

# **Peripheral Driver Library**

## **Texas Instruments CC2538 Family of Products**

# **User's Guide**




# Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License (CC BY-ND 3.0). To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/legalcode> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

# Copyright

Copyright © 2013 Texas Instruments Incorporated. All rights reserved. CC2538 and SmartRF are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

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

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



# Revision Information

User guide literature numbers from Texas Instruments RF Products start with *SWRU*. The document revision is indicated by a letter suffix after the literature number. The initial version of a document does not have a letter suffix (for example *SWRU321*). The first revision is suffixed *A*, the second *B*, and so on (for example *SWRU321B*). The literature number for this document is in the document footer.

This document was updated on May 02, 2013 (build 9965).

# Table of Contents

<b>Document License</b>	<b>2</b>
<b>Copyright</b>	<b>2</b>
<b>Revision Information</b>	<b>2</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Programming Model</b>	<b>7</b>
2.1 Introduction	7
2.2 Direct Register Access Model	7
2.3 Software Driver Model	8
2.4 Combining The Models	8
<b>3 Analog to Digital Converter (ADC)</b>	<b>11</b>
3.1 Introduction	11
3.2 API Functions	12
3.3 Programming Example	15
<b>4 Advanced Encryption Standard Electronic Code Book mode (AES)</b>	<b>17</b>
4.1 Introduction	17
4.2 API Functions	17
4.3 Programming Example	19
<b>5 Advanced Encryption Standard Counter with CBC-MAC (CCM)</b>	<b>21</b>
5.1 Introduction	21
5.2 API Functions	21
5.3 Programming Example	25
<b>6 Flash</b>	<b>27</b>
6.1 Introduction	27
6.2 API Functions	28
6.3 Programming Example	32
<b>7 General-Purpose Inputs/Outputs (GPIO)</b>	<b>33</b>
7.1 Introduction	33
7.2 API Functions	34
7.3 Programming Example	50
<b>8 Inter-Integrated Circuit (I2C)</b>	<b>53</b>
8.1 Introduction	53
8.2 API Functions	54
8.3 Programming Example	67
<b>9 Nested Vectored Interrupt Controller (NVIC)</b>	<b>69</b>
9.1 Introduction	69
9.2 API Functions	70
9.3 Programming Example	78
<b>10 I/O Control (IOC)</b>	<b>79</b>
10.1 Introduction	79
10.2 API Functions	80
10.3 Programming Example	84
<b>11 Public Key HW Accelerator driver (PKA)</b>	<b>85</b>
11.1 Introduction	85
11.2 API Functions	85
<b>12 Sleep Timer</b>	<b>99</b>

12.1	Introduction . . . . .	99
12.2	API Functions . . . . .	100
12.3	Programming Example . . . . .	104
<b>13</b>	<b>Synchronous Serial Interface (SSI)</b> . . . . .	<b>105</b>
13.1	Introduction . . . . .	105
13.2	API Functions . . . . .	105
13.3	Programming Example . . . . .	115
<b>14</b>	<b>System Control (SysCtrl)</b> . . . . .	<b>117</b>
14.1	Introduction . . . . .	117
14.2	API Functions . . . . .	118
14.3	Programming Example . . . . .	127
<b>15</b>	<b>System Tick (SysTick)</b> . . . . .	<b>129</b>
15.1	Introduction . . . . .	129
15.2	API Functions . . . . .	129
15.3	Programming Example . . . . .	133
<b>16</b>	<b>General Purpose Timer</b> . . . . .	<b>135</b>
16.1	Introduction . . . . .	135
16.2	API Functions . . . . .	136
16.3	Programming Example . . . . .	149
<b>17</b>	<b>UART</b> . . . . .	<b>151</b>
17.1	Introduction . . . . .	151
17.2	API Functions . . . . .	151
17.3	Programming Example . . . . .	170
<b>18</b>	<b>uDMA Controller</b> . . . . .	<b>173</b>
18.1	Introduction . . . . .	173
18.2	API Functions . . . . .	174
18.3	Programming Example . . . . .	190
<b>19</b>	<b>Watchdog Timer (WDT)</b> . . . . .	<b>193</b>
19.1	Introduction . . . . .	193
19.2	API Functions . . . . .	193
19.3	Programming Example . . . . .	194
<b>20</b>	<b>Secure Hash Algorithm of digest size 256 (SHA)</b> . . . . .	<b>197</b>
20.1	Introduction . . . . .	197
20.2	API Functions . . . . .	197
20.3	Programming Example . . . . .	199
<b>21</b>	<b>Error Handling</b> . . . . .	<b>201</b>

# 1 Introduction

The CC2538 peripheral driver library from Texas Instruments® is a set of drivers for accessing the peripherals found on the CC2538 family of ARM® Cortex™-M3 based devices. 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 peripherals of the device.

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

- They are written entirely in C language 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 language 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 used 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 useful only during initial program development, it can be removed to improve code size and speed (see chapter 21).

For many applications, the drivers can be used as is. But in some cases, the drivers must be enhanced or rewritten 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.

The driver library includes drivers for all devices in the CC2538 family.

The following tool chains are supported:

- IAR Embedded Workbench®
- TI Code Composer Studio™

## Source Code Overview

A brief overview of the organization of the peripheral driver library source code follows:

`source/` This directory contains the source code for the drivers.

---

<code>hw_*.h</code>	Header files, one per peripheral, that describe all the registers and the bit fields within those registers for each peripheral. These header files are used by the drivers to directly access a peripheral, and can be used by application code to bypass the peripheral driver library API.
<code>inc/</code>	This directory holds the header files used for the direct register access programming model.

## 2 Programming Model

Introduction .....	7
Direct Register Access Model .....	7
Software Driver Model .....	8
Combining The Models .....	8

### 2.1 Introduction

The peripheral driver library provides support for two programming models: the direct register access model and the software driver model. Each model can be used independently or combined, based on the needs of the application or the programming environment desired by the developer.

Each programming model has advantages and disadvantages. Use of the direct register access model generally results in smaller and more efficient code than using the software driver model. However, the direct register access model requires detailed knowledge of the operation of each register and bit field, as well as their interactions and any sequencing required for proper operation of the peripheral; the software driver model insulates the developer from these details, thus generally requiring less time to develop applications.

### 2.2 Direct Register Access Model

In the direct register access model, the peripherals are programmed by the application by writing values directly into the registers in the peripheral. A set of defines that simplifies this process is provided.

These defines are stored in the `inc` directory, there is a single `hw_*.h` header file for each peripheral type. For example, the SSI defines are stored in the `hw_ssi.h` header file.

The defines used by the direct register access model follow a naming convention that makes it easier to know how to use a particular macro. The rules are as follows:

- All register name macros start with the module name (for example, `IOC` for the IOC module) and are followed by the name of the register as it appears in the data sheet (for example, the `GPT0OCP1` register in the data sheet results in `IOC_GPT0OCP1`).
- All register defines for a given peripheral, are listed in the first section of the corresponding header file (for example, in the first section of the `hw_ssi.h` file for the SSI registers).
- In case there are multiple instances of the same peripheral or submodule on the device, the register defines are offset values relative to the base address of a peripheral instance. If an offset is used, this will be identified in the register name using `_O_` (for example, `CR0` register in the data sheet results in `SSI_O_CR0`). The base address of each peripheral is defined in the memory map header file stored in the `inc` directory, under the name `hw_memmap.h`. If there is only a single instance absolute addresses are used (for example, the register defines for the IOC module are defined by their full addresses in the `hw_ioc.h` file).
- Values that end in `_M` represent the mask for a bit field in a register.

- Values that end in `_S` represent the number of bits to shift a value in order to align it with a bit field. These values match the macro with the same base name but ending with `_M`.
- All other macros represent the value of a single bit field.
- All register bit fields start with the module name, followed by the register name, and then followed by the bit field name as it appears in the data sheet. For example, the `SCR` bit field in the `CR0` register in the `SSI` module is identified by `SSI_CR0_SCR...`

Given these defines, the `CR0` register, in the first instance of the `SSI` peripheral (`SSI0`), can be programmed as follows:

```
HWREG(SSI0_BASE + SSI_O_CR0) = ((5 << SSI_CR0_SCR_S) | SSI_CR0_SPH | SSI_CR0_SPO);
```

Alternatively, the following has the same effect (although it is not as easy to understand):

```
HWREG(SSI0_BASE + SSI_O_CR0) = 0x000005c0;
```

The value of the `SCR` field from the `CR0` register can be extracted as follows:

```
ulValue = (HWREG(SSI0_BASE + SSI_O_CR0) & SSI_CR0_SCR_M) >> SSI_CR0_SCR_S;
```

## 2.3 Software Driver Model

In the software driver model, the API provided by the peripheral driver library is used by applications to control the peripherals. Because these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This method provides for rapid development of the application without requiring detailed knowledge of how to program the peripherals.

Corresponding to the direct register access model example, the following call also programs the `CR0` register in the `SSI` module (though the register name is hidden by the API):

```
SSISConfigSetExpClk(SSI0_BASE, 50000000, SSI_FRF_MOTO_MODE_3,
                    SSI_MODE_MASTER, 1000000, 8);
```

The resulting value in the `CR0` register might not be exactly the same because [SSISConfigSetExpClk\(\)](#) may compute a different value for the `SCR` bit field than what was used in the direct register access model example.

All example applications use the software driver model.

driver library. The drivers combine to form the software driver model.

## 2.4 Combining The Models

The direct register access model and software driver model can be used together in a single application, thus applying the most appropriate model as needed to any particular situation within the



application. For example, the software driver model can be used to configure the peripherals (because this is not performance critical) and the direct register access model can be used to operate the peripheral (which may be more performance critical). Or, the software driver model can be used for peripherals that are not performance critical (such as a UART used for data logging) and the direct register access model can be used for performance critical peripherals (such as the ADC module used to capture real-time analog data).



## 3 Analog to Digital Converter (ADC)

Introduction .....	11
API Functions .....	12
Programming Example .....	15

### 3.1 Introduction

The analog to digital converter (ADC) API provides a set of functions to deal with the ADC. Functions are provided to configure the ADC, read the captured data, register an interrupt handler.

The ADC supports 14-bit analog-to-digital conversion with up to 12 effective number of bits (ENOBs). The ADC includes an analog multiplexer with up to eight individually configurable channels and a reference voltage generator.

The main features of the ADC are as follows:

- Selectable decimation rates, which also set the effective resolution (7 to 12 bits)
- Eight individual input channels, single-ended or differential
- Reference voltage selectable as internal, external single-ended, external differential, or AVDD5
- Interrupt request generation
- DMA triggers at end of conversions
- Temperature sensor input
- Battery measurement capability

The signals on the PA pins can be used as ADC inputs. In the following discussion, these port pins are referred to as the AIN0,...,AIN7 pins. The input pins AIN0,...,AIN7 are connected to the ADC. Configure the pins PA0,...,PA7 for use as ADC inputs, as analog inputs with no pullup or pulldown enabled.

The inputs can be configured as single-ended or differential inputs. Differential inputs consist of the input pairs AIN0,...,AIN1, AIN2,...,AIN3, AIN4,...,AIN5, and AIN6,...,AIN7.

**Note:**

No negative supply can be applied to these pins, nor can a supply higher than VDD (unregulated power) be applied. The difference between the pins of each pair is converted in differential mode.

In addition to input pins AIN0,...,AIN7, the output of an on-chip temperature sensor can be selected as an input to the ADC for temperature measurements

It is also possible to select a voltage corresponding to AVDD5/3 as an ADC input. This input allows the implementation of, for example, a battery monitor in applications where this feature is required. The reference in this case must not depend on the battery voltage; for example, the AVDD5 voltage is not a valid reference.

The ADC can be programmed to perform a single conversion from any channel.

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

## 3.2 API Functions

### Functions

- `uint16_t SOCADCDataGet (void)`
- `bool SOCADCEndOfConversionGet (void)`
- `void SOCADCIntRegister (void (*pfnHandler)(void))`
- `void SOCADCIntUnregister (void)`
- `void SOCADCSingleConfigure (uint32_t ui32Resolution, uint32_t ui32Reference)`
- `void SOCADCSingleStart (uint32_t ui32Channel)`

### 3.2.1 Detailed Description

The ADC API is broken into two groups of functions: those that deal with the sample configuration and data access and those that deal with interrupt handling.

The ADC can be configured for a single conversion using `SOCADCSingleConfigure()`. The conversion is started using `SOCADCSingleStart()`. Availability of data is queried using `SOCADCEndOfConversionGet()`, while the actual sample data can be retrieved using `SOCADCDataGet()`.

The interrupt handler for the ADC sample sequencer interrupts are managed with `SOCADCIntRegister()` and `SOCADCIntUnregister()`.

### 3.2.2 Function Documentation

#### 3.2.2.1 SOCADCDataGet

Get data value from conversion

**Prototype:**

```
uint16_t
SOCADCDataGet (void)
```

**Description:**

This function gets the latest conversion data result of the programmed conversion. The function returns 16 bits of data, but depending on the programmed precision, only part of the data is significant. The following defined bit masks can be used to extract the significant data depending on the decimation rate: **SOCADC\_7\_BIT\_MASK**, **SOCADC\_9\_BIT\_MASK**, **SOCADC\_10\_BIT\_MASK** and **SOCADC\_12\_BIT\_MASK**

**See also:**

[SOCADCEndOfConversionGet\(\)](#).

**Returns:**

Data conversion value

### 3.2.2.2 SOCADCEndOfConversionGet

Check if conversion is done

**Prototype:**

```
bool  
SOCADCEndOfConversionGet(void)
```

**Description:**

This function can be used to query the status of the conversion.

**Returns:**

true if conversion is done, otherwise false.

### 3.2.2.3 SOCADCIntRegister

Registers an interrupt handler for ADC interrupt

**Prototype:**

```
void  
SOCADCIntRegister(void (*pfnHandler)(void))
```

**Parameters:**

***pfnHandler*** is a pointer to the function called when the SOC ADC interrupt occurs.

**Description:**

This function does the actual registering of the interrupt handler, which enables the global interrupt in the interrupt controller.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 3.2.2.4 SOCADCIntUnregister

Unregisters an interrupt handler for the ADC interrupt

**Prototype:**

```
void  
SOCADCIntUnregister(void)
```

**Description:**

This function does the actual unregistering of the interrupt handler. This function clears the handler to be called when an ADC interrupt occurs and masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 3.2.2.5 SOCADCSingleConfigure

Configure ADC conversion for a single channel

**Prototype:**

```
void
SOCADCSingleConfigure(uint32_t ui32Resolution,
                      uint32_t ui32Reference)
```

**Parameters:**

**ui32Resolution** is the resolution of the conversion.

**ui32Reference** is the reference voltage to be used for the conversion.

**Description:**

This function configures the ADC for a single channel conversion. The *ui32Resolution* parameter must be one of: **SOCADC\_7\_BIT**, **SOCADC\_9\_BIT**, **SOCADC\_10\_BIT** or **SOCADC\_12\_BIT**. The reference voltage is set using the *ui32Reference* parameter, which must be configured as one of the following: **SOCADC\_REF\_INTERNAL** for internal reference, **SOCADC\_REF\_EXT\_AIN7** for external reference on pin AIN7 (pad PA7), **SOCADC\_REF\_AVDD5** for external AVDD5 pin, **SOCADC\_REF\_EXT\_AIN67** for external reference on differential input pins AIN6-AIN7 (Pads PA6-PA7).

**Note:**

A single conversion triggers an interrupt if this has been registered using [SOCADCIntRegister\(\)](#).

**See also:**

[SOCADCSingleStart\(\)](#) and [SOCADCIntRegister\(\)](#).

**Returns:**

None

### 3.2.2.6 SOCADCSingleStart

Start a configured single conversion

**Prototype:**

```
void
SOCADCSingleStart(uint32_t ui32Channel)
```

**Parameters:**

**ui32Channel** is the input channel to use for the conversion.

**Description:**

This function initiates a configured single channel conversion. The input channel is set using the *ui32Channel* parameter. This parameter must be configured as one of the following values: **SOCADC\_AIN0** for single ended input Pad PA0 **SOCADC\_AIN1** for single ended input Pad PA1 **SOCADC\_AIN2** for single ended input Pad PA2 **SOCADC\_AIN3** for single ended input Pad PA3 **SOCADC\_AIN4** for single ended input Pad PA4 **SOCADC\_AIN5** for single ended input Pad PA5 **SOCADC\_AIN6** for single ended input Pad PA6 **SOCADC\_AIN7** for single ended input Pad PA7 **SOCADC\_AIN01** for differential Pads PA0-PA1 **SOCADC\_AIN23** for differential

Pads PA2-PA3 **SOCADC\_AIN45** for differential Pads PA4-PA5 **SOCADC\_AIN67** for differential Pads PA6-PA7 **SOCADC\_GND** for Ground as input **SOCADC\_TEMP\_SENS** for on-chip temperature sensor **SOCADC\_VDD** for Vdd/3

**Note:**

A single conversion triggers an interrupt if this has been registered using [SOCADCIntRegister\(\)](#).

**See also:**

[SOCADCSingleConfigure\(\)](#) and [SOCADCIntRegister\(\)](#).

**Returns:**

None

## 3.3 Programming Example

The following example shows how to use the ADC API.

```
uint16_t uilDummy;

//
// Configure ADC, Internal reference, 512 decimation rate (12bit)
//
SOCADCSingleConfigure(SOCADC_12_BIT, SOCADC_REF_INTERNAL);

//
// Trigger single conversion on AIN6 (connected to LV_ALS_OUT).
//
SOCADCSingleStart(SOCADC_AIN6);

//
// Wait until conversion is completed
//
while(!SOCADCEndOfConversionGet())
{
}

//
// Get data and shift down based on decimation rate
//
uilDummy = SOCADCDataGet() >> SOCADC_12_BIT_RSHIFT;
```





## 4 Advanced Encryption Standard Electronic Code Book mode (AES)

Introduction .....	17
API Functions .....	17
Programming Example .....	19

### 4.1 Introduction

The Advanced Encryption Standard Electronic Code Book mode (AES) and is a specification for the encryption of electronic data. The AES API provides a set of functions for using the AES-ECB mode that configure AES-ECB operation, configure and handle the AES interrupt.

The AES engine is forced to use keys from the key store module for its operations. A key is provided to the AES engine by triggering the key store module to read an AES key from the key store memory and write it to the AES engine.

The AES can set error bits when a DMA operation fails or key store fails.

To check for completion of the AES-ECB operation an interrupt handler can be used or status bits can be polled.

Please Note that the same AES engine is used for Key Load, SHA256, AES-ECB, and AES-CCM operation. The calling function must protect the hardware and implement the AES mutex so that the AES engine performs only one operation at a time.

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

### 4.2 API Functions

#### Functions

- `uint8_t AESECBCheckResult (void)`
- `uint8_t AESECBGetResult (void)`
- `uint8_t AESECBStart (uint8_t *pui8MsgIn, uint8_t *pui8MsgOut, uint8_t ui8KeyLocation, uint8_t ui8Encrypt, uint8_t ui8IntEnable)`
- `uint8_t AESLoadKey (uint8_t *pui8Key, uint8_t ui8KeyLocation)`

#### 4.2.1 Detailed Description

The AES API has the following four functions:

- `AESLoadKey()` writes the key into the Key Ram. Key Ram location should be specified.
- `AESECBStart()` starts an AES-ECB operation.
- `AESECBCheckResult()` is called to check the result of AES-ECB operation.

- [AESECBGetResult\(\)](#) is to get the result of AES-ECB operation.

## 4.2.2 Function Documentation

### 4.2.2.1 AESECBCheckResult

AESECBCheckResult is called to check the result of AES-ECB AESECBStart operation.

**Prototype:**

```
uint8_t
AESECBCheckResult(void)
```

**Returns:**

if result is available or error occurs returns true. If result is not yet available or no error occurs returns false

### 4.2.2.2 uint8\_t AESECBGetResult (void)

AESECBGetResult gets the result of the AES ECB operation. This function must only be called after AESECBStart function is called.

**Returns:**

AES\_SUCCESS if successful.

### 4.2.2.3 uint8\_t AESECBStart (uint8\_t \* *pui8MsgIn*, uint8\_t \* *pui8MsgOut*, uint8\_t *ui8KeyLocation*, uint8\_t *ui8Encrypt*, uint8\_t *ui8IntEnable*)

AESECBStart starts an AES-ECB operation.

**Parameters:**

***pui8MsgIn*** is pointer to input data.

***pui8MsgOut*** is pointer to output data.

***ui8KeyLocation*** is the location in Key RAM.

***ui8Encrypt*** is set 'true' to ui8Encrypt or set 'false' to decrypt.

***ui8IntEnable*** is set 'true' to enable AES interrupts or 'false' to disable AES interrupt.

**Description:**

The *ui8KeyLocation* parameter is an enumerated type which specifies the Key Ram location in which the key is stored. This parameter can have any of the following values:

- KEY\_AREA\_0
- KEY\_AREA\_1
- KEY\_AREA\_2,
- KEY\_AREA\_3,
- KEY\_AREA\_4,
- KEY\_AREA\_5,
- KEY\_AREA\_6,

#### ■ KEY\_AREA\_7

**Returns:**

AES\_SUCCESS if successful.

#### 4.2.2.4 AESLoadKey

AESLoadKey writes the key into the Key Ram. Key Ram location must be specified.

**Prototype:**

```
uint8_t  
AESLoadKey(uint8_t *pui8Key,  
            uint8_t ui8KeyLocation)
```

**Parameters:**

***pui8Key*** is pointer to AES Key.

***ui8KeyLocation*** is location in Key RAM.

**Description:**

The *ui8KeyLocation* parameter is an enumerated type which specifies the Key Ram location in which the key is stored. This parameter can have any of the following values:

- KEY\_AREA\_0
- KEY\_AREA\_1
- KEY\_AREA\_2,
- KEY\_AREA\_3,
- KEY\_AREA\_4,
- KEY\_AREA\_5,
- KEY\_AREA\_6,
- KEY\_AREA\_7

The pointer *pui8Key* has the address where the Key is stored.

**Returns:**

AES\_SUCCESS if successful.

## 4.3 Programming Example

For AES example code, please refer to the example at `examples/aes`.



## 5 Advanced Encryption Standard Counter with CBC-MAC (CCM)

Introduction .....	21
API Functions .....	21
Programming Example .....	25

### 5.1 Introduction

The Counter with CBC-MAC (CCM) mode of operation for cryptographic block ciphers is an authenticated encryption algorithm designed to provide both authentication and confidentiality. CCM mode is only defined for block ciphers with a block length of 128 bits.

As the name suggests, CCM mode combines the well-known counter mode of encryption with the well-known CBC-MAC mode of authentication. The key insight is that the same encryption key can be used for both, provided that the counter values used in the encryption do not collide with the (pre-)initialization vector used in the authentication. A proof of security exists for this combination, based on the security of the underlying block cipher. The proof also applies to a generalization of CCM for any size block cipher, and for any size cryptographically strong pseudo-random function (since in both counter mode and CBC-MAC, the block cipher is only ever used in one direction).

The CCM API provides a set of functions for using the CCM mode. Functions are provided to program and configure CCM operation, and configure the AES-CCM interrupt.

The AES engine is forced to use keys from the key store module for its operations. A key is provided to the AES engine by triggering the key store module to read an AES key from the key store memory and write it to the AES engine. The [AESLoadKey\(\)](#) in the file `aes.c`, can be used for loading keys.

The AES engine can set error bits when a DMA operation fails or key store fails.

An interrupt handler can be used or status bits can be polled to verify the completion of the AES-CCM operation.

Please Note that the same AES engine is used for Key Load, SHA256, AES-ECB, and AES-CCM operation. The calling function must protect the hardware and implement the AES mutex so that the AES engine performs only one operation at a time.

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

### 5.2 API Functions

#### Functions

- `uint8_t CCMAuthEncryptCheckResult` (void)
- `uint8_t CCMAuthEncryptGetResult` (uint8\_t ui8Mval, uint16\_t ui16LenM, uint8\_t \*pui8Cstate)
- `uint8_t CCMAuthEncryptStart` (bool bEncrypt, uint8\_t ui8Mval, uint8\_t \*pui8N, uint8\_t \*pui8M, uint16\_t ui16LenM, uint8\_t \*pui8A, uint16\_t ui16LenA, uint8\_t ui8KeyLocation, uint8\_t \*pui8Cstate, uint8\_t ui8CCMLVal, uint8\_t ui8IntEnable)

- `uint8_t CCMLnvAuthDecryptCheckResult` (void)
- `uint8_t CCMLnvAuthDecryptGetResult` (`uint8_t ui8Mval`, `uint8_t *pui8C`, `uint16_t ui16LenC`, `uint8_t *pui8Cstate`)
- `uint8_t CCMLnvAuthDecryptStart` (`bool bDecrypt`, `uint8_t ui8Mval`, `uint8_t *pui8N`, `uint8_t *pui8C`, `uint16_t ui16LenC`, `uint8_t *pui8A`, `uint16_t ui16LenA`, `uint8_t ui8KeyLocation`, `uint8_t *pui8Cstate`, `uint8_t ui8CCMLVal`, `uint8_t ui8IntEnable`)

## 5.2.1 Detailed Description

The CCM API can be broken into the following two sets of functions:

- The first is Authentication + Encryption.
- The second is Decryption + Inverse Authentication.

Authentication + Encryption APIs:

- `CCMAuthEncryptStart()` to start CCM Authentication + Encryption operation.
- `CCMAuthEncryptCheckResult()` to check CCM operation.
- `CCMAuthEncryptGetResult()` to get the result of CCM operation.

Decryption + Inverse Authentication APIs:

- `CCMLnvAuthDecryptStart()` to start CCM Decryption + Inverse Authentication.
- `CCMLnvAuthDecryptCheckResult()` function checks CCM operation.
- `CCMLnvAuthDecryptGetResult()` to get the result of CCM operation.

## 5.2.2 Function Documentation

### 5.2.2.1 CCMAuthEncryptCheckResult

`CCMAuthEncryptCheckResult` checks the status of CCM encrypt operation.

**Prototype:**

```
uint8_t
CCMAuthEncryptCheckResult (void)
```

**Returns:**

if result is available or error occurs, function returns true. If result is not yet available or no error occurs, returns false

### 5.2.2.2 `uint8_t CCMAuthEncryptGetResult (uint8_t ui8Mval, uint16_t ui16LenM, uint8_t *pui8Cstate)`

`CCMAuthEncryptGetResult` gets the result of CCM operation. This function should be called after `CCMAuthEncryptStart` is called.

**Parameters:**

***ui8Mval*** is length of authentication field in octets [0, 2, 4, 6, 8, 10, 12, 14 or 16].

***ui16LenM*** is length of message *pui8M*[] in octets.

***pui8Cstate*** is pointer to AES state buffer.

**Returns:**

AES\_SUCCESS if successful.

5.2.2.3 uint8\_t CCMAuthEncryptStart (bool *bEncrypt*, uint8\_t *ui8Mval*, uint8\_t \* *pui8N*, uint8\_t \* *pui8M*, uint16\_t *ui16LenM*, uint8\_t \* *pui8A*, uint16\_t *ui16LenA*, uint8\_t *ui8KeyLocation*, uint8\_t \* *pui8Cstate*, uint8\_t *ui8CCMLVal*, uint8\_t *ui8IntEnable*)

CCMAuthEncryptStart starts the CCM operation

**Parameters:**

***bEncrypt*** if set to 'true' then run encryption and set to 'false' for authentication only.

***ui8Mval*** is the length of authentication field in octets [0, 2, 4, 6, 8, 10, 12, 14 or 16].

***pui8N*** is the pointer to 13-byte or 12-byte Nonce.

***pui8M*** is the pointer to octet string 'm'/input message.

***ui16LenM*** is the length of *pui8M*[] in octets.

***pui8A*** is the pointer to octet string 'a'.

***ui16LenA*** is the Length of *pui8A*[] in octets.

***ui8KeyLocation*** is the location where the Key is stored in Key RAM.

***pui8Cstate*** is the pointer to output buffer.

***ui8CCMLVal*** is the ccm L Value to be used.

***ui8IntEnable*** if set to 'true' to enable interrupts or 'false' to disable interrupts. Should be 'false' if *bEncrypt* is set to 'false'.

**Description:**

The function will place in *pui8Cstate* the first *ui8Mval* bytes containing the Authentication Tag.

The *ui8KeyLocation* parameter is an enumerated type which specifies the Key Ram location in which the key is stored. This parameter can have any of the following values:

- KEY\_AREA\_0
- KEY\_AREA\_1
- KEY\_AREA\_2,
- KEY\_AREA\_3,
- KEY\_AREA\_4,
- KEY\_AREA\_5,
- KEY\_AREA\_6,
- KEY\_AREA\_7

**Returns:**

AES\_SUCCESS if successful.

### 5.2.2.4 CCMLInvAuthDecryptCheckResult

CCMLInvAuthDecryptCheckResult function checks CCM decrypt and Inverse Authentication result.

**Prototype:**

```
uint8_t
CCMLInvAuthDecryptCheckResult (void)
```

**Returns:**

if result is available or error occurs returns true. If result is not yet available or no error occurs returns false

### 5.2.2.5 uint8\_t CCMLInvAuthDecryptGetResult (uint8\_t *ui8Mval*, uint8\_t \* *pui8C*, uint16\_t *ui16LenC*, uint8\_t \* *pui8Cstate*)

CCMLInvAuthDecryptGetResult gets the result of CCM operation. This function should be called only after CCMLInvAuthDecryptStart is called.

**Parameters:**

***ui8Mval*** is length of authentication field in octets [0, 2, 4, 6, 8, 10, 12, 14 or 16].  
***pui8C*** is pointer to octet string 'c' = 'm' || auth tag T.  
***ui16LenC*** is length of message *pui8C*[] in octets.  
***pui8Cstate*** is pointer to AES state buffer, cannot be part of *pui8C*[]).

**Returns:**

AES\_SUCCESS if successful.

### 5.2.2.6 uint8\_t CCMLInvAuthDecryptStart (bool *bDecrypt*, uint8\_t *ui8Mval*, uint8\_t \* *pui8N*, uint8\_t \* *pui8C*, uint16\_t *ui16LenC*, uint8\_t \* *pui8A*, uint16\_t *ui16LenA*, uint8\_t *ui8KeyLocation*, uint8\_t \* *pui8Cstate*, uint8\_t *ui8CCMLVal*, uint8\_t *ui8IntEnable*)

CCMLInvAuthDecryptStart starts the CCM Decryption and Inverse Authentication operation.

**Parameters:**

***bDecrypt*** if set to 'true' then run decryption, set to 'false' if authentication only  
***ui8Mval*** is the length of authentication field in octets [0, 2, 4, 6, 8, 10, 12, 14 or 16].  
***pui8N*** is the pointer to 13-byte or 12-byte Nonce.  
***pui8C*** is the pointer to octet string 'c' = 'm' || auth tag T.  
***ui16LenC*** is the length of *pui8C*[] in octets.  
***pui8A*** is the pointer to octet string 'a'.  
***ui16LenA*** is the Length of *pui8A*[] in octets.  
***ui8KeyLocation*** is the location where the Key is stored in Key RAM.  
***pui8Cstate*** is the pointer to output buffer. (cannot be part of *pui8C*[]).  
***ui8CCMLVal*** is the ccm L Value to be used.  
***ui8IntEnable*** if set to 'true' to enable interrupts or 'false' to disable interrupts. Set to 'false' if *bDecrypt* is set to 'false'.



**Description:**

The function will place in *pui8Cstate* the first *ui8Mval* bytes of containing the Authentication Tag.

The *ui8KeyLocation* parameter is an enumerated type which specifies the Key Ram location in which the key is stored. This parameter can have any of the following values:

- KEY\_AREA\_0
- KEY\_AREA\_1
- KEY\_AREA\_2,
- KEY\_AREA\_3,
- KEY\_AREA\_4,
- KEY\_AREA\_5,
- KEY\_AREA\_6,
- KEY\_AREA\_7

**Returns:**

AES\_SUCCESS if successful.

## 5.3 Programming Example

The CCM example, please refer to the example at `examples/ccm`.



## 6 Flash

Introduction .....	27
API Functions .....	28
Programming Example .....	32

### 6.1 Introduction

The CC2538 flash provides in-circuit programmable nonvolatile program memory for the device. The flash memory is organized as a set of 2kB pages that can be individually erased. Erasing a page causes the entire contents of the page to be reset to all 1s. These pages can be individually protected. Read-only pages cannot be erased or programmed, protecting the contents of those pages from being modified. In addition to holding program code and constants, the nonvolatile memory allows the application to save data that must be preserved such that it is available after restarting the device. Device configuration and page protection bits are stored in the (upper) top page of the flash. Program data is stored in the remaining (main) pages. Please refer to the device User's Guide for the contents and layout of the configuration page.

Depending on the member of the CC2538 family used, the amount of available flash is 128kB, 256kB, or 512kB. The location of the upper page is thus relative to the size of the flash.

A flash page is the smallest erasable unit in the memory, whereas a 32-bit word is the smallest writable unit that can be written to the flash. In general, a page must be erased before writing can begin. The page-erase operation sets all bits in the page to 1. The chip-erase command (through the debug interface) erases all pages in the flash. Erasing is the only way to set bits in the flash to 1. When writing a word to the flash, the 0-bits are programmed to 0 and the 1-bits are ignored (leaves the bit in the flash unchanged). Thus, bits are erased to 1 and can be written to 0. When writing multiple times to a word, some limitations must be observed. Please refer to the internal memory chapter in the device User's Guide for further details.

