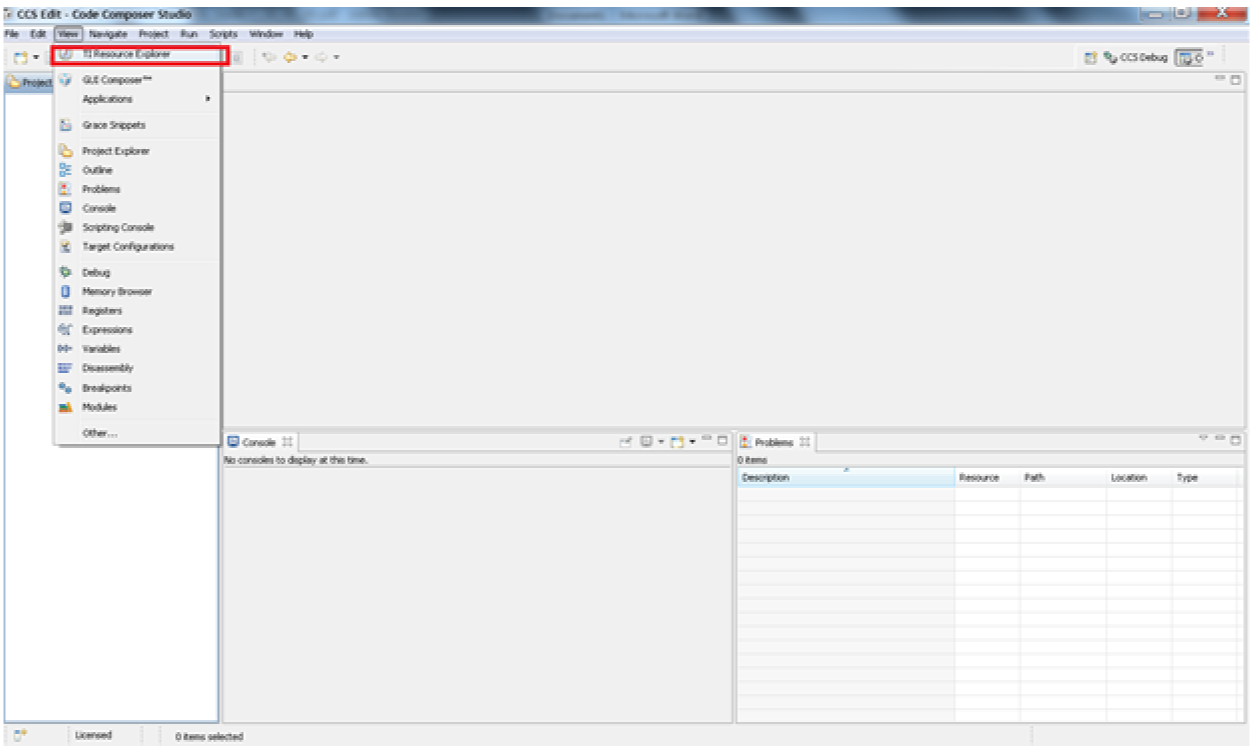
Using assert statements to debug

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

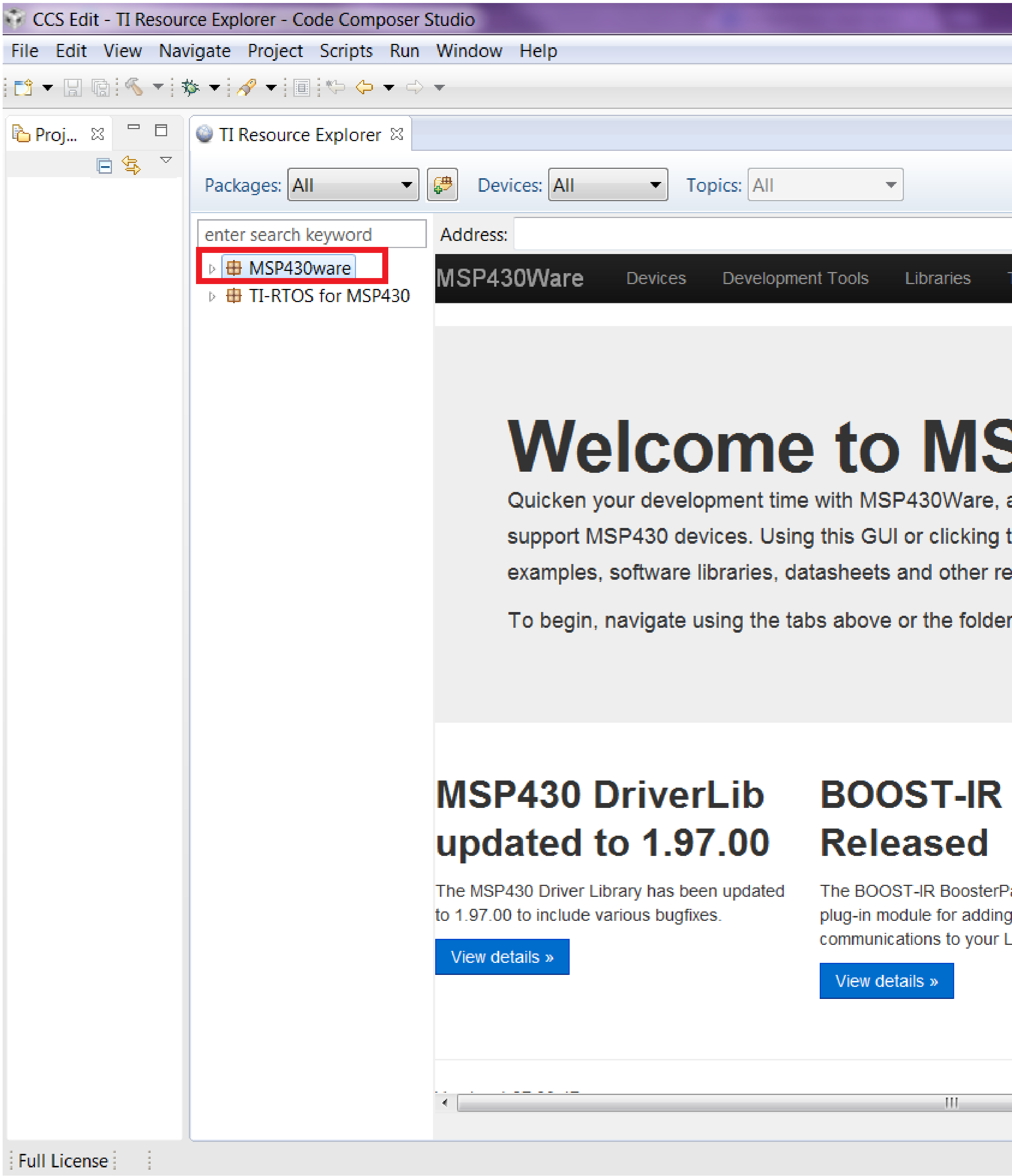
2 Navigating to driverlib through CCS Resource Explorer

2.1 Introduction

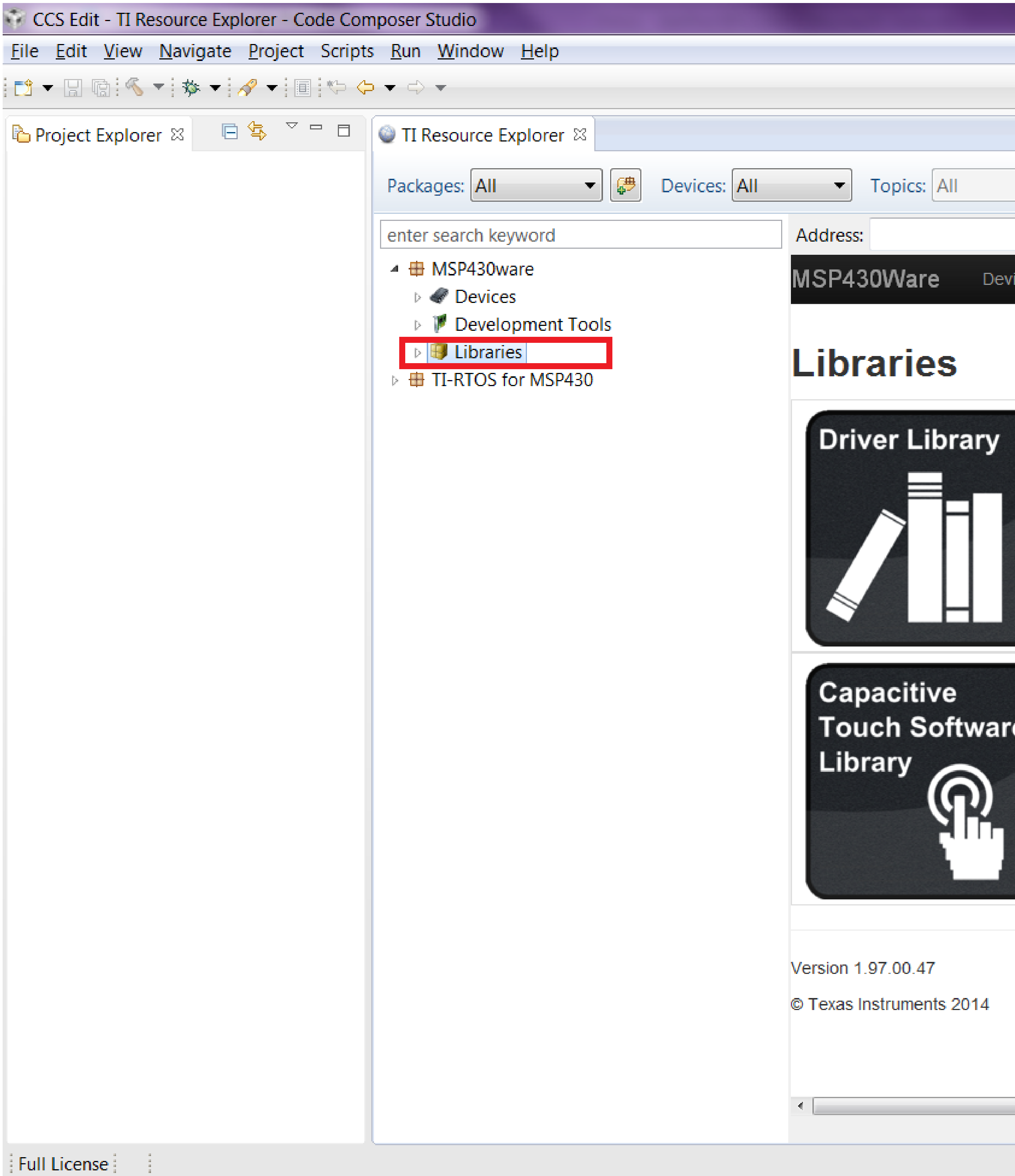
In CCS, click View->TI Resource Explorer

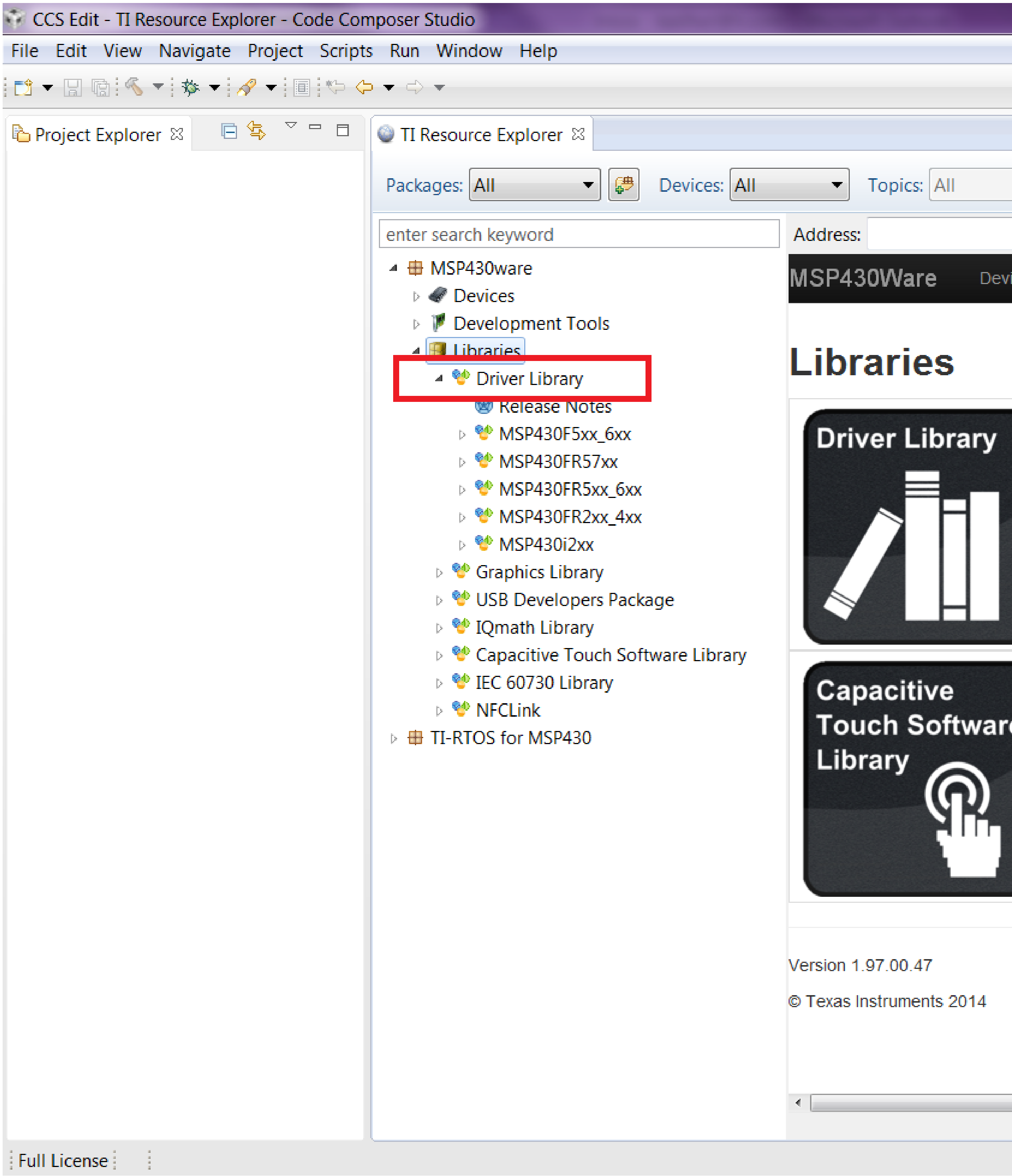


In Resource Explorer View, click on MSP430ware

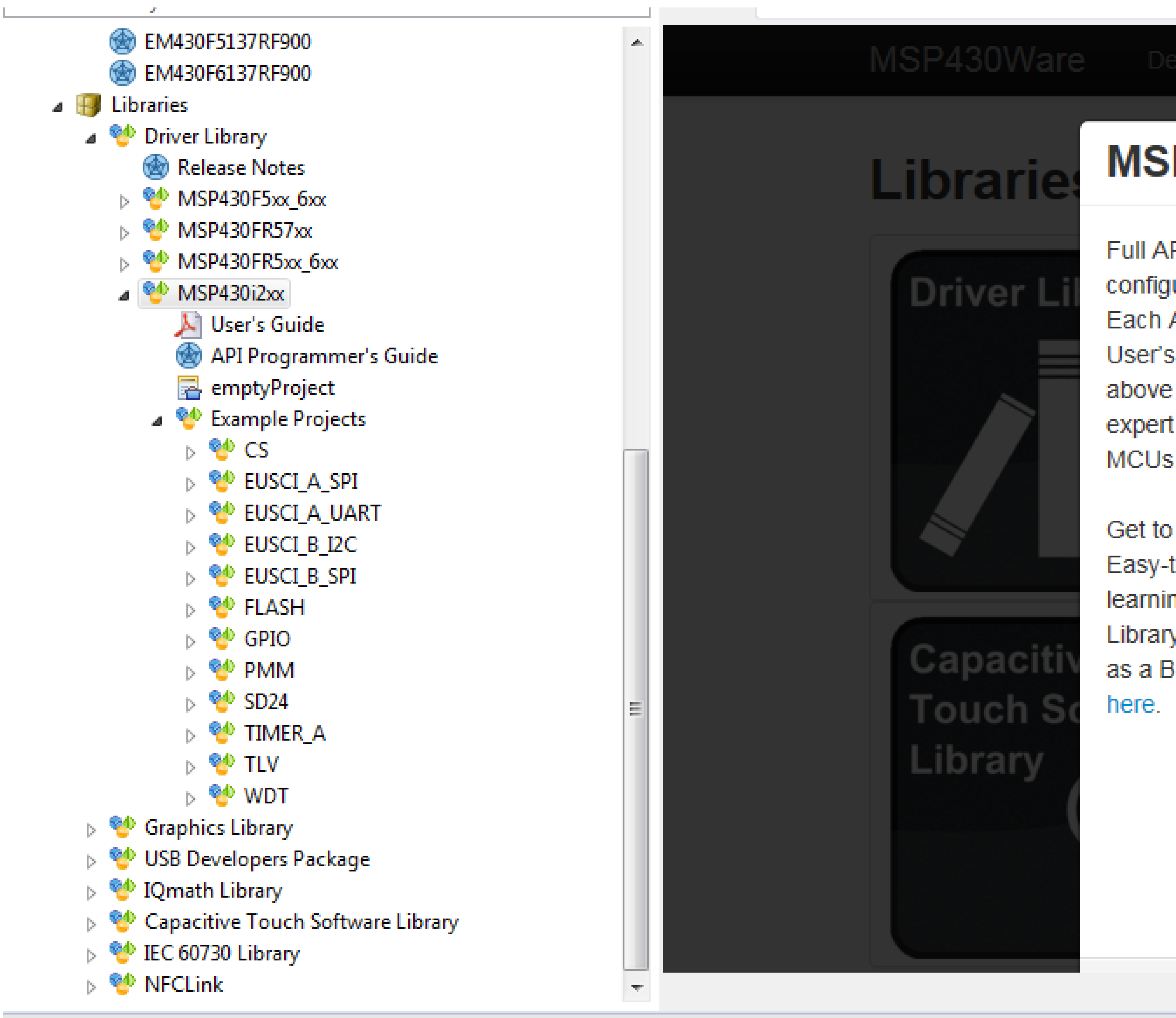


Clicking MSP430ware takes you to the introductory page. The version of the latest MSP430ware installed is available in this page. In this screenshot the version is 1.30.00.15 The various software, collateral, code examples, datasheets and user guides can be navigated by clicking the different topics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