To optimize flash power consumption and access time, the flash system contains a caching feature. The cache has several operating modes:

- Enabling the cache decreases power consumption and increases read performance.
- Cache with prefetching improves performance at the expense of a potential increase in power consumption.
- Real-time mode provides predictable flash read access time, the execution time is equal to cache disabled mode, but the power consumption is lower.
- Disabling the cache increases the power consumption and reduces performance.

The flash API provides a set of functions to deal with the on-chip flash. These functions program and erase the flash, and configure the cache system.

**Note:**

The core of the flash programming code is stored in ROM because executing code from flash while programming the flash is problematic with the single bank flash.

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

## 6.2 API Functions

### Functions

- uint32\_t [FlashCacheModeGet](#) (void)
- void [FlashCacheModeSet](#) (uint32\_t ui32CacheMode)
- uint32\_t [FlashGet](#) (uint32\_t ui32Addr)
- int32\_t [FlashMainPageErase](#) (uint32\_t ui32Address)
- int32\_t [FlashMainPageProgram](#) (uint32\_t \*pui32Data, uint32\_t ui32Address, uint32\_t ui32Count)
- uint32\_t [FlashSizeGet](#) (void)
- uint32\_t [FlashSramSizeGet](#) (void)
- int32\_t [FlashUpperPageErase](#) (void)
- int32\_t [FlashUpperPageProgram](#) (uint32\_t \*pui32Data, uint32\_t ui32Address, uint32\_t ui32Count)

### 6.2.1 Detailed Description

The flash API is broken into two groups of functions: those that deal with programming the flash, those that deal with flash cache configuration. deal with interrupt handling.

Flash programming is managed with the following functions:

- [FlashMainPageErase\(\)](#)
- [FlashMainPageProgram\(\)](#)
- [FlashUpperPageErase\(\)](#)
- [FlashUpperPageProgram\(\)](#)

Flash cache configuration is managed with the following functions:

- [FlashCacheModeGet\(\)](#)
- [FlashCacheModeSet\(\)](#)

Device specific configuration such as flash and SRAM size can be accessed using the following functions:

- [FlashSizeGet\(\)](#)
- [FlashSramSizeGet\(\)](#)

### 6.2.2 Function Documentation

#### 6.2.2.1 FlashCacheModeGet

Gets the current state of the flash Cache Mode

**Prototype:**

```
uint32_t  
FlashCacheModeGet (void)
```

**Description:**

This function gets the current setting for the Cache Mode.

**Returns:**

Returns the CM bits. Return value should match one of the FLASH\_CACHE\_MODE\_<> macros defined in flash.h.

### 6.2.2.2 FlashCacheModeSet

Sets the flash Cache Mode state

**Prototype:**

```
void  
FlashCacheModeSet (uint32_t ui32CacheMode)
```

**Parameters:**

***ui32CacheMode*** is the desired cache mode.

**Description:**

This function sets the flash Cache Mode to the desired state and accepts a right justified 2 bit setting for the Cachemode bits. The function waits for the flash to be idle, reads the FCTL register contents, masks in the requested setting, and writes it into the FCTL register.

The parameter *ui32CacheMode* can have one of the following values:

- FLASH\_CTRL\_CACHE\_MODE\_DISABLE
- FLASH\_CTRL\_CACHE\_MODE\_ENABLE
- FLASH\_CTRL\_CACHE\_MODE\_PREFETCH\_ENABLE
- FLASH\_CTRL\_CACHE\_MODE\_REALTIME

**Returns:**

None

### 6.2.2.3 FlashGet

Gets the current contents of the flash at the designated address

**Prototype:**

```
uint32_t  
FlashGet (uint32_t ui32Addr)
```

**Parameters:**

***ui32Addr*** is the desired address to be read within the flash.

**Description:**

This function helps differentiate flash memory reads from flash register reads.

**Returns:**

Returns the 32bit value as an uint32\_t value.

### 6.2.2.4 FlashMainPageErase

Erases a flash main page with use of ROM function

**Prototype:**

```
int32_t  
FlashMainPageErase(uint32_t ui32Address)
```

**Parameters:**

**ui32Address** is the start address of the flash main page to be erased.

**Description:**

This function erases one 2 kB main page of the on-chip flash. After erasing, the page is filled with 0xFF bytes. Locked pages cannot be erased. The flash main pages do not include the upper page.

This function does not return until the page is erased or an error encountered.

**Returns:**

Returns 0 on success, -1 if erasing error is encountered, or -2 in case of illegal parameter use.

### 6.2.2.5 FlashMainPageProgram

Programs the flash main pages by use of ROM function

**Prototype:**

```
int32_t  
FlashMainPageProgram(uint32_t *pui32Data,  
                     uint32_t ui32Address,  
                     uint32_t ui32Count)
```

**Parameters:**

**pui32Data** is a pointer to the data to be programmed.

**ui32Address** is the starting address in flash to be programmed. Must be a multiple of four and within the flash main pages.

**ui32Count** is the number of bytes to be programmed. Must be a multiple of four.

**Description:**

This function programs a sequence of words into the on-chip flash. Programming each location consists of the result of an AND operation of the new data and the existing data; in other words, bits that contain 1 can remain 1 or be changed to 0, but bits that are 0 cannot be changed to 1. Therefore, a word can be programmed multiple times as long as these rules are followed; if a program operation attempts to change a 0 bit to a 1 bit, that bit will not have its value changed.

Because the flash is programmed one word at a time, the starting address and byte count must both be multiples of four. The caller must verify the programmed contents, if verification is required.

This function does not return until the data is programmed or an error encountered. Locked flash pages cannot be programmed.

**Returns:**

Returns 0 on success, -1 if a programming error is encountered or, -2 in case of illegal parameter use.

### 6.2.2.6 FlashSizeGet

Returns the flash size in number of KBytes

**Prototype:**

```
uint32_t  
FlashSizeGet(void)
```

**Description:**

This function returns the size of the flash in KBytes as determined by examining the FLASH\_DIECFG0 register settings.

**Returns:**

Returns the flash size in KBytes

### 6.2.2.7 FlashSramSizeGet

Returns the SRAM size in number of KBytes

**Prototype:**

```
uint32_t  
FlashSramSizeGet(void)
```

**Description:**

This function returns the size of the SRAM in KBytes as determined by examining the FLASH\_DIECFG0 register settings.

**Returns:**

Returns the SRAM size in KBytes

### 6.2.2.8 FlashUpperPageErase

Erases the upper flash page with use of ROM function

**Prototype:**

```
int32_t  
FlashUpperPageErase(void)
```

**Description:**

This function erases the 2 kB upper page of the on-chip flash. After erasing, the page is filled with 0xFF bytes. A locked page cannot be erased.

This function does not return until the flash page is erased or an error encountered.

**Returns:**

Returns 0 on success, -1 if erasing error is encountered or, -2 in case of illegal parameter use.

### 6.2.2.9 FlashUpperPageProgram

Programs the upper page of the flash by use of ROM function

**Prototype:**

```
int32_t  
FlashUpperPageProgram(uint32_t *pui32Data,  
                      uint32_t ui32Address,  
                      uint32_t ui32Count)
```

**Parameters:**

**pui32Data** is a pointer to the data to be programmed.

**ui32Address** is the starting address within the flash upper page to be programmed. Must be a multiple of four and within the flash upper page.

**ui32Count** is the number of bytes to be programmed. Must be a multiple of four.

**Description:**

This function programs a sequence of words into the on-chip flash. Programming each location consists of the result of an AND operation of the new data and the existing data; in other words, bits that contain 1 can remain 1 or be changed to 0, but bits that are 0 cannot be changed to 1. Therefore, a word can be programmed multiple times as long as these rules are followed; if a program operation attempts to change a 0 bit to a 1 bit, that bit will not have its value changed.

Because the flash is programmed one word at a time, the starting address and byte count must both be multiples of four. The caller must verify the programmed contents, if such verification is required.

This function does not return until the data is programmed or an error encountered. A locked flash page cannot be programmed.

**Returns:**

Returns 0 on success, -1 if a programming error is encountered or, -2 in case of illegal parameter use.

## 6.3 Programming Example

The following example shows how to use the flash API to erase a page of the flash and program a few words.

```
uint32_t pulData[2];  
  
//  
// Erase a 2k page of the flash.  
//  
FlashMainPageErase(0x00202000);  
  
//  
// Program some data into the newly erased pages of the flash.  
//  
pulData[0] = 0x12345678;  
pulData[1] = 0x56789abc;  
FlashMainPageProgram(pulData, 0x00202000, sizeof(pulData));
```



# 7 General-Purpose Inputs/Outputs (GPIO)

Introduction .....	33
API Functions .....	34
Programming Example .....	50

## 7.1 Introduction

The GPIO module is composed of four physical GPIO blocks, each corresponding to an individual GPIO port (A, B, C and D). The GPIO module supports 32 programmable I/O pins. The device is configured for individual access to each of these ports. Additionally, each GPIO block allows for single read and write operations for individual GPIO bits. The GPIO module has the following features:

- Up to 32 GPIOs, depending on configuration
- Highly flexible pin muxing allows use as GPIO or one of several peripheral functions
- Fast toggle capable of a change every two clock cycles
- Programmable control for GPIO interrupts:
  - Interrupt generation masking
  - Edge-triggered on rising, falling, or both
  - Level-sensitive on high or low values
- Bit masking in read and write operations through address lines

Each GPIO can operate in one of 3 modes:

- Software controlled input
- Software controlled output
- Hardware controlled

When programmed in software control mode, the GPIO operates as a software programmable input or output, where the value on an external pin can be set or read using the state access functions [GPIOPinRead\(\)](#) and [GPIOPinWrite\(\)](#). In hardware control mode, the GPIOs I/O pin is controlled by an on-chip peripherals external pin. The pin muxing from pin to peripheral is controlled by the IOC module. To route a GPIO input or output to and from a peripheral, configure the muxing matrix using the [IOCPinConfigPeriphInput\(\)](#) and [IOCPinConfigPeriphOutput\(\)](#) functions. The operation mode of a GPIO pin, software or hardware controlled, is configured using the [GPIODirModeSet\(\)](#) function.

The interrupt capabilities of each GPIO port are controlled by a set by the [GPIOIntTypeSet\(\)](#) function. This function selects the source of the interrupt, its polarity, and the edge detection properties. There are four GPIO interrupts, one for each port (A, B, C, and D). When one or more GPIO inputs cause an interrupt, a single interrupt output is sent to the interrupt controller (NVIC) for the entire GPIO port. For edge-triggered interrupts, software must clear the interrupt to enable any further interrupts. For a level-sensitive interrupt, the external source must hold the level constant for the interrupt to be recognized by the INTC.

Each GPIO pin can be set up to trigger a **power-up** interrupt that wakes up the device from the various power modes. In addition to the GPIO pins, both USB and sleep timer can be used to trigger power-up interrupts. To enable the power-up interrupt, write the corresponding bits in the SYS\_CTRL\_IWE register. To use the power-up interrupt on a GPIO port, USB, or sleep timer, perform the following:

1. Set up GPIO pin(s) as input (if a GPIO port is used as power-up interrupt source) using [GPIODirModeSet\(\)](#).
2. Enable wake up interrupt for the desired port using the SYS\_CTRL\_IWE register.
3. Set the power-up interrupt type for the specified pin(s), using [GPIOPowIntTypeSet\(\)](#).
4. Clear any pending GPIO power-up interrupts for the specified pin(s) using [GPIOPowIntClear\(\)](#).
5. Enable power-up interrupt for the specified port, using [GPIOPowIntEnable\(\)](#).
6. The device can now be put to sleep (all power modes) and will wake up on the configured interrupt.

Most of the GPIO functions can operate on more than one GPIO pin (within a single module) at a time. The *ui8Pins* parameter to these functions specifies the pins affected: only the GPIO pins corresponding to the bits in this parameter that are set are affected (where pin 0 is bit 0, pin 1 in bit 1, and so on). For example, if *ui8Pins* is 0x09, then pins 0 and 3 are affected by the function.

This protocol is most useful for the [GPIOPinRead\(\)](#) and [GPIOPinWrite\(\)](#) functions; a read returns only the values of the requested pins (with the other pin values masked out) and a write only affects the requested pins simultaneously (that is, the state of multiple GPIO pins can be changed at the same time). This data masking for the GPIO pin state occurs in the hardware; a single read or write is issued to the hardware, which interprets some of the address bits as an indication of the GPIO pins to operate upon (and therefore the ones to not affect). See the part data sheet for details of the GPIO data register address-based bit masking.

For functions that have a *ui8Pin* (singular) parameter, only a single pin is affected by the function. In this case, the value specifies the pin number (that is, 0 through 7).

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

## 7.2 API Functions

### Functions

- `uint32_t GPIODirModeGet (uint32_t ui32Port, uint8_t ui8Pin)`
- `void GPIODirModeSet (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32PinIO)`
- `uint32_t GPIOIntTypeGet (uint32_t ui32Port, uint8_t ui8Pin)`
- `void GPIOIntTypeSet (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32IntType)`
- `void GPIOIntWakeupDisable (uint32_t ui32Config)`
- `void GPIOIntWakeupEnable (uint32_t ui32Config)`
- `void GPIOPinIntClear (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinIntDisable (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinIntEnable (uint32_t ui32Port, uint8_t ui8Pins)`
- `uint32_t GPIOPinIntStatus (uint32_t ui32Port, bool bMasked)`
- `uint32_t GPIOPinRead (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinTypeGPIOInput (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinTypeGPIOOutput (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinTypeI2C (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinTypeSSI (uint32_t ui32Port, uint8_t ui8Pins)`
- `void GPIOPinTypeTimer (uint32_t ui32Port, uint8_t ui8Pins)`

- void [GPiOPinTypeUARTInput](#) (uint32\_t ui32Port, uint8\_t ui8Pins)
- void [GPiOPinTypeUARTOutput](#) (uint32\_t ui32Port, uint8\_t ui8Pins)
- void [GPiOPinWrite](#) (uint32\_t ui32Port, uint8\_t ui8Pins, uint8\_t ui8Val)
- void [GPiOPortIntRegister](#) (uint32\_t ui32Port, void (\*pfnHandler)(void))
- void [GPiOPortIntUnregister](#) (uint32\_t ui32Port)
- void [GPiOPowIntClear](#) (uint32\_t ui32Port, uint8\_t ui8Pins)
- void [GPiOPowIntDisable](#) (uint32\_t ui32Port, uint8\_t ui8Pins)
- void [GPiOPowIntEnable](#) (uint32\_t ui32Port, uint8\_t ui8Pins)
- uint32\_t [GPiOPowIntStatus](#) (uint32\_t ui32Port, bool bMasked)
- uint32\_t [GPiOPowIntTypeGet](#) (uint32\_t ui32Port, uint8\_t ui8Pin)
- void [GPiOPowIntTypeSet](#) (uint32\_t ui32Port, uint8\_t ui8Pins, uint32\_t ui32IntType)

## 7.2.1 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 operation mode of GPIO pins are configured with [GPiODirModeSet\(\)](#). The configuration can be read back with [GPiODirModeGet\(\)](#).

The following convenience functions configure in the required or recommended configuration for a particular peripheral:

- [GPiOPinTypeGPIOInput\(\)](#)
- [GPiOPinTypeGPIOOutput\(\)](#)
- [GPiOPinTypeI2C\(\)](#)
- [GPiOPinTypeSSI\(\)](#)
- [GPiOPinTypeTimer\(\)](#)
- [GPiOPinTypeUART\(\)](#)

The GPIO interrupts are handled with:

- [GPiOIntTypeSet\(\)](#)
- [GPiOIntTypeGet\(\)](#)
- [GPiOPinIntEnable\(\)](#)
- [GPiOPinIntDisable\(\)](#)
- [GPiOPinIntStatus\(\)](#)
- [GPiOPinIntClear\(\)](#),
- [GPiOPortIntRegister\(\)](#)
- [GPiOPortIntUnregister\(\)](#)

The power up settings are changed using:

- [GPiOPowIntEnable\(\)](#)
- [GPiOPowIntDisable\(\)](#)
- [GPiOPowIntTypeSet\(\)](#)
- [GPiOPowIntTypeGet\(\)](#)

- [GPIOPowIntStatus\(\)](#)
- [GPIOPowIntClear\(\)](#)
- [GPIOIntWakeupEnable\(\)](#)
- [GPIOIntWakeupDisable\(\)](#)

When a GPIO is programmed for software control mode, the pin state is accessed with [GPIOPinRead\(\)](#) and [GPIOPinWrite\(\)](#).

## 7.2.2 Function Documentation

### 7.2.2.1 GPIODirModeGet

Gets the direction and mode of a pin

**Prototype:**

```
uint32_t
GPIODirModeGet (uint32_t ui32Port,
                uint8_t ui8Pin)
```

**Parameters:**

***ui32Port*** is the base address of the GPIO port.  
***ui8Pin*** is the pin number.

**Description:**

This function gets the direction and control mode for a specified pin on the selected GPIO port. The pin can be configured as either an input or output under software control, or it can be under hardware control. The type of control and direction are returned as an enumerated data type.

**Returns:**

Returns one of the enumerated data types described for [GPIODirModeSet\(\)](#).

### 7.2.2.2 GPIODirModeSet

Sets the direction and mode of the specified pin(s)

**Prototype:**

```
void
GPIODirModeSet (uint32_t ui32Port,
                uint8_t ui8Pins,
                uint32_t ui32PinIO)
```

**Parameters:**

***ui32Port*** is the base address of the GPIO port.  
***ui8Pins*** is the bit-packed representation of the pin(s).  
***ui32PinIO*** is the pin direction and/or mode.

**Description:**

This function sets the specified pin(s) on the selected GPIO port as either an input or output under software control or sets the pin to be under hardware control.

The parameter *ui32PinIO* is an enumerated data type that can be one of the following values:

- **GPIO\_DIR\_MODE\_IN**
- **GPIO\_DIR\_MODE\_OUT**
- **GPIO\_DIR\_MODE\_HW**

where **GPIO\_DIR\_MODE\_IN** specifies that the pin will be programmed as a software controlled input, **GPIO\_DIR\_MODE\_OUT** specifies that the pin will be programmed as a software controlled output, and **GPIO\_DIR\_MODE\_HW** specifies that the pin will be placed under hardware control.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.3 GPIOIntTypeGet

Gets the interrupt type for a pin

**Prototype:**

```
uint32_t
GPIOIntTypeGet (uint32_t ui32Port,
                uint8_t ui8Pin)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pin** is the pin number.

**Description:**

This function gets the interrupt type for a specified pin on the selected GPIO port. The pin can be configured as a falling edge, rising edge, or both edge detected interrupt, or can be configured as a low level or high level detected interrupt. The type of interrupt detection mechanism is returned as an enumerated data type.

**Returns:**

Returns one of the enumerated data types described for [GPIOIntTypeSet\(\)](#).

### 7.2.2.4 GPIOIntTypeSet

Sets the interrupt type for the specified pin(s)

**Prototype:**

```
void
GPIOIntTypeSet (uint32_t ui32Port,
                uint8_t ui8Pins,
                uint32_t ui32IntType)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

***ui32IntType*** specifies the type of interrupt trigger mechanism.

**Description:**

This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

The parameter *ui32IntType* is an enumerated data type that can be one of the following values:

- **GPIO\_FALLING\_EDGE**
- **GPIO\_RISING\_EDGE**
- **GPIO\_BOTH\_EDGES**
- **GPIO\_LOW\_LEVEL**
- **GPIO\_HIGH\_LEVEL**

where the different values describe the interrupt detection mechanism (edge or level) and the particular triggering event (falling, rising, or both edges for edge detect, low or high for level detect).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

To avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

**Returns:**

None

## 7.2.2.5 GPIOIntWakeupDisable

Disable Wake Up Interrupt

**Prototype:**

```
void
GPIOIntWakeupDisable(uint32_t ui32Config)
```

**Parameters:**

***ui32Config*** is the source to disable wake on interrupt from.

**Description:**

This function disables Wake up on interrupt from the selected sources.

The *ui32Config* argument must be one or the logical or of several of the following values:

**GPIO\_IWE\_PORT\_A, GPIO\_IWE\_PORT\_B, GPIO\_IWE\_PORT\_C, GPIO\_IWE\_PORT\_D, GPIO\_IWE\_USB, GPIO\_IWE\_SM\_TIMER,**

**Returns:**

None

### 7.2.2.6 GPIOIntWakeupEnable

Enable Wake Up Interrupt

**Prototype:**

```
void  
GPIOIntWakeupEnable(uint32_t ui32Config)
```

**Parameters:**

***ui32Config*** is the source to enable wake up on interrupt.

**Description:**

This function enables wake up on interrupt from the selected sources.

The *ui32Config* argument must be one or the logical or of several of the following values:

**GPIO\_IWE\_PORT\_A, GPIO\_IWE\_PORT\_B, GPIO\_IWE\_PORT\_C, GPIO\_IWE\_PORT\_D, GPIO\_IWE\_USB, GPIO\_IWE\_SM\_TIMER.**

**Returns:**

None

### 7.2.2.7 GPIOPinIntClear

Clears the interrupt for the specified pin(s)

**Prototype:**

```
void  
GPIOPinIntClear(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

**Parameters:**

***ui32Port*** is the base address of the GPIO port.

***ui8Pins*** is the bit-packed representation of the pin(s).

**Description:**

Clears the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

The write buffer in the Cortex-M3 processor can cause the interrupt source to take several clock cycles before clearing. Therefore, TI recommends clearing the interrupt source early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to clear the interrupt source early can result in the interrupt handler being immediately reentered (because NVIC still sees the interrupt source asserted).

**Returns:**

None

### 7.2.2.8 GPIOPinIntDisable

Disables interrupts for the specified pin(s)

**Prototype:**

```
void  
GPIOPinIntDisable(uint32_t ui32Port,  
                  uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

Masks the interrupt for the specified pin(s)

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.9 GPIOPinIntEnable

Enables interrupts for the specified pin(s)

**Prototype:**

```
void  
GPIOPinIntEnable(uint32_t ui32Port,  
                  uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

Unmasks the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.10 GPIOPinIntStatus

Gets interrupt status for the specified GPIO port



**Prototype:**

```
uint32_t
GPIOPinIntStatus(uint32_t ui32Port,
                 bool bMasked)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**bMasked** specifies whether masked or raw interrupt status is returned.

**Description:**

If **bMasked** is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

**Returns:**

Returns a bit-packed byte, where each bit that is set identifies an active masked or raw interrupt, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Bits 31:8 should be ignored.

### 7.2.2.11 GPIOPinRead

Reads the values present of the specified pin(s)

**Prototype:**

```
uint32_t
GPIOPinRead(uint32_t ui32Port,
            uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The values at the specified pin(s) are read, as specified by **ui8Pins**. Values are returned for both input and output pin(s), and the value for pin(s) that are not specified by **ui8Pins** are set to 0.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

Returns a bit-packed byte providing the state of the specified pin, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Any bit that is not specified by **ui8Pins** is returned as a 0. Bits 31:8 should be ignored.

### 7.2.2.12 GPIOPinTypeGPIOInput

Configures pin(s) for use as GPIO inputs

**Prototype:**

```
void  
GPIOPinTypeGPIOInput (uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The GPIO pins must be properly configured in order to function correctly as GPIO inputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.13 GPIOPinTypeGPIOOutput

Configures pin(s) for use as GPIO outputs

**Prototype:**

```
void  
GPIOPinTypeGPIOOutput (uint32_t ui32Port,  
                      uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The GPIO pins must be properly configured to function correctly as GPIO outputs. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.14 GPIOPinTypeI2C

Configures pin(s) for use by the I2C peripheral

**Prototype:**

```
void  
GPIOPinTypeI2C (uint32_t ui32Port,  
               uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The I2C pins must be properly configured for the I2C peripheral to function correctly. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

This function cannot be used to turn any pin into an I2C pin; it only configures an I2C pin for proper operation.

**Returns:**

None

### 7.2.2.15 GPIOPinTypeSSI

Configures pin(s) for use by the SSI peripheral

**Prototype:**

```
void  
GPIOPinTypeSSI(uint32_t ui32Port,  
                uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The SSI pins must be properly configured for the SSI peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations might work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

This function cannot be used to turn any pin into a SSI pin; but only configures an SSI pin for proper operation.

**Returns:**

None

### 7.2.2.16 GPIOPinTypeTimer

Configures pin(s) for use by the Timer peripheral

**Prototype:**

```
void  
GPIOPinTypeTimer(uint32_t ui32Port,  
                 uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The CCP pins must be properly configured for the timer peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations might work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

This function cannot be used to turn any pin into a timer pin but only configures a timer pin for proper operation.

**Returns:**

None

### 7.2.2.17 GPIOPinTypeUARTInput

Configures input pin(s) for use by the UART peripheral

**Prototype:**

```
void  
GPIOPinTypeUARTInput(uint32_t ui32Port,  
                     uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

The UART input pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations might work as well depending upon the board setup (for example, using the on-chip pull-ups).

**Note:**

For PC0 through PC3 the function GPIOPinTypeUARTHiDrive() should be used to configure these high drive pins.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

This function cannot be used to turn any pin into a UART pin; but only configures a UART pin for proper operation.

**Returns:**  
None

### 7.2.2.18 GPIOPinTypeUARTOutput

Configures output pin(s) for use by the UART peripheral

**Prototype:**  

```
void  
GPIOPinTypeUARTOutput (uint32_t ui32Port,  
                        uint8_t ui8Pins)
```

**Parameters:**  
*ui32Port* is the base address of the GPIO port.  
*ui8Pins* is the bit-packed representation of the pin(s).

**Description:**  
The UART output pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations might work as well depending upon the board setup (for example, using the on-chip pull-ups).  
The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**  
This function cannot be used to turn any pin into a UART pin; but only configures a UART pin for proper operation.

**Returns:**  
None

### 7.2.2.19 GPIOPinWrite

Writes a value to the specified pin(s)

**Prototype:**  

```
void  
GPIOPinWrite (uint32_t ui32Port,  
              uint8_t ui8Pins,  
              uint8_t ui8Val)
```

**Parameters:**  
*ui32Port* is the base address of the GPIO port.  
*ui8Pins* is the bit-packed representation of the pin(s).  
*ui8Val* is the value to write to the pin(s).

**Description:**  
Writes the corresponding bit values to the output pin(s) specified by *ui8Pins*. Writing to a pin configured as an input pin has no effect.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.20 GPIOPortIntRegister

Registers an interrupt handler for a GPIO port

**Prototype:**

```
void  
GPIOPortIntRegister(uint32_t ui32Port,  
                    void (*pfnHandler)(void))
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**pfnHandler** is a pointer to the GPIO port interrupt handling function.

**Description:**

This function ensures that the interrupt handler specified by *pfnHandler* is called when an interrupt is detected from the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with [GPIOPinIntEnable\(\)](#).

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 7.2.2.21 GPIOPortIntUnregister

Removes an interrupt handler for a GPIO port

**Prototype:**

```
void  
GPIOPortIntUnregister(uint32_t ui32Port)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**Description:**

This function unregisters the interrupt handler for the specified GPIO port. This function also disables the corresponding GPIO port interrupt in the interrupt controller; individual GPIO interrupts and interrupt sources must be disabled with [GPIOPinIntDisable\(\)](#).

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 7.2.2.22 GPIOPowIntClear

Clears the power-up interrupt for the specified pin(s)

**Prototype:**

```
void  
GPIOPowIntClear(uint32_t ui32Port,  
                 uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

Clears the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.23 GPIOPowIntDisable

Disables power-up interrupts for the specified pin(s)

**Prototype:**

```
void  
GPIOPowIntDisable(uint32_t ui32Port,  
                   uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

Masks the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.24 GPIOPowIntEnable

Enables power-up interrupts for the specified pin(s)

**Prototype:**

```
void  
GPIOPowIntEnable(uint32_t ui32Port,  
                 uint8_t ui8Pins)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**Description:**

Unmasks the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

### 7.2.2.25 GPIOPowIntStatus

Gets power-up interrupt status for the specified GPIO port

**Prototype:**

```
uint32_t  
GPIOPowIntStatus(uint32_t ui32Port,  
                 bool bMasked)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**bMasked** specifies whether masked or raw interrupt status is returned.

**Description:**

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

**Returns:**

Returns a bit-packed byte, where each bit that is set identifies an active masked or raw interrupt, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Bits 31:8 should be ignored.

### 7.2.2.26 GPIOPowIntTypeGet

Gets the power-up interrupt type for a pin



**Prototype:**

```
uint32_t
GPIOPowIntTypeGet (uint32_t ui32Port,
                  uint8_t ui8Pin)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pin** is the pin number.

**Description:**

This function gets the interrupt type for a specified pin on the selected GPIO port. The pin can be configured as a falling edge, rising edge, or both edge detected interrupt, or it can be configured as a low level or high level detected interrupt. The type of interrupt detection mechanism is returned as an enumerated data type.

**Returns:**

Returns one of the enumerated data types described for [GPIOIntTypeSet\(\)](#).

### 7.2.2.27 GPIOPowIntTypeSet

Sets the power-up interrupt type for the specified pin(s)

**Prototype:**

```
void
GPIOPowIntTypeSet (uint32_t ui32Port,
                  uint8_t ui8Pins,
                  uint32_t ui32IntType)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pins** is the bit-packed representation of the pin(s).

**ui32IntType** specifies type of power-up interrupt trigger mechanism.

**Description:**

This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

The parameter *ui32IntType* is an enumerated data type that can be one of the following values:

- **GPIO\_POW\_FALLING\_EDGE**
- **GPIO\_POW\_RISING\_EDGE**

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

To avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

**Returns:**

None

## 7.3 Programming Example

The following example shows how to use the GPIO API to initialize the GPIO, enable interrupts, read data from pins, and write data to pins.

```
uint32_t ui32Val;

//
// Register the port-level interrupt handler. This handler is the
// first level interrupt handler for all the pin interrupts.
//
GPIOPortIntRegister(GPIO_A_BASE, IntHandler);

//
// Initialize the GPIO pin configuration.
//
// Set pins 2, 4, and 5 as input, SW controlled.
//
GPIOPinTypeGPIOInput(GPIO_A_BASE,
                      GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

//
// Set pins 0 and 3 as output, SW controlled.
//
GPIOPinTypeGPIOOutput(GPIO_A_BASE, GPIO_PIN_0 | GPIO_PIN_3);

//
// Make pins 2 and 4 rising edge triggered interrupts.
//
GPIOIntTypeSet(GPIO_A_BASE, GPIO_PIN_2 | GPIO_PIN_4, GPIO_RISING_EDGE);

//
// Make pin 5 high level triggered interrupts.
//
GPIOIntTypeSet(GPIO_A_BASE, GPIO_PIN_5, GPIO_HIGH_LEVEL);

//
// Read some pins.
//
ui32Val = GPIOPinRead(GPIO_A_BASE,
                      (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
                       GPIO_PIN_4 | GPIO_PIN_5));

//
// Write some pins. Even though pins 2, 4, and 5 are specified, those
// pins are unaffected by this write because they are configured as inputs.
// At the end of this write, pin 0 will be a 0, and pin 3 will be a 1.
//
GPIOPinWrite(GPIO_A_BASE,
              (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
               GPIO_PIN_4 | GPIO_PIN_5),
              0xF8);
```

```
//  
// Enable the pin interrupts.  
//  
GPIOPinIntEnable(GPIO_A_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);
```



## 8 Inter-Integrated Circuit (I2C)

Introduction .....	53
API Functions .....	54
Programming Example .....	67

### 8.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the CC2538 I2C master and slave modules. Functions are provided to perform the following actions:

- Initialize the I2C modules
- Send and receive data
- Obtain status
- Manage interrupts for the I2C modules

The I2C master and slave modules 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 CC2538 I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave. Finally, the CC2538 I2C modules can operate at two speeds: standard (100 kb/s) and fast (400 kb/s).

The master and slave I2C modules can generate interrupts. The I2C master module generates interrupts when a transmit or receive operation completes (or aborts due to an error); and on some devices when a clock low timeout occurs. The I2C slave module generates interrupts when data is sent or requested by a master; and on some devices, when a START or STOP condition is present.

#### 8.1.1 Master Operations

When using this API to drive the I2C master module, the user must first initialize the I2C master module with a call to [I2CMasterInitExpClk\(\)](#). That function sets the bus speed and enables the master module.

The user may transmit or receive data after the successful initialization of the I2C master module. Data is transferred by first setting the slave address using [I2CMasterSlaveAddrSet\(\)](#). That function is also used to define whether the transfer is a send (a write to the slave from the master) or a receive (a read from the slave by the master). Then, if connected to an I2C bus that has multiple masters, the CC2538 I2C master must first call [I2CMasterBusBusy\(\)](#) before trying to initiate the desired transaction. After determining that the bus is not busy, if trying to send data, the user must call the [I2CMasterDataPut\(\)](#) function. The transaction can then be initiated on the bus by calling the [I2CMasterControl\(\)](#) function with any of the following commands:

- **I2C\_MASTER\_CMD\_SINGLE\_SEND**
- **I2C\_MASTER\_CMD\_SINGLE\_RECEIVE**
- **I2C\_MASTER\_CMD\_BURST\_SEND\_START**

### ■ I2C\_MASTER\_CMD\_BURST\_RECEIVE\_START

Any of these commands result in the master arbitrating for the bus, driving the start sequence onto the bus, and sending the slave address and direction bit across the bus. The remainder of the transaction can then be driven using either a polling or interrupt-driven method.

For the single send and receive cases, the polling method involves looping on the return from [I2CMasterBusy\(\)](#). Once that function indicates that the I2C master is no longer busy, the bus transaction is complete and can be checked for errors using [I2CMasterErr\(\)](#). If there are no errors, then the data has been sent or is ready to be read using [I2CMasterDataGet\(\)](#). For the burst send and receive cases, the polling method also involves calling the [I2CMasterControl\(\)](#) function for each byte transmitted or received (using either the **I2C\_MASTER\_CMD\_BURST\_SEND\_CONT** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_CONT** commands), and for the last byte sent or received (using either the **I2C\_MASTER\_CMD\_BURST\_SEND\_FINISH** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_FINISH** commands). If any error is detected during the burst transfer, the appropriate stop command (**I2C\_MASTER\_CMD\_BURST\_SEND\_ERROR\_STOP** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_ERROR\_STOP**) should be used to call the [I2CMasterControl\(\)](#) function.

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C master interrupt; the interrupt occurs when the master is no longer busy.

## 8.1.2 Slave Operations

When using this API to drive the I2C slave module, the user must first initialize the I2C slave module with a call to [I2CSlaveInit\(\)](#). This function enables the I2C slave module and initializes the address of the slave. After the initialization completes, the user may poll the slave status using [I2CSlaveStatus\(\)](#) to determine if a master requested a send or receive operation. Depending on the type of operation requested, the user can call [I2CSlaveDataPut\(\)](#) or [I2CSlaveDataGet\(\)](#) to complete the transaction. Alternatively, the I2C slave can handle transactions using an interrupt handler registered with [I2CIntRegister\(\)](#), and by enabling the I2C slave interrupt.

This driver is contained in `source/i2c.c`, with `source/i2c.h` containing the API definitions for use by applications.

## 8.2 API Functions

### Functions

- void [I2CIntRegister](#) (void (\*pfnHandler)(void))
- void [I2CIntUnregister](#) (void)
- bool [I2CMasterBusBusy](#) (void)
- bool [I2CMasterBusy](#) (void)
- void [I2CMasterControl](#) (uint32\_t ui32Cmd)
- uint32\_t [I2CMasterDataGet](#) (void)
- void [I2CMasterDataPut](#) (uint8\_t ui8Data)
- void [I2CMasterDisable](#) (void)
- void [I2CMasterEnable](#) (void)

- `uint32_t I2CMasterErr` (void)
- `void I2CMasterInitExpClk` (uint32\_t ui32I2CClk, bool bFast)
- `void I2CMasterIntClear` (void)
- `void I2CMasterIntDisable` (void)
- `void I2CMasterIntEnable` (void)
- `bool I2CMasterIntStatus` (bool bMasked)
- `void I2CMasterSlaveAddrSet` (uint8\_t ui8SlaveAddr, bool bReceive)
- `uint32_t I2CSlaveDataGet` (void)
- `void I2CSlaveDataPut` (uint8\_t ui8Data)
- `void I2CSlaveDisable` (void)
- `void I2CSlaveEnable` (void)
- `void I2CSlaveInit` (uint8\_t ui8SlaveAddr)
- `void I2CSlaveIntClear` (void)
- `void I2CSlaveIntClearEx` (uint32\_t ui32IntFlags)
- `void I2CSlaveIntDisable` (void)
- `void I2CSlaveIntDisableEx` (uint32\_t ui32IntFlags)
- `void I2CSlaveIntEnable` (void)
- `void I2CSlaveIntEnableEx` (uint32\_t ui32IntFlags)
- `bool I2CSlaveIntStatus` (bool bMasked)
- `uint32_t I2CSlaveIntStatusEx` (bool bMasked)
- `uint32_t I2CSlaveStatus` (void)

## 8.2.1 Detailed Description

The 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 the following functions:

- `I2CIntRegister()`
- `I2CIntUnregister()`
- `I2CMasterIntEnable()`
- `I2CMasterIntDisable()`
- `I2CMasterIntClear()`
- `I2CMasterIntStatus()`
- `I2CSlaveIntEnable()`
- `I2CSlaveIntDisable()`
- `I2CSlaveIntClear()`
- `I2CSlaveIntStatus()`
- `I2CSlaveIntEnableEx()`
- `I2CSlaveIntDisableEx()`
- `I2CSlaveIntClearEx()`
- `I2CSlaveIntStatusEx()`

Status and initialization functions for the I2C modules are:

- [I2CMasterInitExpClk\(\)](#)
- [I2CMasterEnable\(\)](#)
- [I2CMasterDisable\(\)](#)
- [I2CMasterBusBusy\(\)](#)
- [I2CMasterBusy\(\)](#)
- [I2CMasterErr\(\)](#)
- [I2CSlaveInit\(\)](#)
- [I2CSlaveEnable\(\)](#)
- [I2CSlaveDisable\(\)](#)
- [I2CSlaveStatus\(\)](#)

Sending and receiving data from the I2C modules are handled by the following functions:

- [I2CMasterSlaveAddrSet\(\)](#)
- [I2CMasterControl\(\)](#)
- [I2CMasterDataGet\(\)](#)
- [I2CMasterDataPut\(\)](#)
- [I2CSlaveDataGet\(\)](#)
- [I2CSlaveDataPut\(\)](#)

## 8.2.2 Function Documentation

### 8.2.2.1 I2CIntRegister

Registers an interrupt handler for the I2C module

**Prototype:**

```
void  
I2CIntRegister(void (*pfnHandler) (void))
```

**Parameters:**

***pfnHandler*** is a pointer to the function to be called when the I2C interrupt occurs.

**Description:**

This function sets the handler to be called when an I2C interrupt occurs. This function enables the global interrupt in the interrupt controller; specific I2C interrupts must be enabled through [I2CMasterIntEnable\(\)](#) and [I2CSlaveIntEnable\(\)](#). If necessary, the interrupt handler must clear the interrupt source through [I2CMasterIntClear\(\)](#) and [I2CSlaveIntClear\(\)](#).

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None



### 8.2.2.2 I2CIntUnregister

Unregisters an interrupt handler for the I2C module

**Prototype:**

```
void  
I2CIntUnregister(void)
```

**Description:**

This function clears the handler to be called when an I2C interrupt occurs. The function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 8.2.2.3 I2CMasterBusBusy

Indicates whether or not the I2C bus is busy

**Prototype:**

```
bool  
I2CMasterBusBusy(void)
```

**Description:**

This function returns an indication of whether or not the I2C bus is busy. This function can be used in a multimaster environment to determine if another master is currently using the bus.

**Returns:**

Returns **true** if the I2C bus is busy; otherwise, returns **false**

### 8.2.2.4 I2CMasterBusy

Indicates whether or not the I2C master is busy

**Prototype:**

```
bool  
I2CMasterBusy(void)
```

**Description:**

This function returns an indication of whether or not the I2C master is busy transmitting or receiving data.

**Returns:**

Returns **true** if the I2C master is busy; otherwise, returns **false**

### 8.2.2.5 I2CMasterControl

Controls the state of the I2C master module

**Prototype:**

```
void  
I2CMasterControl (uint32_t ui32Cmd)
```

**Parameters:**

***ui32Cmd*** command to be issued to the I2C master module

**Description:**

This function is used to control the state of the master module send and receive operations. The *ui32Cmd* parameter can be one of the following values:

- I2C\_MASTER\_CMD\_SINGLE\_SEND
- I2C\_MASTER\_CMD\_SINGLE\_RECEIVE
- I2C\_MASTER\_CMD\_BURST\_SEND\_START
- I2C\_MASTER\_CMD\_BURST\_SEND\_CONT
- I2C\_MASTER\_CMD\_BURST\_SEND\_FINISH
- I2C\_MASTER\_CMD\_BURST\_SEND\_ERROR\_STOP
- I2C\_MASTER\_CMD\_BURST\_RECEIVE\_START
- I2C\_MASTER\_CMD\_BURST\_RECEIVE\_CONT
- I2C\_MASTER\_CMD\_BURST\_RECEIVE\_FINISH
- I2C\_MASTER\_CMD\_BURST\_RECEIVE\_ERROR\_STOP

**Returns:**

None

### 8.2.2.6 I2CMasterDataGet

Receives a byte that has been sent to the I2C master

**Prototype:**

```
uint32_t  
I2CMasterDataGet (void)
```

**Description:**

This function reads a byte of data from the I2C master data register.

**Returns:**

Returns the byte received from by the I2C master, cast as an `uint32_t`

### 8.2.2.7 I2CMasterDataPut

Transmits a byte from the I2C master

**Prototype:**

```
void  
I2CMasterDataPut (uint8_t ui8Data)
```

**Parameters:**

**ui8Data** data to be transmitted from the I2C master

**Description:**

This function places the supplied data into I2C master data register.

**Returns:**

None

#### 8.2.2.8 I2CMasterDisable

Disables the I2C master block

**Prototype:**

```
void  
I2CMasterDisable(void)
```

**Description:**

This function disables operation of the I2C master block.

**Returns:**

None

#### 8.2.2.9 I2CMasterEnable

Enables the I2C Master block

**Prototype:**

```
void  
I2CMasterEnable(void)
```

**Description:**

This function will enable operation of the I2C Master block.

**Returns:**

None

#### 8.2.2.10 I2CMasterErr

Gets the error status of the I2C master module

**Prototype:**

```
uint32_t  
I2CMasterErr(void)
```

**Description:**

This function is obtains the error status of the master module send and receive operations.

**Returns:**

Returns the error status as one of the following values:

- I2C\_MASTER\_ERR\_NONE
- I2C\_MASTER\_ERR\_ADDR\_ACK
- I2C\_MASTER\_ERR\_DATA\_ACK
- I2C\_MASTER\_ERR\_ARB\_LOST

### 8.2.2.11 I2CMasterInitExpClk

Initializes the I2C master block

**Prototype:**

```
void  
I2CMasterInitExpClk (uint32_t ui32I2CClk,  
                    bool bFast)
```

**Parameters:**

**ui32I2CClk** is the rate of the clock supplied to the I2C module.

**bFast** set up for fast data transfers

**Description:**

This function initializes operation of the I2C master block. Upon successful initialization of the I2C block, this function has set the bus speed for the master, and has enabled the I2C master block.

If the parameter **bFast** is **true**, then the master block will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

The peripheral clock will be the same as the processor clock. This will be the value returned by [SysCtrlClockGet\(\)](#), or it can be explicitly hardcoded if it is constant and known (to save the code/execution overhead of a call to [SysCtrlClockGet\(\)](#)).

**Returns:**

None

### 8.2.2.12 I2CMasterIntClear

Clears I2C master interrupt sources

**Prototype:**

```
void  
I2CMasterIntClear (void)
```

**Description:**

This function clears the I2C master interrupt source, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None

### 8.2.2.13 I2CMasterIntDisable

Disables the I2C master interrupt

**Prototype:**

```
void  
I2CMasterIntDisable(void)
```

**Description:**

This function disables the I2C master interrupt source.

**Returns:**

None

### 8.2.2.14 I2CMasterIntEnable

Enables the I2C Master interrupt

**Prototype:**

```
void  
I2CMasterIntEnable(void)
```

**Description:**

This function enables the I2C Master interrupt source.

**Returns:**

None

### 8.2.2.15 I2CMasterIntStatus

Gets the current I2C master interrupt status

**Prototype:**

```
bool  
I2CMasterIntStatus(bool bMasked)
```

**Parameters:**

**bMasked** is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

**Description:**

This function returns the interrupt status for the I2C master module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

Returns the current interrupt status, returned as **true** if active or **false** if not active.

### 8.2.2.16 I2CMasterSlaveAddrSet

Sets the address that the I2C master places on the bus

**Prototype:**

```
void  
I2CMasterSlaveAddrSet (uint8_t ui8SlaveAddr,  
                        bool bReceive)
```

**Parameters:**

**ui8SlaveAddr** 7-bit slave address

**bReceive** flag indicating the type of communication with the slave

**Description:**

This function sets the address that the I2C master places on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address indicates that the I2C master is initiating a read from the slave; otherwise, the address indicates that the I2C master is initiating a write to the slave.

**Returns:**

None

### 8.2.2.17 I2CSlaveDataGet

Receives a byte that has been sent to the I2C slave

**Prototype:**

```
uint32_t  
I2CSlaveDataGet (void)
```

**Description:**

This function reads a byte of data from the I2C slave data register.

**Returns:**

Returns the byte received from by the I2C slave, cast as an uint32\_t.

### 8.2.2.18 I2CSlaveDataPut

Transmits a byte from the I2C slave

**Prototype:**

```
void  
I2CSlaveDataPut (uint8_t ui8Data)
```

**Parameters:**

**ui8Data** data to be transmitted from the I2C slave

**Description:**

This function places the supplied data into I2C slave data register.

**Returns:**

None

### 8.2.2.19 I2CSlaveDisable

Disables the I2C slave block

**Prototype:**

```
void  
I2CSlaveDisable(void)
```

**Description:**

This function disables operation of the I2C slave block.

**Returns:**

None

### 8.2.2.20 I2CSlaveEnable

Enables the I2C slave block

**Prototype:**

```
void  
I2CSlaveEnable(void)
```

**Description:**

This function enables operation of the I2C slave block.

**Returns:**

None

### 8.2.2.21 I2CSlaveInit

Initializes the I2C slave block

**Prototype:**

```
void  
I2CSlaveInit(uint8_t ui8SlaveAddr)
```

**Parameters:**

***ui8SlaveAddr*** 7-bit slave address

**Description:**

This function initializes operation of the I2C slave block. Upon successful initialization of the I2C blocks, this function has set the slave address has enabled the I2C slave block.

The parameter *ui8SlaveAddr* is the value that will be compared against the slave address sent by an I2C master.

**Returns:**

None

### 8.2.2.22 I2CSlaveIntClear

Clears I2C slave interrupt sources

**Prototype:**

```
void  
I2CSlaveIntClear(void)
```

**Description:**

This function clears the I2C slave interrupt source, so that it no longer asserts. This must be done in the interrupt handler to keep it from being recalled immediately upon exit.

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None

### 8.2.2.23 I2CSlaveIntClearEx

Clears the I2C slave interrupt sources

**Prototype:**

```
void  
I2CSlaveIntClearEx(uint32_t ui32IntFlags)
```

**Parameters:**

***ui32IntFlags*** is a bit mask of the interrupt sources to be cleared.

**Description:**

This function clears the specified I2C Slave interrupt sources, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CSlaveIntEnableEx\(\)](#).

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None



### 8.2.2.24 I2CSlaveIntDisable

Disables the I2C Slave interrupt

**Prototype:**

```
void  
I2CSlaveIntDisable(void)
```

**Description:**

This function disables the I2C Slave interrupt source

**Returns:**

None

### 8.2.2.25 I2CSlaveIntDisableEx

Disables individual I2C slave interrupt sources

**Prototype:**

```
void  
I2CSlaveIntDisableEx(uint32_t ui32IntFlags)
```

**Parameters:**

***ui32IntFlags*** is the bit mask of the interrupt sources to be disabled.

**Description:**

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

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [I2CSlaveIntEnableEx\(\)](#).

**Returns:**

None

### 8.2.2.26 I2CSlaveIntEnable

Enables the I2C Slave interrupt

**Prototype:**

```
void  
I2CSlaveIntEnable(void)
```

**Description:**

This function enables the I2C Slave interrupt source.

**Returns:**

None

### 8.2.2.27 I2CSlaveIntEnableEx

Enables individual I2C slave interrupt sources

**Prototype:**

```
void  
I2CSlaveIntEnableEx (uint32_t ui32IntFlags)
```

**Parameters:**

**ui32IntFlags** is the bit mask of the interrupt sources to be enabled.

**Description:**

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

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **I2C\_SLAVE\_INT\_STOP** Stop condition detected interrupt
- **I2C\_SLAVE\_INT\_START** Start condition detected interrupt
- **I2C\_SLAVE\_INT\_DATA** Data interrupt

**Returns:**

None

### 8.2.2.28 I2CSlaveIntStatus

Gets the current I2C slave interrupt status

**Prototype:**

```
bool  
I2CSlaveIntStatus (bool bMasked)
```

**Parameters:**

**bMasked** is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

**Description:**

This function returns the interrupt status for the I2C slave module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

Returns the current interrupt status, returned as **true** if active or **false** if not active.

### 8.2.2.29 I2CSlaveIntStatusEx

Gets the current I2C slave interrupt status

**Prototype:**

```
uint32_t  
I2CSlaveIntStatusEx (bool bMasked)
```

**Parameters:**

***bMasked*** is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

**Description:**

This function returns the interrupt status for the I2C slave module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

Returns the current interrupt status, enumerated as a bit field of values described in [I2CSlaveIntEnableEx\(\)](#).

### 8.2.2.30 I2CSlaveStatus

Gets the I2C slave module status

**Prototype:**

```
uint32_t  
I2CSlaveStatus(void)
```

**Description:**

This function returns the action requested from a master, if any. Possible values are:

- **I2C\_SLAVE\_ACT\_NONE**
- **I2C\_SLAVE\_ACT\_RREQ**
- **I2C\_SLAVE\_ACT\_TREQ**
- **I2C\_SLAVE\_ACT\_RREQ\_FBR**

**Returns:**

Returns **I2C\_SLAVE\_ACT\_NONE** to indicate that no action has been requested of the I2C slave module, **I2C\_SLAVE\_ACT\_RREQ** to indicate that an I2C master has sent data to the I2C slave module, **I2C\_SLAVE\_ACT\_TREQ** to indicate that an I2C master has requested that the I2C slave module send data, and **I2C\_SLAVE\_ACT\_RREQ\_FBR** to indicate that an I2C master has sent data to the I2C slave and the first byte following the address of the slave has been received.

## 8.3 Programming Example

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

```
//  
// Initialize Master and Slave  
//  
I2CMasterInitExpClk(SysCtrlClockGet(), true);  
  
//  
// Specify slave address  
//  
I2CMasterSlaveAddrSet(0x3B, false);
```

```
//  
// Place the character to be sent in the data register  
//  
I2CMasterDataPut('Q');  
  
//  
// Initiate send of character from Master to Slave  
//  
I2CMasterControl(I2C_MASTER_CMD_SINGLE_SEND);  
  
//  
// Delay until transmission completes  
//  
while(I2CMasterBusy())  
{  
}
```

## 9 Nested Vectored Interrupt Controller (NVIC)

Introduction .....	69
API Functions .....	70
Programming Example .....	78

### 9.1 Introduction

The interrupt controller API provides a set of functions for dealing with the nested vectored interrupt controller (NVIC). Functions are provided to enable and disable interrupts, register interrupt handlers, and set the priority of interrupts.

The NVIC provides global interrupt masking, prioritization, and handler dispatching. Devices within the CC2538 family support up to 163 interrupt sources and 8 priority levels. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The CC2538 family supports an alternate, or compressed interrupt map, that effectively have the same number of available interrupt sources, but in a shorter more densely populated map.

The NVIC is tightly coupled with the Cortex-M3 microprocessor. When the processor responds to an interrupt, the NVIC supplies the address of the function to handle the interrupt directly to the processor. This action eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, thus reducing interrupt response time.

The interrupt prioritization in the NVIC allows handling of higher priority interrupts before lower priority interrupts, as well as allowing preemption of lower priority interrupt handlers by higher priority interrupts. Again, this helps reduce interrupt response time (for example, a 1-ms system control interrupt is not held off by the execution of a lower priority 1-second housekeeping interrupt handler).

Subprioritization is also possible; instead of having N bits of preemptable prioritization, the NVIC can be configured (through software) for N - M bits of preemptable prioritization and M bits of subpriority. In this scheme, two interrupts with the same preemptable prioritization but different subpriorities do not cause a preemption; instead tail chaining is used to process the two interrupts back-to-back.

If two interrupts with the same priority (and subpriority if so configured) are asserted at the same time, the one with the lower interrupt number is processed first. The NVIC keeps track of the nesting of interrupt handlers, allowing the processor to return from interrupt context only once all nested and pending interrupts have been handled.

Interrupt handlers can be configured in one of two ways; statically at compile time or dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the startup code of the application. When statically configured, the interrupts must be explicitly enabled in the NVIC through [IntEnable\(\)](#) before the processor can respond to the interrupt (in addition to any interrupt enabling required within the peripheral). Statically configuring the interrupt table provides the fastest interrupt response time because the stacking operation (a write to SRAM) can be performed in parallel with the interrupt handler table fetch (a read from Flash), as well as the prefetch of the interrupt handler (assuming it is also in flash).

Alternatively, interrupts can be configured at runtime using [IntRegister\(\)](#) (or the analog in each individual driver). When using [IntRegister\(\)](#), the interrupt must also be enabled as before; when

using the analog in each individual driver, [IntEnable\(\)](#) is called by the driver and does not need to be called by the application. Run-time configuration of interrupts adds a small latency to the interrupt response time because the stacking operation (a write to SRAM) and the interrupt handler table fetch (a read from SRAM) must be performed sequentially.

Run-time configuration of interrupt handlers requires that the interrupt handler table is placed on a 1-kB boundary in SRAM (typically, this is at the beginning of SRAM). Failure to do so results in an incorrect vector address being fetched in response to an interrupt. The vector table is in a section called vtable and should be placed appropriately with a linker script.

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

Symbolic definitions of the available interrupt numbers are contained in `inc/hw_ints.h`.

## 9.2 API Functions

### Functions

- void [IntAltMapDisable](#) (void)
- void [IntAltMapEnable](#) (void)
- bool [IntAltMapIsEnabled](#) (void)
- void [IntDisable](#) (uint32\_t ui32Interrupt)
- void [IntEnable](#) (uint32\_t ui32Interrupt)
- bool [IntMasterDisable](#) (void)
- bool [IntMasterEnable](#) (void)
- void [IntPendClear](#) (uint32\_t ui32Interrupt)
- void [IntPendSet](#) (uint32\_t ui32Interrupt)
- int32\_t [IntPriorityGet](#) (uint32\_t ui32Interrupt)
- uint32\_t [IntPriorityGroupingGet](#) (void)
- void [IntPriorityGroupingSet](#) (uint32\_t ui32Bits)
- uint32\_t [IntPriorityMaskGet](#) (void)
- void [IntPriorityMaskSet](#) (uint32\_t ui32PriorityMask)
- void [IntPrioritySet](#) (uint32\_t ui32Interrupt, uint8\_t ui8Priority)
- void [IntRegister](#) (uint32\_t ui32Interrupt, void (\*pfnHandler)(void))
- void [IntUnregister](#) (uint32\_t ui32Interrupt)

### 9.2.1 Detailed Description

The primary function of the interrupt controller API is to manage the interrupt vector table used by the NVIC to dispatch interrupt requests. Registering an interrupt handler is a simple matter of inserting the handler address into the table. By default, the table is filled with pointers to an internal handler that loops forever; it is an error for an interrupt to occur when there is no interrupt handler registered to process it. Therefore, interrupt sources should not be enabled before a handler has been registered, and interrupt sources should be disabled before a handler is unregistered. Interrupt handlers are managed with [IntRegister\(\)](#) and [IntUnregister\(\)](#).

Each interrupt source can be individually enabled and disabled through [IntEnable\(\)](#) and [IntDisable\(\)](#). The processor interrupt can be enabled and disabled through [IntMasterEnable\(\)](#) and [IntMasterDisable\(\)](#); this does not affect the individual interrupt enable states. Masking of the processor interrupt can be used as a simple critical section (only an NMI can interrupt the processor while the processor interrupt is disabled), although masking the processor interrupt can have adverse effects on the interrupt response time.

The priority of each interrupt source can be set and examined using [IntPrioritySet\(\)](#) and [IntPriorityGet\(\)](#). The priority assignments are defined by the hardware; the upper N bits of the 8-bit priority are examined to determine the priority of an interrupt (for the CC2538 family, N is 3). This protocol allows priorities to be defined without knowledge of the exact number of supported priorities; moving to a device with more or fewer priority bits is made easier as the interrupt source continues to have a similar level of priority. Smaller priority numbers correspond to higher interrupt priority, so 0 is the highest priority.

The alternate interrupt map can be enabled and disabled using [IntAltMapEnable\(\)](#) and [IntAltMapDisable\(\)](#).

**Note:**

When using the alternate map, the application should be compiled with the `USE_ALTERNATE_INTERRUPT_MAP` symbol defined.

## 9.2.2 Function Documentation

### 9.2.2.1 IntAltMapDisable

Disable the alternate interrupt mapping

**Prototype:**

```
void  
IntAltMapDisable(void)
```

**Description:**

This function disables the alternate (that is, smaller) interrupt map.

**See also:**

See also [IntAltMapDisable\(\)](#) and [IntAltMapsEnabled\(\)](#).

**Returns:**

None

### 9.2.2.2 IntAltMapEnable

Enables the alternate interrupt mapping

**Prototype:**

```
void  
IntAltMapEnable(void)
```

**Description:**

This function enables the alternate (that is, smaller) interrupt map.

**See also:**

See also [IntAltMapDisable\(\)](#) and [IntAltMapIsEnabled\(\)](#).

**Returns:**

None

### 9.2.2.3 IntAltMapIsEnabled

Checks to see if the Alternate Interrupt Mapping is in use

**Prototype:**

```
bool  
IntAltMapIsEnabled(void)
```

**See also:**

See also [IntAltMapDisable\(\)](#) and [IntAltMapIsEnabled\(\)](#).

**Returns:**

Returns **true** if the Alternate Mapping is in use and **false** otherwise.

### 9.2.2.4 void IntDisable (uint32\_t *ui32Interrupt*)

Disables an interrupt

**Parameters:**

***ui32Interrupt*** specifies the interrupt to be disabled.

**Description:**

This function disables specified interrupt in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

**Returns:**

None

### 9.2.2.5 IntEnable

Enables an interrupt

**Prototype:**

```
void  
IntEnable(uint32_t ui32Interrupt)
```

**Parameters:**

***ui32Interrupt*** specifies the interrupt to be enabled.

**Description:**

This function enables the specified interrupt in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

**Returns:**

None



### 9.2.2.6 IntMasterDisable

Disables the processor interrupt

**Prototype:**

```
bool  
IntMasterDisable(void)
```

**Description:**

This function prevents the processor from receiving interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

**Returns:**

Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

### 9.2.2.7 IntMasterEnable

Enables the processor interrupt

**Prototype:**

```
bool  
IntMasterEnable(void)
```

**Description:**

This function allows the processor to respond to interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

**Returns:**

Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

### 9.2.2.8 IntPendClear

Unpends an interrupt

**Prototype:**

```
void  
IntPendClear(uint32_t ui32Interrupt)
```

**Parameters:**

**ui32Interrupt** specifies the interrupt to be unpended.

**Description:**

This function unpends the specified interrupt in the interrupt controller. This will cause any previously generated interrupts that have not been handled yet (due to higher priority interrupts or the interrupt not having been enabled yet) to be discarded.

**Returns:**

None

### 9.2.2.9 IntPendSet

Pends an interrupt

**Prototype:**

```
void  
IntPendSet (uint32_t ui32Interrupt)
```

**Parameters:**

***ui32Interrupt*** specifies the interrupt to be pended.

**Description:**

This function pends the specified interrupt in the interrupt controller. This causes the interrupt controller to execute the corresponding interrupt handler at the next available time, based on the current interrupt state priorities. For example, if called by a higher priority interrupt handler, the specified interrupt handler is not called until after the current interrupt handler executes. The interrupt must have been enabled for it to be called.

**Returns:**

None

### 9.2.2.10 IntPriorityGet

Gets the priority of an interrupt

**Prototype:**

```
int32_t  
IntPriorityGet (uint32_t ui32Interrupt)
```

**Parameters:**

***ui32Interrupt*** specifies the interrupt in question.

**Description:**

This function gets the priority of an interrupt. See [IntPrioritySet\(\)](#) for a definition of the priority value.

**Returns:**

Returns the interrupt priority, or -1 if an invalid interrupt was specified

### 9.2.2.11 IntPriorityGroupingGet

Gets the priority grouping of the interrupt controller

**Prototype:**

```
uint32_t  
IntPriorityGroupingGet (void)
```

**Description:**

This function returns the split between preemptable priority levels and subpriority levels in the interrupt priority specification.

**Returns:**

Returns the number of bits of preemptable priority

### 9.2.2.12 IntPriorityGroupingSet

Sets the priority grouping of the interrupt controller

**Prototype:**

```
void  
IntPriorityGroupingSet (uint32_t ui32Bits)
```

**Parameters:**

**ui32Bits** specifies the number of bits of preemptable priority.

**Description:**

This function specifies the split between preemptable priority levels and subpriority levels in the interrupt priority specification. The range of the grouping values depend on the hardware implementation; on the CC2538 device family, 3 bits are available for hardware interrupt prioritization and therefore priority grouping values of three through seven have the same effect.

**Returns:**

None

### 9.2.2.13 IntPriorityMaskGet

Gets the priority masking level

**Prototype:**

```
uint32_t  
IntPriorityMaskGet (void)
```

**Description:**

This function gets the current setting of the interrupt priority masking level. The value returned is the priority level such that all interrupts of that priority and lesser priorities are masked. A value of 0 disables priority masking.

Smaller numbers correspond to higher interrupt priorities. For example, a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater will be blocked.

The hardware priority mechanism looks only at the upper N bits of the priority level (where N is 3 for the CC2538 device family), so any prioritization must be performed in those bits.

**Returns:**

Returns the value of the interrupt priority level mask

### 9.2.2.14 IntPriorityMaskSet

Sets the priority masking level

**Prototype:**

```
void  
IntPriorityMaskSet (uint32_t ui32PriorityMask)
```

**Parameters:**

***ui32PriorityMask*** is the priority level that will be masked.

**Description:**

This function sets the interrupt priority masking level so that all interrupts at the specified or lesser priority level is masked. This can be used to globally disable a set of interrupts with priority below a predetermined threshold. A value of 0 disables priority masking.

Smaller numbers correspond to higher interrupt priorities. For example, a priority level mask of 4 allows interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater are blocked.

The hardware priority mechanism looks only at the upper N bits of the priority level (where N is 3 for the CC2538 device family), so any prioritization must be performed in those bits.

**Returns:**

None

### 9.2.2.15 IntPrioritySet

Sets the priority of an interrupt

**Prototype:**

```
void  
IntPrioritySet (uint32_t ui32Interrupt,  
               uint8_t ui8Priority)
```

**Parameters:**

***ui32Interrupt*** specifies the interrupt in question.

***ui8Priority*** specifies the priority of the interrupt.

**Description:**

This function sets the priority of an interrupt. When multiple interrupts are asserted simultaneously, those with the highest priority are processed before the lower priority interrupts. Smaller numbers correspond to higher interrupt priorities; priority 0 is the highest interrupt priority.

The hardware priority mechanism will look only at the upper N bits of the priority level (where N is 3 for the CC2538 device family), so any prioritization must be performed in those bits. The remaining bits can be used to subprioritize the interrupt sources, and may be used by the hardware priority mechanism on a future part. This arrangement allows priorities to migrate to different NVIC implementations without changing the gross prioritization of the interrupts. Thus for CC2538 to set a priority of 3, the parameter *ui8Priority* must be set to (3<<5).

**Returns:**

None

### 9.2.2.16 IntRegister

Registers a function to be called when an interrupt occurs

**Prototype:**

```
void  
IntRegister(uint32_t ui32Interrupt,  
            void (*pfnHandler)(void))
```

**Parameters:**

**ui32Interrupt** specifies the interrupt in question.  
**pfnHandler** is a pointer to the function to be called.

**Description:**

This function specifies the handler function to be called when the given interrupt is asserted to the processor. When the interrupt occurs, if it is enabled (through [IntEnable\(\)](#)), the handler function is called in interrupt context. Because the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other nonhandler code.

**Note:**

This function (directly or indirectly through a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise NVIC will not look in the correct portion of memory for the vector table (it requires the vector table be on a 1-kB memory alignment). Normally, the SRAM vector table is so placed through the use of linker scripts. See the discussion of compile-time versus runtime interrupt handler registration in the introduction to this chapter.

**Returns:**

None

### 9.2.2.17 IntUnregister

Unregisters the function to be called when an interrupt occurs

**Prototype:**

```
void  
IntUnregister(uint32_t ui32Interrupt)
```

**Parameters:**

**ui32Interrupt** specifies the interrupt in question.

**Description:**

This function indicates that no handler should be called when the given interrupt is asserted to the processor. The interrupt source is automatically disabled (through [IntDisable\(\)](#)) if necessary.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

## 9.3 Programming Example

The following example shows how to use the interrupt controller API to register an interrupt handler and enable the interrupt.