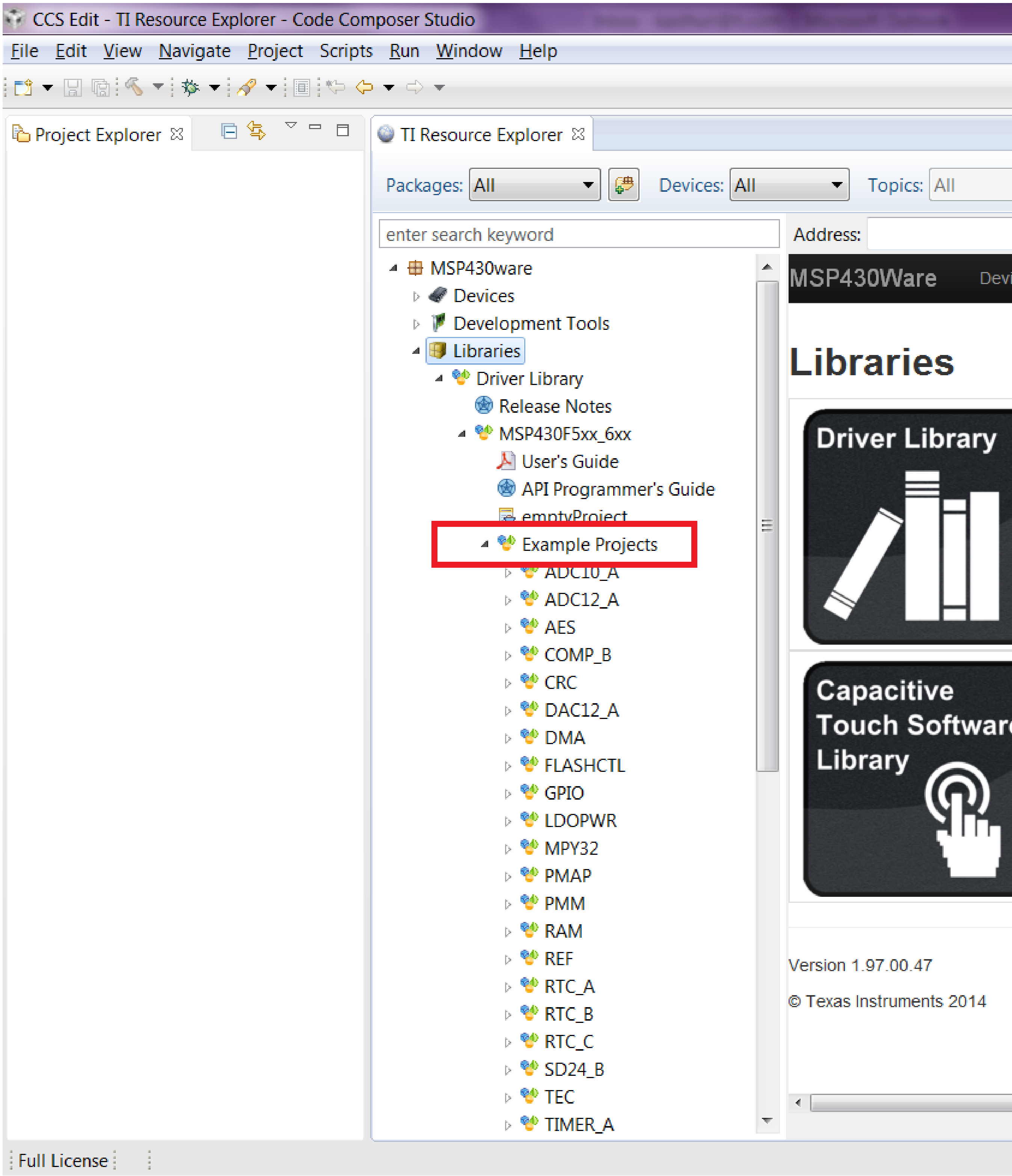

CCS Edit - TI Resource Explorer - Code Composer Studio
File Edit View Navigate Project Scripts Run Window Help
Project Explorer
TI Resource Explorer
Packages: All
Devices: All
Topics: All
enter search keyword
Address:
MSP430ware
Devices
Development Tools
Libraries
TI-RTOS for MSP430
MSP430Ware
Libraries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



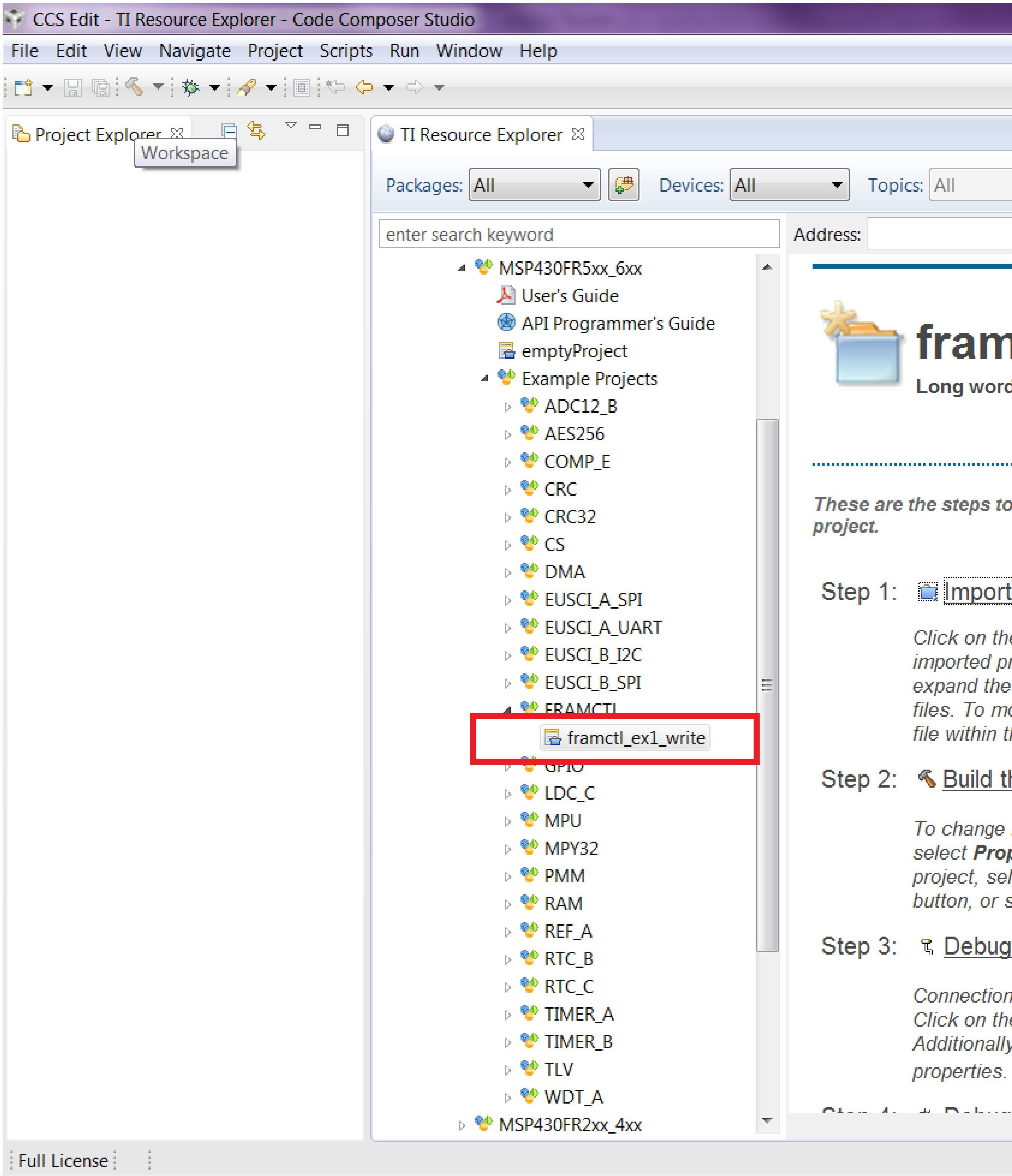
Driverlib is designed per Family. If a common device family user's guide exists for a group of devices, these devices belong to the same 'family'. Currently driverlib is available for the following family of devices. MSP430F5xx_6xx MSP430FR57xx MSP430FR2xx_4xx MSP430FR5xx_6xx MSP430i2xx



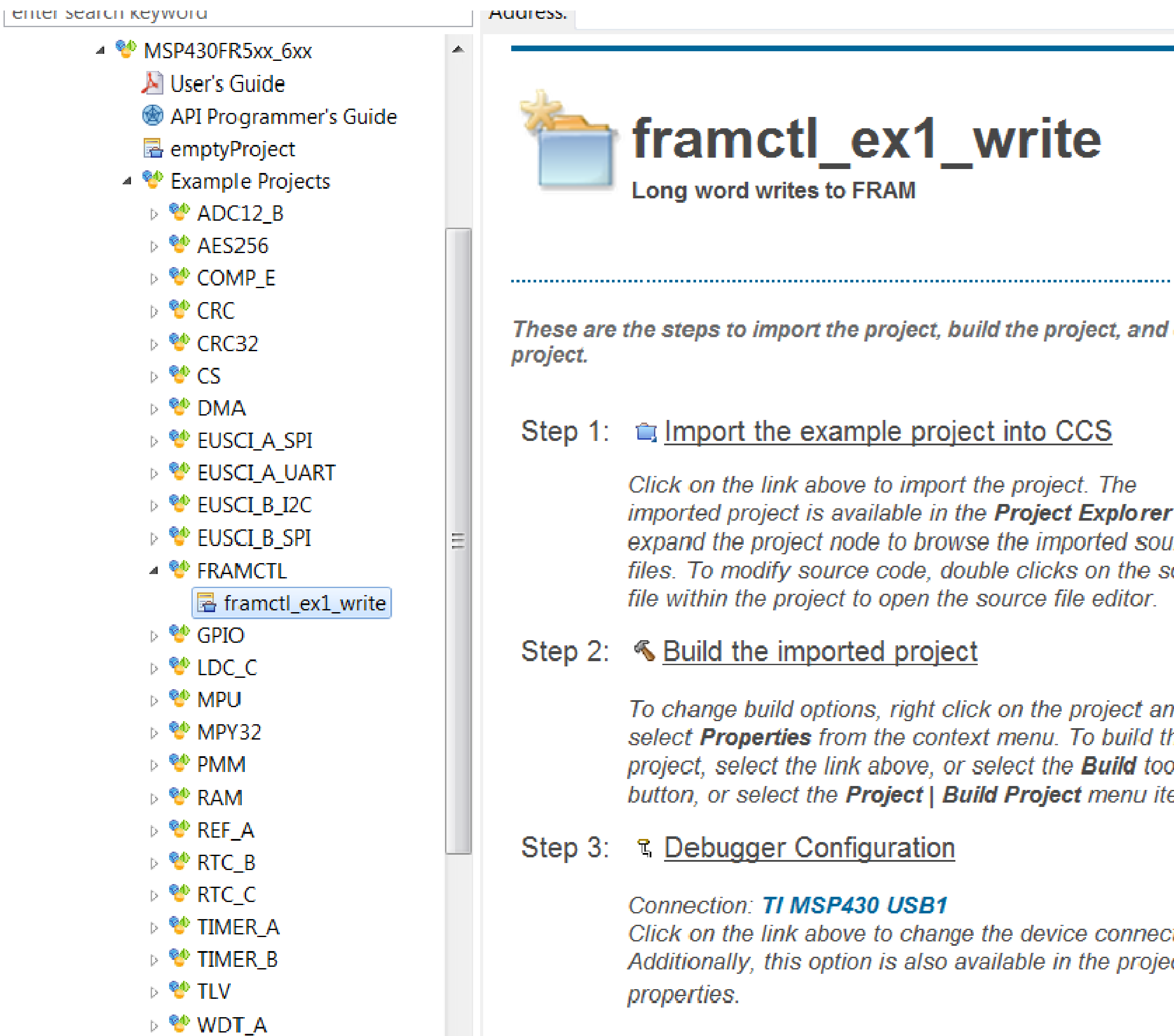
Click on the MSP430i2xx to navigate to the driverlib based example code for that family.



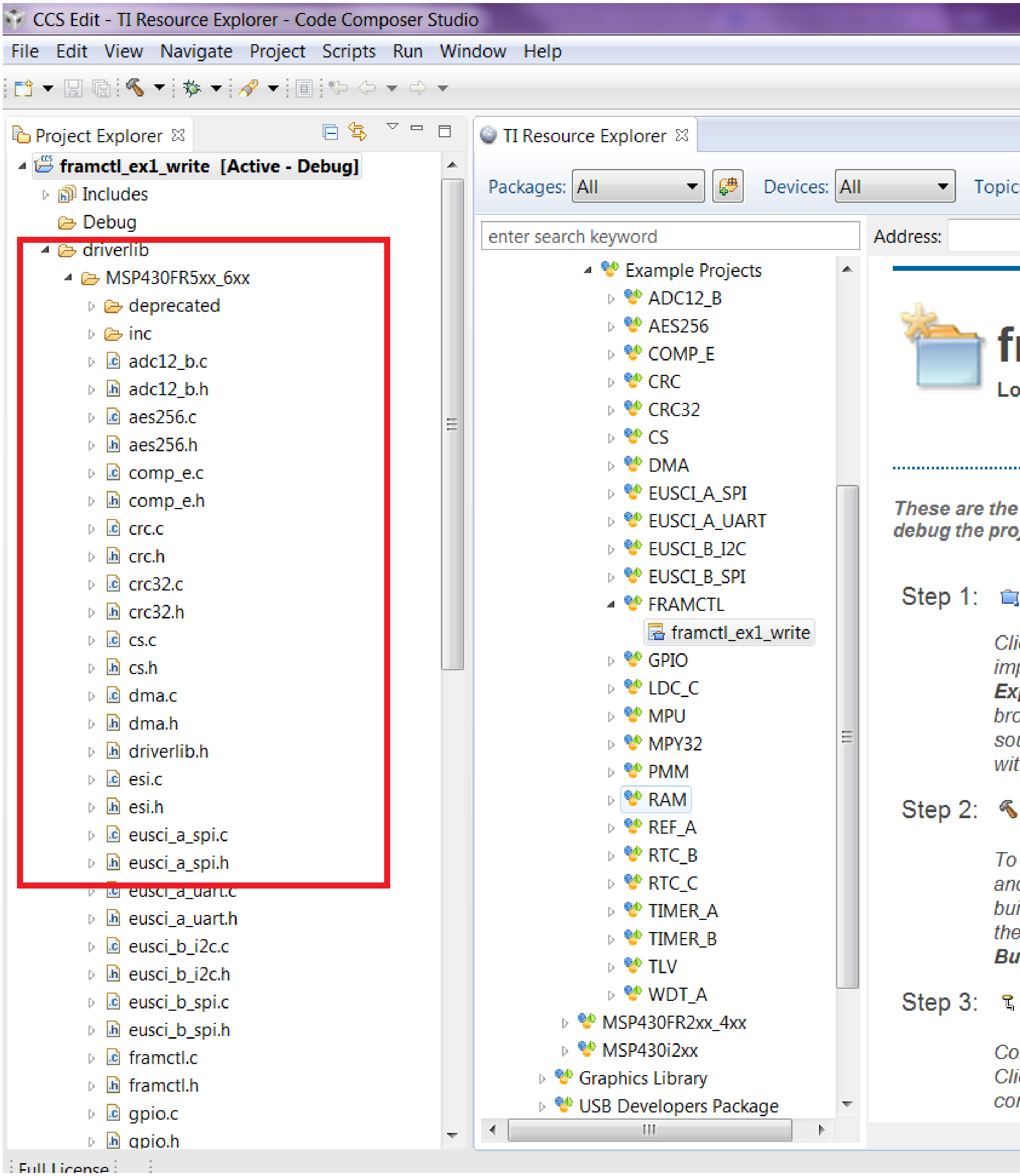
The various peripherals are listed in alphabetical order. The names of peripherals are as in device family user's guide. Clicking on a peripheral name lists the driverlib example code for that peripheral. The screenshot below shows an example when the user clicks on GPIO peripheral.



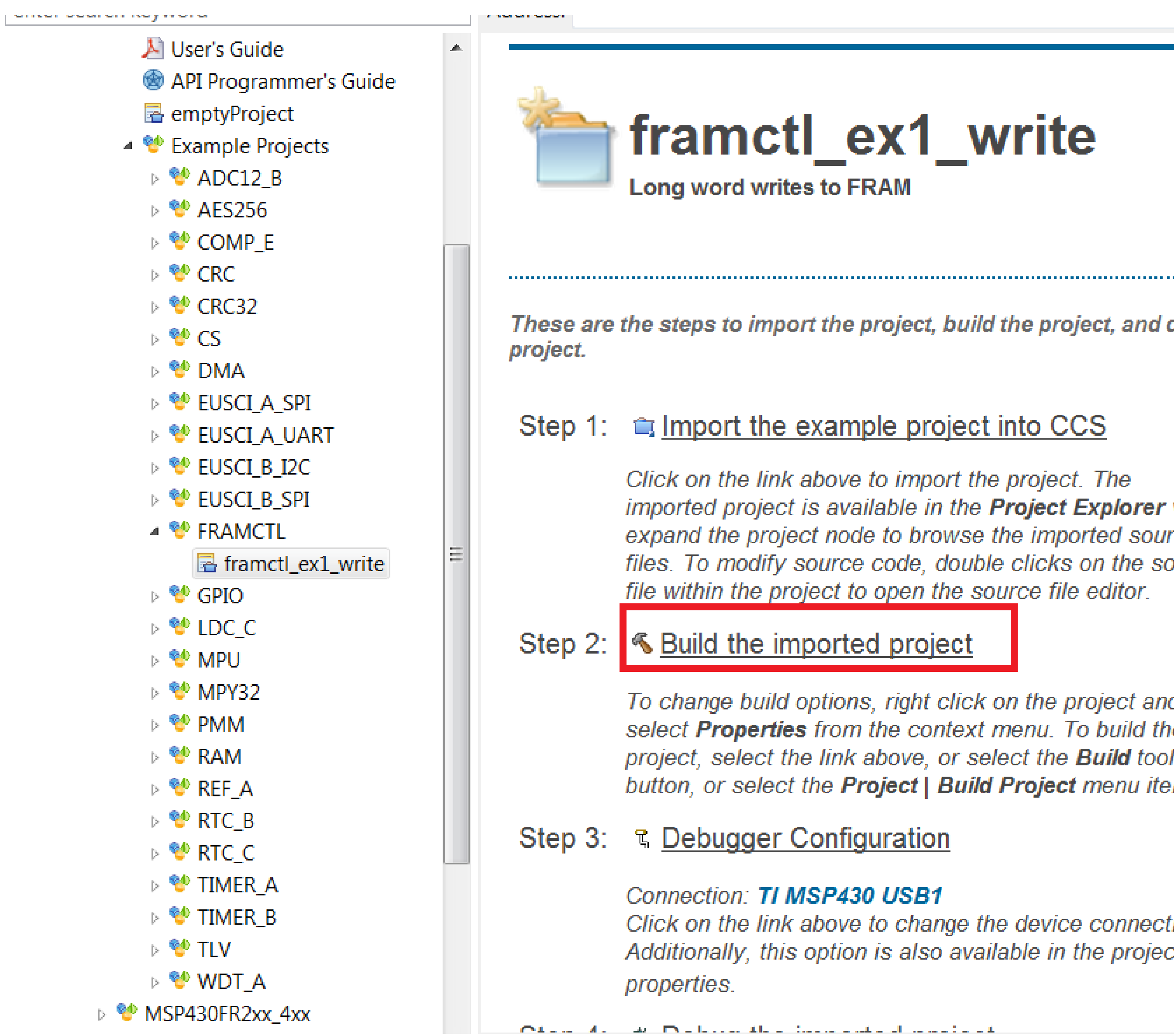
Now click on the specific example you are interested in. On the right side there are options to Import/Build/Download and Debug. Import the project by clicking on the ”Import the example project into CCS"



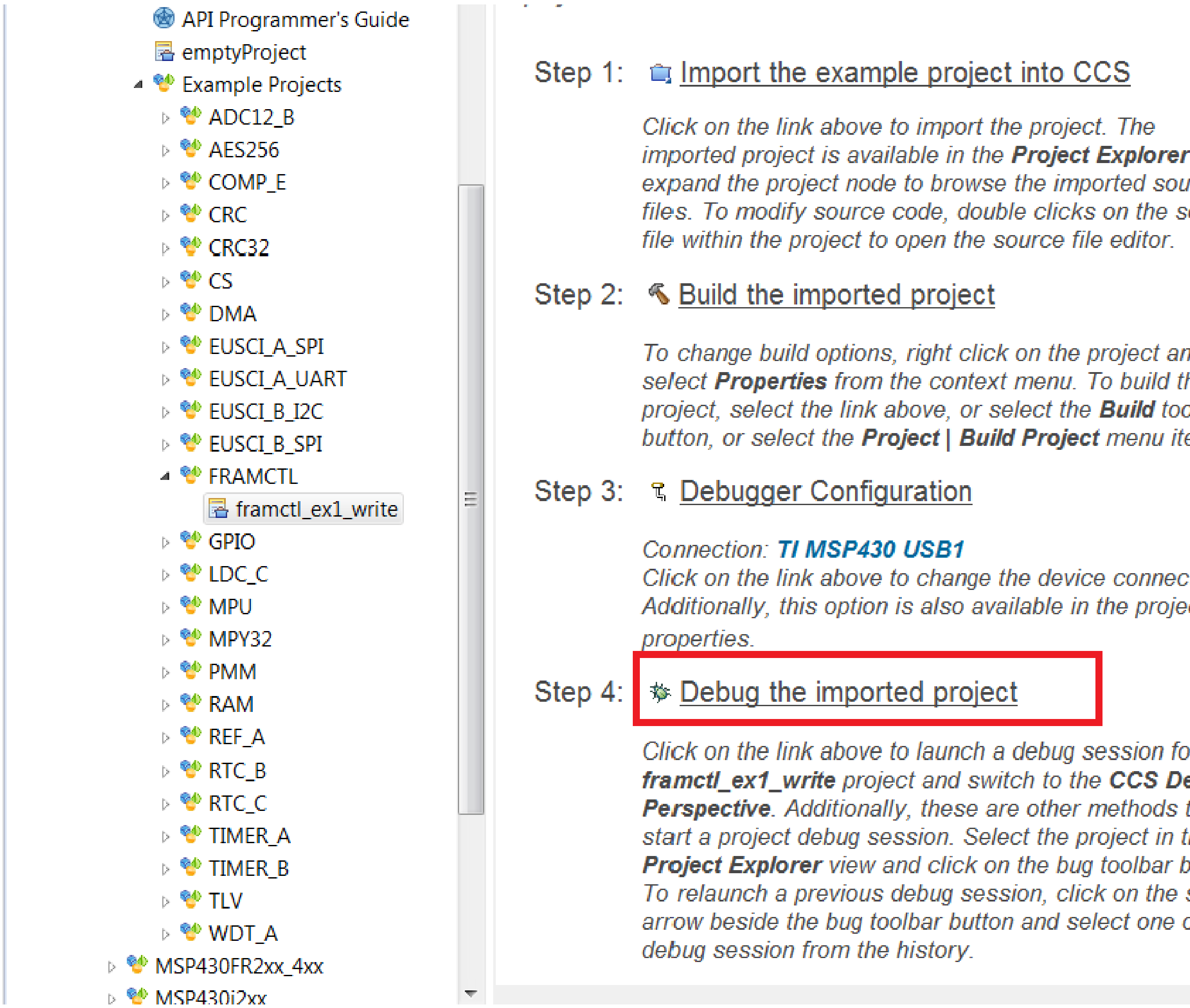
The imported project can be viewed on the left in the Project Explorer. All required driverlib source and header files are included inside the driverlib folder. All driverlib source and header files are linked to the example projects. So if the user modifies any of these source or header files, the original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



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

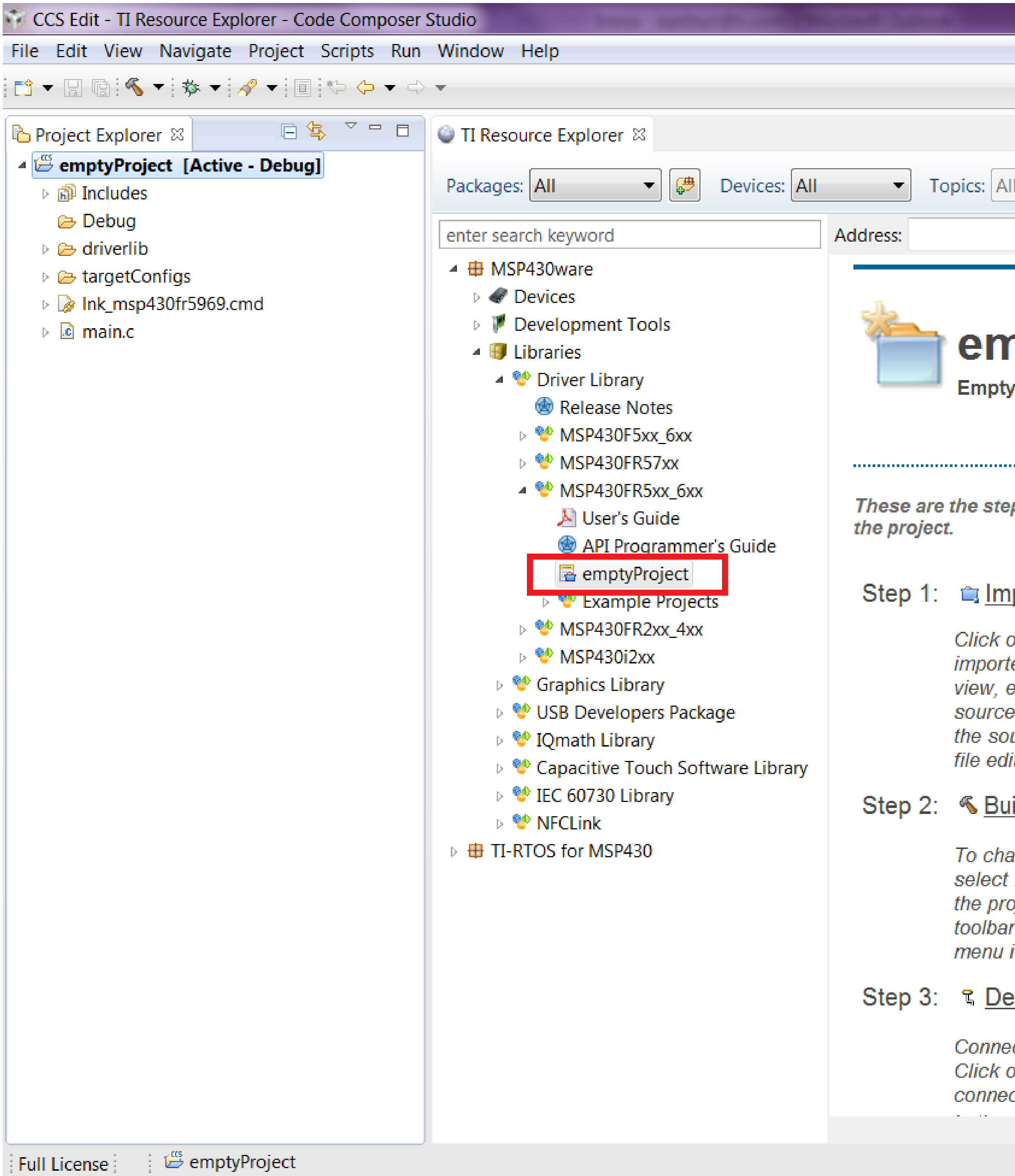


Now click on Build the imported project on the right to build the example project.



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

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

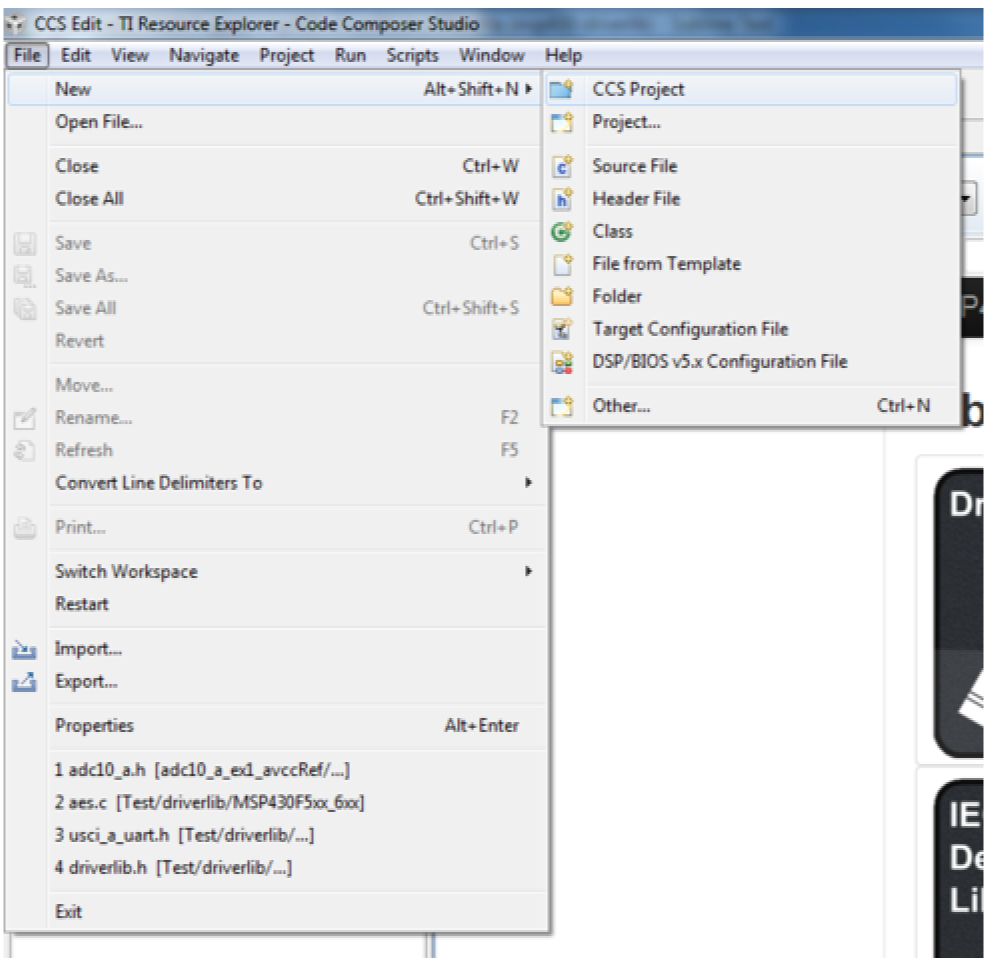


The main.c included with the empty project can be modified to include user code.

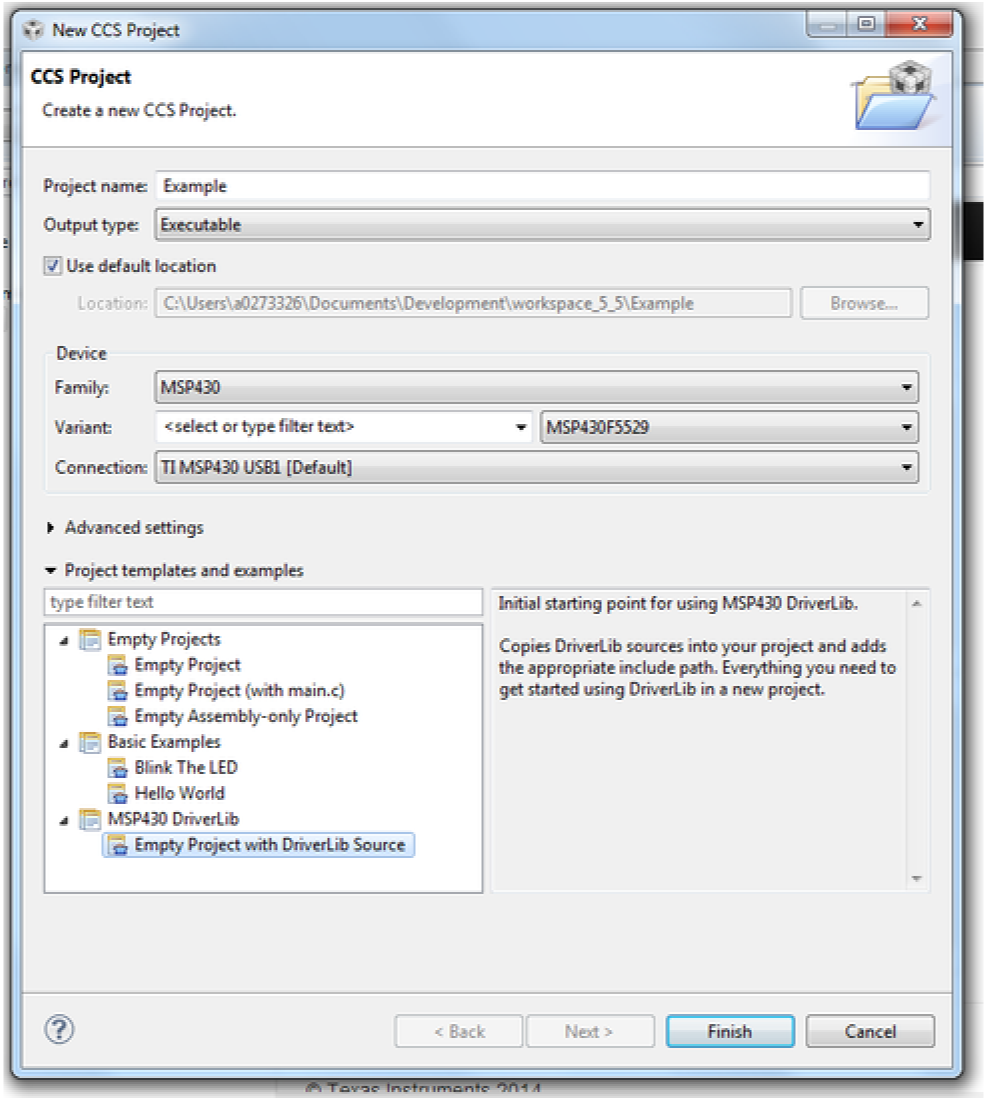
3 How to create a new CCS project that uses Driverlib

3.1 Introduction

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



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



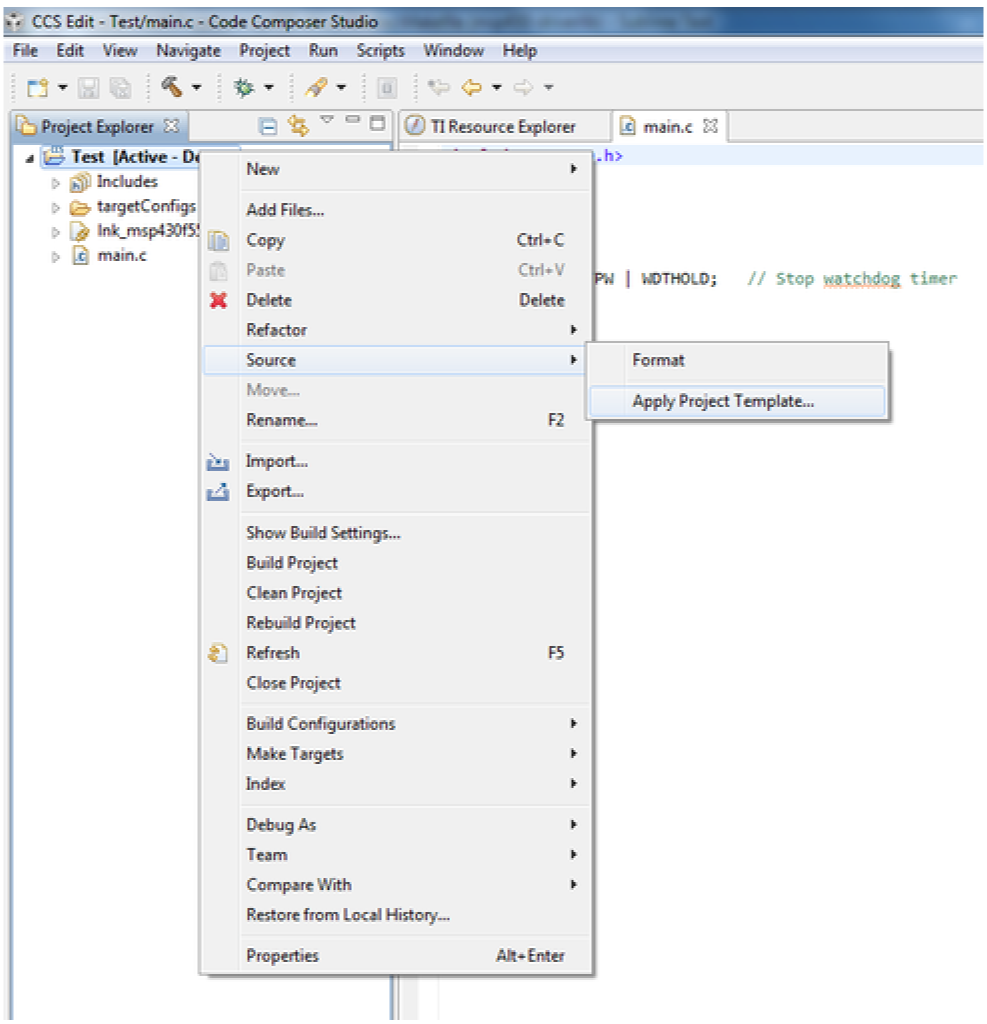
Finally click "Finish" and begin developing with your Driver Library enabled project.