```
//  
// The interrupt handler function.  
//  
extern void IntHandler(void);  
  
//  
// Register the interrupt handler function for  
// interrupt 5 (FAULT_BUS).  
//  
IntRegister(FAULT_BUS, IntHandler);  
  
//  
// Enable interrupt 5 (FAULT_BUS).  
//  
IntEnable(FAULT_BUS);  
  
//  
// Enable interrupts.  
//  
IntMasterEnable();
```

## 10 I/O Control (IOC)

Introduction .....	79
API Functions .....	80
Programming Example .....	84

### 10.1 Introduction

The I/O control module connects to the on-chip peripherals external I/Os and routes these signals through a muxing matrix, after which they are routed to the GPIO block and on to the I/O pads. Both external input and output signals are routed through this chain of modules. The 32 available I/O pins are logically grouped into 4 port groups with 8 I/Os in each group (Port A, Port B, Port C and Port D), where each I/O has an associated GPIO instance. For a detailed description of the signal routing see the General-Purpose Inputs/Outputs section of the CC2538 User's Guide.

The I/O control module allows the following peripheral signals to be routed to or from any of the 32 GPIO pads:

- UART0, UART1
- SSI0 and SSI1
- I2C
- General Purpose Timers 0, 1, 2 and 3.

One exception is the GPIO signals, each of which is assigned to a single fixed GPIO pad.

The pad characteristic can be controlled by the peripheral or explicitly set by configuring the pad with one of the following override values:

- Output enable
- Pullup enable
- Pulldown enable
- Analog enable
- Override disable

If the on-chip peripheral can configure the pad, TI recommends that the peripheral determine the pad configuration using the override disable option. Analog inputs and outputs are applicable for ADC, comparator, and external clock sources.

To configure the device for external operation with a specific peripheral, the chosen inputs and outputs should be routed to and from the peripheral by configuring the muxing matrix using the [IOCPinConfigPeriphInput\(\)](#) and [IOCPinConfigPeriphOutput\(\)](#) functions. The related input and output pads can then be configured using the [IOCPadConfigSet\(\)](#) function. A simpler method is to use the dedicated peripheral functions in the GPIO driver, (for example use [GPIOPinTypeSSI\(\)](#) to configure input and output pins as required for use with an on-chip SSI peripheral).

Most of the IOC functions can operate on more than one pin (within a single port) at a time. The *ui8Pins* parameter functions specify the pins that are affected: Only the pins corresponding to the bits that are set in this parameter are affected (where pin 0 is bit 0, pin 1 is bit 1, and so on). For example, if *ui8Pins* is 0x09, then pins 0 and 3 are affected by the function. For functions that have a *ui8Pin* (singular) parameter, only a single pin is affected by the function. In this case, the value specifies the pin number (that is, 0 through 7).

**Note:**

Port C pin 0 through Port C pin 3 (PC0-PC3) are bidirectional high-drive (20 mA) pad cells. They do not support on-die pullup or pulldown resistors or analog connectivity. Port A is the only port supporting ADC access.

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

## 10.2 API Functions

### Functions

- `uint32_t IOCPadConfigGet (uint32_t ui32Port, uint8_t ui8Pin)`
- `void IOCPadConfigSet (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32PinDrive)`
- `void IOCPinConfigPeriphInput (uint32_t ui32Port, uint8_t ui8Pin, uint32_t ui32PinSelectReg)`
- `void IOCPinConfigPeriphOutput (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32OutputSignal)`

### 10.2.1 Detailed Description

The following functions configure the pin mux matrix:

- `IOCPinConfigPeriphInput()`
- `IOCPinConfigPeriphOutput()`

The following functions access the pad configuration:

- `IOCPadConfigSet()`
- `IOCPadConfigGet()`

### 10.2.2 Function Documentation

#### 10.2.2.1 IOCPadConfigGet

Get drive type on the pad for the desired port pin.

**Prototype:**

```
uint32_t
IOCPadConfigGet (uint32_t ui32Port,
                 uint8_t ui8Pin)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pin** is the bit-packed representation of the port pin.



**Description:**

This function returns the configured pin drive type for the desired pin on the selected GPIO port.

The pin in *ui8Pin* is specified using a bit-packed byte, where the bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

Returns the logical OR of the enumerated data types described for *ui32PinDrive* parameter in [IOCPadConfigSet\(\)](#).

**See also:**

[IOCPadConfigSet\(\)](#)

### 10.2.2.2 IOCPadConfigSet

Set desired drive type on the pad for the desired port pin(s).

**Prototype:**

```
void
IOCPadConfigSet (uint32_t ui32Port,
                 uint8_t ui8Pins,
                 uint32_t ui32PinDrive)
```

**Parameters:**

***ui32Port*** is the base address of the GPIO port.

***ui8Pins*** is the bit-packed representation of the port pin(s).

***ui32PinDrive*** is the drive configuration of the desired port pin.

**Description:**

This function sets the desired pin drive type for the desired pin(s) on the selected GPIO port.

The *ui32PinDrive* parameter controls the configuration of the pin drive on the pad for the desired pin(s). The parameter is the logical OR of any of the following:

- **IOC\_OVERRIDE\_OE**
- **IOC\_OVERRIDE\_PUE**
- **IOC\_OVERRIDE\_PDE**
- **IOC\_OVERRIDE\_ANA**
- **IOC\_OVERRIDE\_DIS**

where **IOC\_OVERRIDE\_OE** is the output enable bit connected directly to the output enable pin for the IO driver cell, after it is ORed with any OE signal from the desired peripheral. The OE is driven from the SSI, I2C and GPT peripherals. **IOC\_OVERRIDE\_PUE** is the enable bit for the pull-up. **IOC\_OVERRIDE\_PDE** is the enable bit for the pull-down. **IOC\_OVERRIDE\_ANA** must be set for the analog signal.

The pin(s) in *ui8Pins* are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Note:**

PC0 through PC3 are bidirectional high-drive pad-cells. They do not support on-die pullup or pulldown resistors or analog connectivity. For these four pins the *ui32PinDrive* parameter must be set to either **IOC\_OVERRIDE\_OE** or **IOC\_OVERRIDE\_DIS**.

**Returns:**

None

### 10.2.2.3 IOCPinConfigPeriphInput

Mux the desired port pin to the desired on-chip peripheral input signal

**Prototype:**

```
void
IOCPinConfigPeriphInput (uint32_t ui32Port,
                          uint8_t ui8Pin,
                          uint32_t ui32PinSelectReg)
```

**Parameters:**

**ui32Port** is the base address of the GPIO port.

**ui8Pin** is the bit-packed representation of the desired port pin.

**ui32PinSelectReg** is the address of the IOC mux-register for the desired peripheral input signal to which the desired port pin shall be routed.

**Description:**

This function routes the desired port pin to the desired on-chip peripheral input signal. Functions are available within the GPIO device driver that set the peripheral signal to be under hardware control and configures the pin drive type on the desired port pin.

The parameter *ui32PinSelectReg* is an enumerated data type that can be one of the following values:

- **IOC\_UARTRXD\_UART0**
- **IOC\_UARTCTS\_UART1**
- **IOC\_UARTRXD\_UART1**
- **IOC\_CLK\_SSI\_SSI0**
- **IOC\_SSIRXD\_SSI0**
- **IOC\_SSIFSSIN\_SSI0**
- **IOC\_CLK\_SSIIN\_SSI0**
- **IOC\_CLK\_SSI\_SSI1**
- **IOC\_SSIRXD\_SSI1**
- **IOC\_SSIFSSIN\_SSI1**
- **IOC\_CLK\_SSIIN\_SSI1**
- **IOC\_I2CMSSDA**
- **IOC\_I2CMSSCL**
- **IOC\_GPT0OCP1**
- **IOC\_GPT0OCP2**
- **IOC\_GPT1OCP1**
- **IOC\_GPT1OCP2**
- **IOC\_GPT2OCP1**

- IOC\_GPT2OCP2
- IOC\_GPT3OCP1
- IOC\_GPT3OCP2

The pin in `ui8Pin` is specified using a bit-packed byte, where the bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

#### 10.2.2.4 IOCPinConfigPeriphOutput

Mux desired on-chip peripheral output signal to the desired port pin(s).

**Prototype:**

```
void
IOCPinConfigPeriphOutput (uint32_t ui32Port,
                          uint8_t ui8Pins,
                          uint32_t ui32OutputSignal)
```

**Parameters:**

***ui32Port*** is the base address of the GPIO port.

***ui8Pins*** is the bit-packed representation of the port pin(s).

***ui32OutputSignal*** is the desired peripheral output signal to drive the desired port pin(s).

**Description:**

This function routes the desired on-chip peripheral signal to the desired pin(s) on the selected GPIO port. Functions are available within the GPIO device driver that can set the peripheral signal to be under hardware control. The [IOCPadConfigSet\(\)](#) function can be used to set the pin drive type on the desired port pin.

The *ui32OutputSignal* parameter is an enumerated type that controls which peripheral output signal to route to the desired port pin(s). This parameter can have any of the following values:

- IOC\_MUX\_OUT\_SEL\_UART0\_TXD
- IOC\_MUX\_OUT\_SEL\_UART1\_RTS
- IOC\_MUX\_OUT\_SEL\_UART1\_TXD
- IOC\_MUX\_OUT\_SEL\_SSI0\_TXD
- IOC\_MUX\_OUT\_SEL\_SSI0\_CLKOUT
- IOC\_MUX\_OUT\_SEL\_SSI0\_FSSOUT
- IOC\_MUX\_OUT\_SEL\_SSI0\_STXSER\_EN
- IOC\_MUX\_OUT\_SEL\_SSI1\_TXD
- IOC\_MUX\_OUT\_SEL\_SSI1\_CLKOUT
- IOC\_MUX\_OUT\_SEL\_SSI1\_FSSOUT
- IOC\_MUX\_OUT\_SEL\_SSI1\_STXSER\_EN
- IOC\_MUX\_OUT\_SEL\_I2C\_CMSSDA
- IOC\_MUX\_OUT\_SEL\_I2C\_CMSSCL
- IOC\_MUX\_OUT\_SEL\_GPT0\_ICP1
- IOC\_MUX\_OUT\_SEL\_GPT0\_ICP2

- IOC\_MUX\_OUT\_SEL\_GPT1\_ICP1
- IOC\_MUX\_OUT\_SEL\_GPT1\_ICP2
- IOC\_MUX\_OUT\_SEL\_GPT2\_ICP1
- IOC\_MUX\_OUT\_SEL\_GPT2\_ICP2
- IOC\_MUX\_OUT\_SEL\_GPT3\_ICP1
- IOC\_MUX\_OUT\_SEL\_GPT3\_ICP2

The pin(s) in *ui8Pins* are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None

## 10.3 Programming Example

The following example shows how to use the IOC API to map UART signals to the GPIO pins.

```
//  
// Map UART signals to the GPIO pins and configure them as  
// hardware controlled.  
//  
IOCPinConfigPeriphOutput(GPIO_A_BASE, GPIO_PIN_1, IOC_MUX_OUT_SEL_UART0_TXD);  
GPIOPinTypeUARTOutput(GPIO_A_BASE, GPIO_PIN_1);  
  
IOCPinConfigPeriphInput(GPIO_A_BASE, GPIO_PIN_0, IOC_UARTRXD_UART0);  
GPIOPinTypeUARTInput(GPIO_A_BASE, GPIO_PIN_0);
```

# 11 Public Key HW Accelerator driver (PKA)

Introduction .....	85
API Functions .....	85

## 11.1 Introduction

The public key accelerator driver (PKA) API provides a set of functions to deal with the PKA hardware module. Functions perform a set of operations on the big numbers needed for the Elliptical curve cryptography (ECC) implementation.

This driver is contained in `source/pka.c`, with `source/pka.h` containing the low level API definitions for use by applications.

## 11.2 API Functions

### Functions

- tPKAStatus [PKABigNumAddGetResult](#) (uint32\_t \*pui32ResultBuf, uint32\_t \*pui32Len, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKABigNumAddStart](#) (uint32\_t \*pui32BN1, uint8\_t ui8BN1Size, uint32\_t \*pui32BN2, uint8\_t ui8BN2Size, uint32\_t \*pui32ResultVector)
- tPKAStatus [PKABigNumCmpGetResult](#) (void)
- tPKAStatus [PKABigNumCmpStart](#) (uint32\_t \*pui32BNum1, uint32\_t \*pui32BNum2, uint8\_t ui8Size)
- tPKAStatus [PKABigNumInvModGetResult](#) (uint32\_t \*pui32ResultBuf, uint8\_t ui8Size, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKABigNumInvModStart](#) (uint32\_t \*pui32BNum, uint8\_t ui8BNSize, uint32\_t \*pui32Modulus, uint8\_t ui8Size, uint32\_t \*pui32ResultVector)
- tPKAStatus [PKABigNumModGetResult](#) (uint32\_t \*pui32ResultBuf, uint8\_t ui8Size, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKABigNumModStart](#) (uint32\_t \*pui32BNum, uint8\_t ui8BNSize, uint32\_t \*pui32Modulus, uint8\_t ui8ModSize, uint32\_t \*pui32ResultVector)
- tPKAStatus [PKABigNumMultGetResult](#) (uint32\_t \*pui32ResultBuf, uint32\_t \*pui32Len, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKABigNumMultiplyStart](#) (uint32\_t \*pui32Xplicand, uint8\_t ui8XplicandSize, uint32\_t \*pui32Xplier, uint8\_t ui8XplierSize, uint32\_t \*pui32ResultVector)
- void [PKAClearInt](#) (void)
- void [PKADisableInt](#) (void)
- tPKAStatus [PKAECCAddGetResult](#) (tECPt \*ptOutEcPt, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKAECCAddStart](#) (tECPt \*ptEcPt1, tECPt \*ptEcPt2, tECCCurveInfo \*ptCurve, uint32\_t \*pui32ResultVector)
- tPKAStatus [PKAECCMultGenPtGetResult](#) (tECPt \*ptOutEcPt, uint32\_t ui32ResVectorLoc)

- tPKAStatus [PKAECCMultGenPtStart](#) (uint32\_t \*pui32Scalar, tECCCurveInfo \*ptCurve, uint32\_t \*pui32ResultVector)
- tPKAStatus [PKAECCMultiplyGetResult](#) (tECPt \*ptOutEcPt, uint32\_t ui32ResVectorLoc)
- tPKAStatus [PKAECCMultiplyStart](#) (uint32\_t \*pui32Scalar, tECPt \*ptEcPt, tECCCurveInfo \*ptCurve, uint32\_t \*pui32ResultVector)
- void [PKAEnableInt](#) (void)
- tPKAStatus [PKAGetOpsStatus](#) (void)
- void [PKARegInt](#) (void (\*pfnHandler)(void))
- void [PKAUnRegInt](#) (void)

## 11.2.1 Detailed Description

The PKA low level driver API is broken into three groups of functions:

- Operations on the big numbers.
- ECC operations.
- Utility functions to control the pka interrupt operations.

The `pkaBigNumAddStart()` and `pkaBigNumAddGetResult`, `pkaBigNumMultiplyStart()` and `pkaBigNumMultGetResult()`, `pkaBigNumCmpStart()` and `pkaBigNumCmpGetResult()` provides the basic addition, multiplication, and compare operations on the big numbers; `pkaBigNumModStart()` and `pkaBigNumModGetResult()`, `pkaBigNumInvModStart()` and `pkaBigNumInvModGetResult()` provides the modulus and inverse of modulus of the big number respectively.

The `pkaECCAddStart()` and `pkaECCAddGetResult()` performs the addition of the two ECC points given the ECC curve information. The `pkaECCMultiplyStart()` and `pkaECCMultiplyGetResult()` provides the ability to multiply with the scalar quantity given the ECC curve information; `pkaECCMultGenPtStart()` and `pkaECCMultGenPtGetResult()` provides the ability to multiply the given scalar quantity with the known ECC (generator) point on the ECC curve.

To check the whether the status of the PKA operation, the PKA interrupt can be used along with the interrupt handler, alternatively the status of PKA operation can be polled using `pkaGetOpsStatus()`.

## 11.2.2 Function Documentation

### 11.2.2.1 PKABigNumAddGetResult

Gets the result of the addition operation on two big number.

#### Prototype:

```
tPKAStatus
PKABigNumAddGetResult (uint32_t *pui32ResultBuf,
                      uint32_t *pui32Len,
                      uint32_t ui32ResVectorLoc)
```

#### Parameters:

***pui32ResultBuf*** is the pointer to buffer where the result needs to be stored.

***pui32Len*** is the address of the variable containing the length of the buffer. After the operation the actual length of the resultant is stored at this address.

***ui32ResVectorLoc*** is the address of the result location which was provided by the start function

**See also:**

[PKABigNumAddStart\(\)](#).

This function gets the result of the addition operation on two big numbers, previously started using the function

**See also:**

[PKABigNumAddStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_FAILURE** if the operation is not successful.
- **PKA\_STATUS\_BUF\_UNDERFLOW** if the length of the provided buffer is less than the length of the result.

#### 11.2.2.2 tPKAStatus PKABigNumAddStart (uint32\_t \* *pui32BN1*, uint8\_t *ui8BN1Size*, uint32\_t \* *pui32BN2*, uint8\_t *ui8BN2Size*, uint32\_t \* *pui32ResultVector*)

Starts the addition of two big number.

**Parameters:**

***pui32BN1*** is the pointer to the buffer containing the first big number.

***ui8BN1Size*** is the size of the first big number in 32-bit word.

***pui32BN2*** is the pointer to the buffer containing the second big number.

***ui8BN2Size*** is the size of the second big number in 32-bit word.

***pui32ResultVector*** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the addition of the two big numbers.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

#### 11.2.2.3 PKABigNumCmpGetResult

Gets the result of the comparison operation of two big numbers.

**Prototype:**

```
tPKAStatus
PKABigNumCmpGetResult(void)
```

**Description:**

This function provides the results of the comparison of two big numbers which was started using the

**See also:**

[PKABigNumCmpStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_OPERATION\_INPRG** if the operation is in progress.
- **PKA\_STATUS\_SUCCESS** if the two big numbers are equal.
- **PKA\_STATUS\_A\_GR\_B** if the first number is greater than the second.
- **PKA\_STATUS\_A\_LT\_B** if the first number is less than the second.

#### 11.2.2.4 PKABigNumCmpStart

Starts the comparison of two big numbers.

**Prototype:**

```
tPKAStatus
PKABigNumCmpStart(uint32_t *pui32BNum1,
                  uint32_t *pui32BNum2,
                  uint8_t ui8Size)
```

**Parameters:**

***pui32BNum1*** is the pointer to the first big number.  
***pui32BNum2*** is the pointer to the second big number.  
***ui8Size*** is the size of the big number in 32 bit size word.

**Description:**

This function starts the comparison of two big numbers pointed by *pui32BNum1* and *pui32BNum2*. Note this function expects the size of the two big numbers equal.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

#### 11.2.2.5 PKABigNumInvModGetResult

Gets the result of the big number inverse modulo operation.



**Prototype:**

```
tPKAStatus
PKABigNumInvModGetResult (uint32_t *pui32ResultBuf,
                          uint8_t ui8Size,
                          uint32_t ui32ResVectorLoc)
```

**Parameters:**

***pui32ResultBuf*** is the pointer to buffer where the result needs to be stored.

***ui8Size*** is the size of the provided buffer in 32 bit *ui8Size* word.

***ui32ResVectorLoc*** is the address of the result location which was provided by the start function

**See also:**

[PKABigNumInvModStart\(\)](#).

This function gets the result of the big number inverse modulo operation previously started using the function

**See also:**

[PKABigNumInvModStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_BUF\_UNDERFLOW** if the length of the provided buffer is less then the result.

11.2.2.6 tPKAStatus PKABigNumInvModStart (uint32\_t \* *pui32BNum*, uint8\_t *ui8BNSize*, uint32\_t \* *pui32Modulus*, uint8\_t *ui8Size*, uint32\_t \* *pui32ResultVector*)

Starts the big number inverse modulo operation.

**Parameters:**

***pui32BNum*** is the pointer to the buffer containing the big number (dividend).

***ui8BNSize*** is the size of the *pui32BNum* in 32 bit word.

***pui32Modulus*** is the pointer to the buffer containing the divisor.

***ui8Size*** is the size of the divisor in 32 bit word.

***pui32ResultVector*** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the the inverse modulo operation on *pui32BNum* using the divisor *pui32Modulus*.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

### 11.2.2.7 PKABigNumModGetResult

Gets the result of the big number modulus operation.

**Prototype:**

```
tPKAStatus
PKABigNumModGetResult (uint32_t *pui32ResultBuf,
                       uint8_t ui8Size,
                       uint32_t ui32ResVectorLoc)
```

**Parameters:**

***pui32ResultBuf*** is the pointer to buffer where the result needs to be stored.

***ui8Size*** is the size of the provided buffer in 32 bit size word.

***ui32ResVectorLoc*** is the address of the result location which was provided by the start function

**See also:**

[PKABigNumModStart\(\)](#).

This function gets the result of the big number modulus operation which was previously started using the function

**See also:**

[PKABigNumModStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_BUF\_UNDERFLOW**, if the *ui8Size* is less than the length of the result.

### 11.2.2.8 tPKAStatus PKABigNumModStart (uint32\_t \* *pui32BNum*, uint8\_t *ui8BNSize*, uint32\_t \* *pui32Modulus*, uint8\_t *ui8ModSize*, uint32\_t \* *pui32ResultVector*)

Starts the big number modulus operation.

**Parameters:**

***pui32BNum*** is the pointer to the big number on which modulo operation needs to be carried out.

***ui8BNSize*** is the size of the big number

**See also:**

*pui32BNum* in 32-bit word.

**Parameters:**

***pui32Modulus*** is the pointer to the divisor.

***ui8ModSize*** is the size of the divisor

**See also:**

*pui32Modulus*.

**Parameters:**

***pui32ResultVector*** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the modulo operation on the big num

**See also:**

pui32BNum using the divisor

pui32Modulus. The PKA RAM location where the result will be available is stored in

pui32ResultVector.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

### 11.2.2.9 PKABigNumMultGetResult

Gets the results of the big number multiplication.

**Prototype:**

```
tPKAStatus
PKABigNumMultGetResult (uint32_t *pui32ResultBuf,
                        uint32_t *pui32Len,
                        uint32_t ui32ResVectorLoc)
```

**Parameters:**

***pui32ResultBuf*** is the pointer to buffer where the result needs to be stored.

***pui32Len*** is the address of the variable containing the length of the buffer. After the operation, the actual length of the resultant is stored at this address.

***ui32ResVectorLoc*** is the address of the result location which was provided by the start function

**See also:**

[PKABigNumMultiplyStart\(\)](#).

This function gets the result of the multiplication of two big numbers operation previously started using the function

**See also:**

[PKABigNumMultiplyStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_FAILURE** if the operation is not successful.

- **PKA\_STATUS\_BUF\_UNDERFLOW** if the length of the provided buffer is less than the length of the result.

#### 11.2.2.10 tPKAStatus PKABigNumMultiplyStart (uint32\_t \* *pui32Xplicand*, uint8\_t *ui8XplicandSize*, uint32\_t \* *pui32Xplier*, uint8\_t *ui8XplierSize*, uint32\_t \* *pui32ResultVector*)

Starts the big number multiplication.

##### Parameters:

- pui32Xplicand*** is the pointer to the buffer containing the big number multiplicand.
- ui8XplicandSize*** is the size of the multiplicand in 32-bit word.
- pui32Xplier*** is the pointer to the buffer containing the big number multiplier.
- ui8XplierSize*** is the size of the multiplier in 32-bit word.
- pui32ResultVector*** is the pointer to the result vector location which will be set by this function.

##### Description:

This function starts the multiplication of the two big numbers.

##### Returns:

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

#### 11.2.2.11 PKAClearInt

Clears the PKA interrupt.

##### Prototype:

```
void
PKAClearInt (void)
```

##### Description:

This function unpendes PKA interrupt. This will cause any previously generated PKA interrupts that have not been handled yet to be discarded.

##### Returns:

None.

#### 11.2.2.12 PKADisableInt

Disables the PKA interrupt.

##### Prototype:

```
void
PKADisableInt (void)
```

**Description:**

This function disables the PKA interrupt.

**Returns:**

None.

### 11.2.2.13 PKAECCAddGetResult

Gets the result of the ECC Addition

**Prototype:**

```
tPKAStatus
PKAECCAddGetResult (tECPt *ptOutEcPt,
                    uint32_t ui32ResVectorLoc)
```

**Parameters:**

**ptOutEcPt** is the pointer to the structure where the resultant point will be stored. The callee is responsible to allocate memory, for the ec point structure including the memory for x and y co-ordinate values.

**ui32ResVectorLoc** is the address of the result location which was provided by the function

**See also:**

[PKAECCAddStart\(\)](#).

This function gets the result of ecc point addition operation on the on the two given ec points, previously started using the function

**See also:**

[PKAECCAddStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_FAILURE** if the operation is not successful.

### 11.2.2.14 tPKAStatus PKAECCAddStart (tECPt \* ptEcPt1, tECPt \* ptEcPt2, tECCCurveInfo \* ptCurve, uint32\_t \* pui32ResultVector)

Starts the ECC Addition.

**Parameters:**

**ptEcPt1** is the pointer to the structure containing the first ecc point.

**ptEcPt2** is the pointer to the structure containing the second ecc point.

**ptCurve** is the pointer to the structure containing the curve info.

**pui32ResultVector** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the ecc point addition operation on the two given ec points and generates the resultant ecc point.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

### 11.2.2.15 PKAECCMultGenPtGetResult

Gets the result of ECC Multiplication with Generator point.

**Prototype:**

```
tPKAStatus
PKAECCMultGenPtGetResult (tECPt *ptOutEcPt,
                          uint32_t ui32ResVectorLoc)
```

**Parameters:**

**ptOutEcPt** is the pointer to the structure where the resultant EC point will be stored. The callee is responsible to allocate the space for the ec point structure and the x and y co-ordinate as well.

**ui32ResVectorLoc** is the address of the result location which was provided by the start function

**See also:**

[PKAECCMultGenPtStart\(\)](#).

This function gets the result of ecc point multiplication operation on the scalar point and the known generator point on the curve, previously started using the function

**See also:**

[PKAECCMultGenPtStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_FAILURE** if the operation is not successful.

### 11.2.2.16 tPKAStatus PKAECCMultGenPtStart (uint32\_t \* *pui32Scalar*, tECCCurveInfo \* *ptCurve*, uint32\_t \* *pui32ResultVector*)

Starts the ECC Multiplication with Generator point.

**Parameters:**

***pt32Scalar*** is the pointer to the buffer containing the scalar value.

***ptCurve*** is the pointer to the structure containing the curve info.

***pt32ResultVector*** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the ecc point multiplication operation of the scalar value with the well known generator point of the given curve.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

### 11.2.2.17 PKAECCMultiplyGetResult

Gets the result of ECC Multiplication

**Prototype:**

```
tPKAStatus  
PKAECCMultiplyGetResult (tECPt *ptOutEcPt,  
                        uint32_t ui32ResVectorLoc)
```

**Parameters:**

***ptOutEcPt*** is the pointer to the structure where the resultant EC point will be stored. The callee is responsible to allocate the space for the ec point structure and the x and y co-ordinate as well.

***ui32ResVectorLoc*** is the address of the result location which was provided by the start function

**See also:**

[PKAECCMultiplyStart\(\)](#).

This function gets the result of ecc point multiplication operation on the ec point and the scalar value, previously started using the function

**See also:**

[PKAECCMultiplyStart\(\)](#).

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if the operation is successful.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy performing the operation.
- **PKA\_STATUS\_RESULT\_0** if the result is all zeroes.
- **PKA\_STATUS\_FAILURE** if the operation is not successful.

#### 11.2.2.18 tPKAStatus PKAECCMultiplyStart (uint32\_t \* *pui32Scalar*, tECPt \* *ptEcPt*, tECCCurveInfo \* *ptCurve*, uint32\_t \* *pui32ResultVector*)

Starts ECC Multiplication.

**Parameters:**

***pui32Scalar*** is pointer to the buffer containing the scalar value to be multiplied.

***ptEcPt*** is the pointer to the structure containing the elliptic curve point to be multiplied. The point should be on the given curve.

***ptCurve*** is the pointer to the structure containing the curve info.

***pui32ResultVector*** is the pointer to the result vector location which will be set by this function.

**Description:**

This function starts the Elliptical curve cryptography (ECC) point multiplication operation on the EC point and the scalar value.

**Returns:**

Returns:

- **PKA\_STATUS\_SUCCESS** if successful in starting the operation.
- **PKA\_STATUS\_OPERATION\_INPRG**, if the PKA hw module is busy doing some other operation.

#### 11.2.2.19 PKAEnableInt

Enables the PKA interrupt.

**Prototype:**

```
void  
PKAEnableInt(void)
```

**Description:**

This function enables the PKA interrupt.

**Returns:**

None.

#### 11.2.2.20 PKAGetOpsStatus

Provides the PKA operation status.

**Prototype:**

```
tPKAStatus  
PKAGetOpsStatus(void)
```

**Description:**

This function provides information on whether any PKA operation is in progress or not. This function allows to check the PKA operation status before starting any new PKA operation.

**Returns:**

Returns:



- **PKA\_STATUS\_INPRG** if the PKA operation is in progress.
- **PKA\_STATUS\_OPERATION\_NOT\_INPRG** if the PKA operation is not in progress.

#### 11.2.2.21 PKARegInt

Registers an interrupt handler for PKA interrupt.

**Prototype:**

```
void  
PKARegInt(void (*pfnHandler)(void))
```

**Parameters:**

***pfnHandler*** is a pointer to the function to be called when the PKA interrupt occurs.

**Description:**

This function does the actual registering of the interrupt handler. This will not enable the PKA interrupt in the interrupt controller, a call to the function

**See also:**

[PKAEnableInt\(\)](#) is needed to enable the PKA interrupt.

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None.

#### 11.2.2.22 PKAUnRegInt

Unregisters an interrupt handler for the PKA interrupt.

**Prototype:**

```
void  
PKAUnRegInt(void)
```

**Description:**

This function deregisters the interrupt service routine. This function will not disable the interrupt and an explicit call to

**See also:**

[PKADisableInt\(\)](#) is needed.

**Returns:**

None.



# 12 Sleep Timer

Introduction .....	99
API Functions .....	100
Programming Example .....	104

## 12.1 Introduction

The sleep timer sets the period during which the system enters and exits low-power modes PM1 and PM2. The sleep timer also maintains timing in timer 2 when entering power mode PM1 or PM2. The main features of the sleep timer are the following:

- 32-bit timer up-counter operating at 32-kHz clock rate
- 32-bit compare with interrupt and DMA trigger
- 32-bit capture

The sleep timer is a 32-bit timer running on the 32-kHz clock (Clk32k). The timer starts running immediately after a reset and continues to run uninterrupted.

### Note:

The sleep timer runs when operating in all power modes except PM3. The value of the sleep timer is not preserved in PM3. When returning from PM1 or PM2 (where the system clock is shut down), the sleep timer value is not up to date until a positive edge on the 32-kHz clock is detected after the system clock restarts. To ensure an updated value is read, wait for a positive transition on the 32-kHz clock by polling the SYS\_CTRL\_CLOCK\_STA.SYNC\_32K bit before reading the sleep timer value.

A **timer compare event** occurs when the timer value is equal to the 32-bit compare value and there is a positive edge on the 32-kHz clock. The compare value is set by [SleepModeTimerCompareSet\(\)](#). In PM1 and PM2, the sleep timer compare event can be used to wake up the device and return to active operation in active mode. The default value of the compare value after reset is 0xFFFFFFFF. A compare event can trigger a sleep mode interrupt.

The **timer capture** occurs when the interrupt flag for a selected I/O pin is set and the 32-kHz clock has detected this. Sleep timer capture is enabled by [SleepModeCaptureNew\(\)](#). The I/O pin that triggers the capture is configured using [SleepModeCaptureConfig\(\)](#). The captured value is retrieved using [SleepModeCaptureGet\(\)](#). The returned value is one more than the value at the instant for the event on the I/O pin. Software should therefore subtract 1 from the captured value if absolute timing is required.

### Note:

Switching the input-capture pin is not possible while capture is enabled. Capture must be disabled before a new input-capture pin can be selected.

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

## 12.2 API Functions

### Functions

- void [SleepModeCaptureConfig](#) (uint32\_t ui32Port, uint32\_t ui32Pin)
- uint32\_t [SleepModeCaptureGet](#) (void)
- bool [SleepModeCapturesValid](#) (void)
- void [SleepModeCaptureNew](#) (void)
- void [SleepModeIntRegister](#) (void (\*pfnHandler)(void))
- void [SleepModeIntUnregister](#) (void)
- void [SleepModeTimerCompareSet](#) (uint32\_t ui32Compare)
- uint32\_t [SleepModeTimerCountGet](#) (void)

### 12.2.1 Detailed Description

The sleep timer is broken into three groups of functions:

- those that deal with the timer compare mode
- those that deal with the timer capture mode
- those that deal with timer interrupts

The timer compare value is set using [SleepModeTimerCompareSet\(\)](#). The value of the sleep mode counter can be retrieved using [SleepModeTimerCountGet\(\)](#).

The capture mode is configured and accessed using:

- [SleepModeCaptureConfig\(\)](#)
- [SleepModeCaptureGet\(\)](#)
- [SleepModeCapturesValid\(\)](#)

Timer interrupts are registered and unregistered using:

- [SleepModeIntRegister\(\)](#)
- [SleepModeIntUnregister\(\)](#)

### 12.2.2 Function Documentation

#### 12.2.2.1 SleepModeCaptureConfig

Selects capture port and pin

**Prototype:**

```
void  
SleepModeCaptureConfig (uint32_t ui32Port,  
                        uint32_t ui32Pin)
```

**Parameters:**

**ui32Port** is the port.

**ui32Pin** is the pin number.

**Description:**

This function sets the port and pin on which values are to be captured.

The *ui32Port* argument must be only one of the following values: **SLEEPMODE\_PORT\_A**, **SLEEPMODE\_PORT\_B**, **SLEEPMODE\_PORT\_C**, **SLEEPMODE\_PORT\_D**, **SLEEPMODE\_PORT\_USB**.

The *ui32Pin* argument must be only one of the following values: **SLEEPMODE\_PIN\_0**, **SLEEPMODE\_PIN\_1**, **SLEEPMODE\_PIN\_2**, **SLEEPMODE\_PIN\_3**, **SLEEPMODE\_PIN\_4**, **SLEEPMODE\_PIN\_5**, **SLEEPMODE\_PIN\_6**, **SLEEPMODE\_PIN\_7**.

**Note:**

if *ui32Port* is set to **SLEEPMODE\_PORT\_USB**, only *ui32Pin* **SLEEPMODE\_PIN\_0** can be used.

**Returns:**

None

### 12.2.2.2 SleepModeCaptureGet

Get last capture value

**Prototype:**

```
uint32_t  
SleepModeCaptureGet(void)
```

**Description:**

This function returns the last captured value.

**Note:**

The captured value is one more than the value at the instant for the event on the I/O pin. Software should therefore subtract 1 from the captured value if absolute timing is required.

**See also:**

[SleepModeCaptureNew\(\)](#), [SleepModeCapturesValid\(\)](#)

**Returns:**

Last captured value

### 12.2.2.3 SleepModeCapturesValid

Checks if capture value has been updated

**Prototype:**

```
bool  
SleepModeCaptureIsValid(void)
```

**Description:**

This function returns true if a value has been captured.

**See also:**

[SleepModeCaptureGet\(\)](#), [SleepModeCaptureNew\(\)](#)

**Returns:**

Returns true if capture value has been updated

#### 12.2.2.4 SleepModeCaptureNew

Prepares for a new value to be captured

**Prototype:**

```
void  
SleepModeCaptureNew(void)
```

**Description:**

This function prepares the capture logic to capture a new value.

The relevant pin interrupt flag must be cleared after calling this function using [IntPendClear\(\)](#).

**See also:**

[SleepModeCaptureGet\(\)](#), [SleepModeCaptureIsValid\(\)](#)

**Returns:**

None

#### 12.2.2.5 SleepModeIntRegister

Registers an interrupt handler for Sleep Mode Timer interrupt

**Prototype:**

```
void  
SleepModeIntRegister(void (*pfnHandler)(void))
```

**Parameters:**

***pfnHandler*** is a pointer to the function to be called when the Sleep Mode Timer interrupt occurs.

**Description:**

This function does the actual registering of the interrupt handler, thus enabling the global interrupt in the interrupt controller.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 12.2.2.6 SleepModeIntUnregister

Unregisters an interrupt handler for the sleep mode timer interrupt

**Prototype:**

```
void  
SleepModeIntUnregister(void)
```

**Description:**

This function does the actual unregistering of the interrupt handler. This function clears the handler to be called when a compare interrupt occurs and masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 12.2.2.7 SleepModeTimerCompareSet

Set compare value of the sleep mode timer

**Prototype:**

```
void  
SleepModeTimerCompareSet(uint32_t ui32Compare)
```

**Parameters:**

**ui32Compare** is a 32-bit compare value.

**Description:**

This function sets the compare value of the sleep mode timer. A timer compare interrupt is generated when the timer value is equal to the compare value.

**Note:**

When setting a new compare value, the value must be at least 5 more than the current sleep timer value. Otherwise, the timer compare event might be lost.

**Returns:**

None

### 12.2.2.8 SleepModeTimerCountGet

Get current value of the sleep mode timer

**Prototype:**

```
uint32_t  
SleepModeTimerCountGet(void)
```

**Description:**

This function returns the current value of the sleep mode timer (that is, the timer count)

**Returns:**

Current value of the sleep mode timer

## 12.3 Programming Example

The following example shows how to use the sleep timer API to wake up the system after 1000 32kHz cycles (interrupt handler not shown).

```
uint32_t ui32Val;

//
// Enable sleep mode interrupt
//
IntEnable(INT_SMTIM);

//
// Set timer to 1000 above current value
//
ui32Val = SleepModeTimerCountGet();
SleepModeTimerCompareSet(ui32Val + 1000);

//
// Go to sleep
//
SysCtrlSleep();
```



# 13 Synchronous Serial Interface (SSI)

Introduction .....	105
API Functions .....	105
Programming Example .....	115

## 13.1 Introduction

The synchronous serial interface (SSI) module provides the functionality for synchronous serial communications with peripheral devices, and can be configured to use either the Motorola® SPI™, National Semiconductor® Microwire, or the Texas Instruments® synchronous serial interface frame formats. The size of the data frame is also configurable, and can be set from 4 to 16 bits, inclusive.

The SSI module performs serial-to-parallel data conversion on data received from a peripheral device, and parallel-to-serial conversion on data transmitted to a peripheral device. The TX and RX paths are buffered with internal FIFOs allowing up to eight 16-bit values to be stored independently.

The SSI module can be configured as either a master or a slave device. As a slave device, the SSI module can also be configured to disable its output, which allows a master device to be coupled with multiple slave devices.

The SSI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the input clock of the SSI module. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

For parts that include a DMA controller, the SSI module also provides a DMA interface to facilitate data transfer through DMA.

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

## 13.2 API Functions

### Functions

- bool [SSIBusy](#) (uint32\_t ui32Base)
- uint32\_t [SSIClockSourceGet](#) (uint32\_t ui32Base)
- void [SSIClockSourceSet](#) (uint32\_t ui32Base, uint32\_t ui32Source)
- void [SSIConfigSetExpClk](#) (uint32\_t ui32Base, uint32\_t ui32SSIClk, uint32\_t ui32Protocol, uint32\_t ui32Mode, uint32\_t ui32BitRate, uint32\_t ui32DataWidth)
- void [SSIDataGet](#) (uint32\_t ui32Base, uint32\_t \*pui32Data)
- int32\_t [SSIDataGetNonBlocking](#) (uint32\_t ui32Base, uint32\_t \*pui32Data)
- void [SSIDataPut](#) (uint32\_t ui32Base, uint32\_t ui32Data)
- int32\_t [SSIDataPutNonBlocking](#) (uint32\_t ui32Base, uint32\_t ui32Data)
- void [SSIDisable](#) (uint32\_t ui32Base)
- void [SSIDMADisable](#) (uint32\_t ui32Base, uint32\_t ui32DMAFlags)
- void [SSIDMAEnable](#) (uint32\_t ui32Base, uint32\_t ui32DMAFlags)

- void [SSIEnable](#) (uint32\_t ui32Base)
- void [SSIIntClear](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [SSIIntDisable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [SSIIntEnable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [SSIIntRegister](#) (uint32\_t ui32Base, void (\*pfnHandler)(void))
- uint32\_t [SSIIntStatus](#) (uint32\_t ui32Base, bool bMasked)
- void [SSIIntUnregister](#) (uint32\_t ui32Base)

## 13.2.1 Detailed Description

The SSI API is broken into three groups of functions:

- those that deal with configuration and state
- those that handle data
- those that manage interrupts.

The configuration of the SSI module is managed by the [SSIConfigSetExpClk\(\)](#) function, while state is managed by the [SSIEnable\(\)](#) and [SSIDisable\(\)](#) functions. The DMA interface is enabled or disabled by the [SSIDMAEnable\(\)](#) and [SSIDMADisable\(\)](#) functions. The [SSIClockSourceGet\(\)](#) and [SSIClockSourceSet\(\)](#) functions manage the SSI baud clock.

The [SSIDataPut\(\)](#), [SSIDataPutNonBlocking\(\)](#), [SSIDataGet\(\)](#), and [SSIDataGetNonBlocking\(\)](#) functions perform data handling.

The [SSIIntClear\(\)](#), [SSIIntDisable\(\)](#), [SSIIntEnable\(\)](#), [SSIIntRegister\(\)](#), [SSIIntStatus\(\)](#), and [SSIIntUnregister\(\)](#) functions manage interrupts from the SSI module.

## 13.2.2 Function Documentation

### 13.2.2.1 SSIBusy

Determines whether the SSI transmitter is busy or not

**Prototype:**

```
bool
SSIBusy(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the SSI port.

**Description:**

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the hardware shift register.

**Returns:**

Returns **true** if the SSI is transmitting or **false** if all transmissions are complete.

### 13.2.2.2 SSIClockSourceGet

Gets the data clock source for the specified SSI peripheral

**Prototype:**

```
uint32_t  
SSIClockSourceGet (uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the SSI port.

**Description:**

This function returns the data clock source for the specified SSI. The possible data clock source are the system clock (**SSI\_CLOCK\_SYSTEM**) or the precision internal oscillator (**SSI\_CLOCK\_PIOSC**).

**Returns:**

None

### 13.2.2.3 SSIClockSourceSet

Sets the data clock source for the specified SSI peripheral

**Prototype:**

```
void  
SSIClockSourceSet (uint32_t ui32Base,  
                  uint32_t ui32Source)
```

**Parameters:**

**ui32Base** is the base address of the SSI port.

**ui32Source** is the baud clock source for the SSI.

**Description:**

This function allows the baud clock source for the SSI to be selected. The possible clock source are the system clock (**SSI\_CLOCK\_SYSTEM**) or the precision internal oscillator (**SSI\_CLOCK\_PIOSC**), i.e. the IO clock in the SysCtrl. If **SSI\_CLOCK\_SYSTEM** is chosen, the IO clock frequency must thus be queried by [SysCtrlClockSet\(\)](#). If **SSI\_CLOCK\_PIOSC** the [SysCtrlIOClockSet\(\)](#) function must be used.

Changing the baud clock source will change the data rate generated by the SSI. Therefore, the data rate should be reconfigured after any change to the SSI clock source.

**Returns:**

None

### 13.2.2.4 SSIConfigSetExpClk

Configures the synchronous serial interface

**Prototype:**

```
void
SSIConfigSetExpClk(uint32_t ui32Base,
                   uint32_t ui32SSIClk,
                   uint32_t ui32Protocol,
                   uint32_t ui32Mode,
                   uint32_t ui32BitRate,
                   uint32_t ui32DataWidth)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32SSIClk** is the rate of the clock supplied to the SSI module.

**ui32Protocol** specifies the data transfer protocol.

**ui32Mode** specifies the mode of operation.

**ui32BitRate** specifies the clock rate.

**ui32DataWidth** specifies number of bits transferred per frame.

**Description:**

This function configures the synchronous serial interface. It sets the SSI protocol, mode of operation, bit rate, and data width.

The *ui32Protocol* parameter defines the data frame format. The *ui32Protocol* parameter can be one of the following values: **SSI\_FRF\_MOTO\_MODE\_0**, **SSI\_FRF\_MOTO\_MODE\_1**, **SSI\_FRF\_MOTO\_MODE\_2**, **SSI\_FRF\_MOTO\_MODE\_3**, **SSI\_FRF\_TI**, or **SSI\_FRF\_NMW**. The Motorola frame formats imply the following polarity and phase configurations:

Polarity	Phase	Mode
0	0	SSI_FRF_MOTO_MODE_0
0	1	SSI_FRF_MOTO_MODE_1
1	0	SSI_FRF_MOTO_MODE_2
1	1	SSI_FRF_MOTO_MODE_3

The *ui32Mode* parameter defines the operating mode of the SSI module. The SSI module can operate as a master or slave; if a slave, the SSI can be configured to disable output on its serial output line. The *ui32Mode* parameter can be one of the following values: **SSI\_MODE\_MASTER**, **SSI\_MODE\_SLAVE**, or **SSI\_MODE\_SLAVE\_OD**.

The *ui32BitRate* parameter defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria:

- $F_{SSI} \geq 2 * \text{bit rate (master mode)}$
- $F_{SSI} \geq 12 * \text{bit rate (slave modes)}$

where  $F_{SSI}$  is the frequency of the clock supplied to the SSI module.

The *ui32DataWidth* parameter defines the width of the data transfers, and can be a value between 4 and 16, inclusive.

The peripheral clock is set in the System Control module. The frequency of the system clock is the value returned by [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#) depending on the chosen clock source as set by [SSIClockSourceSet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#)).

**Returns:**  
None

### 13.2.2.5 SSIDataGet

Gets a data element from the SSI receive FIFO

**Prototype:**

```
void  
SSIDataGet (uint32_t ui32Base,  
            uint32_t *pui32Data)
```

**Parameters:**  
*ui32Base* specifies the SSI module base address.  
*pui32Data* is a pointer to a storage location for data that was received over the SSI interface.

**Description:**  
This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the *pui32Data* parameter.

**Note:**  
Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SSISetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

**Returns:**  
None

### 13.2.2.6 SSIDataGetNonBlocking

Gets a data element from the SSI receive FIFO

**Prototype:**

```
int32_t  
SSIDataGetNonBlocking (uint32_t ui32Base,  
                       uint32_t *pui32Data)
```

**Parameters:**  
*ui32Base* specifies the SSI module base address.  
*pui32Data* is a pointer to a storage location for data that was received over the SSI interface.

**Description:**  
This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the *ui32Data* parameter. If there is no data in the FIFO, then this function returns a zero.

**Note:**  
Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SSISetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

**Returns:**  
Returns the number of elements read from the SSI receive FIFO.

### 13.2.2.7 SSIDataPut

Puts a data element into the SSI transmit FIFO

**Prototype:**

```
void  
SSIDataPut (uint32_t ui32Base,  
            uint32_t ui32Data)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32Data** is the data to be transmitted over the SSI interface.

**Description:**

This function places the supplied data into the transmit FIFO of the specified SSI module.

**Note:**

The upper 32 - N bits of the *ui32Data* are discarded by the hardware, where N is the data width as configured by [SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

**Returns:**

None

### 13.2.2.8 SSIDataPutNonBlocking

Puts a data element into the SSI transmit FIFO

**Prototype:**

```
int32_t  
SSIDataPutNonBlocking (uint32_t ui32Base,  
                       uint32_t ui32Data)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32Data** is the data to be transmitted over the SSI interface.

**Description:**

This function places the supplied data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function returns a zero.

**Note:**

The upper 32 - N bits of the *ui32Data* are discarded by the hardware, where N is the data width as configured by [SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

**Returns:**

Returns the number of elements written to the SSI transmit FIFO.

### 13.2.2.9 SSIDisable

Disables the synchronous serial interface

**Prototype:**

```
void  
SSIDisable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**Description:**

This function disables operation of the synchronous serial interface.

**Returns:**

None

### 13.2.2.10 SSIDMADisable

Disable SSI DMA operation

**Prototype:**

```
void  
SSIDMADisable(uint32_t ui32Base,  
               uint32_t ui32DMAFlags)
```

**Parameters:**

**ui32Base** is the base address of the SSI port.

**ui32DMAFlags** is a bit mask of the DMA features to disable.

**Description:**

This function is used to disable SSI DMA features that were enabled by [SSIDMAEnable\(\)](#). The specified SSI DMA features are disabled. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- SSI\_DMA\_RX - disable DMA for receive
- SSI\_DMA\_TX - disable DMA for transmit

**Returns:**

None

### 13.2.2.11 SSIDMAEnable

Enable SSI DMA operation

**Prototype:**

```
void  
SSIDMAEnable(uint32_t ui32Base,  
              uint32_t ui32DMAFlags)
```

**Parameters:**

**ui32Base** is the base address of the SSI port.

**ui32DMAFlags** is a bit mask of the DMA features to enable.

**Description:**

The specified SSI DMA features are enabled. The SSI can be configured to use DMA for transmit and/or receive data transfers. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- SSI\_DMA\_RX - enable DMA for receive
- SSI\_DMA\_TX - enable DMA for transmit

**Note:**

The uDMA controller must also be set up before DMA can be used with the SSI.

**Returns:**

None

### 13.2.2.12 SSIEnable

Enables the synchronous serial interface

**Prototype:**

```
void  
SSIEnable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**Description:**

This function enables operation of the synchronous serial interface. The synchronous serial interface must be configured before it is enabled.

**Returns:**

None

### 13.2.2.13 SSIIntClear

Clears SSI interrupt sources

**Prototype:**

```
void  
SSIIntClear(uint32_t ui32Base,  
            uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32IntFlags** is a bit mask of the interrupt sources to be cleared.



**Description:**

The specified SSI interrupt sources are cleared so that they no longer assert. This function must be called in the interrupt handler to keep the interrupts from being recognized again immediately upon exit. The *ui32IntFlags* parameter can consist of either or both the **SSI\_RXTO** and **SSI\_RXOR** values.

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None

### 13.2.2.14 SSIntDisable

Disables individual SSI interrupt sources

**Prototype:**

```
void  
SSIntDisable(uint32_t ui32Base,  
             uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32IntFlags** is a bit mask of the interrupt sources to be disabled.

**Description:**

Disables the indicated SSI interrupt sources. The *ui32IntFlags* parameter can be any of the **SSI\_TXFF**, **SSI\_RXFF**, **SSI\_RXTO**, or **SSI\_RXOR** values.

**Returns:**

None

### 13.2.2.15 SSIntEnable

Enables individual SSI interrupt sources

**Prototype:**

```
void  
SSIntEnable(uint32_t ui32Base,  
            uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**ui32IntFlags** is a bit mask of the interrupt sources to be enabled.

**Description:**

Enables the indicated SSI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The *ui32IntFlags* parameter can be any of the **SSI\_TXFF**, **SSI\_RXFF**, **SSI\_RXTO**, or **SSI\_RXOR** values.

**Returns:**

None

### 13.2.2.16 SSIntRegister

Registers an interrupt handler for the synchronous serial interface

**Prototype:**

```
void
SSIntRegister(uint32_t ui32Base,
              void (*pfnHandler) (void))
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**pfnHandler** is a pointer to the function to be called when the synchronous serial interface interrupt occurs.

**Description:**

This sets the handler to be called when an SSI interrupt occurs. This will enable the global interrupt in the interrupt controller; specific SSI interrupts must be enabled via [SSIntEnable\(\)](#). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via [SSIntClear\(\)](#).

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 13.2.2.17 SSIntStatus

Gets the current interrupt status

**Prototype:**

```
uint32_t
SSIntStatus(uint32_t ui32Base,
            bool bMasked)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**bMasked** is **false** if the raw interrupt status is required or **true** if the masked interrupt status is required.

**Description:**

This function returns the interrupt status for the SSI module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

The current interrupt status, enumerated as a bit field of **SSI\_TXFF**, **SSI\_RXFF**, **SSI\_RXT0**, and **SSI\_RXOR**.

### 13.2.2.18 SSIntUnregister

Unregisters an interrupt handler for the synchronous serial interface

**Prototype:**

```
void  
SSIntUnregister(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** specifies the SSI module base address.

**Description:**

This function will clear the handler to be called when a SSI interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

## 13.3 Programming Example

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

```
char *pcChars = "SSI Master send data.";
long lIdx;

//
// Configure the SSI.
//
SSIconfigSetExpClk(SSI0_BASE, SysCtrlIOClockGet(), SSI_FRF_MOTO_MODE_3,
                  SSI_MODE_MASTER, SysCtrlClockGet()/2, 8);

//
// Enable the SSI module.
//
SSIEnable(SSI0_BASE);

//
// Send some data.
//
lIdx = 0;
while(pcChars[lIdx])
```

```
{  
    SSIDataPut (SSI0_BASE, pcChars[lIdx]);  
    lIdx++;  
}
```

## 14 System Control (SysCtrl)

Introduction .....	117
API Functions .....	118
Programming Example .....	127

### 14.1 Introduction

The system controller determines the overall operation of the device, controlling the clocking of the device, the set of peripherals that are enabled under various power saving scenarios, and the configuration of the device and its resets. The system controller also provides information about the device.

The system controller controls the power management features that enables optimized power consumption for an application. Power management is based on three levels of power saving actions:

- Clock gating of unused or not required peripheral clocks
- Power down of clock sources
- Power down the power supply

Clock gating is the simplest power saving action and also the fastest to enter and exit. Powering down the supply is the most effective power saving action, but a power down / up sequence is more time demanding than clock gating or power down of clock sources. Please refer to the device user guide for a complete description of the power management features.

The device can be clocked from two sources: an external 32-MHz crystal oscillator or the internal 16-MHz RC oscillator. The 16-MHz RC oscillator consumes less power than the crystal oscillator but is not as accurate as the crystal oscillator and cannot be used for RF transceiver operation.

The CC2538 family supports three modes of operation:

- run mode
- sleep mode
- deep-sleep mode

In run mode, the processor is actively executing code. In sleep mode, the clocking of the device is unchanged but the processor no longer executes code (and is no longer clocked). In deep-sleep mode, the clocking of the device can change (depending on the power mode configuration) and the processor no longer executes code (and is no longer clocked). An interrupt returns the device to run mode from one of the sleep modes. The sleep modes are entered on request from the code.

Each peripheral in the device can be individually enabled, disabled, or reset. Additionally, the set of peripherals that remain enabled during sleep mode and deep-sleep mode can be configured, allowing custom sleep and deep-sleep modes to be defined.

In deep-sleep mode, the device can enter one of four available power modes (PM0 - PM3). In these power modes, increasingly aggressive power saving measures are applied, by powering clock sources and power supplies down. Please refer to the device user guide for a complete description of the features.

The members of the CC2538 family have a varying peripheral set and memory sizes. This information can be used to write adaptive software that can run on more than one member of the CC2538 family.

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

## 14.2 API Functions

### Functions

- `uint32_t SysCtrlClockGet` (void)
- `void SysCtrlClockLossDetectEnable` (void)
- `void SysCtrlClockSet` (bool bExternalOsc32k, bool bInternalOsc, uint32\_t ui32SysDiv)
- `void SysCtrlDeepSleep` (void)
- `void SysCtrlDelay` (uint32\_t ui32Count)
- `uint32_t SysCtrlIOClockGet` (void)
- `void SysCtrlIOClockSet` (uint32\_t ui32IODiv)
- `void SysCtrlPeripheralDeepSleepDisable` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralDeepSleepEnable` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralDisable` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralEnable` (uint32\_t ui32Peripheral)
- `bool SysCtrlPeripheralPresent` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralReset` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralSleepDisable` (uint32\_t ui32Peripheral)
- `void SysCtrlPeripheralSleepEnable` (uint32\_t ui32Peripheral)
- `uint32_t SysCtrlPowerModeGet` (void)
- `void SysCtrlPowerModeSet` (uint32\_t ui32PowerMode)
- `void SysCtrlReset` (void)
- `void SysCtrlSleep` (void)

### 14.2.1 Detailed Description

The SysCtrl API is broken up into five groups of functions:

- those that provide device information
- those that deal with device clocking
- those that provide power mode access
- those that deal clock loss detection
- those that deal with sleep modes and device reset

Information about the device is provided by `SysCtrlPeripheralPresent()`.

Clocking of the device and peripherals is configured with `SysCtrlClockSet()` and `SysCtrlIOClockSet()`. Information about device and peripheral clocking is provided by `SysCtrlClockGet()` and `SysCtrlIOClockGet()`.

Peripheral enabling and reset are controlled using the following functions:

- `SysCtrlPeripheralReset()`

- [SysCtrlPeripheralEnable\(\)](#)
- [SysCtrlPeripheralDisable\(\)](#)
- [SysCtrlPeripheralSleepEnable\(\)](#)
- [SysCtrlPeripheralSleepDisable\(\)](#)
- [SysCtrlPeripheralDeepSleepEnable\(\)](#)
- [SysCtrlPeripheralDeepSleepDisable\(\)](#)

The power mode that is entered when the Cortex-M3 enters Deep Sleep mode, is configured using [SysCtrlPowerModeSet\(\)](#). The current configuration is retrieved using [SysCtrlPowerModeGet\(\)](#).

The device is put into sleep modes with [SysCtrlSleep\(\)](#) and [SysCtrlDeepSleep\(\)](#). A full device reset can be initiated by calling the [SysCtrlReset\(\)](#) function.

## 14.2.2 Function Documentation

### 14.2.2.1 SysCtrlClockGet

Gets the processor clock rate

**Prototype:**

```
uint32_t  
SysCtrlClockGet(void)
```

**Description:**

This function determines the clock rate of the processor clock.

**Returns:**

The processor clock rate

### 14.2.2.2 SysCtrlClockLossDetectEnable

Enable Clock Loss Detection

**Prototype:**

```
void  
SysCtrlClockLossDetectEnable(void)
```

**Description:**

This function enables clock loss detection.

**Returns:**

None

### 14.2.2.3 SysCtrlClockSet

Sets the general clocking of the device

**Prototype:**

```
void
SysCtrlClockSet (bool bExternalOsc32k,
                 bool bInternalOsc,
                 uint32_t ui32SysDiv)
```

**Parameters:**

**bExternalOsc32k** is set to true for applications with external 32kHz crystal.

**bInternalOsc** selects internal 1-16MHz RC oscillator. If set to false, the external 0-32MHz crystal is used.

**ui32SysDiv** System Clock Setting.

**Description:**

This function configures the clocking of the device. The oscillator used and the system clock divider settings are configured with this function.

The *ui32SysDiv* argument must be only one of the following values:  
**SYS\_CTRL\_SYSDIV\_32MHZ**, **SYS\_CTRL\_SYSDIV\_16MHZ**, **SYS\_CTRL\_SYSDIV\_8MHZ**,  
**SYS\_CTRL\_SYSDIV\_4MHZ**, **SYS\_CTRL\_SYSDIV\_2MHZ**, **SYS\_CTRL\_SYSDIV\_1MHZ**,  
**SYS\_CTRL\_SYSDIV\_500KHZ**, **SYS\_CTRL\_SYSDIV\_250KHZ**. Note  
**SYS\_CTRL\_SYSDIV\_32MHZ** can not be selected when Internal Oscillator is selected.

**Returns:**

None

#### 14.2.2.4 SysCtrlDeepSleep

Puts the processor into deep-sleep mode

**Prototype:**

```
void
SysCtrlDeepSleep (void)
```

**Description:**

This function places the processor into deep-sleep mode and does not return until the processor returns to run mode. The peripherals that are enabled by [SysCtrlPeripheralDeepSleepEnable\(\)](#) continue to operate and can wake up the processor (if not in power mode 1, 2, or 3). Note the power mode should be set before going into deep sleep.

**See also:**

[SysCtrlPowerModeSet\(\)](#), [SysCtrlPeripheralDeepSleepEnable\(\)](#)

**Returns:**

None

#### 14.2.2.5 SysCtrlDelay

Provides a small delay

**Prototype:**

```
void
SysCtrlDelay (uint32_t ui32Count)
```



**Parameters:**

***ui32Count*** is the number of delay loop iterations to perform.

**Description:**

This function provides a means of generating a constant length delay and is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

**Returns:**

None

#### 14.2.2.6 SysCtrlIOClockGet

Gets the IO clock rate

**Prototype:**

```
uint32_t  
SysCtrlIOClockGet (void)
```

**Description:**

This function returns the IO clocking of the device, i.e. the Baud rate clock for SSI and UART.

**Returns:**

The IO clock rate

#### 14.2.2.7 SysCtrlIOClockSet

Sets the IO clocking of the device

**Prototype:**

```
void  
SysCtrlIOClockSet (uint32_t ui32IODiv)
```

**Parameters:**

***ui32IODiv*** System Clock Setting.

**Description:**

This function configures the IO clocking of the device (that is, the Baud rate clock for SSI and UART).

The *ui32IODiv* argument must be only one of the following values:  
**SYS\_CTRL\_SYSDIV\_32MHZ, SYS\_CTRL\_SYSDIV\_16MHZ, SYS\_CTRL\_SYSDIV\_8MHZ,  
SYS\_CTRL\_SYSDIV\_4MHZ, SYS\_CTRL\_SYSDIV\_2MHZ, SYS\_CTRL\_SYSDIV\_1MHZ,  
SYS\_CTRL\_SYSDIV\_500KHZ, SYS\_CTRL\_SYSDIV\_250KHZ.**

Note **SYS\_CTRL\_SYSDIV\_32MHZ** cannot be selected when Internal Oscillator is selected.

**Returns:**

None

### 14.2.2.8 SysCtrlPeripheralDeepSleepDisable

Disables a peripheral in deep-sleep mode

**Prototype:**

```
void  
SysCtrlPeripheralDeepSleepDisable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to disable in deep-sleep mode.

**Description:**

This function causes a peripheral to stop operating when the processor goes into deep-sleep mode. Disabling peripherals while in deep-sleep mode helps to lower the current draw of the device, and can keep peripherals that require a particular clock frequency from operating when the clock changes as a result of entering deep-sleep mode. If enabled (by [SysCtrlPeripheralEnable\(\)](#)), the peripheral automatically resumes operation when the processor leaves deep-sleep mode, maintaining its entire state from before entry into deep-sleep mode.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

None

### 14.2.2.9 SysCtrlPeripheralDeepSleepEnable

Enables a peripheral in deep-sleep mode

**Prototype:**

```
void  
SysCtrlPeripheralDeepSleepEnable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to enable in deep-sleep mode.

**Description:**

This function allows a peripheral to continue operating when the processor goes into deep-sleep mode. Because the clocking configuration of the device can change, not all peripherals can safely continue operating while the processor is in sleep mode. Safe operation depends on the chosen power mode. The caller must make sensible choices.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

None

### 14.2.2.10 SysCtrlPeripheralDisable

Disables a peripheral (in Run mode)

**Prototype:**

```
void  
SysCtrlPeripheralDisable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to disable.

**Description:**

Peripherals are disabled with this function. Once disabled, peripherals do not operate or respond to register reads/writes.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

None

### 14.2.2.11 SysCtrlPeripheralEnable

Enables a peripheral (in Run mode)

**Prototype:**

```
void  
SysCtrlPeripheralEnable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to enable.

**Description:**

Peripherals are enabled with this function. At power-up, some peripherals are disabled and must be enabled to operate or respond to register reads/writes.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Note:**

The actual enabling of the peripheral might be delayed until some time after this function returns. Ensure that the peripheral is not accessed until enabled.

**Returns:**

None

### 14.2.2.12 SysCtrlPeripheralPresent

Determines if a peripheral is present

**Prototype:**

```
bool  
SysCtrlPeripheralPresent (uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral in question.

**Description:**

Determines if a particular peripheral is present in the device (that is, is not permanently disabled).

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

Returns **true** if the specified peripheral is present and **false** if it is permanently disabled.

### 14.2.2.13 SysCtrlPeripheralReset

Performs a software reset of a peripheral

**Prototype:**

```
void  
SysCtrlPeripheralReset (uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to reset.

**Description:**

This function performs a software reset of the specified peripheral. An individual peripheral reset signal is asserted for a brief period and then deasserted, leaving the peripheral in a operating state but in its reset condition.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**.

**Returns:**

None

### 14.2.2.14 SysCtrlPeripheralSleepDisable

Disables a peripheral in sleep mode

**Prototype:**

```
void  
SysCtrlPeripheralSleepDisable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to disable in sleep mode.

**Description:**

This function causes a peripheral to stop operating when the processor goes into sleep mode. Disabling peripherals while in sleep mode helps lower the current draw of the device. If enabled (by [SysCtrlPeripheralEnable\(\)](#)), the peripheral automatically resume operation when the processor leaves sleep mode, maintaining the entire state before entry into sleep mode.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

None

#### 14.2.2.15 SysCtrlPeripheralSleepEnable

Enables a peripheral in sleep mode

**Prototype:**

```
void  
SysCtrlPeripheralSleepEnable(uint32_t ui32Peripheral)
```

**Parameters:**

***ui32Peripheral*** is the peripheral to enable in sleep mode.

**Description:**

This function allows a peripheral to continue operating when the processor goes into sleep mode. Because the clocking configuration of the device does not change, any peripheral can safely continue operating while the processor is in sleep mode, and can therefore wake the processor from sleep mode.

The *ui32Peripheral* parameter must be one of the values: **SYS\_CTRL\_PERIPH\_GPT0**, **SYS\_CTRL\_PERIPH\_GPT1**, **SYS\_CTRL\_PERIPH\_GPT2**, **SYS\_CTRL\_PERIPH\_GPT3**, **SYS\_CTRL\_PERIPH\_SSI0**, **SYS\_CTRL\_PERIPH\_SSI1**, **SYS\_CTRL\_PERIPH\_UART0**, **SYS\_CTRL\_PERIPH\_UART1**, **SYS\_CTRL\_PERIPH\_I2C**, **SYS\_CTRL\_PERIPH\_PKA**, **SYS\_CTRL\_PERIPH\_AES**, **SYS\_CTRL\_PERIPH\_RFC**.

**Returns:**

None

#### 14.2.2.16 SysCtrlPowerModeGet

Get Power Mode

**Prototype:**

```
uint32_t  
SysCtrlPowerModeGet (void)
```

**Description:**

This function returns the current Power Mode setting.

**Returns:**

Power mode, i.e. one of the following values: **SYS\_CTRL\_PM\_NOACTION**, **SYS\_CTRL\_PM\_1**, **SYS\_CTRL\_PM\_2** or **SYS\_CTRL\_PM\_3**.