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

4 How to include driverlib into your existing CCS project

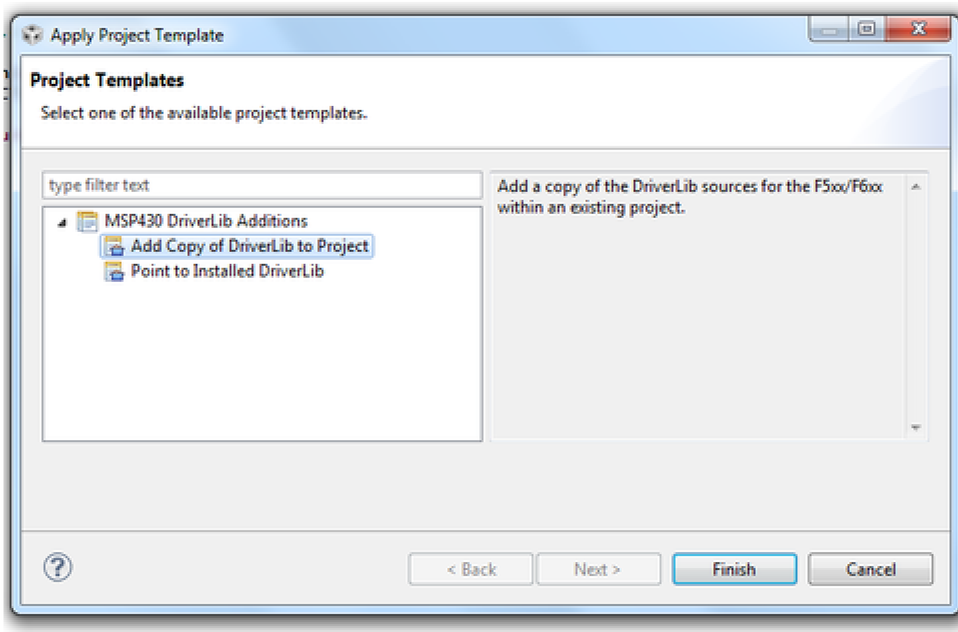
4.1 Introduction

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



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

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

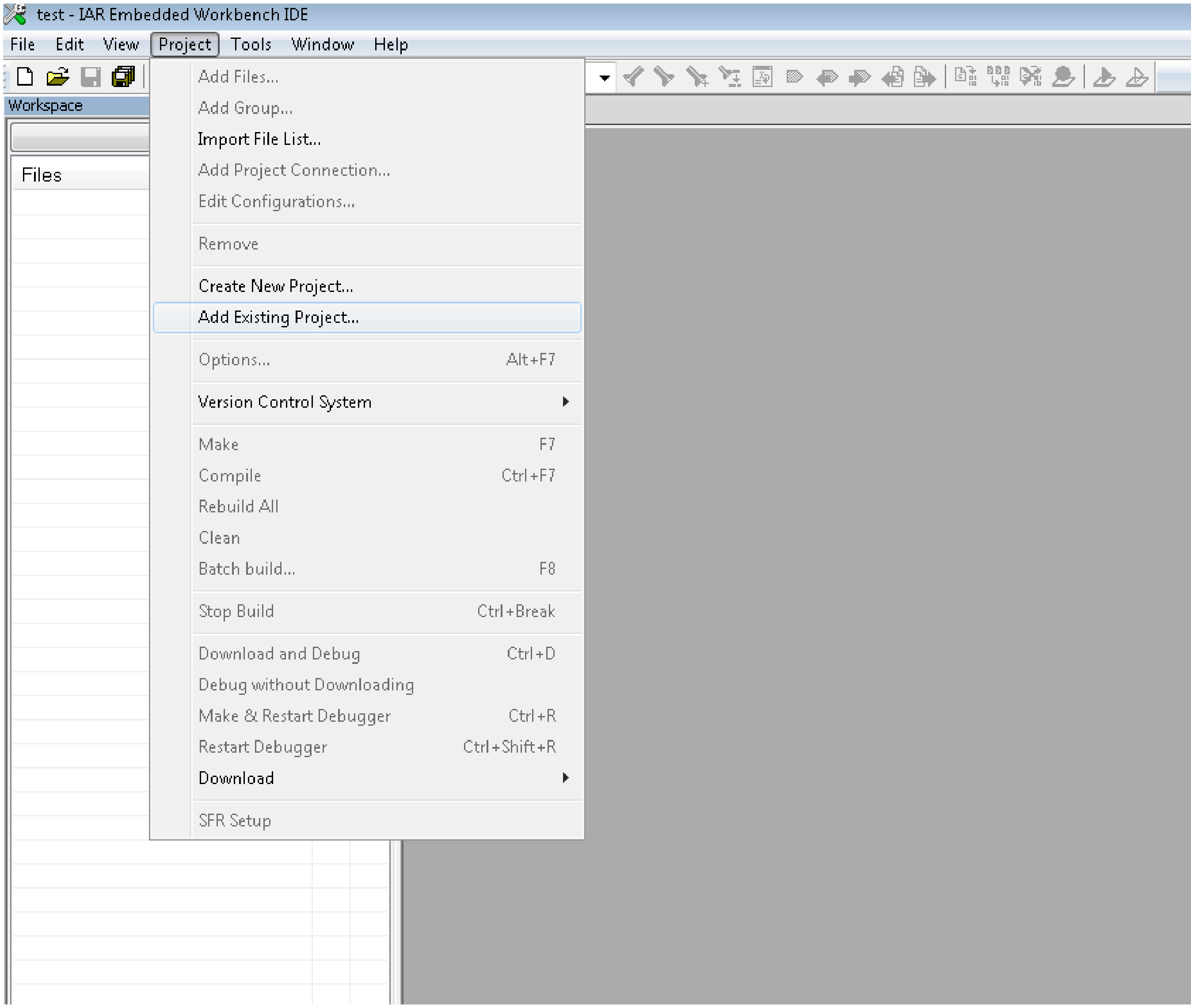


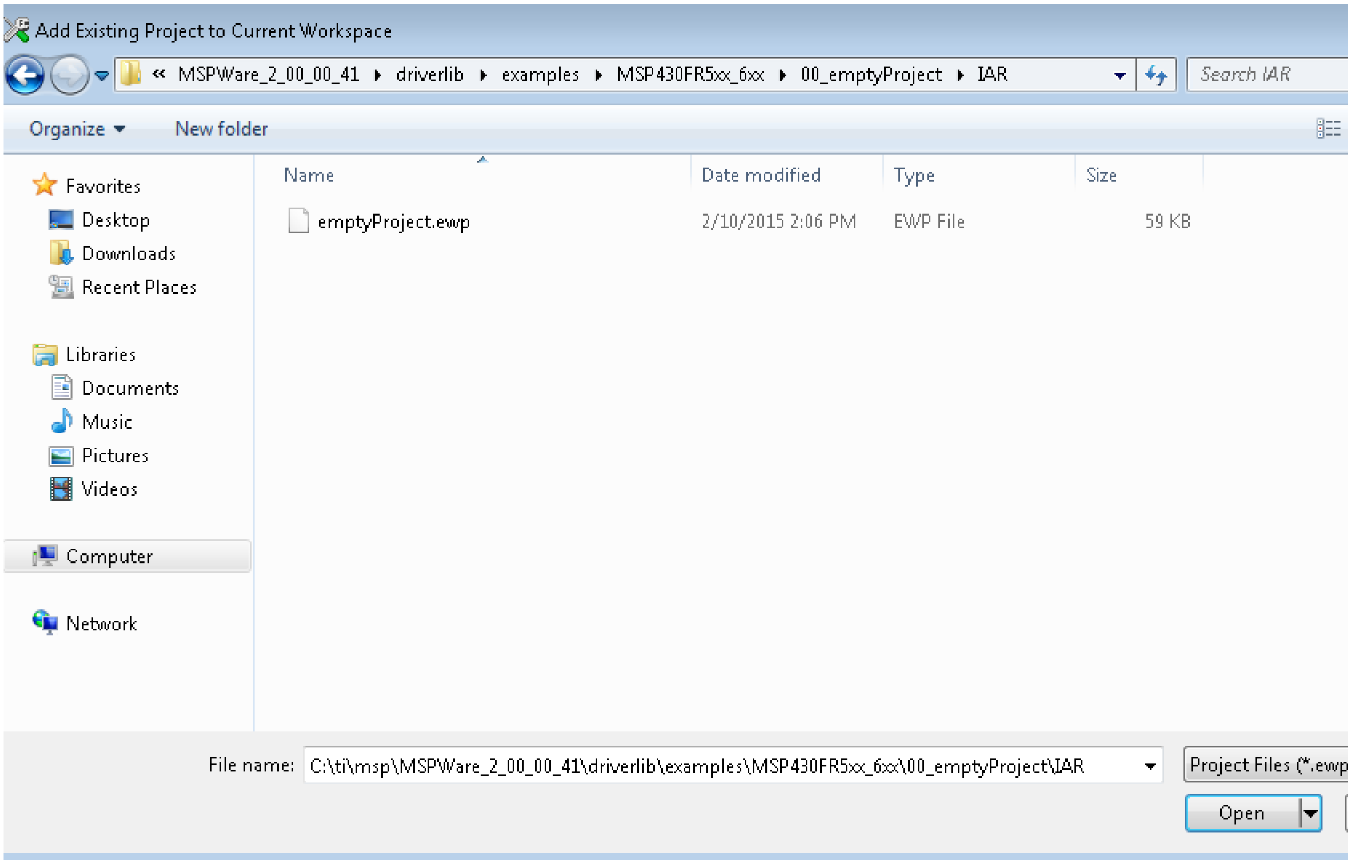
Click "Finish" and start developing with driver library in your project.

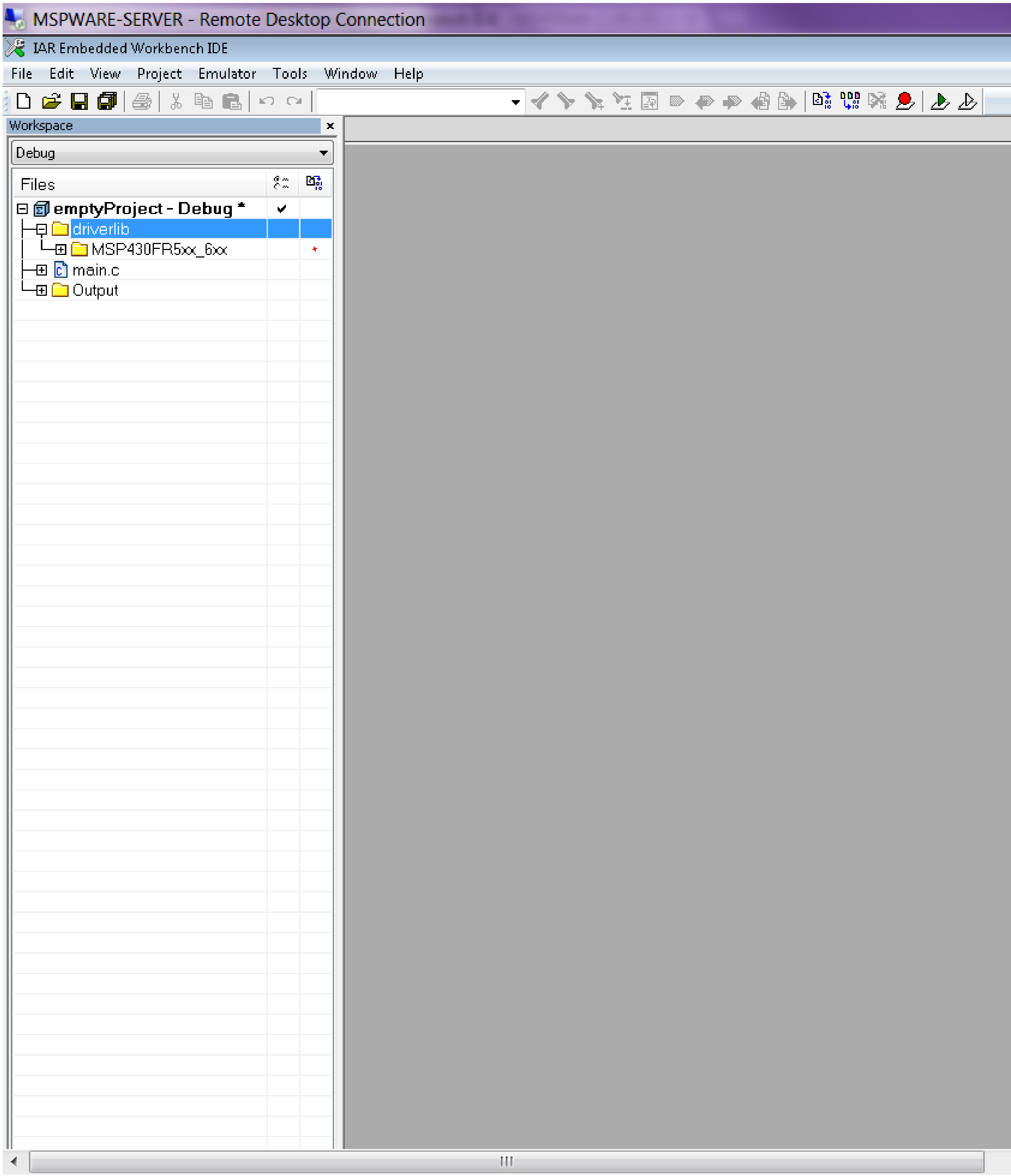
5 How to create a new IAR project that uses Driverlib

5.1 Introduction

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



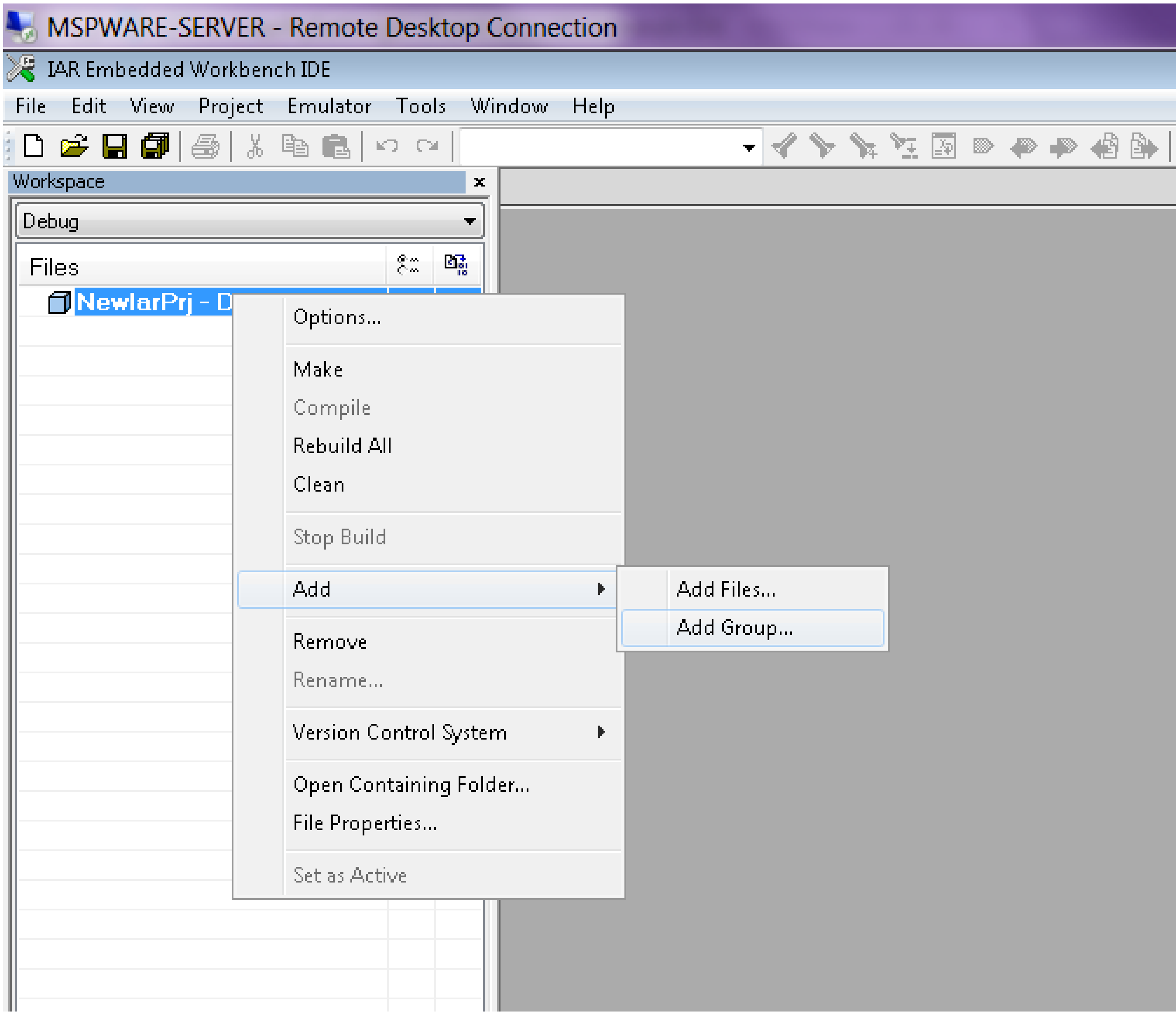

Add Existing Project to Current Workspace
« MSPWare_2_00_00_41 ▸ driverlib ▸ examples ▸ MSP430FR5xx_6xx ▸ 00_emptyProject ▸ IAR
Search IAR
Organize
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Network
Name
Date modified
Type
Size
emptyProject.ewp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


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

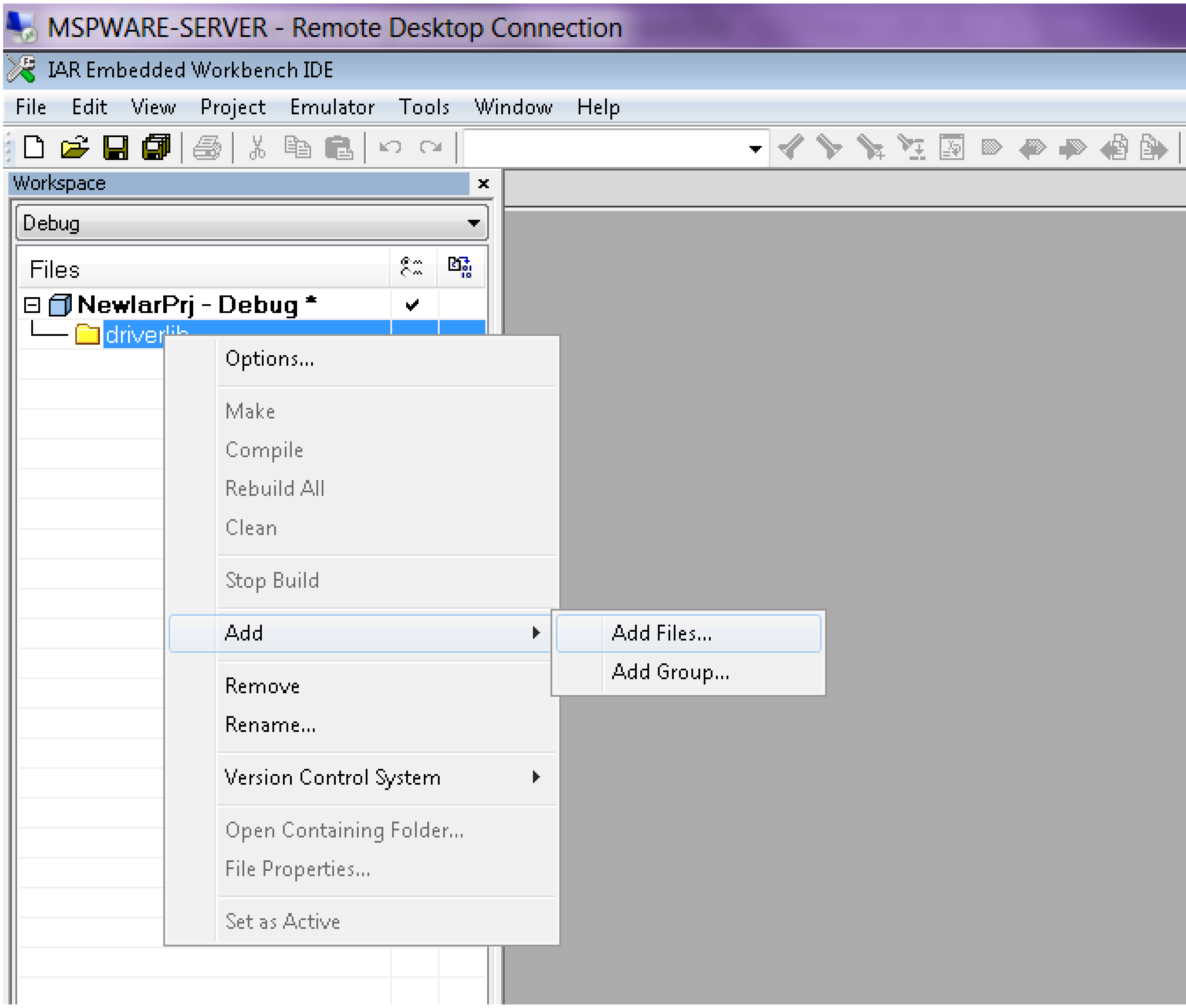
6 How to include driverlib into your existing IAR project

6.1 Introduction

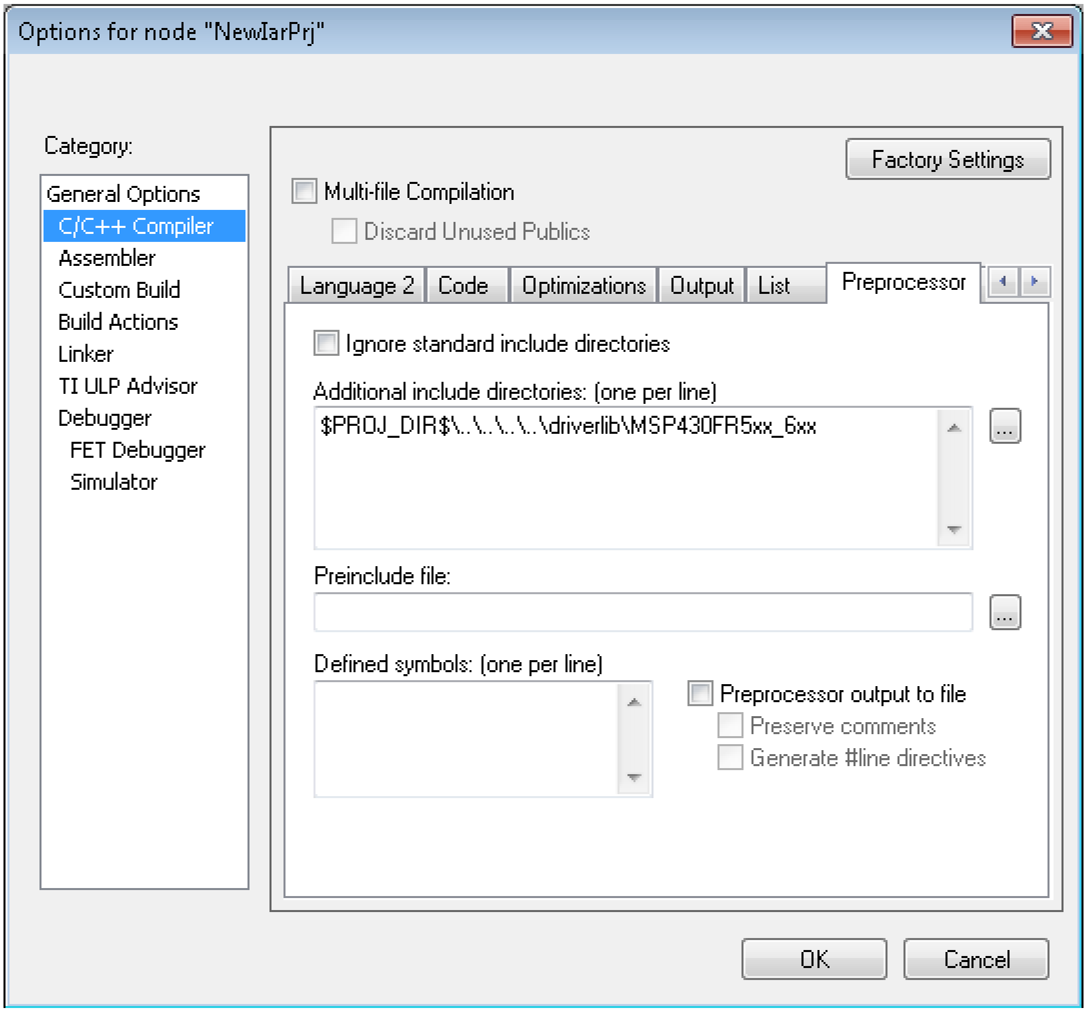
To add driver library to an existing project, right click project click on Add Group - ”driverlib”



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



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



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

7 Clock System (CS)

Introduction . . . 28
API Functions . . . 28
Programming Example . . . 31

7.1 Introduction

The clock system module supports low system cost and low power consumption. Using four 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 an external resistor or to bypass the DCO entirely.

Four system clock signals are available from the clock module:

- ACLK - Auxiliary clock. The ACLK is fixed at 32kHz when running using the DCO. If the device is set to DCO bypass mode ACLK runs at the bypass clock frequency / 512.
- MCLK - Master clock. MCLK can be divided by 1, 2, 4, 8 or 16. MCLK is used by the CPU and system.
- SMCLK - Subsystem master clock. SMCLK can be divided by 1, 2, 4, 8 or 16. SMCLK is software selectable by individual peripheral modules.
- SD24CLK - SD24 Clock provides a 1.024-MHz fixed-frequency clock to the Sigma-Delta ADC (SD24). The clock is only delivered when a clock request from SD24 is asserted. If SD24 functionality is needed in DCO bypass mode then the external clock frequency must be 16.384-MHz.

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

7.2 API Functions

Macros

- #define **CS_DCO_FREQ** 16384000

Functions

- void CS_setupDCO (uint8_t mode)
  Sets up the DCO using the selected mode.
- void CS_initClockSignal (uint8_t clockSource, uint8_t clockSourceDivider)
  Initializes a clock singal with a divider.
- uint32_t CS_getACLK (void)
  Get the current ACLK frequency in Hz.
- uint32_t CS_getSMCLK (void)
  Get the current SMCLK frequency in Hz.

- uint32_t CS_getMCLK (void)
  Get the current MCLK frequency in Hz.
- uint8_t CS_getFaultFlagStatus (uint8_t mask)
  Get the DCO fault flag status.

7.2.1 Detailed Description

The CS API is broken into three groups of functions: those that initializes the clock module, those that determine the clock speeds, and CS fault flag handling.

General CS configuration and initialization are handled by the following APIs:

- CS_setupDCO()
- CS_initClockSignal()

Determining clock speeds are handled by the following APIs:

- CS_getACLK()
- CS_getSMCLK()
- CS_getMCLK()

CS fault flags are handled by:

- CS_getFaultFlagStatus()

The CS_getMCLK, CS_getSMCLK or CS_getACLK APIs are only accurate when using the DCO with an internal or external resistor or the bypass clock is at 16.384MHz.

7.2.2 Function Documentation

uint32_t CS_getACLK (void)

Get the current ACLK frequency in Hz.

This API returns the current ACLK frequency in Hz. It does not work when the device is setup in DCO bypass mode. Also, CS_setupDCO() should be called before this API so that the DCO has been calibrated and this calculation is accurate.

Returns

Current ACLK frequency in Hz, 0 when in bypass mode

uint8_t CS_getFaultFlagStatus (uint8_t *mask*)

Get the DCO fault flag status.

Reads and returns DCO fault flag. The DCO fault flag is set when the DCO is operating in external resistor mode and the DCO detects an abnormality. An abnormality could be if the ROSC pin is left open or shorted to ground, or if the resistance connected at the ROSC pin is far away from the recommended value. If the fault persists the DCO automatically switches to the internal resistor mode as a fail-safe mechanism.

Parameters

mask	Mask of fault flags to check Mask value is the logical OR of any of the following: ■ **CS_DCO_FAULT_FLAG**

Returns

Logical OR of any of the following:

- **CS_DCO_FAULT_FLAG**
  indicating if the masked fault flags are set

uint32_t CS_getMCLK (void)

Get the current MCLK frequency in Hz.

This API returns the current MCLK frequency in Hz. It does not work when the device is setup in DCO bypass mode. Also, CS_setupDCO() should be called before this API so that the DCO has been calibrated and this calculation is accurate.

Returns

Current MCLK frequency in Hz, 0 when in bypass mode

uint32_t CS_getSMCLK (void)

Get the current SMCLK frequency in Hz.

This API returns the current SMCLK frequency in Hz. It does not work when the device is setup in DCO bypass mode. Also, CS_setupDCO() should be called before this API so that the DCO has been calibrated and this calculation is accurate.

Returns

Current SMCLK frequency in Hz, 0 when in bypass mode

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

Initializes a clock singal with a divider.

Sets up a clock signal with a divider. If the DCO is in bypass mode the frequency will be CLKIN / divider. If the DCO is not in bypass mode the frequency will 16.384MHz / divider.

Parameters

clockSource	Clock signal to initialize Valid values are: ■ **CS_MCLK** ■ **CS_SMCLK**
clockSource←Divider	Divider setting for the selected clock signal Valid values are: ■ **CS_CLOCK_DIVIDER_1** ■ **CS_CLOCK_DIVIDER_2** ■ **CS_CLOCK_DIVIDER_4** ■ **CS_CLOCK_DIVIDER_8** ■ **CS_CLOCK_DIVIDER_16**

Returns

None

void CS_setupDCO (uint8_t *mode*)

Sets up the DCO using the selected mode.

Sets up the DCO using the selected mode. If the bypass mode is selected than an external digital clock is required on the CLKIN pin to drive all clocks on the device. ACLK frequency is not programmable and is fixed to the bypass clock frequency divided by 512. For external resistor mode a 20kOhm resistor is recommended at the ROSC pin. External resistor mode offers higer clock accuracy in terms of absolute tolerance and temperature drift compared to the internal resistor mode. Please check your device datasheet for details and ratings for the differnt modes.

Parameters

mode	Mode to put the DCO into Valid values are: ■ **CS_INTERNAL_RESISTOR** - DCO operation with internal resistor ■ **CS_EXTERNAL_RESISTOR** - DCO operation with external resistor ■ **CS_BYPASS_MODE** - Bypass mode, provide external clock signal

Returns

None

7.3 Programming Example

The following example shows the configuration of the CS module that sets SMCLK = DCO / 2 and MCLK = DCO / 8.

```
// Set DCO Frequency to 16.384MHz
CS_setupDCO(CS_INTERNAL_RESISTOR);
```

```
// Configure MCLK and SMCLK
CS_initClockSignal(CS_MCLK, CS_CLOCK_DIVIDER_8);
CS_initClockSignal(CS_SMCLK, CS_CLOCK_DIVIDER_2);
```

8 EUSCI Universal Asynchronous Receiver/Transmitter (EUSCI_A_UART)

Introduction . . . 33
API Functions . . . 33
Programming Example . . . 42

8.1 Introduction

The MSP430i2xx Driver Library for EUSC_A_UART 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 eUSCI 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 eUSCI. The transmit and receive functions use the same baud-rate frequency.

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

8.2 API Functions

Functions

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

- void EUSCI_A_UART_disableInterrupt (uint16_t baseAddress, uint8_t mask)
  Disables individual UART interrupt sources.
- uint8_t EUSCI_A_UART_getInterruptStatus (uint16_t baseAddress, uint8_t mask)
  Gets the current UART interrupt status.
- void EUSCI_A_UART_clearInterrupt (uint16_t baseAddress, 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.

8.2.1 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.

Configuration and control of the EUSI_UART are handled by the

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

Sending and receiving data via the EUSI_UART is handled by the

- EUSCI_A_UART_transmitData()

- EUSCI_A_UART_receiveData()
- EUSCI_A_UART_transmitAddress()
- EUSCI_A_UART_transmitBreak()

Managing the EUSI_UART interrupts and status are handled by the

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

8.2.2 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/MSP430BaudRateConverter/index.html`

Parameters

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

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

Returns

STATUS_SUCCESS or STATUS_FAIL of the initialization process

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

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

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

Parameters

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

Modified bits of **UCAxTXBUF** register.

Returns

None

8.3 Programming Example

The following example shows how to use the EUSI_A_UART API to initialize the EUSI_A_UART and start transmiting characters.

```
// Configuration for 115200 UART with SMCLK at 16384000
// These values were generated using the online tool available at:
// http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.html
EUSCI_A_UART_initParam uartConfig = {
    EUSCI_A_UART_CLOCKSOURCE_SMCLK,          // SMCLK Clock Source
    8,                                        // BRDIV = 8
    14,                                       // UCxBRF = 14
    34,                                       // UCxBRS = 34
    EUSCI_A_UART_NO_PARITY,                  // No Parity
    EUSCI_A_UART_MSB_FIRST,                  // MSB First
    EUSCI_A_UART_ONE_STOP_BIT,               // One stop bit
    EUSCI_A_UART_MODE,                       // UART mode
    EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION  // Oversampling Baudrate
};

WDT_hold(WDT_BASE);

// Setting the DCO to use the internal resistor. DCO will be at 16.384MHz
CS_setupDCO(CS_INTERNAL_RESISTOR);

// SMCLK should be same speed as DCO. SMCLK = 16.384MHz
CS_initClockSignal(CS_SMCLK, CS_CLOCK_DIVIDER_1);

// Settings P1.2 and P1.3 as UART pins. P1.4 as LED output
GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
                                           GPIO_PIN2 | GPIO_PIN3,
                                           GPIO_PRIMARY_MODULE_FUNCTION);
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN4);
GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN4);

// Configure and enable the UART peripheral
EUSCI_A_UART_init(EUSCI_A0_BASE, &uartConfig);
EUSCI_A_UART_enable(EUSCI_A0_BASE);

EUSCI_A_UART_enableInterrupt(EUSCI_A0_BASE,
                             EUSCI_A_UART_RECEIVE_INTERRUPT);

while(1) {
    EUSCI_A_UART_transmitData(EUSCI_A0_BASE, TXData);

    // Go to sleep and wait for LPM exit
    __bis_SR_register(LPM0_bits | GIE);
}
```

9 EUSCI Synchronous Peripheral Interface (EUSCI_A_SPI)

Introduction 44
API Functions 44
Programming Example 53

9.1 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_A.

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

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

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

9.2 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.

9.2.1 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_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()

9.2.2 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.
select4PinFunctionality	selects 4 pin functionality Valid values are: ■ **EUSCI_A_SPI_PREVENT_CONFLICTS_WITH_OTHER_MASTERS** ■ **EUSCI_A_SPI_ENABLE_SIGNAL_FOR_4WIRE_SLAVE**

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

Returns

None

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

9.3 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.

```
EUSCI_A_SPI_initMasterParam spiMasterConfig = {
    EUSCI_A_SPI_CLOCKSOURCE_ACLK,              // ACLK Clock Source
    32000,                                      // ACLK = 32kHz
    16000,                                      // SPICLK = 16kHz
    EUSCI_A_SPI_MSB_FIRST,                     // MSB First
    EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,    // Phase
    EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_HIGH, // High polarity
    EUSCI_A_SPI_3PIN                           // 3Wire SPI Mode
};

WDT_hold(WDT_BASE);

// Setting P1.1, P1.2 and P1.3 as SPI pins.
GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
                                           GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3,
                                           GPIO_PRIMARY_MODULE_FUNCTION);

// Setting P1.4 as LED Pin
GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN4);
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN4);

// Setting the DCO to use the internal resistor. DCO will be at 16.384MHz
// ACLK is at 32kHz
CS_setupDCO(CS_INTERNAL_RESISTOR);

// Configure and enable the SPI peripheral
EUSCI_A_SPI_initMaster(EUSCI_A0_BASE, &spiMasterConfig);
EUSCI_A_SPI_enable(EUSCI_A0_BASE);

// Put the first byte in the transfer buffer
EUSCI_A_SPI_transmitData(EUSCI_A0_BASE, TXData);

EUSCI_A_SPI_enableInterrupt(EUSCI_A0_BASE, EUSCI_A_SPI_RECEIVE_INTERRUPT);

// Go into LPM0 with interrupts enabled
__bis_SR_register(LPM0_bits | GIE);
```

10 EUSCI Synchronous Peripheral Interface (EUSCI_B_SPI)

Introduction 54
API Functions 54
Programming Example 63

10.1 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_B.

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

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

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

10.2 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.