### 14.2.2.17 SysCtrlPowerModeSet

Set Power Mode

**Prototype:**

```
void  
SysCtrlPowerModeSet (uint32_t ui32PowerMode)
```

**Parameters:**

***ui32PowerMode*** is the power mode to be entered.

**Description:**

This function selects the power mode to enter when CM3 enters Deep Sleep mode. Power mode PM0 (**SYS\_CTRL\_PM\_NOACTION**) is entered when the CPU wakes up due to an interrupt. Note only transitions to and from PM0 are legal (that is, PM1 to PM2 cannot happen).

The *ui32PowerMode* argument must be only one of the following values: **SYS\_CTRL\_PM\_NOACTION**, **SYS\_CTRL\_PM\_1**, **SYS\_CTRL\_PM\_2** or **SYS\_CTRL\_PM\_3**.

**See also:**

[SysCtrlDeepSleep\(\)](#).

**Returns:**

None

### 14.2.2.18 SysCtrlReset

Resets the device

**Prototype:**

```
void  
SysCtrlReset (void)
```

**Description:**

This function performs a software reset of the entire device. The processor and all peripherals are reset and all device registers are returned to their default values.

**Returns:**

This function does not return.

### 14.2.2.19 SysCtrlSleep

Puts the processor into sleep mode

**Prototype:**

```
void  
SysCtrlSleep(void)
```

**Description:**

This function places the processor into sleep mode and does not return until the processor returns to run mode. The peripherals that are enabled by [SysCtrlPeripheralSleepEnable\(\)](#) continue to operate and can wake up the processor.

**See also:**

[SysCtrlPeripheralSleepEnable\(\)](#)

**Returns:**

None

## 14.3 Programming Example

The following example shows how to use the SysCtrl API to configure the device for normal operation.

```
//  
// Set the clocking to run directly from the external crystal/oscillator.  
// (no ext 32k osc, no internal osc)  
//  
SysCtrlClockSet(false, false, SYS_CTRL_SYSDIV_32MHZ);  
  
//  
// Set IO clock to the same as system clock  
//  
SysCtrlIOClockSet(SYS_CTRL_SYSDIV_32MHZ);
```





# 15 System Tick (SysTick)

Introduction .....	129
API Functions .....	129
Programming Example .....	133

## 15.1 Introduction

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M microprocessor. Its intended purpose is to provide a periodic interrupt for an RTOS, but it can be used for other simple timing purposes.

The SysTick interrupt handler does not need to clear the SysTick interrupt source because the NVIC automatically does so when the SysTick interrupt handler is called.

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

## 15.2 API Functions

### Functions

- void [SysTickDisable](#) (void)
- void [SysTickEnable](#) (void)
- void [SysTickIntDisable](#) (void)
- void [SysTickIntEnable](#) (void)
- void [SysTickIntRegister](#) (void (\*pfnHandler)(void))
- void [SysTickIntUnregister](#) (void)
- uint32\_t [SysTickPeriodGet](#) (void)
- void [SysTickPeriodSet](#) (uint32\_t ui32Period)
- uint32\_t [SysTickValueGet](#) (void)

### 15.2.1 Detailed Description

Like SysTick, the SysTick API is fairly simple. There are functions for configuring and enabling SysTick ([SysTickEnable\(\)](#), [SysTickDisable\(\)](#), [SysTickPeriodSet\(\)](#), [SysTickPeriodGet\(\)](#), and [SysTickValueGet\(\)](#)) and functions for dealing with an interrupt handler for SysTick ([SysTickIntRegister\(\)](#), [SysTickIntUnregister\(\)](#), [SysTickIntEnable\(\)](#), and [SysTickIntDisable\(\)](#)).

### 15.2.2 Function Documentation

#### 15.2.2.1 SysTickDisable

Disables the SysTick counter

**Prototype:**

```
void  
SysTickDisable(void)
```

**Description:**

This will stop the SysTick counter. If an interrupt handler has been registered, it will no longer be called until SysTick is restarted.

**Returns:**

None

### 15.2.2.2 SysTickEnable

Enables the SysTick counter

**Prototype:**

```
void  
SysTickEnable(void)
```

**Description:**

This function starts the SysTick counter. If an interrupt handler has been registered, it is called when the SysTick counter rolls over.

**Note:**

Calling this function causes the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to [SysTickPeriodSet\(\)](#). If an immediate reload is required, the **NVIC\_ST\_CURRENT** register must be written to force the reload. Any write to this register clears the SysTick counter to 0 and causes a reload with the supplied period on the next clock.

**Returns:**

None

### 15.2.2.3 SysTickIntDisable

Disables the SysTick interrupt

**Prototype:**

```
void  
SysTickIntDisable(void)
```

**Description:**

This function will disable the SysTick interrupt, preventing it from being reflected to the processor.

**Returns:**

None

#### 15.2.2.4 SysTickIntEnable

Enables the SysTick interrupt

**Prototype:**

```
void  
SysTickIntEnable(void)
```

**Description:**

This function will enable the SysTick interrupt, allowing it to be reflected to the processor.

**Note:**

The SysTick interrupt handler does not need to clear the SysTick interrupt source as this is done automatically by NVIC when the interrupt handler is called.

**Returns:**

None

#### 15.2.2.5 SysTickIntRegister

Registers an interrupt handler for the SysTick interrupt

**Prototype:**

```
void  
SysTickIntRegister(void (*pfnHandler) (void))
```

**Parameters:**

***pfnHandler*** is a pointer to the function to be called when the SysTick interrupt occurs.

**Description:**

This sets the handler to be called when a SysTick interrupt occurs.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

#### 15.2.2.6 SysTickIntUnregister

Unregisters the interrupt handler for the SysTick interrupt

**Prototype:**

```
void  
SysTickIntUnregister(void)
```

**Description:**

This function will clear the handler to be called when a SysTick interrupt occurs.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 15.2.2.7 SysTickPeriodGet

Gets the period of the SysTick counter

**Prototype:**

```
uint32_t  
SysTickPeriodGet(void)
```

**Description:**

This function returns the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

**Returns:**

Returns the period of the SysTick counter.

### 15.2.2.8 SysTickPeriodSet

Sets the period of the SysTick counter

**Prototype:**

```
void  
SysTickPeriodSet(uint32_t ui32Period)
```

**Parameters:**

**ui32Period** is the number of clock ticks in each period of the SysTick counter; must be between 1 and 16, 777, 216, inclusive.

**Description:**

This function sets the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

**Note:**

Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC\_ST\_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and will cause a reload with the *ui32Period* supplied here on the next clock after the SysTick is enabled.

**Returns:**

None

### 15.2.2.9 SysTickValueGet

Gets the current value of the SysTick counter

**Prototype:**

```
uint32_t  
SysTickValueGet(void)
```

**Description:**

This function returns the current value of the SysTick counter; this will be a value between the period - 1 and zero, inclusive.

**Returns:**

Returns the current value of the SysTick counter.

## 15.3 Programming Example

The following example shows how to use the SysTick API to configure the SysTick counter and read its value.

```
unsigned long ulValue;

//
// Configure and enable the SysTick counter.
//
SysTickPeriodSet(1000);
SysTickEnable();

//
// Delay for some time...
//

//
// Read the current SysTick value.
//
ulValue = SysTickValueGet();
```



## 16 General Purpose Timer

Introduction .....	135
API Functions .....	136
Programming Example .....	149

### 16.1 Introduction

The timer API provides a set of functions for using the timer module. Functions are provided to configure and control the timer, modify timer/counter values, and manage timer interrupt handling.

The timer module provides two half-width timers/counters that can be configured to operate independently as timers or event counters or to operate as a combined full-width timer. The timers provide 16-bit half-width timers and a 32-bit full-width timer. For the purposes of this API, the two half-width timers provided by a timer module are referred to as TimerA and TimerB, and the full-width timer is referred to as TimerA.

When configured as either a full-width or half-width timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured in one-shot mode, the timer ceases counting when it reaches 0 when counting down or the load value when counting up. If configured in continuous mode, the timer counts to 0 (counting down) or the load value (counting up), then reloads and continues counting. When configured as a full-width timer, the timer can also be configured to operate as an RTC. In this mode, the timer expects to be driven by a 32.768-KHz external clock, which is divided down to produce 1-second clock ticks.

When in half-width mode, the timer can also be configured for event capture or as a pulse width modulation (PWM) generator. When configured for event capture, the timer acts as a counter. It can be configured to count either the time between events or the events themselves. The type of event being counted can be configured as a positive edge, a negative edge, or both edges. When a timer is configured as a PWM generator, the input signal used to capture events becomes an output signal, and the timer drives an edge-aligned pulse onto that signal.

The timer module also provides the ability to control other functional parameters, such as output inversion, output triggers, and timer behavior during stalls.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured, or that a certain number of events have been captured. Interrupts can also be generated when the timer has counted down to 0 or when the timer matches a certain value.

On some parts, the counters from multiple timer modules can be synchronized. Synchronized counters are useful in PWM and edge time capture modes. In PWM mode, the PWM outputs from multiple timers can be in lock-step by having the same load value and synchronizing the counters (meaning that the counters always have the same value). Similarly, by using the same load value and synchronized counters in edge time capture mode, the absolute time between two input edges is easily measured.

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

## 16.2 API Functions

### Functions

- void [TimerConfigure](#) (uint32\_t ui32Base, uint32\_t ui32Config)
- void [TimerControlEvent](#) (uint32\_t ui32Base, uint32\_t ui32Timer, uint32\_t ui32Event)
- void [TimerControlLevel](#) (uint32\_t ui32Base, uint32\_t ui32Timer, bool bInvert)
- void [TimerControlStall](#) (uint32\_t ui32Base, uint32\_t ui32Timer, bool bStall)
- void [TimerControlTrigger](#) (uint32\_t ui32Base, uint32\_t ui32Timer, bool bEnable)
- void [TimerControlWaitOnTrigger](#) (uint32\_t ui32Base, uint32\_t ui32Timer, bool bWait)
- void [TimerDisable](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- void [TimerEnable](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- void [TimerIntClear](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [TimerIntDisable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [TimerIntEnable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [TimerIntRegister](#) (uint32\_t ui32Base, uint32\_t ui32Timer, void (\*pfnHandler)(void))
- uint32\_t [TimerIntStatus](#) (uint32\_t ui32Base, bool bMasked)
- void [TimerIntUnregister](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- uint32\_t [TimerLoadGet](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- void [TimerLoadSet](#) (uint32\_t ui32Base, uint32\_t ui32Timer, uint32\_t ui32Value)
- uint32\_t [TimerMatchGet](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- void [TimerMatchSet](#) (uint32\_t ui32Base, uint32\_t ui32Timer, uint32\_t ui32Value)
- uint32\_t [TimerPrescaleGet](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- uint32\_t [TimerPrescaleMatchGet](#) (uint32\_t ui32Base, uint32\_t ui32Timer)
- void [TimerPrescaleMatchSet](#) (uint32\_t ui32Base, uint32\_t ui32Timer, uint32\_t ui32Value)
- void [TimerPrescaleSet](#) (uint32\_t ui32Base, uint32\_t ui32Timer, uint32\_t ui32Value)
- void [TimerSynchronize](#) (uint32\_t ui32Base, uint32\_t ui32Timers)
- uint32\_t [TimerValueGet](#) (uint32\_t ui32Base, uint32\_t ui32Timer)

### 16.2.1 Detailed Description

The timer API is broken into three groups of functions:

- those that deal with timer configuration and control
- those that deal with timer contents
- those that deal with interrupt handling.

Timer configuration is handled by [TimerConfigure\(\)](#), which performs the high level setup of the timer module; that is, it is used to set up full- or half-width modes, and to select between PWM, capture, and timer operations.

The [TimerEnable\(\)](#), [TimerDisable\(\)](#), [TimerControlLevel\(\)](#), [TimerControlTrigger\(\)](#), [TimerControlEvent\(\)](#) and [TimerControlStall\(\)](#) functions perform timer control.

The [TimerLoadSet\(\)](#), [TimerLoadGet\(\)](#), [TimerPrescaleSet\(\)](#), [TimerPrescaleGet\(\)](#), [TimerMatchSet\(\)](#), [TimerMatchGet\(\)](#), [TimerPrescaleMatchSet\(\)](#), [TimerPrescaleMatchGet\(\)](#), [TimerValueGet\(\)](#), and [TimerSynchronize\(\)](#) functions manage timer content.



The [TimerIntRegister\(\)](#) and [TimerIntUnregister\(\)](#) functions manage the interrupt handler for the Timer interrupt. The individual interrupt sources within the timer module are managed with [TimerIntEnable\(\)](#), [TimerIntDisable\(\)](#), [TimerIntStatus\(\)](#), and [TimerIntClear\(\)](#).

## 16.2.2 Function Documentation

### 16.2.2.1 TimerConfigure

Configures the timer(s)

**Prototype:**

```
void
TimerConfigure(uint32_t ui32Base,
               uint32_t ui32Config)
```

**Parameters:**

***ui32Base*** is the base address of the timer module.

***ui32Config*** is the configuration for the timer.

**Description:**

This function configures the operating mode of the timer(s). The timer module is disabled before being configured, and is left in the disabled state. The 16/32-bit timer is comprised of two 16-bit timers that can operate independently or be concatenated to form a 32-bit timer.

The configuration is specified in *ui32Config* as one of the following values:

- **GPTIMER\_CFG\_ONE\_SHOT** - Full-width one-shot timer
- **GPTIMER\_CFG\_ONE\_SHOT\_UP** - Full-width one-shot timer that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_PERIODIC** - Full-width periodic timer
- **GPTIMER\_CFG\_PERIODIC\_UP** - Full-width periodic timer that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_SPLIT\_PAIR** - Two half-width timers

When configured for a pair of half-width timers, each timer is separately configured. The first timer is configured by setting *ui32Config* to the result of a logical OR operation between one of the following values and *ui32Config*:

- **GPTIMER\_CFG\_A\_ONE\_SHOT** - Half-width one-shot timer
- **GPTIMER\_CFG\_A\_ONE\_SHOT\_UP** - Half-width one-shot timer that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_A\_PERIODIC** - Half-width periodic timer
- **GPTIMER\_CFG\_A\_PERIODIC\_UP** - Half-width periodic timer that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_A\_CAP\_COUNT** - Half-width edge count capture
- **GPTIMER\_CFG\_A\_CAP\_COUNT\_UP** - Half-width edge count capture that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_A\_CAP\_TIME** - Half-width edge time capture
- **GPTIMER\_CFG\_A\_CAP\_TIME\_UP** - Half-width edge time capture that counts up instead of down (not available on all parts)
- **GPTIMER\_CFG\_A\_PWM** - Half-width PWM output

Similarly, the second timer is configured by setting *ui32Config* to the result of a logical OR operation between one of the corresponding **GPTIMER\_CFG\_B\_\*** values and *ui32Config*.

**Returns:**

None

### 16.2.2.2 TimerControlEvent

Controls the event type

**Prototype:**

```
void  
TimerControlEvent (uint32_t ui32Base,  
                   uint32_t ui32Timer,  
                   uint32_t ui32Event)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to be adjusted; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**ui32Event** specifies the type of event; must be one of **GPTIMER\_EVENT\_POS\_EDGE**, **GPTIMER\_EVENT\_NEG\_EDGE**, or **GPTIMER\_EVENT\_BOTH\_EDGES**.

**Description:**

This function sets the signal edge(s) that triggers the timer when in capture mode.

**Returns:**

None

### 16.2.2.3 TimerControlLevel

Controls the output level

**Prototype:**

```
void  
TimerControlLevel (uint32_t ui32Base,  
                  uint32_t ui32Timer,  
                  bool bInvert)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to adjust; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**bInvert** specifies the output level.

**Description:**

This function sets the PWM output level for the specified timer. If the *bInvert* parameter is **true**, then the timer's output is made active low; otherwise, it is made active high.

**Returns:**

None

#### 16.2.2.4 TimerControlStall

Controls the stall handling

**Prototype:**

```
void  
TimerControlStall(uint32_t ui32Base,  
                  uint32_t ui32Timer,  
                  bool bStall)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to be adjusted; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**bStall** specifies the response to a stall signal.

**Description:**

This function controls the stall response for the specified timer. If the *bStall* parameter is **true**, then the timer stops counting if the processor enters debug mode; otherwise the timer keeps running while in debug mode.

**Returns:**

None

#### 16.2.2.5 TimerControlTrigger

Enables or disables the trigger output

**Prototype:**

```
void  
TimerControlTrigger(uint32_t ui32Base,  
                    uint32_t ui32Timer,  
                    bool bEnable)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer to adjust; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**bEnable** specifies the desired trigger state.

**Description:**

This function controls the trigger output for the specified timer. If the *bEnable* parameter is **true**, then the timer's output trigger is enabled; otherwise it is disabled.

**Returns:**

None

#### 16.2.2.6 TimerControlWaitOnTrigger

Controls the wait on trigger handling

**Prototype:**

```
void  
TimerControlWaitOnTrigger(uint32_t ui32Base,  
                           uint32_t ui32Timer,  
                           bool bWait)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to be adjusted; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**bWait** specifies if the timer should wait for a trigger input.

**Description:**

This function controls whether or not a timer waits for a trigger input to start counting. When enabled, the previous timer in the trigger chain must count to its timeout in order for this timer to start counting. Refer to the part's data sheet for a description of the trigger chain.

**Note:**

This functionality is not available on all parts.

**Returns:**

None

### 16.2.2.7 TimerDisable

Disables the timer(s)

**Prototype:**

```
void  
TimerDisable(uint32_t ui32Base,  
              uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to disable; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**Description:**

This function disables operation of the timer module.

**Returns:**

None

### 16.2.2.8 TimerEnable

Enables the timer(s)

**Prototype:**

```
void  
TimerEnable(uint32_t ui32Base,  
             uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to enable; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**Description:**

This function enables operation of the timer module. The timer must be configured before it is enabled.

**Returns:**

None

### 16.2.2.9 TimerIntClear

Clears timer interrupt sources

**Prototype:**

```
void  
TimerIntClear(uint32_t ui32Base,  
              uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32IntFlags** is a bit mask of the interrupt sources to be cleared.

**Description:**

The specified timer interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [TimerIntEnable\(\)](#).

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None

### 16.2.2.10 TimerIntDisable

Disables individual timer interrupt sources

**Prototype:**

```
void  
TimerIntDisable(uint32_t ui32Base,  
                uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32IntFlags** is the bit mask of the interrupt sources to be disabled.

**Description:**

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

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [TimerIntEnable\(\)](#).

**Returns:**

None

### 16.2.2.11 TimerIntEnable

Enables individual timer interrupt sources

**Prototype:**

```
void  
TimerIntEnable(uint32_t ui32Base,  
               uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32IntFlags** is the bit mask of the interrupt sources to be enabled.

**Description:**

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

The *ui32IntFlags* parameter must be the logical OR of any combination of the following:

- **GPTIMER\_CAPB\_EVENT** - Capture B event interrupt
- **GPTIMER\_CAPB\_MATCH** - Capture B match interrupt
- **GPTIMER\_TIMB\_TIMEOUT** - Timer B timeout interrupt
- **GPTIMER\_CAPA\_EVENT** - Capture A event interrupt
- **GPTIMER\_CAPA\_MATCH** - Capture A match interrupt
- **GPTIMER\_TIMA\_TIMEOUT** - Timer A timeout interrupt

**Returns:**

None

### 16.2.2.12 TimerIntRegister

Registers an interrupt handler for the timer interrupt

**Prototype:**

```
void  
TimerIntRegister(uint32_t ui32Base,  
                uint32_t ui32Timer,  
                void (*pfnHandler)(void))
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s); must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**pfnHandler** is a pointer to the function to be called when the timer interrupt occurs.

**Description:**

This function sets the handler to be called when a timer interrupt occurs. In addition, this function enables the global interrupt in the interrupt controller; specific timer interrupts must be enabled via [TimerIntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source via [TimerIntClear\(\)](#).

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 16.2.2.13 TimerIntStatus

Gets the current interrupt status

**Prototype:**

```
uint32_t  
TimerIntStatus(uint32_t ui32Base,  
               bool bMasked)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**bMasked** is false if the raw interrupt status is required and true if the masked interrupt status is required.

**Description:**

This function returns the interrupt status for the timer module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

The current interrupt status, enumerated as a bit field of values described in [TimerIntEnable\(\)](#).

### 16.2.2.14 TimerIntUnregister

Unregisters an interrupt handler for the timer interrupt

**Prototype:**

```
void  
TimerIntUnregister(uint32_t ui32Base,  
                  uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s); must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**Description:**

This function clears the handler to be called when a timer interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

See [IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 16.2.2.15 TimerLoadGet

Gets the timer load value

**Prototype:**

```
uint32_t
TimerLoadGet (uint32_t ui32Base,
              uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer; must be one of **GPTIMER\_A** or **GPTIMER\_B**. Only **GPTIMER\_A** should be used when the timer is configured for 32-bit operation.

**Description:**

This function gets the currently programmed interval load value for the specified timer.

**Returns:**

Returns the load value for the timer.

### 16.2.2.16 TimerLoadSet

Sets the timer load value

**Prototype:**

```
void
TimerLoadSet (uint32_t ui32Base,
              uint32_t ui32Timer,
              uint32_t ui32Value)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to adjust; must be one of: **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**. Only **GPTIMER\_A** should be used when the timer is configured for 32-bit operation.

**ui32Value** is the load value.



**Description:**

This function sets the timer load value; if the timer is running then the value will be immediately loaded into the timer.

**Returns:**

None

### 16.2.2.17 TimerMatchGet

Gets the timer match value

**Prototype:**

```
uint32_t  
TimerMatchGet (uint32_t ui32Base,  
               uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer; must be one of **GPTIMER\_A** or **GPTIMER\_B**. Only **GPTIMER\_A** should be used when the timer is configured for 32-bit operation.

**Description:**

This function gets the match value for the specified timer.

**Returns:**

Returns the match value for the timer.

### 16.2.2.18 TimerMatchSet

Sets the timer match value

**Prototype:**

```
void  
TimerMatchSet (uint32_t ui32Base,  
               uint32_t ui32Timer,  
               uint32_t ui32Value)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to adjust; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**. Only **GPTIMER\_A** should be used when the timer is configured for 32-bit operation.

**ui32Value** is the match value.

**Description:**

This function sets the match value for a timer. This is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal.

**Returns:**

None

### 16.2.2.19 TimerPrescaleGet

Get the timer prescale value

**Prototype:**

```
uint32_t  
TimerPrescaleGet (uint32_t ui32Base,  
                  uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer; must be one of **GPTIMER\_A** or **GPTIMER\_B**.

**Description:**

This function gets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

**Note:**

The availability of the prescaler varies with the timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

**Returns:**

The value of the timer prescaler

### 16.2.2.20 TimerPrescaleMatchGet

Get the timer prescale match value

**Prototype:**

```
uint32_t  
TimerPrescaleMatchGet (uint32_t ui32Base,  
                       uint32_t ui32Timer)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer; must be one of **GPTIMER\_A** or **GPTIMER\_B**.

**Description:**

This function gets the value of the input clock prescaler match value. When in a 16-bit mode that uses the counter match and prescaler, the prescale match effectively extends the range of the counter to 24-bits.

**Note:**

The availability of the prescaler match varies with the timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

**Returns:**

The value of the timer prescale match.

### 16.2.2.21 TimerPrescaleMatchSet

Set the timer prescale match value

**Prototype:**

```
void  
TimerPrescaleMatchSet (uint32_t ui32Base,  
                        uint32_t ui32Timer,  
                        uint32_t ui32Value)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to adjust; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**ui32Value** is the timer prescale match value; must be between 0 and 255, inclusive.

**Description:**

This function sets the value of the input clock prescaler match value. When in a 16-bit mode that uses the counter match and the prescaler, the prescale match effectively extends the range of the counter to 24-bits.

**Note:**

The availability of the prescaler match varies with the timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

**Returns:**

None

### 16.2.2.22 TimerPrescaleSet

Set the timer prescale value

**Prototype:**

```
void  
TimerPrescaleSet (uint32_t ui32Base,  
                  uint32_t ui32Timer,  
                  uint32_t ui32Value)
```

**Parameters:**

**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer(s) to adjust; must be one of **GPTIMER\_A**, **GPTIMER\_B**, or **GPTIMER\_BOTH**.

**ui32Value** is the timer prescale value; must be between 0 and 255, inclusive.

**Description:**

This function sets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

**Note:**

The availability of the prescaler varies with the timer mode in use. Please consult the datasheet for the part you are using to determine whether this support is available.

**Returns:**  
None

### 16.2.2.23 TimerSynchronize

Synchronizes the counters in a set of timers

**Prototype:**  

```
void  
TimerSynchronize(uint32_t ui32Base,  
                 uint32_t ui32Timers)
```

**Parameters:**  
**ui32Base** is the base address of the timer module. This must be the base address of Timer0 (in other words, **GPTIMER0\_BASE**).  
**ui32Timers** is the set of timers to synchronize.

**Description:**  
This function will synchronize the counters in a specified set of timers. When a timer is running in half-width mode, each half can be included or excluded in the synchronization event. When a timer is running in full-width mode, only the A timer can be synchronized (specifying the B timer has no effect).

The *ui32Timers* parameter is the logical OR of any of the following defines:

- **GPTIMER\_0A\_SYNC**
- **GPTIMER\_0B\_SYNC**
- **GPTIMER\_1A\_SYNC**
- **GPTIMER\_1B\_SYNC**
- **GPTIMER\_2A\_SYNC**
- **GPTIMER\_2B\_SYNC**
- **GPTIMER\_3A\_SYNC**
- **GPTIMER\_3B\_SYNC**

**Note:**  
This functionality is not available on all parts.

**Returns:**  
None

### 16.2.2.24 TimerValueGet

Gets the current timer value

**Prototype:**  

```
uint32_t  
TimerValueGet(uint32_t ui32Base,  
              uint32_t ui32Timer)
```

**Parameters:**  
**ui32Base** is the base address of the timer module.

**ui32Timer** specifies the timer; must be one of **GPTIMER\_A** or **GPTIMER\_B**. Only **GPTIMER\_A** should be used when the timer is configured for 32-bit operation.

**Description:**

This function reads the current value of the specified timer.

**Returns:**

Returns the current value of the timer.

## 16.3 Programming Example

The following example shows how to use the timer API to configure the timer as a half-width one-shot timer and a half-width edge capture counter.

```
//  
// Configure TimerA as a half-width one-shot timer, and TimerB as a  
// half-width edge capture counter.  
//  
TimerConfigure(GPTIMER0_BASE, (GPTIMER_CFG_SPLIT_PAIR | GPTIMER_CFG_A_ONE_SHOT  
                                GPTIMER_CFG_B_CAP_COUNT));  
  
//  
// Set the count time for the the one-shot timer (TimerA).  
//  
TimerLoadSet(GPTIMER0_BASE, GPTIMER_A, 3000);  
  
//  
// Configure the counter (TimerB) to count both edges.  
//  
TimerControlEvent(GPTIMER0_BASE, GPTIMER_B, GPTIMER_EVENT_BOTH_EDGES);  
  
//  
// Enable the timers.  
//  
TimerEnable(GPTIMER0_BASE, GPTIMER_BOTH);
```



# 17 UART

Introduction .....	151
API Functions .....	151
Programming Example .....	170

## 17.1 Introduction

The universal asynchronous receiver/transmitter (UART) API provides a set of functions for using the CC2538 UART modules. Functions are provided to configure and control the UART modules, to send and receive data, and to manage interrupts for the UART modules.

The CC2538 UART performs the functions of parallel-to-serial and serial-to-parallel conversions. It is very similar in functionality to a 16C550 UART, but is not register-compatible.

Some of the features of the CC2538 UART are:

- Separate receive (RX) FIFO and transmit (TX) 16x8-bit FIFOs
- Programmable baud rate generator
- Automatic generation and stripping of start, stop, and parity bits
- Line break generation and detection
- Programmable serial interface
  - 5, 6, 7, or 8 data bits
  - Even, odd, stick, or no parity bit generation and detection
  - 1 or 2 stop bit generation
- Modem control/flow control
- IrDA serial-IR (SIR) encoder/decoder
- DMA interface
- 9-bit operation

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

## 17.2 API Functions

### Functions

- void [UART9BitAddrSend](#) (uint32\_t ui32Base, uint8\_t ui8Addr)
- void [UART9BitAddrSet](#) (uint32\_t ui32Base, uint8\_t ui8Addr, uint8\_t ui8Mask)
- void [UART9BitDisable](#) (uint32\_t ui32Base)
- void [UART9BitEnable](#) (uint32\_t ui32Base)
- void [UARTBreakCtl](#) (uint32\_t ui32Base, bool bBreakState)
- bool [UARTBusy](#) (uint32\_t ui32Base)
- int32\_t [UARTCharGet](#) (uint32\_t ui32Base)

- int32\_t [UARTCharGetNonBlocking](#) (uint32\_t ui32Base)
- void [UARTCharPut](#) (uint32\_t ui32Base, uint8\_t ui8Data)
- bool [UARTCharPutNonBlocking](#) (uint32\_t ui32Base, uint8\_t ui8Data)
- bool [UARTCharsAvail](#) (uint32\_t ui32Base)
- uint32\_t [UARTClockSourceGet](#) (uint32\_t ui32Base)
- void [UARTClockSourceSet](#) (uint32\_t ui32Base, uint32\_t ui32Source)
- void [UARTConfigGetExpClk](#) (uint32\_t ui32Base, uint32\_t ui32UARTClk, uint32\_t \*pui32Baud, uint32\_t \*pui32Config)
- void [UARTConfigSetExpClk](#) (uint32\_t ui32Base, uint32\_t ui32UARTClk, uint32\_t ui32Baud, uint32\_t ui32Config)
- void [UARTDisable](#) (uint32\_t ui32Base)
- void [UARTDisableSIR](#) (uint32\_t ui32Base)
- void [UARTDMADisable](#) (uint32\_t ui32Base, uint32\_t ui32DMAFlags)
- void [UARTDMAEnable](#) (uint32\_t ui32Base, uint32\_t ui32DMAFlags)
- void [UARTEnable](#) (uint32\_t ui32Base)
- void [UARTEnableSIR](#) (uint32\_t ui32Base, bool bLowPower)
- void [UARTFIFODisable](#) (uint32\_t ui32Base)
- void [UARTFIFOEnable](#) (uint32\_t ui32Base)
- void [UARTFIFOLevelGet](#) (uint32\_t ui32Base, uint32\_t \*pui32TxLevel, uint32\_t \*pui32RxLevel)
- void [UARTFIFOLevelSet](#) (uint32\_t ui32Base, uint32\_t ui32TxLevel, uint32\_t ui32RxLevel)
- void [UARTIntClear](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [UARTIntDisable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [UARTIntEnable](#) (uint32\_t ui32Base, uint32\_t ui32IntFlags)
- void [UARTIntRegister](#) (uint32\_t ui32Base, void (\*pfnHandler)(void))
- uint32\_t [UARTIntStatus](#) (uint32\_t ui32Base, bool bMasked)
- void [UARTIntUnregister](#) (uint32\_t ui32Base)
- uint32\_t [UARTParityModeGet](#) (uint32\_t ui32Base)
- void [UARTParityModeSet](#) (uint32\_t ui32Base, uint32\_t ui32Parity)
- void [UARTRxErrorClear](#) (uint32\_t ui32Base)
- uint32\_t [UARTRxErrorGet](#) (uint32\_t ui32Base)
- bool [UARTSpaceAvail](#) (uint32\_t ui32Base)
- uint32\_t [UARTTxIntModeGet](#) (uint32\_t ui32Base)
- void [UARTTxIntModeSet](#) (uint32\_t ui32Base, uint32\_t ui32Mode)

## 17.2.1 Detailed Description

The UART API provides the set of functions required to implement an interrupt-driven UART driver. These functions may be used to control any of the available UART ports on a CC2538 device and can be used with one port without causing conflicts with the other port.

The UART API is broken into three groups of functions:

- those that deal with configuration and control of the UART modules
- those used to send and receive data
- those that deal with interrupt handling.



The [UARTClockSourceSet\(\)](#) and [UARTClockSourceGet\(\)](#) functions handle the clock source for the baud rate generator.

The [UARTConfigGetExpClk\(\)](#), [UARTConfigSetExpClk\(\)](#), [UARTDisable\(\)](#), [UARTEnable\(\)](#), [UARTParityModeGet\(\)](#), and [UARTParityModeSet\(\)](#) functions handle configuration and control of the UART.

The [UARTDMAEnable\(\)](#) and [UARTDMADisable\(\)](#) functions enabled or disabled the DMA interface.

The [UARTCharGet\(\)](#), [UARTCharGetNonBlocking\(\)](#), [UARTCharPut\(\)](#), [UARTCharPutNonBlocking\(\)](#), [UARTBreakCtl\(\)](#), [UARTCharsAvail\(\)](#), and [UARTSpaceAvail\(\)](#) functions handle sending and receiving data through the UART.

The [UARTIntClear\(\)](#), [UARTIntDisable\(\)](#), [UARTIntEnable\(\)](#), [UARTIntRegister\(\)](#), [UARTIntStatus\(\)](#), and [UARTIntUnregister\(\)](#) functions manage the UART interrupts.

The [UART9BitEnable\(\)](#), [UART9BitDisable\(\)](#), [UART9BitAddrSet\(\)](#), and [UART9BitAddrSend\(\)](#) functions handle the 9-bit operation mode.