10.2.1 Detailed Description

To use the module as a master, the user must call EUSCI_B_SPI_initMaster() 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_initSlave() 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_initMaster()
- EUSCI_B_SPI_initSlave()
- EUSCI_B_SPI_disable()
- EUSCI_B_SPI_enable()
- 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()

10.2.2 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

10.3 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.

```
EUSCI_B_SPI_initMasterParam spiMasterConfig = {
    EUSCI_B_SPI_CLOCKSOURCE_ACLK,              // ACLK Clock Source
    32000,                                      // ACLK = 32kHz
    16000,                                      // SPICLK = 16kHz
    EUSCI_B_SPI_MSB_FIRST,                     // MSB First
    EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,    // Phase
    EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH, // High polarity
    EUSCI_B_SPI_3PIN                           // 3Wire SPI Mode
};

WDT_hold(WDT_BASE);

// Setting P1.1, P1.2 and P1.3 as SPI pins.
GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
                                           GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3,
                                           GPIO_PRIMARY_MODULE_FUNCTION);

// Setting P1.4 as LED Pin
GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN4);
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN4);

// Setting the DCO to use the internal resistor. DCO will be at 16.384MHz
// ACLK is at 32kHz
CS_setupDCO(CS_INTERNAL_RESISTOR);

// Configure and enable the SPI peripheral
EUSCI_B_SPI_initMaster(EUSCI_B0_BASE, &spiMasterConfig);
EUSCI_B_SPI_enable(EUSCI_B0_BASE);

// Put the first byte in the transfer buffer
EUSCI_B_SPI_transmitData(EUSCI_B0_BASE, TXData);

EUSCI_B_SPI_enableInterrupt(EUSCI_B0_BASE, EUSCI_B_SPI_RECEIVE_INTERRUPT);

// Go into LPM0 with interrupts enabled
__bis_SR_register(LPM0_bits | GIE);
```

11 EUSCI Inter-Integrated Circuit (EUSCI_B_I2C)

Introduction 64
API Functions 66
Programming Example 86

11.1 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 MSP430i2xx eUSCI_B_I2C module. 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 driver library EUSCI_B_I2C module supports both sending and receiving data as either a master or a slave, and also supports 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.

11.2 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_masterReceiveMultiByteStart()
- 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.

11.3 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() function.

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.

This driver is contained in `eusci_b_i2c.c`, with `eusci_b_i2c.h` containing the API definitions for use by applications.

11.4 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.

11.4.1 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_clearInterruptFlag()
- 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_slaveInit()
- EUSCI_B_I2C_interruptStatus()
- EUSCI_B_I2C_setSlaveAddress()
- EUSCI_B_I2C_setMode()
- EUSCI_B_I2C_masterIsStopSent()
- EUSCI_B_I2C_masterIsStartSent()
- EUSCI_B_I2C_selectMasterEnvironmentSelect()

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

- EUSCI_B_I2C_slavePutData()
- EUSCI_B_I2C_slaveGetData()

Sending and receiving data from the I2C master 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()

11.4.2 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

11.5 Programming Example

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

```
EUSCI_B_I2C_initMasterParam i2cConfig = {
    EUSCI_B_I2C_CLOCKSOURCE_SMCLK,          // SMCLK Clock Source
    4096000,                                  // SMCLK = 4.096MHz
    EUSCI_B_I2C_SET_DATA_RATE_400KBPS,      // Desired I2C Clock of 400kHz
    0,                                        // No byte counter threshold
    EUSCI_B_I2C_NO_AUTO_STOP                 // No Autostop
};

WDT_hold(WDT_BASE);

// Setting the DCO to use the internal resistor. DCO will be at 16.384MHz
CS_setupDCO(CS_INTERNAL_RESISTOR);

// Setting SMCLK = DCO / 4 = 4.096 MHz
CS_initClockSignal(CS_SMCLK, CS_CLOCK_DIVIDER_4);

// Setting P1.6 and P1.7 as I2C pins
GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
                                           GPIO_PIN6 | GPIO_PIN7,
                                           GPIO_PRIMARY_MODULE_FUNCTION);

// Setting up I2C communication at 400kHz using SMCLK
EUSCI_B_I2C_initMaster(EUSCI_B0_BASE, &i2cConfig);

// Settings slave address
EUSCI_B_I2C_setSlaveAddress(EUSCI_B0_BASE, SLAVE_ADDRESS);

// Enable the module for operation
EUSCI_B_I2C_enable(EUSCI_B0_BASE);

// Enable needed I2C interrupts
EUSCI_B_I2C_clearInterruptFlag(EUSCI_B0_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0 |
                                              EUSCI_B_I2C_NAK_INTERRUPT);
EUSCI_B_I2C_enableInterrupt(EUSCI_B0_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0 |
                                           EUSCI_B_I2C_NAK_INTERRUPT);

while(1) {

    TXByteCtr = 4;
    TXData = 0;

    // Make sure last transaction is done sending
    while(EUSCI_B_I2C_masterIsStopSent(EUSCI_B0_BASE) ==
      EUSCI_B_I2C_SENDING_STOP);

    TXByteCtr--;
    EUSCI_B_I2C_masterSendMultiByteStart(EUSCI_B0_BASE, TXData++);

    // Go to sleep and wait for LPM exit
    __bis_SR_register(LPM0_bits | GIE);
}
```

12 FlashCtl - Flash Memory Controller

Introduction 87
API Functions .... ??
Programming Example 93

12.1 Introduction

The flash memory module has an integrated controller that controls programming and erase operations. Single bits, bytes, or words can be written to flash memory, but a segment is the smallest size of the flash memory that can be erased. The flash memory is partitioned into main and information memory sections. There is no difference in the operation of the main and information memory sections. Code and data can be located in either section. See the device-specific data sheet for the start and end addresses of each bank, when available, and for the complete memory map of a device. This library provides the API for flash segment erase, flash writes and flash operation status check.

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

12.2 API Functions

Functions

- void FlashCtl_eraseSegment (uint8_t ∗flash_ptr)
  Erase a single segment of the flash memory.
- void FlashCtl_performMassErase (uint8_t ∗flash_ptr)
  Erase all flash memory.
- bool FlashCtl_performEraseCheck (uint8_t ∗flash_ptr, uint16_t numberOfBytes)
  Erase check of the flash memory.
- void FlashCtl_write8 (uint8_t ∗data_ptr, uint8_t ∗flash_ptr, uint16_t count)
  Write data into the flash memory in byte format, pass by reference.
- void FlashCtl_write16 (uint16_t ∗data_ptr, uint16_t ∗flash_ptr, uint16_t count)
  Write data into the flash memory in 16-bit word format, pass by reference.
- void FlashCtl_write32 (uint32_t ∗data_ptr, uint32_t ∗flash_ptr, uint16_t count)
  Write data into the flash memory in 32-bit word format, pass by reference.
- void FlashCtl_fillMemory32 (uint32_t value, uint32_t ∗flash_ptr, uint16_t count)
  Write data into the flash memory in 32-bit word format, pass by value.
- uint8_t FlashCtl_getStatus (uint8_t mask)
  Check FlashCtl status to see if it is currently busy erasing or programming.
- void FlashCtl_lockInfo (void)
  Locks the information flash memory segment.
- void FlashCtl_unlockInfo (void)
  Unlocks the information flash memory segment.
- uint8_t FlashCtl_setupClock (uint32_t clockTargetFreq, uint32_t clockSourceFreq, uint16_t clockSource)
  Sets up the clock for the flash module.

12.2.1 Detailed Description

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

FlashCtl_performEraseCheck() helps check if a specific number of bytes in flash are currently erased. A pointer to the starting location of the erase check and the number of bytes to be checked is passed into this function.

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

FlashCtl_write8() facilitates writing into the flash memory in byte format. FlashCtl_write16() facilitates writing into the flash memory in word format. FlashCtl_write32() facilitates writing into the flash memory in long format, pass by reference. FlashCtl_memoryFill32() facilitates writing into the flash memory in long format, pass by value. FlashCtl_getStatus() checks if the flash is currently busy erasing or programming. FlashCtl_lockInfo() locks information memory. FlashCtl_unlockInfo() unlocks information memory.

The Flash API is broken into 4 groups of functions: those that deal with flash erase, those that write into flash, those that give status of flash, and those that lock/unlock information memory.

The flash erase operations are managed by:

- FlashCtl_segmentErase()
- FlashCtl_massErase()

Flash writes are managed by:

- FlashCtl_write8()
- FlashCtl_write16()
- FlashCtl_write32()
- FlashCtl_memoryFill32()

The status is given by:

- FlashCtl_getStatus()
- FlashCtl_performEraseCheck()

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

- FlashCtl_lockInfo()
- FlashCtl_unlockInfo()

The Flash clock is managed by:

- FlashCtl_setupClock()

12.2.2 Function Documentation

void FlashCtl_eraseSegment (uint8_t ∗ *flash_ptr*)

Erase a single segment of the flash memory.

For devices like MSP430i204x, if the specified segment is the information flash segment, the FLASH_unlockInfo API must be called prior to calling this API.

Parameters

flash_ptr	is the pointer into the flash segment to be erased

Returns

None

void FlashCtl_fillMemory32 (uint32_t *value,* uint32_t ∗ *flash_ptr,* uint16_t *count*)

Write data into the flash memory in 32-bit word format, pass by value.

This function writes a 32-bit data value into flash memory, count times. Assumes the flash memory is already erased and unlocked. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

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

Returns

None

uint8_t FlashCtl_getStatus (uint8_t *mask*)

Check FlashCtl status to see if it is currently busy erasing or programming.

This function checks the status register to determine if the flash memory is ready for writing.

Parameters

mask	FLASHCTL status to read Mask value is the logical OR of any of the following: ■ **FLASHCTL_READY_FOR_NEXT_WRITE** ■ **FLASHCTL_ACCESS_VIOLATION_INTERRUPT_FLAG** ■ **FLASHCTL_PASSWORD_WRITTEN_INCORRECTLY** ■ **FLASHCTL_BUSY**

Returns

Logical OR of any of the following:
- **FlashCtl_READY_FOR_NEXT_WRITE**
- **FlashCtl_ACCESS_VIOLATION_INTERRUPT_FLAG**
- **FlashCtl_PASSWORD_WRITTEN_INCORRECTLY**
- **FlashCtl_BUSY**
  indicating the status of the FlashCtl

void FlashCtl_lockInfo (void)

Locks the information flash memory segment.

This function is typically called after an erase or write operation on the information flash segment is performed by any of the other API functions in order to re-lock the information flash segment.

Returns

None

bool FlashCtl_performEraseCheck (uint8_t * *flash_ptr,* uint16_t *numberOfBytes*)

Erase check of the flash memory.

This function checks bytes in flash memory to make sure that they are in an erased state (are set to 0xFF).

Parameters

flash_ptr	is the pointer to the starting location of the erase check
numberOfBytes	is the number of bytes to be checked

Returns

STATUS_SUCCESS or STATUS_FAIL

void FlashCtl_performMassErase (uint8_t * *flash_ptr*)

Erase all flash memory.

This function erases all the flash memory banks. For devices like MSP430i204x, this API erases main memory and information flash memory if the FLASH_unlockInfo API was previously executed (otherwise the information flash is not erased). Also note that erasing information flash memory in the MSP430i204x impacts the TLV calibration constants located at the information memory.

Parameters

flash_ptr	is a pointer into the bank to be erased

Returns

None

uint8_t FlashCtl_setupClock (uint32_t *clockTargetFreq,* uint32_t *clockSourceFreq,* uint16_t *clockSource*)

Sets up the clock for the flash module.

This function sets up the clock for the flash module. This function is typically called before any of the other flash API functions are called.

Parameters

clockTargetFreq	is the target clock source frequency in Hz.
clockSource←Freq	is the clock source frequency in Hz.
clockSource	is the clock source type for the flash. Valid values are: ■ **FLASHCTL_MCLK** [Default] ■ **FLASHCTL_SMCLK**

Returns

clock setup result
indicating clock setup succeed or failed

void FlashCtl_unlockInfo (void)

Unlocks the information flash memory segment.

This function must be called before an erase or write operation on the information flash segment is performed by any of the other API functions.

Returns

None

void FlashCtl_write16 (uint16_t ∗ *data_ptr,* uint16_t ∗ *flash_ptr,* uint16_t *count*)

Write data into the flash memory in 16-bit word format, pass by reference.

This function writes a 16-bit word array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

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

Returns

None

void FlashCtl_write32 (uint32_t ∗ *data_ptr,* uint32_t ∗ *flash_ptr,* uint16_t *count*)

Write data into the flash memory in 32-bit word format, pass by reference.

This function writes a 32-bit array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

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

Returns

None

void FlashCtl_write8 (uint8_t * *data_ptr,* uint8_t * *flash_ptr,* uint16_t *count*)

Write data into the flash memory in byte format, pass by reference.

This function writes a byte array of size count into flash memory. Assumes the flash memory is already erased and unlocked. FlashCtl_eraseSegment can be used to erase a segment.

Parameters

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

Returns

None

12.3 Programming Example

The following example shows some flash operations using the APIs

```
do {
    FlashCtl_segmentErase((uint8_t *)INFO_START);
    status = FlashCtl_performEraseCheck((uint8_t *)INFO_START, 128);
} while(status == STATUS_FAIL);

// Flash write
FlashCtl_write32(calibration_data, (uint32_t *)(INFO_START), 1);
```

13 GPIO

Introduction 95
API Functions 96
Programming Example 123

13.1 Introduction

The Digital I/O (GPIO) API provides a set of functions for using the MP430i2xx GPIO module. 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.

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() or GPIO_setAsInputPin(). The GPIO pin may instead be configured to operate in the Peripheral

Module assigned function by configuring the GPIO using GPIO_setAsPeripheralModuleFunctionOutputPin() or GPIO_setAsPeripheralModuleFunctionInputPin().

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

13.2 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.
- 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.

13.2.1 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_disableInterrupt()
- GPIO_clearInterrupt()
- GPIO_getInterruptStatus()
- GPIO_selectInterruptEdge()

The GPIO pin state is accessed with

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

13.2.2 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: ■ **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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

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

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

Returns

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

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

Returns

Logical OR of any of the following:

- **GPIO_PIN0**
- **GPIO_PIN1**
- **GPIO_PIN2**
- **GPIO_PIN3**
- **GPIO_PIN4**
- **GPIO_PIN5**
- **GPIO_PIN6**
- **GPIO_PIN7**
- **GPIO_PIN8**
- **GPIO_PIN9**
- **GPIO_PIN10**
- **GPIO_PIN11**
- **GPIO_PIN12**
- **GPIO_PIN13**
- **GPIO_PIN14**
- **GPIO_PIN15**
- **GPIO_PIN_ALL8**
- **GPIO_PIN_ALL16**
  indicating the interrupt status of the selected pins [Default: 0]

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

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

Modified bits of **PxIES** register.

Returns

None

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

Modified bits of **PxDIR** register, bits of **PxREN** register and bits of **PxSEL** 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** ■ **GPIO_PIN_ALL8** ■ **GPIO_PIN_ALL16**

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

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

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

Returns

None

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

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

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

Returns

None

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

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

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

Modified bits of **PxOUT** register.

Returns

None

13.3 Programming Example

The following example shows how to use the GPIO API.

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

// Set P1.4 to input direction
GPIO_setAsInputPin(GPIO_PORT_P1, GPIO_PIN4);

while (1) {

  // Test P1.4
  if(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN4)) {

    // if P1.4 set, set P1.0
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);
  } else {
    // else reset
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0);
  }
}
```

14 16-Bit Hardware Multiplier (MPY)

Introduction 124
API Functions 124
Programming Example 127

14.1 Introduction

The 16-Bit Hardware Multiplier (MPY) API provides a set of functions for using the MPY module. Functions are provided to setup the MPY module, set the operand registers, and obtain the results.

The MPY Modules does not generate any interrupts.

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

14.2 API Functions

Functions

- void MPY_setOperandOne8Bit (uint8_t multiplicationType, uint8_t operand)
  Sets an 8-bit value into operand 1.
- void MPY_setOperandOne16Bit (uint8_t multiplicationType, uint16_t operand)
  Sets an 16-bit value into operand 1.
- void MPY_setOperandTwo8Bit (uint8_t operand)
  Sets an 8-bit value into operand 2, which starts the multiplication.
- void MPY_setOperandTwo16Bit (uint16_t operand)
  Sets an 16-bit value into operand 2, which starts the multiplication.
- uint32_t MPY_getResult (void)
  Returns an 64-bit result of the last multiplication operation.
- uint16_t MPY_getSumExtension (void)
  Returns the Sum Extension of the last multiplication operation.

14.2.1 Detailed Description

The MPY API is broken into two groups of functions: those that set the operand registers, and those that return the results.

The operand registers are set by

- MPY_setOperandOne8Bit()
- MPY_setOperandOne16Bit()
- MPY_setOperandTwo8Bit()
- MPY_setOperandTwo16Bit()

The results can be returned by

- MPY_getResult()
- MPY_getSumExtension()

14.2.2 Function Documentation

uint32_t MPY_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

uint16_t MPY_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 MPY module Sum Extension.

void MPY_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

multiplicationType	is the type of multiplication to perform once the second operand is set. Valid values are: ■ **MPY_MULTIPLY_UNSIGNED** ■ **MPY_MULTIPLY_SIGNED** ■ **MPY_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY_MULTIPLYACCUMULATE_SIGNED**

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

Returns

None

void MPY_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: ■ **MPY_MULTIPLY_UNSIGNED** ■ **MPY_MULTIPLY_SIGNED** ■ **MPY_MULTIPLYACCUMULATE_UNSIGNED** ■ **MPY_MULTIPLYACCUMULATE_SIGNED**
operand	is the 8-bit value to load into the 1st operand.

Returns

None

void MPY_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 MPY_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

14.3 Programming Example

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

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

// Set a 16-bit operand into the specific Operand 1 register to specify

// Unsigned multiplication
MPY_setOperandOne16Bit(MPY_MULTIPLY_UNSIGNED,
                       0x1234);
// Set Operand 2 to begin the multiplication operation
MPY_setOperandTwo16Bit(0x5678);
```

15 Power Management Module (PMM)

Introduction .... ??
API Functions 128
Programming Example 132
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, to provide mechanisms for the supervising and monitoring of both the voltage applied to the device (VCC) and the voltage generated for the core (VCORE).

The PMM uses an integrated low-dropout voltage regulator (LDO) to produce a secondary core voltage (VCORE) from the primary voltage applied to the device (VCC). VCORE supplies the CPU, memories (flash/RAM), and the digital modules, while VCC supplies the I/Os and analog modules. The VCORE output is maintained using a voltage reference generated by the reference block within the PMM. 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.

PMM features include:

- Supply voltage (VCC) range: 2.2 V to 3.6 V
- High-side brownout reset (BORH)
- Supply voltage monitor (VMON) for VCC with programmable threshold levels and monitoring of external pin (VMONIN) against internal reference
- Generation of fixed voltage of 1.8-V for the device core (VCORE)
- Supply voltage supervisor (SVS) for VCORE
- Precise 1.16-V reference for the entire device and integrated temperature sensor.

15.1 API Functions

Functions

- void PMM_setupVoltageMonitor (uint8_t voltageMonitorLevel)
  Sets up the voltage monitor.
- void PMM_calibrateReference (void)
  Setup the calibration.
- void PMM_setRegulatorStatus (uint8_t status)
  Set the status of the PMM regulator.
- void PMM_unlockIOConfiguration (void)
  Unlocks the IO.
- void PMM_enableInterrupt (uint8_t mask)
  Enables interrupts.
- void PMM_disableInterrupt (uint8_t mask)
  Disables interrupts.
- uint8_t PMM_getInterruptStatus (uint8_t mask)
  Returns the interrupt status.
- void PMM_clearInterrupt (uint8_t mask)
  Clears the masked interrupts.

15.1.1 Detailed Description

The PMM API is broken into three groups of functions: those for setting up the PMM, those for using LPM4.5 mode and those used for PMM interrupts.

Setting up the PMM is done by:

- PMM_calibrateReference()
- PMM_setupVoltageMonitor()

Using LPM4.5 mode is done by:

- PMM_setRegulatorStatus()
- PMM_unlockIOConfiguration()

Using PMM interrupts is done by:

- PMM_enableInterrupt()
- PMM_disableInterrupt()
- PMM_getInterruptStatus()
- PMM_clearInterrupt()

15.1.2 Function Documentation

void PMM_calibrateReference (void)

Setup the calibration.

Modified bits of **REFCAL0** register and bits of **REFCAL1** register.

Returns

None

void PMM_clearInterrupt (uint8_t *mask*)

Clears the masked interrupts.

Parameters

mask	Mask value is the logical OR of any of the following: ■ **PMM_LPM45_INTERRUPT** - LPM 4.5 Interrupt

Returns

None

void PMM_disableInterrupt (uint8_t *mask*)

Disables interrupts.

Parameters

mask	Mask value is the logical OR of any of the following: ■ **PMM_VMON_INTERRUPT** - Voltage Monitor Interrupt

Returns

None

void PMM_enableInterrupt (uint8_t *mask*)

Enables interrupts.

Parameters

mask	Mask value is the logical OR of any of the following: ■ **PMM_VMON_INTERRUPT** - Voltage Monitor Interrupt

Returns

None

uint8_t PMM_getInterruptStatus (uint8_t *mask*)

Returns the interrupt status.

Parameters

mask	Mask value is the logical OR of any of the following: ■ **PMM_VMON_INTERRUPT** - Voltage Monitor Interrupt ■ **PMM_LPM45_INTERRUPT** - LPM 4.5 Interrupt

Returns

Logical OR of any of the following:
- **PMM_VMON_INTERRUPT** Voltage Monitor Interrupt
- **PMM_LPM45_INTERRUPT** LPM 4.5 Interrupt
  indicating the status of the masked interrupts

void PMM_setRegulatorStatus (uint8_t *status*)

Set the status of the PMM regulator.

Parameters

status	Valid values are: ■ **PMM_REGULATOR_ON** - Turn the PMM regulator off ■ **PMM_REGULATOR_OFF** - Turn the PMM regulator on Modified bits are **REGOFF** of **LPM45CTL** register.

Modified bits of **LPM45CTL** register.

Returns

None

void PMM_setupVoltageMonitor (uint8_t *voltageMonitorLevel*)

Sets up the voltage monitor.

Parameters

voltage←MonitorLevel	Valid values are: ■ **PMM_DISABLE_VMON** - Disable the voltage monitor ■ **PMM_DVCC_2350MV** - Compare DVCC to 2350mV ■ **PMM_DVCC_2650MV** - Compare DVCC to 2650mV ■ **PMM_DVCC_2850MV** - Compare DVCC to 2850mV ■ **PMM_VMONIN_1160MV** - Compare VMONIN to 1160mV Modified bits are **VMONLVLx** of **VMONCTL** register.

Modified bits of **VMONCTL** register.

Returns

None

void PMM_unlockIOConfiguration (void)

Unlocks the IO.

Modified bits are **LOCKLPM45** of **LPM45CTL** register.

Returns

None

15.2 Programming Example

The following example shows some pmm operations using the APIs

```
//Use the line below to bring the level back to 0
    status = PMM_setVCore(PMM_BASE,
        PMMCOREV_0
        );

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

    //continuous loop
    while (1)
    {
        //Toggle P1.0
        GPIO_toggleOutputOnPin(
            GPIO_PORT_P1,
            GPIO_PIN0
            );
        //Delay
        __delay_cycles(20000);
    }
```

16 24-Bit Sigma Delta Converter (SD24)

Introduction 133
API Functions 133
Programming Example 144

16.1 Introduction

The SD24 module consists of up to four independent sigma-delta analog-to-digital converters. The converters are based on second-order oversampling sigma-delta modulators and digital decimation filters. The decimation filters are comb type filters with selectable oversampling ratios of up to 256. Additional filtering can be done in software.

A sigma-delta analog-to-digital converter basically consists of two parts: the analog part - called modulator - and the digital part - a decimation filter. The modulator of the SD24 provides a bit stream of zeros and ones to the digital decimation filter. The digital filter averages the bitstream from the modulator over a given number of bits (specified by the oversampling rate) and provides samples at a reduced rate for further processing to the CPU.

As commonly known averaging can be used to increase the signal-to-noise performance of a conversion. With a conventional ADC each factor-of-4 oversampling can improve the SNR by about 6 dB or 1 bit. To achieve a 16-bit resolution out of a simple 1-bit ADC would require an impractical oversampling rate of 415 = 1.073.741.824. To overcome this limitation the sigma-delta modulator implements a technique called noise-shaping - due to an implemented feedback-loop and integrators the quantization noise is pushed to higher frequencies and thus much lower oversampling rates are sufficient to achieve high resolutions.

16.2 API Functions

Functions

- void SD24_init (uint16_t baseAddress, uint8_t referenceSelect)
  Initializes the SD24 Module.
- void SD24_initConverter (uint16_t baseAddress, uint16_t converter, uint16_t conversionMode)
  Configure SD24 converter.
- void SD24_initConverterAdvanced (uint16_t baseAddress, SD24_initConverterAdvancedParam ∗param)
  Configure SD24 converter - Advanced Configure.
- void SD24_setConverterDataFormat (uint16_t baseAddress, uint16_t converter, uint16_t dataFormat)
  Set SD24 converter data format.
- void SD24_startConverterConversion (uint16_t baseAddress, uint8_t converter)
  Start Conversion for Converter.
- void SD24_stopConverterConversion (uint16_t baseAddress, uint8_t converter)
  Stop Conversion for Converter.
- void SD24_setInputChannel (uint16_t baseAddress, uint8_t converter, uint8_t inputChannel)
  Configures the input channel.
- void SD24_setInterruptDelay (uint16_t baseAddress, uint8_t converter, uint8_t interruptDelay)

Configures the delay for an interrupt to trigger.
- void SD24_setOversampling (uint16_t baseAddress, uint8_t converter, uint16_t oversampleRatio)
  Configures the oversampling ratio for a converter.
- void SD24_setGain (uint16_t baseAddress, uint8_t converter, uint8_t gain)
  Configures the gain for the converter.
- uint32_t SD24_getResults (uint16_t baseAddress, uint8_t converter)
  Returns the results for a converter.
- uint16_t SD24_getHighWordResults (uint16_t baseAddress, uint8_t converter)
  Returns the high word results for a converter.
- void SD24_enableInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
  Enables interrupts for the SD24 Module.
- void SD24_disableInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
  Disables interrupts for the SD24 Module.
- void SD24_clearInterrupt (uint16_t baseAddress, uint8_t converter, uint16_t mask)
  Clears interrupts for the SD24 Module.
- uint16_t SD24_getInterruptStatus (uint16_t baseAddress, uint8_t converter, uint16_t mask)
  Returns the interrupt status for the SD24 Module.

16.2.1 Detailed Description

The SD24 API is broken into three groups of functions: those that deal with initialization and conversions, those that handle interrupts, and those that handle auxiliary features of the SD24.

The SD24 initialization and conversion functions are

- SD24_init()
- SD24_initConverter()
- SD24_initConverterAdvanced()
- SD24_startConverterConversion()
- SD24_stopConverterConversion()
- SD24_getResults()
- SD24_getHighWordResults()

The SD24 interrupts are handled by

- SD24_enableInterrupt()
- SD24_disableInterrupt()
- SD24_clearInterrupt()
- SD24_getInterruptStatus()

Auxiliary features of the SD24 are handled by

- SD24_setInputChannel()
- SD24_setConverterDataFormat()
- SD24_setInterruptDelay()
- SD24_setOversampling()
- SD24_setGain()

16.2.2 Function Documentation

void SD24_clearInterrupt (uint16_t *baseAddress,* uint8_t *converter,* uint16_t *mask*)

Clears interrupts for the SD24 Module.

This function clears interrupt flags for the SD24 module.

Parameters

baseAddress	is the base address of the SD24 module.
converter	is the selected converter. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
mask	is the bit mask of the converter interrupt sources to clear. Mask value is the logical OR of any of the following: ■ **SD24_CONVERTER_INTERRUPT** ■ **SD24_CONVERTER_OVERFLOW_INTERRUPT** Modified bits are **SD24OVIFGx** of **SD24BIFG** register.

Returns

None

void SD24_disableInterrupt (uint16_t *baseAddress,* uint8_t *converter,* uint16_t *mask*)

Disables interrupts for the SD24 Module.

This function disables interrupts for the SD24 module.

Parameters

baseAddress	is the base address of the SD24 module.
converter	is the selected converter. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**

mask	is the bit mask of the converter interrupt sources to be disabled. Mask value is the logical OR of any of the following: ■ **SD24_CONVERTER_INTERRUPT** ■ **SD24_CONVERTER_OVERFLOW_INTERRUPT** Modified bits are **SD24OVIEx** of **SD24BIE** register.

Modified bits of **SD24BIE** register.

Returns

None

void SD24_enableInterrupt (uint16_t *baseAddress,* uint8_t *converter,* uint16_t *mask*)

Enables interrupts for the SD24 Module.

This function enables interrupts for the SD24 module. Does not clear interrupt flags.

Parameters

baseAddress	is the base address of the SD24 module.
converter	is the selected converter. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
mask	is the bit mask of the converter interrupt sources to be enabled. Mask value is the logical OR of any of the following: ■ **SD24_CONVERTER_INTERRUPT** ■ **SD24_CONVERTER_OVERFLOW_INTERRUPT** Modified bits are **SD24OVIEx** of **SD24BIE** register.

Returns

None

uint16_t SD24_getHighWordResults (uint16_t *baseAddress,* uint8_t *converter*)

Returns the high word results for a converter.

This function gets the upper 16-bit result from the SD24MEMx register and returns it.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter who's results will be returned Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**

Returns

Result of conversion

uint16_t SD24_getInterruptStatus (uint16_t *baseAddress,* uint8_t *converter,* uint16_t *mask*)

Returns the interrupt status for the SD24 Module.

This function returns interrupt flag statuses for the SD24 module.

Parameters

baseAddress	is the base address of the SD24 module.
converter	is the selected converter. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
mask	is the bit mask of the converter interrupt sources to return. Mask value is the logical OR of any of the following: ■ **SD24_CONVERTER_INTERRUPT** ■ **SD24_CONVERTER_OVERFLOW_INTERRUPT**

Returns

Logical OR of any of the following:

- **SD24_CONVERTER_INTERRUPT**
- **SD24_CONVERTER_OVERFLOW_INTERRUPT**
  indicating the status of the masked interrupts

uint32_t SD24_getResults (uint16_t *baseAddress,* uint8_t *converter*)

Returns the results for a converter.

This function gets the results from the SD24MEMx register for upper 16-bit and lower 16-bit results, and concatenates them to form a long. The actual result is a maximum 24 bits.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter who's results will be returned Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**

Returns

Result of conversion

void SD24_init (uint16_t *baseAddress,* uint8_t *referenceSelect*)

Initializes the SD24 Module.

This function initializes the SD24 module sigma-delta analog-to-digital conversions. Specifically the function sets up the clock source for the SD24 core to use for conversions. Upon completion of the initialization the SD24 interrupt registers will be reset and the given parameters will be set. The converter configuration settings are independent of this function.

Parameters

baseAddress	is the base address of the SD24 module.
referenceSelect	selects the reference source for the SD24 core Valid values are: ■ **SD24_REF_EXTERNAL** [Default] ■ **SD24_REF_INTERNAL** Modified bits are **SD24REFS** of **SD24BCTL0** register.

Returns

None

void SD24_initConverter (uint16_t *baseAddress,* uint16_t *converter,* uint16_t *conversionMode*)

Configure SD24 converter.

This function initializes a converter of the SD24 module. Upon completion the converter will be ready for a conversion and can be started with the SD24_startConverterConversion(). Additional configuration such as data format can be configured in SD24_setConverterDataFormat().

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be configured. Check check datasheet for available converters on device. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
conversionMode	determines whether the converter will do continuous samples or a single sample Valid values are: ■ **SD24_CONTINUOUS_MODE** [Default] ■ **SD24_SINGLE_MODE** Modified bits are **SD24SNGL** of **SD24CCTLx** register.

Returns

None

void SD24_initConverterAdvanced (uint16_t *baseAddress,* **SD24_initConverterAdvancedParam** ∗ *param*)

Configure SD24 converter - Advanced Configure.

This function initializes a converter of the SD24 module. Upon completion the converter will be ready for a conversion and can be started with the SD24_startConverterConversion().

Parameters

baseAddress	is the base address of the SD24 module.
param	is the pointer to struct for converter advanced configuration.

Returns

None

References SD24_initConverterAdvancedParam::conversionMode, SD24_initConverterAdvancedParam::converter, SD24_initConverterAdvancedParam::dataFormat, SD24_initConverterAdvancedParam::gain, SD24_initConverterAdvancedParam::groupEnable, SD24_initConverterAdvancedParam::inputChannel, SD24_initConverterAdvancedParam::interruptDelay, and SD24_initConverterAdvancedParam::oversampleRatio.

void SD24_setConverterDataFormat (uint16_t *baseAddress,* uint16_t *converter,* uint16_t *dataFormat*)

Set SD24 converter data format.

This function sets the converter format so that the resulting data can be viewed in either binary or 2's complement.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be configured. Check check datasheet for available converters on device. Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
dataFormat	selects how the data format of the results Valid values are: ■ **SD24_DATA_FORMAT_BINARY** [Default] ■ **SD24_DATA_FORMAT_2COMPLEMENT** Modified bits are **SD24DFx** of **SD24CCTLx** register.

Returns

None

void SD24_setGain (uint16_t *baseAddress,* uint8_t *converter,* uint8_t *gain*)

Configures the gain for the converter.

This function configures the gain for a single converter.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be configured Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
gain	selects the gain for the converter Valid values are: ■ **SD24_GAIN_1** [Default] ■ **SD24_GAIN_2** ■ **SD24_GAIN_4** ■ **SD24_GAIN_8** ■ **SD24_GAIN_16** Modified bits are **SD24GAINx** of **SD24INCTLx** register.

Returns

None

void SD24_setInputChannel (uint16_t *baseAddress,* uint8_t *converter,* uint8_t *inputChannel*)

Configures the input channel.

This function configures the input channel. For MSP430i2xx devices, users can choose either analog input or internal temperature input.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be configured Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
inputChannel	selects oversampling ratio for the converter Valid values are: ■ **SD24_INPUT_CH_ANALOG** ■ **SD24_INPUT_CH_TEMPERATURE** Modified bits are **SD24INCHx** of **SD24INCTLx** register.

Returns

None

void SD24_setInterruptDelay (uint16_t *baseAddress,* uint8_t *converter,* uint8_t *interruptDelay*)

Configures the delay for an interrupt to trigger.

This function configures the delay for the first interrupt service request for the corresponding converter. This feature delays the interrupt request for a completed conversion by up to four conversion cycles allowing the digital filter to settle prior to generating an interrupt request.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be stopped Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
interruptDelay	selects the delay for the interrupt Valid values are: ■ **SD24_FIRST_SAMPLE_INTERRUPT** ■ **SD24_FOURTH_SAMPLE_INTERRUPT** [Default] Modified bits are **SD24INTDLYx** of **SD24INCTLx** register.

Returns

None

void SD24_setOversampling (uint16_t *baseAddress,* uint8_t *converter,* uint16_t *oversampleRatio*)

Configures the oversampling ratio for a converter.

This function configures the oversampling ratio for a given converter.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be configured Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3**
oversampleRatio	selects oversampling ratio for the converter Valid values are: ■ **SD24_OVERSAMPLE_32** ■ **SD24_OVERSAMPLE_64** ■ **SD24_OVERSAMPLE_128** ■ **SD24_OVERSAMPLE_256** Modified bits are **SD24OSRx** of **SD24OSRx** register.

Returns

None

void SD24_startConverterConversion (uint16_t *baseAddress,* uint8_t *converter*)

Start Conversion for Converter.

This function starts a single converter.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be started Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3** Modified bits are **SD24SC** of **SD24CCTLx** register.

Returns

None

void SD24_stopConverterConversion (uint16_t *baseAddress,* uint8_t *converter*)

Stop Conversion for Converter.

This function stops a single converter.

Parameters

baseAddress	is the base address of the SD24 module.
converter	selects the converter that will be stopped Valid values are: ■ **SD24_CONVERTER_0** ■ **SD24_CONVERTER_1** ■ **SD24_CONVERTER_2** ■ **SD24_CONVERTER_3** Modified bits are **SD24SC** of **SD24CCTLx** register.