## 17.2.2 Function Documentation

### 17.2.2.1 UART9BitAddrSend

Sends an address character from the specified port when operating in 9-bit mode

**Prototype:**

```
void
UART9BitAddrSend(uint32_t ui32Base,
                 uint8_t ui8Addr)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

***ui8Addr*** is the address to be transmitted.

**Description:**

This function waits until all data has been sent from the specified port and then sends the given address as an address byte. It then waits until the address byte has been transmitted before returning.

The normal data functions ([UARTCharPut\(\)](#), [UARTCharPutNonBlocking\(\)](#), [UARTCharGet\(\)](#), and [UARTCharGetNonBlocking\(\)](#)) are used to send and receive data characters in 9-bit mode.

**Returns:**

None

### 17.2.2.2 UART9BitAddrSet

Sets the device address(es) for 9-bit mode

**Prototype:**

```
void
UART9BitAddrSet(uint32_t ui32Base,
                uint8_t ui8Addr,
                uint8_t ui8Mask)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui8Addr** is the device address.

**ui8Mask** is the device address mask.

**Description:**

This function sets the device address, or range of device addresses, that respond to requests on the 9-bit UART port. The received address is masked with the mask and then compared against the given address, allowing either a single address (if **ui8Mask** is 0xff) or a set of addresses to be matched.

**Returns:**

None

### 17.2.2.3 UART9BitDisable

Disables 9-bit mode on the specified UART

**Prototype:**

```
void  
UART9BitDisable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function disables the 9-bit operational mode of the UART.

**Returns:**

None

### 17.2.2.4 UART9BitEnable

Enables 9-bit mode on the specified UART

**Prototype:**

```
void  
UART9BitEnable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function enables the 9-bit operational mode of the UART.

**Returns:**

None

### 17.2.2.5 UARTBreakCtl

Causes a BREAK to be sent

**Prototype:**

```
void
UARTBreakCtl(uint32_t ui32Base,
              bool bBreakState)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**bBreakState** controls the output level.

**Description:**

Calling this function with *bBreakState* set to **true** asserts a break condition on the UART. Calling this function with *bBreakState* set to **false** removes the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

**Returns:**

None

### 17.2.2.6 UARTBusy

Determines whether the UART transmitter is busy or not

**Prototype:**

```
bool
UARTBusy(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit FIFO is empty and all bits of the last transmitted character, including all stop bits, have left the hardware shift register.

**Returns:**

Returns **true** if the UART is transmitting or **false** if all transmissions are complete.

### 17.2.2.7 UARTCharGet

Waits for a character from the specified port

**Prototype:**

```
int32_t
UARTCharGet(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

**Returns:**

Returns the character read from the specified port, cast as a *int32\_t*.

### 17.2.2.8 UARTCharGetNonBlocking

Receives a character from the specified port

**Prototype:**

```
int32_t
UARTCharGetNonBlocking(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

**Description:**

This function gets a character from the receive FIFO for the specified port.

**Returns:**

Returns the character read from the specified port, cast as a *int32\_t*. A -1 is returned if there are no characters present in the receive FIFO. The [UARTCharsAvail\(\)](#) function should be called before attempting to call this function.

### 17.2.2.9 UARTCharPut

Waits to send a character from the specified port

**Prototype:**

```
void
UARTCharPut(uint32_t ui32Base,
            uint8_t ui8Data)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

***ui8Data*** is the character to be transmitted.

**Description:**

This function sends the character *ui8Data* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

**Returns:**

None

### 17.2.2.10 UARTCharPutNonBlocking

Sends a character to the specified port

**Prototype:**

```
bool
UARTCharPutNonBlocking(uint32_t ui32Base,
                       uint8_t ui8Data)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui8Data** is the character to be transmitted.

**Description:**

This function writes the character *ui8Data* to the transmit FIFO for the specified port. This function does not block, so if there is no space available, then a **false** is returned, and the application must retry the function later.

**Returns:**

Returns **true** if the character was successfully placed in the transmit FIFO or **false** if there was no space available in the transmit FIFO.

### 17.2.2.11 UARTCharsAvail

Determines if there are any characters in the receive FIFO

**Prototype:**

```
bool
UARTCharsAvail(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function returns a flag indicating whether or not there is data available in the receive FIFO.

**Returns:**

Returns **true** if there is data in the receive FIFO or **false** if there is no data in the receive FIFO.

### 17.2.2.12 UARTClockSourceGet

Gets the baud clock source for the specified UART

**Prototype:**

```
uint32_t
UARTClockSourceGet(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function returns the baud clock source for the specified UART. The possible baud clock source are the system clock (**UART\_CLOCK\_SYSTEM**) or the precision internal oscillator (**UART\_CLOCK\_PIOSC**).

**Returns:**

None

### 17.2.2.13 UARTClockSourceSet

Sets the baud clock source for the specified UART

**Prototype:**

```
void
UARTClockSourceSet (uint32_t ui32Base,
                    uint32_t ui32Source)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32Source** is the baud clock source for the UART.

**Description:**

This function allows the baud clock source for the UART to be selected. The possible clock source are the system clock (**UART\_CLOCK\_SYSTEM**) or the precision internal oscillator (**UART\_CLOCK\_PIOSC**).

If **UART\_CLOCK\_SYSTEM** is chosen, the IO clock frequency must thus be queried by SysCtrlClkSet(). If **UART\_CLOCK\_PIOSC** the SysCtrlIOClkSet() function must be used.

Changing the baud clock source will change the baud rate generated by the UART. Therefore, the baud rate should be reconfigured after any change to the baud clock source.

**Note:**

If the precision internal oscillator (**UART\_CLOCK\_PIOSC**) is used for the UART baud clock, the system clock frequency must be at least 9 MHz in Run mode.

**See also:**

[UARTConfigSetExpClk\(\)](#)

**Returns:**

None

### 17.2.2.14 UARTConfigGetExpClk

Gets the current configuration of a UART

**Prototype:**

```
void
UARTConfigGetExpClk (uint32_t ui32Base,
                    uint32_t ui32UARTClk,
                    uint32_t *pui32Baud,
                    uint32_t *pui32Config)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.  
***ui32UARTClk*** is the rate of the clock supplied to the UART module.  
***pui32Baud*** is a pointer to storage for the baud rate.  
***pui32Config*** is a pointer to storage for the data format.

**Description:**

The baud rate and data format for the UART is determined, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an “official” baud rate. The data format returned in *pui32Config* is enumerated the same as the *ui32Config* parameter of [UARTConfigSetExpClk\(\)](#).

The peripheral clock is set in the System Control module. The frequency of the system clock is the value returned by [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#) depending on the chosen clock source as set by [UARTClockSourceSet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#)).

The CC2538 part has the ability to specify the UART baud clock source (via [UARTClockSourceSet\(\)](#)), the peripheral clock can be changed to PIOSC. In this case, the peripheral clock should be specified as 16, 000, 000 (the nominal rate of PIOSC).

**Returns:**

None

### 17.2.2.15 UARTConfigSetExpClk

Sets the configuration of a UART

**Prototype:**

```
void
UARTConfigSetExpClk(uint32_t ui32Base,
                    uint32_t ui32UARTClk,
                    uint32_t ui32Baud,
                    uint32_t ui32Config)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.  
***ui32UARTClk*** is the rate of the clock supplied to the UART module.  
***ui32Baud*** is the desired baud rate.  
***ui32Config*** is the data format for the port (number of data bits, number of stop bits, and parity).

**Description:**

This function configures the UART for operation in the specified data format. The baud rate is provided in the *ui32Baud* parameter and the data format in the *ui32Config* parameter.

The *ui32Config* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **UART\_CONFIG\_WLEN\_8**, **UART\_CONFIG\_WLEN\_7**, **UART\_CONFIG\_WLEN\_6**, and **UART\_CONFIG\_WLEN\_5** select from eight to five data bits per byte (respectively). **UART\_CONFIG\_STOP\_ONE** and **UART\_CONFIG\_STOP\_TWO** select one or two stop bits (respectively). **UART\_CONFIG\_PAR\_NONE**, **UART\_CONFIG\_PAR\_EVEN**, **UART\_CONFIG\_PAR\_ODD**,

**UART\_CONFIG\_PAR\_ONE**, and **UART\_CONFIG\_PAR\_ZERO** select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

The peripheral clock is set in the System Control module. The frequency of the system clock is the value returned by [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#) depending on the chosen clock source as set by [UARTClockSourceSet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [SysCtrlClockGet\(\)](#) or [SysCtrlIOClockGet\(\)](#)).

The CC2538 part has the ability to specify the UART baud clock source (via [UARTClockSourceSet\(\)](#)), the peripheral clock can be changed to PIOSC. In this case, the peripheral clock should be specified as 16, 000, 000 (the nominal rate of PIOSC).

**See also:**

See [UARTClockSourceSet\(\)](#)

**Returns:**

None

### 17.2.2.16 UARTDisable

Disables transmitting and receiving

**Prototype:**

```
void  
UARTDisable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function clears the UARTEN, TXE, and RXE bits, waits for the end of transmission of the current character, and flushes the transmit FIFO.

**Returns:**

None

### 17.2.2.17 UARTDisableSIR

Disables SIR (IrDA) mode on the specified UART

**Prototype:**

```
void  
UARTDisableSIR(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function clears the SIREN (IrDA) and SIRLP (Low Power) bits.



**Returns:**

None

### 17.2.2.18 UARTDMADisable

Disable UART DMA operation

**Prototype:**

```
void
UARTDMADisable (uint32_t ui32Base,
                 uint32_t ui32DMAFlags)
```

**Parameters:*****ui32Base*** is the base address of the UART port.***ui32DMAFlags*** is a bit mask of the DMA features to disable.**Description:**

This function is used to disable UART DMA features that were enabled by [UARTDMAEnable\(\)](#). The specified UART DMA features are disabled. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- UART\_DMA\_RX - disable DMA for receive
- UART\_DMA\_TX - disable DMA for transmit
- UART\_DMA\_ERR\_RXSTOP - do not disable DMA receive on UART error

**Returns:**

None

### 17.2.2.19 UARTDMAEnable

Enable UART DMA operation

**Prototype:**

```
void
UARTDMAEnable (uint32_t ui32Base,
               uint32_t ui32DMAFlags)
```

**Parameters:*****ui32Base*** is the base address of the UART port.***ui32DMAFlags*** is a bit mask of the DMA features to enable.**Description:**

The specified UART DMA features are enabled. The UART can be configured to use DMA for transmit or receive, and to disable receive if an error occurs. The *ui32DMAFlags* parameter is the logical OR of any of the following values:

- UART\_DMA\_RX - enable DMA for receive
- UART\_DMA\_TX - enable DMA for transmit
- UART\_DMA\_ERR\_RXSTOP - disable DMA receive on UART error

**Note:**

The uDMA controller must also be set up before DMA can be used with the UART.

**Returns:**

None

### 17.2.2.20 UARTEnable

Enables transmitting and receiving

**Prototype:**

```
void  
UARTEnable(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function sets the UARTEN, TXE, and RXE bits, and enables the transmit and receive FIFOs.

**Returns:**

None

### 17.2.2.21 UARTEnableSIR

Enables SIR (IrDA) mode on the specified UART

**Prototype:**

```
void  
UARTEnableSIR(uint32_t ui32Base,  
               bool bLowPower)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**bLowPower** indicates if SIR Low Power Mode is to be used.

**Description:**

This function enables the SIREN control bit for IrDA mode on the UART. If the *bLowPower* flag is set, then SIRLP bit will also be set.

**Returns:**

None

### 17.2.2.22 UARTFIFODisable

Disables the transmit and receive FIFOs

**Prototype:**

```
void  
UARTFIFODisable(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

**Description:**

This functions disables the transmit and receive FIFOs in the UART.

**Returns:**

None

### 17.2.2.23 UARTFIFOEnable

Enables the transmit and receive FIFOs

**Prototype:**

```
void
UARTFIFOEnable (uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

**Description:**

This functions enables the transmit and receive FIFOs in the UART.

**Returns:**

None

### 17.2.2.24 UARTFIFOLevelGet

Gets the FIFO level at which interrupts are generated

**Prototype:**

```
void
UARTFIFOLevelGet (uint32_t ui32Base,
                  uint32_t *pui32TxLevel,
                  uint32_t *pui32RxLevel)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

***pui32TxLevel*** is a pointer to storage for the transmit FIFO level, returned as one of **UART\_FIFO\_TX1\_8**, **UART\_FIFO\_TX2\_8**, **UART\_FIFO\_TX4\_8**, **UART\_FIFO\_TX6\_8**, or **UART\_FIFO\_TX7\_8**.

***pui32RxLevel*** is a pointer to storage for the receive FIFO level, returned as one of **UART\_FIFO\_RX1\_8**, **UART\_FIFO\_RX2\_8**, **UART\_FIFO\_RX4\_8**, **UART\_FIFO\_RX6\_8**, or **UART\_FIFO\_RX7\_8**.

**Description:**

This function gets the FIFO level at which transmit and receive interrupts are generated.

**Returns:**

None

### 17.2.2.25 UARTFIFOLevelSet

Sets the FIFO level at which interrupts are generated

**Prototype:**

```
void
UARTFIFOLevelSet (uint32_t ui32Base,
                  uint32_t ui32TxLevel,
                  uint32_t ui32RxLevel)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32TxLevel** is the transmit FIFO interrupt level, specified as one of **UART\_FIFO\_TX1\_8**, **UART\_FIFO\_TX2\_8**, **UART\_FIFO\_TX4\_8**, **UART\_FIFO\_TX6\_8**, or **UART\_FIFO\_TX7\_8**.

**ui32RxLevel** is the receive FIFO interrupt level, specified as one of **UART\_FIFO\_RX1\_8**, **UART\_FIFO\_RX2\_8**, **UART\_FIFO\_RX4\_8**, **UART\_FIFO\_RX6\_8**, or **UART\_FIFO\_RX7\_8**.

**Description:**

This function sets the FIFO level at which transmit and receive interrupts are generated.

**Returns:**

None

### 17.2.2.26 UARTIntClear

Clears UART interrupt sources

**Prototype:**

```
void
UARTIntClear (uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32IntFlags** is a bit mask of the interrupt sources to be cleared.

**Description:**

The specified UART interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being recognized again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [UARTIntEnable\(\)](#).

**Note:**

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

**Returns:**

None

### 17.2.2.27 UARTIntDisable

Disables individual UART interrupt sources

**Prototype:**

```
void
UARTIntDisable (uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32IntFlags** is the bit mask of the interrupt sources to be disabled.

**Description:**

This function 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.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [UARTIntEnable\(\)](#).

**Returns:**

None

### 17.2.2.28 UARTIntEnable

Enables individual UART interrupt sources

**Prototype:**

```
void
UARTIntEnable (uint32_t ui32Base,
               uint32_t ui32IntFlags)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32IntFlags** is the bit mask of the interrupt sources to be enabled.

**Description:**

This function enables 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.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **UART\_INT\_9BIT** - 9-bit address match interrupt
- **UART\_INT\_OE** - Overrun Error interrupt
- **UART\_INT\_BE** - Break Error interrupt
- **UART\_INT\_PE** - Parity Error interrupt
- **UART\_INT\_FE** - Framing Error interrupt
- **UART\_INT\_RT** - Receive Timeout interrupt
- **UART\_INT\_TX** - Transmit interrupt
- **UART\_INT\_RX** - Receive interrupt
- **UART\_INT\_CTS** - CTS interrupt (UART1 only)

**Returns:**

None

### 17.2.2.29 UARTIntRegister

Registers an interrupt handler for a UART interrupt

**Prototype:**

```
void
UARTIntRegister(uint32_t ui32Base,
                void (*pfnHandler)(void))
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**pfnHandler** is a pointer to the function to be called when the UART interrupt occurs.

**Description:**

This function does the actual registering of the interrupt handler. This function enables the global interrupt in the interrupt controller; specific UART interrupts must be enabled via [UART-IntEnable\(\)](#). It is the interrupt handler's responsibility to clear the interrupt source.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 17.2.2.30 UARTIntStatus

Gets the current interrupt status

**Prototype:**

```
uint32_t
UARTIntStatus(uint32_t ui32Base,
               bool bMasked)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**bMasked** is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

**Description:**

This function returns the interrupt status for the specified UART. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

Returns the current interrupt status, enumerated as a bit field of values described in [UARTIntEnable\(\)](#).

### 17.2.2.31 UARTIntUnregister

Unregisters an interrupt handler for a UART interrupt

**Prototype:**

```
void
UARTIntUnregister(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function does the actual unregistering of the interrupt handler. It clears the handler to be called when a UART interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

### 17.2.2.32 UARTParityModeGet

Gets the type of parity currently being used

**Prototype:**

```
uint32_t
UARTParityModeGet(uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function gets the type of parity used for transmitting data and expected when receiving data.

**Returns:**

Returns the current parity settings, specified as one of **UART\_CONFIG\_PAR\_NONE**, **UART\_CONFIG\_PAR\_EVEN**, **UART\_CONFIG\_PAR\_ODD**, **UART\_CONFIG\_PAR\_ONE**, or **UART\_CONFIG\_PAR\_ZERO**.

### 17.2.2.33 UARTParityModeSet

Sets the type of parity

**Prototype:**

```
void
UARTParityModeSet(uint32_t ui32Base,
                  uint32_t ui32Parity)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32Parity** specifies the type of parity to use.

**Description:**

This function sets the type of parity to use for transmitting and expect when receiving. The *ui32Parity* parameter must be one of **UART\_CONFIG\_PAR\_NONE**, **UART\_CONFIG\_PAR\_EVEN**, **UART\_CONFIG\_PAR\_ODD**, **UART\_CONFIG\_PAR\_ONE**, or **UART\_CONFIG\_PAR\_ZERO**. The last two allow direct control of the parity bit; it is always either one or zero based on the mode.

**Returns:**

None

### 17.2.2.34 UARTRxErrorClear

Clears all reported receiver errors

**Prototype:**

```
void  
UARTRxErrorClear(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

**Description:**

This function is used to clear all receiver error conditions reported via [UARTRxErrorGet\(\)](#). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

**Returns:**

None

### 17.2.2.35 UARTRxErrorGet

Gets current receiver errors

**Prototype:**

```
uint32_t  
UARTRxErrorGet(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the base address of the UART port.

**Description:**

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to [UARTCharGet\(\)](#) or [UARTCharGetNonBlocking\(\)](#) with the exception that the overrun error is set immediately the overrun occurs rather than when a character is next read.

**Returns:**

Returns a logical OR combination of the receiver error flags, **UART\_RXERROR\_FRAMING**, **UART\_RXERROR\_PARITY**, **UART\_RXERROR\_BREAK** and **UART\_RXERROR\_OVERRUN**.



### 17.2.2.36 UARTSpaceAvail

Determines if there is any space in the transmit FIFO

**Prototype:**

```
bool
UARTSpaceAvail (uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function returns a flag indicating whether or not there is space available in the transmit FIFO.

**Returns:**

Returns **true** if there is space available in the transmit FIFO or **false** if there is no space available in the transmit FIFO.

### 17.2.2.37 UARTTxIntModeGet

Returns the current operating mode for the UART transmit interrupt

**Prototype:**

```
uint32_t
UARTTxIntModeGet (uint32_t ui32Base)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**Description:**

This function returns the current operating mode for the UART transmit interrupt. The return value is **UART\_TXINT\_MODE\_EOT** if the transmit interrupt is currently set to be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter. The return value is **UART\_TXINT\_MODE\_FIFO** if the interrupt is set to be asserted based upon the level of the transmit FIFO.

**Returns:**

Returns **UART\_TXINT\_MODE\_FIFO** or **UART\_TXINT\_MODE\_EOT**.

### 17.2.2.38 UARTTxIntModeSet

Sets the operating mode for the UART transmit interrupt

**Prototype:**

```
void
UARTTxIntModeSet (uint32_t ui32Base,
                  uint32_t ui32Mode)
```

**Parameters:**

**ui32Base** is the base address of the UART port.

**ui32Mode** is the operating mode for the transmit interrupt. It may be **UART\_TXINT\_MODE\_EOT** to trigger interrupts when the transmitter is idle or **UART\_TXINT\_MODE\_FIFO** to trigger based on the current transmit FIFO level.

**Description:**

This function allows the mode of the UART transmit interrupt to be set. By default, the transmit interrupt is asserted when the FIFO level falls past a threshold set via a call to [UARTFIFOLevelSet\(\)](#). Alternatively, if this function is called with **ui32Mode** set to **UART\_TXINT\_MODE\_EOT**, the transmit interrupt is asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter.

**Returns:**

None

## 17.3 Programming Example

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

```
//
// Initialize the UART. Set the baud rate, number of data bits, turn off
// parity, number of stop bits, and stick mode.
//
UARTConfigSetExpClk(UART0_BASE, SysCtrlClockGet(), 38400,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                     UART_CONFIG_PAR_NONE));

//
// Enable the UART.
//
UARTEnable(UART0_BASE);

//
// Check for characters. Spin here until a character is placed
// into the receive FIFO.
//
while(!UARTCharsAvail(UART0_BASE))
{
}

//
// Get the character(s) in the receive FIFO.
//
while(UARTCharGetNonBlocking(UART0_BASE))
{
}

//
// Put a character in the output buffer.
//
```

```
UARTCharPut (UART0_BASE, 'c');  
  
//  
// Disable the UART.  
//  
UARTDisable (UART0_BASE);
```



# 18 uDMA Controller

Introduction .....	173
API Functions .....	174
Programming Example .....	190

## 18.1 Introduction

The micro direct memory access (uDMA) API provides functions to configure the CC2538 uDMA controller. The uDMA controller is designed to work with the ARM Cortex-M3 processor and provides an efficient and low-overhead means of transferring blocks of data in the system.

The uDMA controller has the following features:

- Dedicated channels for supported peripherals
- One channel each for receive and transmit for devices with receive and transmit paths
- Dedicated channel for software initiated data transfers
- Channels can be independently configured and operated
- An arbitration scheme that is configurable per channel
- Two levels of priority
- Subordinate to Cortex-M processor bus usage
- Data sizes of 8, 16, or 32 bits
- Address increment of byte, half-word, word, or none
- Maskable device requests
- Optional software initiated transfers on any channel
- Interrupt on transfer completion

The uDMA controller supports several different transfer modes, allowing for complex transfer schemes. The following transfer modes are provided:

- **Basic** mode performs a simple transfer when a request is asserted by a device. This mode is appropriate to use with peripherals where the peripheral asserts the request signal whenever data should be transferred. The transfer pauses if the request is deasserted, even if the transfer is not complete.
- **Auto-request** mode performs a simple transfer that is started by a request, but always completes the entire transfer, even if the request is deasserted. This mode is appropriate to use with software-initiated transfers.
- **Ping-pong** mode is used to transfer data to or from two buffers, switching from one buffer to the other as each buffer fills. This mode is appropriate to use with peripherals as a way to ensure a continuous flow of data to or from the peripheral. However, it is more complex to set up and requires code to manage the ping-pong buffers in the interrupt handler.
- **Memory scatter-gather** mode is a complex mode that provides a way to set up a list of transfer tasks for the uDMA controller. Blocks of data can be transferred to and from arbitrary locations in memory.

- **Peripheral scatter-gather** mode is similar to memory scatter-gather mode except that it is controlled by a peripheral request.

Detailed explanation of the various transfer modes is beyond the scope of this document. Please refer to the device data sheet for more information on the operation of the uDMA controller.

The naming convention for the microDMA controller is to use the Greek letter mu to represent micro. For the purposes of this document, and in the software library function names, a lower case u will be used in place of mu when the controller is referred to as uDMA.

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

## 18.2 API Functions

### Defines

- `uDMATaskStructEntry`(ui32TransferCount, ui32ItemSize, ui32SrcIncrement, pvSrcAddr, ui32DstIncrement, pvDstAddr, ui32ArbSize, ui32Mode)

### Functions

- void `uDMAChannelAssign` (uint32\_t ui32Mapping)
- void `uDMAChannelAttributeDisable` (uint32\_t ui32ChannelNum, uint32\_t ui32Attr)
- void `uDMAChannelAttributeEnable` (uint32\_t ui32ChannelNum, uint32\_t ui32Attr)
- uint32\_t `uDMAChannelAttributeGet` (uint32\_t ui32ChannelNum)
- void `uDMAChannelControlSet` (uint32\_t ui32ChannelStructIndex, uint32\_t ui32Control)
- void `uDMAChannelDisable` (uint32\_t ui32ChannelNum)
- void `uDMAChannelEnable` (uint32\_t ui32ChannelNum)
- bool `uDMAChannelsEnabled` (uint32\_t ui32ChannelNum)
- uint32\_t `uDMAChannelModeGet` (uint32\_t ui32ChannelStructIndex)
- void `uDMAChannelRequest` (uint32\_t ui32ChannelNum)
- void `uDMAChannelScatterGatherSet` (uint32\_t ui32ChannelNum, uint32\_t ui32TaskCount, void \*pvTaskList, uint32\_t ui32IsPeriphSG)
- uint32\_t `uDMAChannelSizeGet` (uint32\_t ui32ChannelStructIndex)
- void `uDMAChannelTransferSet` (uint32\_t ui32ChannelStructIndex, uint32\_t ui32Mode, void \*pvSrcAddr, void \*pvDstAddr, uint32\_t ui32TransferSize)
- void \* `uDMAControlAlternateBaseGet` (void)
- void \* `uDMAControlBaseGet` (void)
- void `uDMAControlBaseSet` (void \*pControlTable)
- void `uDMADisable` (void)
- void `uDMAEnable` (void)
- void `uDMAErrorStatusClear` (void)
- uint32\_t `uDMAErrorStatusGet` (void)
- void `uDMAIntClear` (uint32\_t ui32ChanMask)
- void `uDMAIntRegister` (uint32\_t ui32IntChannel, void (\*pfnHandler)(void))
- uint32\_t `uDMAIntStatus` (void)
- void `uDMAIntUnregister` (uint32\_t ui32IntChannel)

## 18.2.1 Detailed Description

The uDMA API functions provide a means to enable and configure the CC2538 uDMA controller to perform DMA transfers.

The general order of function calls to set up and perform a uDMA transfer is the following:

- `uDMAEnable()` is called once to enable the controller.
- `uDMAControlBaseSet()` is called once to set the channel control table.
- `uDMAChannelAttributeEnable()` is called once or infrequently to configure the behavior of the channel.
- `uDMAChannelControlSet()` is used to set up characteristics of the data transfer. It is called only once if the nature of the data transfer does not change.
- `uDMAChannelTransferSet()` is used to set the buffer pointers and size for a transfer. It is called before each new transfer.
- `uDMAChannelEnable()` enables a channel to perform data transfers.
- `uDMAChannelRequest()` is used to initiate a software-based transfer. This is normally not used for peripheral based transfers.

To use the uDMA controller, you must first enable it by calling `uDMAEnable()`. You can later disable it, if no longer needed, by calling `uDMADisable()`.

Once the uDMA controller is enabled, the controller must be told where to find the channel control structures in system memory by using the function `uDMAControlBaseSet()` and passing a pointer to the base of the channel control structure. The control structure must be allocated by the application. One way to allocate the control structure is to declare an array of data type `char` or `unsigned char`. To support all channels and transfer modes, the control table array should be 1024 bytes, but it can be fewer depending on transfer modes used and number of channels actually used.

**Note:**

The control table must be aligned on a 1024-byte boundary.

The uDMA controller supports multiple channels. Each channel has a set of attribute flags to control certain uDMA features and channel behavior. The attribute flags are configured with the function `uDMAChannelAttributeEnable()` and cleared with `uDMAChannelAttributeDisable()`. The setting of the channel attribute flags can be queried using the function `uDMAChannelAttributeGet()`.

Next, the control parameters of the DMA transfer must be configured. These parameters control the size and address increment of the data items to transfer. The function `uDMAChannelControlSet()` is used to set up these control parameters.

All of the functions mentioned so far are used only once or infrequently to set up the uDMA channel and transfer. To configure the transfer addresses, transfer size, and transfer mode, use the function `uDMAChannelTransferSet()`. This function must be called for each new transfer. Once everything is set up, the channel is enabled by calling `uDMAChannelEnable()`, which must be done before each new transfer. The uDMA controller automatically disables the channel when a transfer completes. A channel can be manually disabled by using `uDMAChannelDisable()`.

Additional functions can be used to query the status of a channel, either from an interrupt handler or in polling fashion. The function `uDMAChannelSizeGet()` is used to find the amount of data remaining to transfer on a channel. This value is 0 when a transfer is completed. The function `uDMAChannelModeGet()` can be used to find the transfer mode of a uDMA channel. This function is usually used to determine if the mode indicates stopped, meaning that a transfer has completed.

on a channel that was previously running. The function `uDMAChannelsEnabled()` can be used to determine if a particular channel is enabled.

If the application is using run-time interrupt registration (see `IntRegister()`), then the function `uDMAIntRegister()` can be used to install an interrupt handler for the uDMA controller. This function also enables the interrupt on the system interrupt controller. If compile-time interrupt registration is used, then call the function `IntEnable()` to enable uDMA interrupts. When an interrupt handler has been installed with `uDMAIntRegister()`, it can be removed by calling `uDMAIntUnregister()`.

This interrupt handler is only for software-initiated transfers or errors. uDMA interrupts for a peripheral occur on the dedicated interrupt channel of the peripheral and should be handled by the peripheral interrupt handler. It is not necessary to acknowledge or clear uDMA interrupt sources. They are cleared automatically when they are serviced.

The uDMA interrupt handler should use the function `uDMAErrorStatusGet()` to test if a uDMA error occurred. If so, the interrupt must be cleared by calling `uDMAErrorStatusClear()`.

**Note:**

Many of the API functions take a channel parameter that includes the logical OR of one of the values `UDMA_PRI_SELECT` or `UDMA_ALT_SELECT` to choose the primary or alternate control structure. For basic and auto transfer modes, only the primary control structure is needed. The alternate control structure is needed only for the complex transfer modes of ping-pong or scatter-gather. See the device data sheet for detailed information about transfer modes.

### Special Considerations for Using Scatter-Gather Operations

To use the scatter-gather modes of the uDMA controller, a task list in memory that describes the scatter-gather operations must be prepared. There is a helper macro, `uDMATaskStructEntry`, provided to help create the initialization values for the task list structure. See the documentation for this macro which includes a code snippet showing how it is used.

Once the task list is prepared, the appropriate uDMA channel must be configured for a scatter-gather operation. The best way to do this is to use the function `uDMAChannelScatterGatherSet()`. Alternatively, the functions `uDMAChannelControlSet()` followed by `uDMAChannelTransferSet()` can also be used.

**Note:**

The scatter-gather task list must reside in SRAM. The uDMA controller cannot read from flash memory.

### About uDMA Channel Function Parameters

Many of the uDMA API functions require a channel number as a parameter. There are two different uses of the channel number. In some cases, it is the number of the uDMA channel and is used to read or write registers within the uDMA controller. In this case, it is simply the channel number with no additional qualifier.

However, in other cases the channel number that is supplied as a parameter is really an index into the uDMA channel control structure. Because every uDMA channel has a primary and an alternate channel control structure, this index must also be specified as part of the channel number. The index is specified by passing a value for the channel parameter that is the logical OR of the actual channel number either of `UDMA_PRI_SELECT` or `UDMA_ALT_SELECT`. The default is the same as `UDMA_PRI_SELECT` so left unspecified, the primary channel control structure is used, which is the correct control structure in most cases.



**Note:**

When **UDMA\_ALT\_SELECT** is specified, what is actually happening is that channel index 32-63 is being used, because the alternate channel control structures for channels 0-31 are at index locations 32-63 in the channel control table.

An example of the first case follows. In this example, a uDMA channel is enabled, and only the channel number is used because this is programming a register in the uDMA controller.

```
uDMAChannelEnable(UDMA_CH8_UART0RX);
```

An example of the second case follows. In this example, the channel control structure is modified to configure some transfer parameters. Therefore, in addition to specifying the channel index, the primary or alternate control structure must also be selected.