Returns

None

16.3 Programming Example

16.4 Programming Example

The following example shows how to initialize and use the SD24 API to start a single channel, single conversion.

```
  unsigned long results;

  SD24_init(SD24_BASE, SD24_REF_INTERNAL);       // Select internal REF

SD24_initConverterAdvancedParam param = {0};
param.converter = SD24_CONVERTER_2;         // Select converter
param.conversionMode = SD24_SINGLE_MODE;    // Select single mode
param.groupEnable = SD24_NOT_GROUPED;       // No grouped
param.inputChannel = SD24_INPUT_CH_ANALOG; // Input from analog signal
param.dataFormat = SD24_DATA_FORMAT_2COMPLEMENT;  // 2's complement data format
param.interruptDelay = SD24_FOURTH_SAMPLE_INTERRUPT;  // 4th sample causes interrupt
param.oversampleRatio = SD24_OVERSAMPLE_256;  // Oversampling ratio 256
param.gain = SD24_GAIN_1;  // Preamplifier gain x1
SD24_initConverterAdvanced(SD24_BASE, &param);

 __delay_cycles(0x3600);                    // Delay for 1.5V REF startup

 while (1)
 {
     SD24_startConverterConversion(SD24_BASE,
         SD24_CONVERTER_2);                             // Set bit to start conversion

     // Poll interrupt flag for channel 2
     while( SD24_getInterruptStatus(SD24_BASE,
```

```
            SD24_CONVERTER_2
            SD24_CONVERTER_INTERRUPT) == 0 );

    results = SD24_getResults(SD24_BASE,
        SD24_CONVERTER_2);                // Save CH2 results (clears IFG)

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

17 Special Function Register (SFR)

Introduction 145
API Functions 145
Programming Example 147

17.1 Introduction

The Special Function Registers API provides a set of functions for using the MSP430i2xx SFR module. Functions are provided to enable and disable interrupts.

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

17.2 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.

17.2.1 Detailed Description

The SFR interrupts are handled by

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

17.2.2 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_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERRUPT** - Watchdog interrupt ■ **SFR_EXTERNAL_RESET_INTERRUPT** - External reset interrupt ■ **SFR_BROWN_OUT_RESET_INTERRUPT** - Brown out reset 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_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERRUPT** - Watchdog interrupt ■ **SFR_FLASH_ACCESS_VIOLATION_INTERRUPT** - Flash access violation 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_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERRUPT** - Watchdog interrupt ■ **SFR_FLASH_ACCESS_VIOLATION_INTERRUPT** - Flash access violation 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_NMI_PIN_INTERRUPT** - NMI pin interrupt, if NMI function is chosen ■ **SFR_OSCILLATOR_FAULT_INTERRUPT** - Oscillator fault interrupt ■ **SFR_WATCHDOG_INTERRUPT** - Watchdog interrupt ■ **SFR_EXTERNAL_RESET_INTERRUPT** - External reset interrupt ■ **SFR_BROWN_OUT_RESET_INTERRUPT** - Brown out reset interrupt

Returns

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

- **SFR_NMI_PIN_INTERRUPT** NMI pin interrupt, if NMI function is chosen
- **SFR_OSCILLATOR_FAULT_INTERRUPT** Oscillator fault interrupt
- **SFR_WATCHDOG_INTERRUPT** Watchdog interrupt
- **SFR_EXTERNAL_RESET_INTERRUPT** External reset interrupt
- **SFR_BROWN_OUT_RESET_INTERRUPT** Brown out reset interrupt
  indicating the status of the masked interrupts

17.3 Programming Example

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

```
do {
  // Clear SFR Fault Flag
  SFR_clearInterrupt(SFR_OSCILLATOR_FAULT_INTERRUPT);
```

```
  // Test oscillator fault flag
} while (SFR_getInterruptStatus(SFR_OSCILLATOR_FAULT_INTERRUPT));
```

18 16-Bit Timer_A (TIMER_A)

Introduction 149
API Functions 150
Programming Example 163

18.1 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.

18.2 API Functions

Functions

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

Stops the timer.
- void Timer_A_setCompareValue (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareValue)
  Sets the value of the capture-compare register.
- void Timer_A_setOutputMode (uint16_t baseAddress, uint16_t compareRegister, uint16_t compareOutputMode)
  Sets the output mode.
- void Timer_A_clearTimerInterrupt (uint16_t baseAddress)
  Clears the Timer TAIFG interrupt flag.
- void Timer_A_clearCaptureCompareInterrupt (uint16_t baseAddress, uint16_t captureCompareRegister)
  Clears the capture-compare interrupt flag.
- uint16_t Timer_A_getCounterValue (uint16_t baseAddress)
  Reads the current timer count value.

18.2.1 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()

18.2.2 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**

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.
captureCompareRegister	is the selected capture compare register Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**

Modified bits of **TAxCCTLn** register.

Returns

None

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**

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**

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**
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**

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**
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_initContinuousModeParam** ∗ *param*)

Configures Timer_A in continuous mode.

Parameters

baseAddress	is the base address of the TIMER_A module.
param	is the pointer to struct for continuous mode initialization.

Modified bits of **TAxCTL** register.

Returns

None

References Timer_A_initContinuousModeParam::clockSource, Timer_A_initContinuousModeParam::clockSourceDivider, Timer_A_initContinuousModeParam::startTimer, Timer_A_initContinuousModeParam::timerClear, and Timer_A_initContinuousModeParam::timerInterruptEnable_TAIE.

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**
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.
captureCompareRegister	Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
outputModeOutBitValue	is the value to be set for out bit Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_HIGH** ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE_LOW**

Modified bits of **TAxCCTLn** register.

Returns

None

void Timer_A_setOutputMode (uint16_t *baseAddress,* uint16_t *compareRegister,* uint16_t *compareOutputMode*)

Sets the output mode.

Sets the output mode for the timer even the timer is already running.

Parameters

baseAddress	is the base address of the TIMER_A module.
compare←Register	selects the compare register being used. Valid values are: ■ **TIMER_A_CAPTURECOMPARE_REGISTER_0** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_1** ■ **TIMER_A_CAPTURECOMPARE_REGISTER_2**
compare←OutputMode	specifies the output mode. Valid values are: ■ **TIMER_A_OUTPUTMODE_OUTBITVALUE** [Default] ■ **TIMER_A_OUTPUTMODE_SET** ■ **TIMER_A_OUTPUTMODE_TOGGLE_RESET** ■ **TIMER_A_OUTPUTMODE_SET_RESET** ■ **TIMER_A_OUTPUTMODE_TOGGLE** ■ **TIMER_A_OUTPUTMODE_RESET** ■ **TIMER_A_OUTPUTMODE_TOGGLE_SET** ■ **TIMER_A_OUTPUTMODE_RESET_SET**

Modified bits are **OUTMOD** of **TAxCCTLn** register.

Returns

None

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

18.3 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();
}
```

19 Tag Length Value (TLV)

Introduction 165
API Functions 165
Programming Example 167

19.1 Introduction

The TLV structure is a table stored in flash memory that contains device-specific information. 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 TLV section), and an explanation on its functionality is available in the MSP430i2xx Family User's Guide.

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

19.2 API Functions

Functions

- void TLV_getInfo (uint8_t tag, uint8_t *length, uint16_t **data_address)
  Gets TLV Info.
- bool TLV_performChecksumCheck (void)
  Performs checksum check on TLV.

19.2.1 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_performChecksumCheck() This function performs a CRC check on the TLV.

19.2.2 Function Documentation

void TLV_getInfo (uint8_t *tag,* 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. This can be used to retrieve calibration constants for the device or find out more information about the device. This function retrieves the address of a tag and the length of the tag request. Please check the device datasheet for tags available on your device.

Parameters

tag	represents the tag for which the information needs to be retrieved. Valid values are: ■ **TLV_CHECKSUM** ■ **TLV_TAG_DIE_RECORD** ■ **TLV_LENGTH_DIE_RECORD** ■ **TLV_WAFER_LOT_ID** ■ **TLV_DIE_X_POSITION** ■ **TLV_DIE_Y_POSITION** ■ **TLV_TEST_RESULTS** ■ **TLV_REF_CALIBRATION_TAG** ■ **TLV_REF_CALIBRATION_LENGTH** ■ **TLV_REF_CALIBRATION_REFCAL1** ■ **TLV_REF_CALIBRATION_REFCAL0** ■ **TLV_DCO_CALIBRATION_TAG** ■ **TLV_DCO_CALIBRATION_LENGTH** ■ **TLV_DCO_CALIBRATION_CSIRFCAL** ■ **TLV_DCO_CALIBRATION_CSIRTCAL** ■ **TLV_DCO_CALIBRATION_CSERFCAL** ■ **TLV_DCO_CALIBRATION_CSERTCAL** ■ **TLV_SD24_CALIBRATION_TAG** ■ **TLV_SD24_CALIBRATION_LENGTH** ■ **TLV_SD24_CALIBRATION_SD24TRIM**

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.
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.

Returns

None

bool TLV_performChecksumCheck (void)

Performs checksum check on TLV.

The 2's complement checksum is calculated on the data stored in the TLV. If the calculated checksum is equal to the checksum stored in the TLV then the user knows that the TLV has not been corrupted. This API can be used after a BOR before writing configuration constants to the appropriate registers.

Returns

true if checksum of TLV matches value stored in TLV, false otherwise

19.3 Programming Example

The following example shows some tlv operations using the APIs

```
bool result;

// Stop the WDT
WDT_hold(WDT_BASE);

// Check the TLV checksum
result = TLV_performChecksumCheck();

// Turn on LED if test passed
if(result) {
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN4);
} else {
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN4);
}

// LED for indicating checksum result
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN4);
```

20 WatchDog Timer (WDT)

Introduction .... ??
API Functions 168
Programming Example 171 The Watchdog Timer (WDT) API provides a set of functions for using the WDT module. 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 module can generate only 1 kind of interrupt in timer interval mode. If in watchdog mode, then the WDT module will assert a reset once the timer has finished.

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

20.1 API Functions

Functions

- void WDT_hold (uint16_t baseAddress)
  Holds the Watchdog Timer.
- void WDT_start (uint16_t baseAddress)
  Starts the Watchdog Timer.
- void WDT_resetTimer (uint16_t baseAddress)
  Resets the timer counter of the Watchdog Timer.
- void WDT_initWatchdogTimer (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
  Sets the clock source for the Watchdog Timer in watchdog mode.
- void WDT_initIntervalTimer (uint16_t baseAddress, uint8_t clockSelect, uint8_t clockDivider)
  Sets the clock source for the Watchdog Timer in timer interval mode.

20.1.1 Detailed Description

The WDT API is one group that controls the WDT module.

- WDT_hold()
- WDT_start()
- WDT_resetTimer()
- WDT_initWatchdogTimer()
- WDT_initIntervalTimer()

20.1.2 Function Documentation

void WDT_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 module.

Returns

None

void WDT_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 module.
clockSelect	is the clock source that the watchdog timer will use. Valid values are: ■ **WDT_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_CLOCKSOURCE_ACLK** 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_CLOCKDIVIDER_32K** [Default] ■ **WDT_CLOCKDIVIDER_8192** ■ **WDT_CLOCKDIVIDER_512** ■ **WDT_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

void WDT_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_resetTimer() before the timer runs out.

Parameters

baseAddress	is the base address of the WDT module.
clockSelect	is the clock source that the watchdog timer will use. Valid values are: ■ **WDT_CLOCKSOURCE_SMCLK** [Default] ■ **WDT_CLOCKSOURCE_ACLK** 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_CLOCKDIVIDER_32K** [Default] ■ **WDT_CLOCKDIVIDER_8192** ■ **WDT_CLOCKDIVIDER_512** ■ **WDT_CLOCKDIVIDER_64** Modified bits are **WDTIS** and **WDTHOLD** of **WDTCTL** register.

Returns

None

void WDT_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 module.

Returns

None

void WDT_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 module.

Returns

None

20.2 Programming Example

The following example shows how to initialize and use the WDT to expire every 32ms.

```
// Stop the WDT
WDT_hold(WDT_BASE);

// Check if WDT caused reset
if(SFR_getInterruptStatus(SFR_WATCHDOG_INTERRUPT)) {
    SFR_clearInterrupt(SFR_WATCHDOG_INTERRUPT);
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN4);
}

GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN4);

// Setup DCO to use internal resistor. DCO calibrated at 16.384 MHz
CS_setupDCO(CS_INTERNAL_RESISTOR);

// Setup SMCLK as DCO / 16 = 1.024 MHz
CS_initClockSignal(CS_SMCLK, CS_CLOCK_DIVIDER_16);

// Configure interval to SMCLK / 32768 = 32ms
WDT_initWatchdogTimer(WDT_BASE,
                      WDT_CLOCKSOURCE_SMCLK,
                      WDT_CLOCKDIVIDER_32K);
WDT_resetTimer(WDT_BASE);
```

21 Data Structure Documentation

21.1 Data Structures

Here are the data structures with brief descriptions:

EUSCI_A_SPI_changeMasterClockParam
Used in the EUSCI_A_SPI_changeMasterClock() function as the param parameter . 172
EUSCI_A_SPI_initMasterParam
Used in the EUSCI_A_SPI_initMaster() function as the param parameter 173
EUSCI_A_SPI_initSlaveParam
Used in the EUSCI_A_SPI_initSlave() function as the param parameter 175
EUSCI_A_UART_initParam
Used in the EUSCI_A_UART_init() function as the param parameter 176
EUSCI_B_I2C_initMasterParam
Used in the EUSCI_B_I2C_initMaster() function as the param parameter 178
EUSCI_B_I2C_initSlaveParam
Used in the EUSCI_B_I2C_initSlave() function as the param parameter 180
EUSCI_B_SPI_changeMasterClockParam
Used in the EUSCI_B_SPI_changeMasterClock() function as the param parameter . 181
EUSCI_B_SPI_initMasterParam
Used in the EUSCI_B_SPI_initMaster() function as the param parameter 181
EUSCI_B_SPI_initSlaveParam
Used in the EUSCI_B_SPI_initSlave() function as the param parameter 183
SD24_initConverterAdvancedParam
Used in the SD24_initConverterAdvanced() function as the param parameter 184
Timer_A_initCaptureModeParam
Used in the Timer_A_initCaptureMode() function as the param parameter 187
Timer_A_initCompareModeParam
Used in the Timer_A_initCompareMode() function as the param parameter 189
Timer_A_initContinuousModeParam
Used in the Timer_A_initContinuousMode() function as the param parameter 190
Timer_A_initUpDownModeParam
Used in the Timer_A_initUpDownMode() function as the param parameter 192
Timer_A_initUpModeParam
Used in the Timer_A_initUpMode() function as the param parameter 193
Timer_A_outputPWMParam
Used in the Timer_A_outputPWM() function as the param parameter 195

21.2 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.

21.2.1 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

21.3 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

21.3.1 Detailed Description

Used in the EUSCI_A_SPI_initMaster() function as the param parameter.

21.3.2 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. Refer to device specific datasheet for available options.
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

21.4 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

21.4.1 Detailed Description

Used in the EUSCI_A_SPI_initSlave() function as the param parameter.

21.4.2 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

21.5 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

21.5.1 Detailed Description

Used in the EUSCI_A_UART_init() function as the param parameter.

21.5.2 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. Refer to device specific datasheet for available options.
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

21.6 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

21.6.1 Detailed Description

Used in the EUSCI_B_I2C_initMaster() function as the param parameter.

21.6.2 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

Selects the clocksource. Refer to device specific datasheet for available options.
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

21.7 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

21.7.1 Detailed Description

Used in the EUSCI_B_I2C_initSlave() function as the param parameter.

21.7.2 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

21.8 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.

21.8.1 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

21.9 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

21.9.1 Detailed Description

Used in the EUSCI_B_SPI_initMaster() function as the param parameter.

21.9.2 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. Refer to device specific datasheet for available options.
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

21.10 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

21.10.1 Detailed Description

Used in the EUSCI_B_SPI_initSlave() function as the param parameter.

21.10.2 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

21.11 SD24_initConverterAdvancedParam Struct Reference

Used in the SD24_initConverterAdvanced() function as the param parameter.

```
#include <sd24.h>
```

Data Fields

- uint8_t converter
- uint16_t conversionMode
- uint8_t groupEnable
- uint8_t inputChannel
- uint8_t dataFormat
- uint8_t interruptDelay
- uint16_t oversampleRatio
- uint8_t gain

21.11.1 Detailed Description

Used in the SD24_initConverterAdvanced() function as the param parameter.

21.11.2 Field Documentation

uint16_t SD24_initConverterAdvancedParam::conversionMode

Determines whether the converter will do continuous samples or a single sample
Valid values are:

- **SD24_CONTINUOUS_MODE** [Default]
- **SD24_SINGLE_MODE**

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::converter

Selects the converter that will be configured. Check check datasheet for available converters on device.
Valid values are:

- **SD24_CONVERTER_0**
- **SD24_CONVERTER_1**
- **SD24_CONVERTER_2**
- **SD24_CONVERTER_3**

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::dataFormat

Selects how the data format of the results
Valid values are:

- **SD24_DATA_FORMAT_BINARY** [Default]
- **SD24_DATA_FORMAT_2COMPLEMENT**

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::gain

Selects the gain for the converter
Valid values are:

- **SD24_GAIN_1** [Default]
- **SD24_GAIN_2**
- **SD24_GAIN_4**
- **SD24_GAIN_8**
- **SD24_GAIN_16**

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::groupEnable

Valid values are:

- **SD24_NOT_GROUPED**
- **SD24_GROUPED** [Default]

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::inputChannel

Selects oversampling ratio for the converter
Valid values are:

- **SD24_INPUT_CH_ANALOG**
- **SD24_INPUT_CH_TEMPERATURE**

Referenced by SD24_initConverterAdvanced().

uint8_t SD24_initConverterAdvancedParam::interruptDelay

Selects the delay for the interrupt
Valid values are:

- **SD24_FIRST_SAMPLE_INTERRUPT**
- **SD24_FOURTH_SAMPLE_INTERRUPT** [Default]

Referenced by SD24_initConverterAdvanced().

uint16_t SD24_initConverterAdvancedParam::oversampleRatio

Selects oversampling ratio for the converter
Valid values are:

- **SD24_OVERSAMPLE_32**
- **SD24_OVERSAMPLE_64**
- **SD24_OVERSAMPLE_128**
- **SD24_OVERSAMPLE_256**

Referenced by SD24_initConverterAdvanced().

The documentation for this struct was generated from the following file:

- sd24.h

21.12 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

21.12.1 Detailed Description

Used in the Timer_A_initCaptureMode() function as the param parameter.

21.12.2 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**

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

21.13 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.

21.13.1 Detailed Description

Used in the Timer_A_initCompareMode() function as the param parameter.

21.13.2 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**

Referenced by Timer_A_initCompareMode().

The documentation for this struct was generated from the following file:

- timer_a.h

21.14 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.

21.14.1 Detailed Description

Used in the Timer_A_initContinuousMode() function as the param parameter.

21.14.2 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_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**

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

21.15 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.

21.15.1 Detailed Description

Used in the Timer_A_initUpDownMode() function as the param parameter.

21.15.2 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_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**

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

21.16 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.

21.16.1 Detailed Description

Used in the Timer_A_initUpMode() function as the param parameter.

21.16.2 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_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**

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

21.17 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.

21.17.1 Detailed Description

Used in the Timer_A_outputPWM() function as the param parameter.

21.17.2 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_4**
- **TIMER_A_CLOCKSOURCE_DIVIDER_8**

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**

Referenced by Timer_A_outputPWM().

The documentation for this struct was generated from the following file:

- timer_a.h

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265