```
uDMAChannelControlSet(UDMA_CH8_UART0RX | UDMA_PRI_SELECT, ...);
```

To help make it clear when one form or the other is to be used, the parameters are named differently in the API description. For functions that require only the channel number, the name of the parameter is *ulChannelNum*. For functions that require the channel index of the channel control structure, the name of the parameter is *ulChannelStructIdx*.

**Selecting uDMA Channels**

The uDMA controller has 32 channels, and therefore, most of the API functions take a channel number with a value from 0-31 or a channel index with a value from 0-63 (the 32-63 is specified with the logical OR of the channel number with **UDMA\_ALT\_SELECT**). To avoid the need for hard coded channel numbers in code, macros are provided that map channel names to channel numbers (such as UDMA\_CH8\_UART0RX).

The CC2538 device provides up to five possible channel assignments. Use the [uDMAChannelAssign\(\)](#) function to configure the channel assignments independently for each channel.

For example, the default mapping for channel 9 is UART0TX. However, this channel also has a mapping to UART1TX. If an application requires use of uDMA with UART1 and does not use UART0, then this channel should be remapped to UART1TX with the following function call:

```
uDMAChannelAssign(UDMA_CH9_UART1TX);
```

## 18.2.2 Define Documentation

### 18.2.2.1 uDMATaskStructEntry

A helper macro for building scatter-gather task table entries.

**Definition:**

```
#define uDMATaskStructEntry(ui32TransferCount,
                           ui32ItemSize,
                           ui32SrcIncrement,
                           pvSrcAddr,
                           ui32DstIncrement,
                           pvDstAddr,
                           ui32ArbSize,
                           ui32Mode)
```

**Parameters:**

***ui32TransferCount*** is the count of items to transfer for this task.

***ui32ItemSize*** is the bit size of the items to transfer for this task.

***ui32SrcIncrement*** is the bit size increment for source data.

***pvSrcAddr*** is the starting address of the data to transfer.

***ui32DstIncrement*** is the bit size increment for destination data.

***pvDstAddr*** is the starting address of the destination data.

***ui32ArbSize*** is the arbitration size to use for the transfer task.

***ui32Mode*** is the transfer mode for this task.

**Description:**

This macro is intended to be used to help populate a table of uDMA tasks for a scatter-gather transfer. This macro will calculate the values for the fields of a task structure entry based on the input parameters.

There are specific requirements for the values of each parameter. No checking is done so it is up to the caller to ensure that correct values are used for the parameters.

The *ui32TransferCount* parameter is the number of items that will be transferred by this task. It must be in the range 1-1024.

The *ui32ItemSize* parameter is the bit size of the transfer data. It must be one of **UDMA\_SIZE\_8**, **UDMA\_SIZE\_16**, or **UDMA\_SIZE\_32**.

The *ui32SrcIncrement* parameter is the increment size for the source data. It must be one of **UDMA\_SRC\_INC\_8**, **UDMA\_SRC\_INC\_16**, **UDMA\_SRC\_INC\_32**, or **UDMA\_SRC\_INC\_NONE**.

The *pvSrcAddr* parameter is a void pointer to the beginning of the source data.

The *ui32DstIncrement* parameter is the increment size for the destination data. It must be one of **UDMA\_DST\_INC\_8**, **UDMA\_DST\_INC\_16**, **UDMA\_DST\_INC\_32**, or **UDMA\_DST\_INC\_NONE**.

The *pvDstAddr* parameter is a void pointer to the beginning of the location where the data will be transferred.

The *ui32ArbSize* parameter is the arbitration size for the transfer, and must be one of **UDMA\_ARB\_1**, **UDMA\_ARB\_2**, **UDMA\_ARB\_4**, and so on up to **UDMA\_ARB\_1024**. This is used to select the arbitration size in powers of 2, from 1 to 1024.

The *ui32Mode* parameter is the mode to use for this transfer task. It must be one of **UDMA\_MODE\_BASIC**, **UDMA\_MODE\_AUTO**, **UDMA\_MODE\_MEM\_SCATTER\_GATHER**, or **UDMA\_MODE\_PER\_SCATTER\_GATHER**. Note that normally all tasks will be one of the scatter-gather modes while the last task in a task list will be AUTO or BASIC.

This macro is intended to be used to initialize individual entries of a structure of **tDMAControlTable** type, like this:

```
tDMAControlTable MyTaskList[] =
{
    uDMATaskStructEntry(Task1Count,  UDMA_SIZE_8,
                        UDMA_SRC_INC_8,  MySourceBuf,
                        UDMA_DST_INC_8,  MyDestBuf,
                        UDMA_ARB_8,   UDMA_MODE_MEM_SCATTER_GATHER),
    uDMATaskStructEntry(Task2Count,  ... ),
}
```

**Returns:**

Nothing; this is not a function.

## 18.2.3 Function Documentation

### 18.2.3.1 uDMAChannelAssign

Assigns a peripheral mapping for a uDMA channel

**Prototype:**

```
void
uDMAChannelAssign(uint32_t ui32Mapping)
```

**Parameters:**

***ui32Mapping*** is a macro specifying the peripheral assignment for a channel

**Description:**

This function assigns a peripheral mapping to a uDMA channel. It is used to select which peripheral is used for a uDMA channel. The parameter *ui32Mapping* should be one of the macros named **UDMA\_CHn\_tttt** from the header file *udma.h*. For example, to assign uDMA channel 8 to the UART0RX channel, the parameter should be the macro **UDMA\_CH8\_UART0RX**.

Please consult the cc2538 data sheet for a table showing all the possible peripheral assignments for the uDMA channels for a particular device.

**Note:**

This function is only available on devices that have the DMA Channel Map Select registers (DMACHMAP0-3). Please consult the data sheet for your part.

**Returns:**

None

### 18.2.3.2 uDMAChannelAttributeDisable

Disables attributes of a uDMA channel

**Prototype:**

```
void
uDMAChannelAttributeDisable(uint32_t ui32ChannelNum,
                             uint32_t ui32Attr)
```

**Parameters:**

***ui32ChannelNum*** is the channel to configure.

***ui32Attr*** is a combination of attributes for the channel.

**Description:**

This function is used to disable attributes of a uDMA channel.

The *ui32Attr* parameter is the logical OR of any of the following:

- **UDMA\_ATTR\_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA\_ATTR\_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA\_ATTR\_HIGH\_PRIORITY** is used to set this channel to high priority.
- **UDMA\_ATTR\_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

**Returns:**  
None

### 18.2.3.3 uDMAChannelAttributeEnable

Enables attributes of a uDMA channel

**Prototype:**

```
void  
uDMAChannelAttributeEnable (uint32_t ui32ChannelNum,  
                             uint32_t ui32Attr)
```

**Parameters:**

***ui32ChannelNum*** is the channel to configure.

***ui32Attr*** is a combination of attributes for the channel.

**Description:**

This function is used to enable attributes of a uDMA channel.

The *ui32Attr* parameter is the logical OR of any of the following:

- **UDMA\_ATTR\_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA\_ATTR\_ALTSELECT** is used to select the alternate control structure for this channel (it is very unlikely that this flag should be used).
- **UDMA\_ATTR\_HIGH\_PRIORITY** is used to set this channel to high priority.
- **UDMA\_ATTR\_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

**Returns:**  
None

### 18.2.3.4 uDMAChannelAttributeGet

Gets the enabled attributes of a uDMA channel

**Prototype:**

```
uint32_t  
uDMAChannelAttributeGet (uint32_t ui32ChannelNum)
```

**Parameters:**

***ui32ChannelNum*** is the channel to configure.

**Description:**

This function returns a combination of flags representing the attributes of the uDMA channel.

**Returns:**

Returns the logical OR of the attributes of the uDMA channel, which can be any of the following:

- **UDMA\_ATTR\_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA\_ATTR\_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA\_ATTR\_HIGH\_PRIORITY** is used to set this channel to high priority.

- **UDMA\_ATTR\_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

### 18.2.3.5 uDMAChannelControlSet

Sets the control parameters for a uDMA channel control structure

**Prototype:**

```
void
uDMAChannelControlSet (uint32_t ui32ChannelStructIndex,
                      uint32_t ui32Control)
```

**Parameters:**

**ui32ChannelStructIndex** is the logical OR of the uDMA channel number with **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT**.

**ui32Control** is logical OR of several control values to set the control parameters for the channel.

**Description:**

This function is used to set control parameters for a uDMA transfer. These are typically parameters that are not changed often.

The *ui32ChannelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT** to choose whether the primary or alternate data structure is used.

The *ui32Control* parameter is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. The choices available for each of these values is described below.

Choose the data size from one of **UDMA\_SIZE\_8**, **UDMA\_SIZE\_16**, or **UDMA\_SIZE\_32** to select a data size of 8, 16, or 32 bits.

Choose the source address increment from one of **UDMA\_SRC\_INC\_8**, **UDMA\_SRC\_INC\_16**, **UDMA\_SRC\_INC\_32**, or **UDMA\_SRC\_INC\_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

Choose the destination address increment from one of **UDMA\_DST\_INC\_8**, **UDMA\_DST\_INC\_16**, **UDMA\_DST\_INC\_32**, or **UDMA\_DST\_INC\_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

The arbitration size determines how many items are transferred before the uDMA controller re-arbitrates for the bus. Choose the arbitration size from one of **UDMA\_ARB\_1**, **UDMA\_ARB\_2**, **UDMA\_ARB\_4**, **UDMA\_ARB\_8**, through **UDMA\_ARB\_1024** to select the arbitration size from 1 to 1024 items, in powers of 2.

The value **UDMA\_NEXT\_USEBURST** is used to force the channel to only respond to burst requests at the tail end of a scatter-gather transfer.

**Note:**

The address increment cannot be smaller than the data size.

**Returns:**

None

### 18.2.3.6 uDMAChannelDisable

Disables a uDMA channel for operation

**Prototype:**

```
void  
uDMAChannelDisable (uint32_t ui32ChannelNum)
```

**Parameters:**

***ui32ChannelNum*** is the channel number to disable.

**Description:**

This function disables a specific uDMA channel. Once disabled, a channel will not respond to uDMA transfer requests until re-enabled via [uDMAChannelEnable\(\)](#).

**Returns:**

None

### 18.2.3.7 uDMAChannelEnable

Enables a uDMA channel for operation

**Prototype:**

```
void  
uDMAChannelEnable (uint32_t ui32ChannelNum)
```

**Parameters:**

***ui32ChannelNum*** is the channel number to enable.

**Description:**

This function enables a specific uDMA channel for use. This function must be used to enable a channel before it can be used to perform a uDMA transfer.

When a uDMA transfer is completed, the channel will be automatically disabled by the uDMA controller. Therefore, this function should be called prior to starting up any new transfer.

**Returns:**

None

### 18.2.3.8 uDMAChannelIsEnabled

Checks if a uDMA channel is enabled for operation

**Prototype:**

```
bool  
uDMAChannelIsEnabled (uint32_t ui32ChannelNum)
```

**Parameters:**

***ui32ChannelNum*** is the channel number to check.

**Description:**

This function checks to see if a specific uDMA channel is enabled. This can be used to check the status of a transfer, since the channel will be automatically disabled at the end of a transfer.

**Returns:**

Returns **true** if the channel is enabled, **false** if disabled.

### 18.2.3.9 uDMAChannelModeGet

Gets the transfer mode for a uDMA channel control structure

**Prototype:**

```
uint32_t  
uDMAChannelModeGet (uint32_t ui32ChannelStructIndex)
```

**Parameters:**

**ui32ChannelStructIndex** is the logical OR of the uDMA channel number with either **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT**.

**Description:**

This function is used to get the transfer mode for the uDMA channel. It can be used to query the status of a transfer on a channel. When the transfer is complete the mode will be **UDMA\_MODE\_STOP**.

**Returns:**

Returns the transfer mode of the specified channel and control structure, which will be one of the following values: **UDMA\_MODE\_STOP**, **UDMA\_MODE\_BASIC**, **UDMA\_MODE\_AUTO**, **UDMA\_MODE\_PINGPONG**, **UDMA\_MODE\_MEM\_SCATTER\_GATHER**, or **UDMA\_MODE\_PER\_SCATTER\_GATHER**.

### 18.2.3.10 uDMAChannelRequest

Requests a uDMA channel to start a transfer

**Prototype:**

```
void  
uDMAChannelRequest (uint32_t ui32ChannelNum)
```

**Parameters:**

**ui32ChannelNum** is the channel number on which to request a uDMA transfer.

**Description:**

This function allows software to request a uDMA channel to begin a transfer. This could be used for performing a memory to memory transfer, or if for some reason a transfer needs to be initiated by software instead of the peripheral associated with that channel.

**Note:**

If the channel is **UDMA\_CH30\_SW** and interrupts are used, then the completion will be signaled on the uDMA dedicated interrupt. If a peripheral channel is used, then the completion will be signaled on the peripheral's interrupt.

**Returns:**

None

### 18.2.3.11 uDMAChannelScatterGatherSet

Configures a uDMA channel for scatter-gather mode

**Prototype:**

```
void  
uDMAChannelScatterGatherSet (uint32_t ui32ChannelNum,  
                             uint32_t ui32TaskCount,  
                             void *pvTaskList,  
                             uint32_t ui32IsPeriphSG)
```

**Parameters:**

**ui32ChannelNum** is the uDMA channel number.

**ui32TaskCount** is the number of scatter-gather tasks to execute.

**pvTaskList** is a pointer to the beginning of the scatter-gather task list.

**ui32IsPeriphSG** is a flag to indicate it is a peripheral scatter-gather transfer (else it will be memory scatter-gather transfer)

**Description:**

This function is used to configure a channel for scatter-gather mode. The caller must have already set up a task list, and pass a pointer to the start of the task list as the *pvTaskList* parameter. The *ui32TaskCount* parameter is the count of tasks in the task list, not the size of the task list. The flag *blsPeriphSG* should be used to indicate if the scatter-gather should be configured for a peripheral or memory scatter-gather operation.

**See also:**

[uDMATaskStructEntry](#)

**Returns:**

None

### 18.2.3.12 uDMAChannelSizeGet

Gets the current transfer size for a uDMA channel control structure

**Prototype:**

```
uint32_t  
uDMAChannelSizeGet (uint32_t ui32ChannelStructIndex)
```

**Parameters:**

**ui32ChannelStructIndex** is the logical OR of the uDMA channel number with either **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT**.

**Description:**

This function is used to get the uDMA transfer size for a channel. The transfer size is the number of items to transfer, where the size of an item might be 8, 16, or 32 bits. If a partial transfer has already occurred, then the number of remaining items will be returned. If the transfer is complete, then 0 will be returned.

**Returns:**

Returns the number of items remaining to transfer.



### 18.2.3.13 uDMAChannelTransferSet

Sets the transfer parameters for a uDMA channel control structure

**Prototype:**

```
void
uDMAChannelTransferSet (uint32_t ui32ChannelStructIndex,
                        uint32_t ui32Mode,
                        void *pvSrcAddr,
                        void *pvDstAddr,
                        uint32_t ui32TransferSize)
```

**Parameters:**

**ui32ChannelStructIndex** is the logical OR of the uDMA channel number with either **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT**.

**ui32Mode** is the type of uDMA transfer.

**pvSrcAddr** is the source address for the transfer.

**pvDstAddr** is the destination address for the transfer.

**ui32TransferSize** is the number of data items to transfer.

**Description:**

This function is used to set the parameters for a uDMA transfer. These are typically parameters that are changed often. The function [uDMAChannelControlSet\(\)](#) MUST be called at least once for this channel prior to calling this function.

The *ui32ChannelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA\_PRI\_SELECT** or **UDMA\_ALT\_SELECT** to choose whether the primary or alternate data structure is used.

The *ui32Mode* parameter should be one of the following values:

- **UDMA\_MODE\_STOP** stops the uDMA transfer. The controller sets the mode to this value at the end of a transfer.
- **UDMA\_MODE\_BASIC** to perform a basic transfer based on request.
- **UDMA\_MODE\_AUTO** to perform a transfer that will always complete once started even if request is removed.
- **UDMA\_MODE\_PINGPONG** to set up a transfer that switches between the primary and alternate control structures for the channel. This allows use of ping-pong buffering for uDMA transfers.
- **UDMA\_MODE\_MEM\_SCATTER\_GATHER** to set up a memory scatter-gather transfer.
- **UDMA\_MODE\_PER\_SCATTER\_GATHER** to set up a peripheral scatter-gather transfer.

The *pvSrcAddr* and *pvDstAddr* parameters are pointers to the first location of the data to be transferred. These addresses should be aligned according to the item size. The compiler will take care of this if the pointers are pointing to storage of the appropriate data type.

The *ui32TransferSize* parameter is the number of data items, not the number of bytes.

The two scatter/gather modes, memory and peripheral, are actually different depending on whether the primary or alternate control structure is selected. This function will look for the **UDMA\_PRI\_SELECT** and **UDMA\_ALT\_SELECT** flag along with the channel number and will set the scatter/gather mode as appropriate for the primary or alternate control structure.

The channel must also be enabled using [uDMAChannelEnable\(\)](#) after calling this function. The transfer will not begin until the channel has been set up and enabled. Note that the channel

is automatically disabled after the transfer is completed, meaning that `uDMAChannelEnable()` must be called again after setting up the next transfer.

**Note:**

Great care must be taken to not modify a channel control structure that is in use or else the results will be unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the `uDMAChannelModeGet()` returns **UDMA\_MODE\_STOP**. For PING-PONG or one of the SCATTER\_GATHER modes, it is safe to modify the primary or alternate control structure only when the other is being used. The `uDMAChannelModeGet()` function will return **UDMA\_MODE\_STOP** when a channel control structure is inactive and safe to modify.

**Returns:**

None

### 18.2.3.14 uDMAControlAlternateBaseGet

Gets the base address for the channel control table alternate structures

**Prototype:**

```
void *  
uDMAControlAlternateBaseGet (void)
```

**Description:**

This function gets the base address of the second half of the channel control table that holds the alternate control structures for each channel.

**Returns:**

Returns a pointer to the base address of the second half of the channel control table.

### 18.2.3.15 uDMAControlBaseGet

Gets the base address for the channel control table

**Prototype:**

```
void *  
uDMAControlBaseGet (void)
```

**Description:**

This function gets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel.

**Returns:**

Returns a pointer to the base address of the channel control table.

### 18.2.3.16 uDMAControlBaseSet

Sets the base address for the channel control table

**Prototype:**

```
void  
uDMAControlBaseSet(void *pControlTable)
```

**Parameters:**

***pControlTable*** is a pointer to the 1024 byte aligned base address of the uDMA channel control table.

**Description:**

This function sets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel. The table must be aligned on a 1024 byte boundary. The base address must be set before any of the channel functions can be used.

The size of the channel control table depends on the number of uDMA channels, and which transfer modes are used. Refer to the introductory text and the microcontroller datasheet for more information about the channel control table.

**Returns:**

None

### 18.2.3.17 uDMADisable

Disables the uDMA controller for use

**Prototype:**

```
void  
uDMADisable(void)
```

**Description:**

This function disables the uDMA controller. Once disabled, the uDMA controller will not operate until re-enabled with [uDMAEnable\(\)](#).

**Returns:**

None

### 18.2.3.18 uDMAEnable

Enables the uDMA controller for use

**Prototype:**

```
void  
uDMAEnable(void)
```

**Description:**

This function enables the uDMA controller. The uDMA controller must be enabled before it can be configured and used.

**Returns:**

None

### 18.2.3.19 uDMAErrorStatusClear

Clears the uDMA error interrupt

**Prototype:**

```
void  
uDMAErrorStatusClear(void)
```

**Description:**

This function clears a pending uDMA error interrupt. It should be called from within the uDMA error interrupt handler to clear the interrupt.

**Returns:**

None

### 18.2.3.20 uDMAErrorStatusGet

Gets the uDMA error status

**Prototype:**

```
uint32_t  
uDMAErrorStatusGet(void)
```

**Description:**

This function returns the uDMA error status. It should be called from within the uDMA error interrupt handler to determine if a uDMA error occurred.

**Returns:**

Returns non-zero if a uDMA error is pending.

### 18.2.3.21 uDMAIntClear

Clears uDMA interrupt status

**Prototype:**

```
void  
uDMAIntClear(uint32_t ui32ChanMask)
```

**Parameters:**

***ui32ChanMask*** is a 32-bit mask with one bit for each uDMA channel.

**Description:**

Clears bits in the uDMA interrupt status register according to which bits are set in *ui32ChanMask*. There is one bit for each channel. If a bit is set in *ui32ChanMask*, then that corresponding channel's interrupt status will be cleared (if it was set).

**Note:**

This function is only available on devices that have the DMA Channel Interrupt Status Register (DMACHIS). Please consult the data sheet for your part.

**Returns:**

None

### 18.2.3.22 uDMAIntRegister

Registers an interrupt handler for the uDMA controller

**Prototype:**

```
void  
uDMAIntRegister(uint32_t ui32IntChannel,  
                void (*pfnHandler)(void))
```

**Parameters:**

**ui32IntChannel** identifies which uDMA interrupt is to be registered.

**pfnHandler** is a pointer to the function to be called when the interrupt is activated.

**Description:**

This sets and enables the handler to be called when the uDMA controller generates an interrupt. The *ui32IntChannel* parameter should be one of the following:

- **UDMA\_INT\_SW** to register an interrupt handler to process interrupts from the uDMA software channel (UDMA\_CH30\_SW)
- **UDMA\_INT\_ERR** to register an interrupt handler to process uDMA error interrupts

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Note:**

The interrupt handler for uDMA is for transfer completion when the channel UDMA\_CH30W is used, and for error interrupts. The interrupts for each peripheral channel are handled through the individual peripheral interrupt handlers.

**Returns:**

None

### 18.2.3.23 uDMAIntStatus

Gets the uDMA controller channel interrupt status

**Prototype:**

```
uint32_t  
uDMAIntStatus(void)
```

**Description:**

This function is used to get the interrupt status of the uDMA controller. The returned value is a 32-bit bit mask that indicates which channels are requesting an interrupt. This function can be used from within an interrupt handler to determine or confirm which uDMA channel has requested an interrupt.

**Note:**

This function is only available on devices that have the DMA Channel Interrupt Status Register (DMACHIS). Please consult the data sheet for your part.

**Returns:**

Returns a 32-bit mask which indicates requesting uDMA channels. There is a bit for each channel, and a 1 in a bit indicates that channel is requesting an interrupt. Multiple bits can be set.

### 18.2.3.24 uDMAIntUnregister

Unregisters an interrupt handler for the uDMA controller

**Prototype:**

```
void  
uDMAIntUnregister(uint32_t ui32IntChannel)
```

**Parameters:**

**ui32IntChannel** identifies which uDMA interrupt to unregister.

**Description:**

This function will disable and clear the handler to be called for the specified uDMA interrupt. The *ui32IntChannel* parameter should be one of **UDMA\_INT\_SW** or **UDMA\_INT\_ERR** as documented for the function [uDMAIntRegister\(\)](#).

**See also:**

[IntRegister\(\)](#) for important information about registering interrupt handlers.

**Returns:**

None

## 18.3 Programming Example

The following example sets up the uDMA controller to perform a software-initiated memory-to-memory transfer:

```
//  
// The application must allocate the channel control table.  
// This one is a full table for all modes and channels.  
// NOTE: This table must be 1024-byte aligned.  
//  
unsigned char ucDMAControlTable[1024];  
  
//  
// Source and destination buffers used for the DMA transfer.  
//  
unsigned char ucSourceBuffer[256];  
unsigned char ucDestBuffer[256];  
  
//  
// Enable the uDMA controller.  
//  
uDMAEnable();  
  
//  
// Set the base for the channel control table.  
//  
uDMAControlBaseSet(&ucDMAControlTable[0]);  
  
//
```

```
// No attributes must be set for a software-based transfer.
// The attributes are cleared by default, but are explicitly cleared
// here, in case they were set elsewhere.
//
uDMAChannelAttributeDisable(UDMA_CH30_SW, UDMA_ATTR_ALL);

//
// Now set up the characteristics of the transfer for
// 8-bit data size, with source and destination increments
// in bytes, and a byte-wise buffer copy. A bus arbitration
// size of 8 is used.
//
uDMAChannelControlSet(UDMA_CH30_SW | UDMA_PRI_SELECT,
                      UDMA_SIZE_8 | UDMA_SRC_INC_8 |
                      UDMA_DST_INC_8 | UDMA_ARB_8);

//
// The transfer buffers and transfer size are now configured.
// The transfer uses AUTO mode, which means that the
// transfer automatically runs to completion after the first
// request.
//
uDMAChannelTransferSet(UDMA_CH30_SW | UDMA_PRI_SELECT,
                      UDMA_MODE_AUTO, ucSourceBuffer, ucDestBuffer,
                      sizeof(ucDestBuffer));

//
// Finally, the channel must be enabled. Because this is a
// software-initiated transfer, a request must also be made.
// The request starts the transfer.
//
uDMAChannelEnable(UDMA_CH30_SW);
uDMAChannelRequest(UDMA_CH30_SW);
```





# 19 Watchdog Timer (WDT)

Introduction .....	193
API Functions .....	193
Programming Example .....	194

## 19.1 Introduction

The watchdog timer (WDT) module is intended as a recovery method in situations that can subject the CPU to a software upset. The WDT resets the system when software fails to clear the WDT within the selected time interval. The watchdog can be used in applications that are subject to electrical noise, power glitches, electrostatic discharge, and so forth, or where high reliability is required. The WDT requires that both the 32-kHz watchdog clock and the 32-MHz system clock run when not in power-down mode. The Watchdog Timer features four selectable timer intervals.

The WDT consists of a 15-bit counter clocked by the 32-kHz clock source. The contents of the 15-bit counter are not user-accessible. The contents of the 15-bit counter are retained during all power modes, and the WDT continues counting when entering active mode again.

When the counter reaches the selected timer interval value, the WDT generates a reset signal for the system. If a watchdog clear sequence is performed before the counter reaches the selected timer interval value, the counter is reset to 0 and continues incrementing its value. In watchdog mode, the WDT does not produce interrupt requests.

**Note:**

The WDT is disabled after a system reset.

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

## 19.2 API Functions

### Functions

- void [WatchdogClear](#) (void)
- void [WatchdogEnable](#) (uint32\_t ui32Interval)

### 19.2.1 Detailed Description

The Watchdog Timer API provides a set of functions to use the CC2538 watchdog timer module. Functions are provided to handle setup and configuration of the watchdog timer.

The WDT operates with a watchdog timer clock frequency of 32.768 kHz (when the 32-kHz XOSC is used). This clock frequency gives time-out periods equal to 1.9 ms, 15.625 ms, 0.25 s, and 1 s, corresponding to the interval value settings 64, 512, 8192, and 32768, respectively. The interval is configured when enabling the Watchdog Timer module using the [WatchdogEnable\(\)](#) function. To clear the counter the [WatchdogClear\(\)](#) function can be used.

## 19.2.2 Function Documentation

### 19.2.2.1 WatchdogClear

Clear watch dog timer

**Prototype:**

```
void  
WatchdogClear(void)
```

**Description:**

This function clears the watch dog timer. Timer must be enabled for the clear operation to take effect.

**Returns:**

None

### 19.2.2.2 WatchdogEnable

Enables the watchdog timer

**Prototype:**

```
void  
WatchdogEnable(uint32_t ui32Interval)
```

**Parameters:**

*ui32Interval* is the timer interval setting.

**Description:**

This function sets the timer interval and enables the watchdog timer. A timeout causes a chip reset.

The *ui32Interval* argument must be only one of the following values: **WATCHDOG\_INTERVAL\_32768**, **WATCHDOG\_INTERVAL\_8192**, **WATCHDOG\_INTERVAL\_512**, **WATCHDOG\_INTERVAL\_64**.

**See also:**

WatchdogDisable()

**Returns:**

None

## 19.3 Programming Example

The following example shows how to set up the watchdog timer API to reset the processor a timeout.

```
//  
// Enable watchdog  
//  
WatchdogEnable(WATCHDOG_INTERVAL_32768);
```

```
while (1)
{
}
```



## 20 Secure Hash Algorithm of digest size 256 (SHA)

Introduction .....	197
API Functions .....	197
Programming Example .....	199

### 20.1 Introduction

SHA256 is a cryptographic hash functions published in NIST whose digest is 256 bits. SHA stands for Secure Hash Algorithm. The input is of size 512 bits.

The hash module supports basic SHA-256 operations and requires reading of input data from an external source. This data is transferred through the DMAC modules.

The hash engine has a 32-bit write interface for input data from the TCM master controller, for data to be hashed. The hash engine also has a 32-bit read interface to (optionally not supported in the current API) read the result hash digest through the TCM master interface. The module internally collects the 32-bit data into a 512-bit input block (SHA-256 block size) and when a full block is received (internally indicated through the DMAC), the hash calculation for the received block is started. When the last word is received (aligned or misaligned), the DMAC and master controller generate the last block signal to the hash engine. The mode of the hash engine and the length of the message (for hash finalization) is programmed using the SHA256 APIs.

The SHA256 engine has the ability to set error bits when a DMA operation fails or SHA256 operation fails.

Please Note that the same AES engine is used for Key Load, SHA256, AES-ECB and AES-CCM operation. The calling function has to protect the hardware and implement the AES mutex so that the AES engine is performs only one operation at a time.

This driver is contained in `source/sha256.c`, with `source/sha256.h` containing the API definitions for use by applications.

### 20.2 API Functions

#### Functions

- `uint8_t SHA256Done` (`tSHA256State *psMd`, `uint8_t *ui8Out`)
- `uint8_t SHA256Init` (`tSHA256State *psMd`)
- `uint8_t SHA256Process` (`tSHA256State *psMd`, `uint8_t *ui8In`, `uint32_t ui32InLen`)

#### 20.2.1 Detailed Description

The SHA-256 API has the following three functions:

- `SHA256Init()` initializes the hash state.

- [SHA256Process\(\)](#) is used to process a block of memory through the hash.
- [SHA256Done\(\)](#) terminates hash session to get the digest.

## 20.2.2 Function Documentation

### 20.2.2.1 SHA256Done

SHA256Done function terminates hash session to get the digest. This function must be called only after [SHA256Process\(\)](#).

**Prototype:**

```
uint8_t  
SHA256Done(tSHA256State *psMd,  
           uint8_t *ui8Out)
```

**Parameters:**

***psMd*** is the pointer to hash state.  
***ui8Out*** is the pointer to hash.

**Description:**

For the pointer to hash state parameter ***psMd*** the calling function has to allocate the hash state structure and pass the pointer to the structure.

**Returns:**

SHA256\_SUCCESS if successful.

### 20.2.2.2 SHA256Init

SHA256init initializes the hash state.

**Prototype:**

```
uint8_t  
SHA256Init(tSHA256State *psMd)
```

**Parameters:**

***psMd*** is the pointer to hash state you wish to initialize.

**Description:**

For the pointer to hash state parameter ***psMd*** the calling function has to allocate the hash state structure and pass the pointer to the structure.

**Returns:**

SHA256\_SUCCESS if successful.

### 20.2.2.3 SHA256Process

SHA256Process processes a block of memory through the hash. This function must be called only after SHA256init().

**Prototype:**

```
uint8_t  
SHA256Process(tSHA256State *psMd,  
              uint8_t *ui8In,  
              uint32_t ui32InLen)
```

**Parameters:**

***psMd*** is the pointer to hash state.

***ui8In*** is the pointer to the data to hash.

***ui32InLen*** is the length of the data to hash *ui8In* bytes (octets).

**Description:**

For the pointer to hash state parameter *psMd* the calling function must allocate the hash state structure and pass the pointer to the structure.

**Returns:**

SHA256\_SUCCESS if successful.

## 20.3 Programming Example

For SHA-256 example code, please refer to the example at `examples/sha256`.





## 21 Error Handling

Invalid arguments and error conditions are handled in a non-traditional manner in the peripheral driver library. Typically, a function would check its arguments to make sure that they are valid (if required; some may be unconditionally valid such as a 32-bit value used as the load value for a 32-bit timer). If an invalid argument is provided, an error code would be returned. The caller then has to check the return code from each invocation of the function to make sure that it succeeded.

This method results in a sizable amount of argument-checking code in each function and return-code-checking code at each call site. For a self-contained application, this extra code becomes an unneeded burden once the application is debugged. Having a means of removing it allows the final code to be smaller and therefore run faster.

In the peripheral driver library, most functions do not return errors ([FlashMainPageProgram\(\)](#), [FlashUpperPageErase\(\)](#), [FlashUpperPageProgram\(\)](#), and [FlashMainPageErase\(\)](#) are the notable exceptions). Argument checking is done via a call to the `ASSERT` macro (provided in `source/debug.h`). This macro has the usual definition of an assert macro; it takes an expression that “must” be true. By making this macro be empty, the argument checking is removed from the code.

There are two definitions of the `ASSERT` macro provided in `source/debug.h`; one that is empty (used for normal situations) and one that evaluates the expression (used when the library is built with debugging). The debug version calls the `__error__` function whenever the expression is not true, passing the file name and line number of the `ASSERT` macro invocation. The `__error__` function is prototyped in `source/debug.h` and must be provided by the application because it is the application’s responsibility to deal with error conditions.

### Note:

To enable the `ASSERT` macro make sure that the `ENABLE_ASSERT` symbol is defined within your project and/or compiler setup.

By setting a breakpoint on the `__error__` function, the debugger immediately stops whenever an error occurs anywhere in the application (something that would be very difficult to do with other error checking methods). When the debugger stops, the arguments to the `__error__` function and the backtrace of the stack pinpoint the function that found an error, what it found to be a problem, and where it was called from. As an example:

```
void
UARTParityModeSet(uint32_t ui32Base, uint32_t ui32Parity)
{
    //
    // Check the arguments.
    //
    ASSERT(UARTBaseValid(ui32Base));
    ASSERT((ui32Parity == UART_CONFIG_PAR_NONE) ||
           (ui32Parity == UART_CONFIG_PAR_EVEN) ||
           (ui32Parity == UART_CONFIG_PAR_ODD) ||
           (ui32Parity == UART_CONFIG_PAR_ONE) ||
           (ui32Parity == UART_CONFIG_PAR_ZERO));
```

Each argument is individually checked, so the line number of the failing `ASSERT` indicates the argument that is invalid. The debugger is able to display the values of the arguments (from the stack backtrace) as well as the caller of the function that had the argument error. This method allows the problem to be quickly identified at the cost of a small amount of code.

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

### Products

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
OMAP Applications Processors	<a href="http://www.ti.com/omap">www.ti.com/omap</a>
Wireless Connectivity	<a href="http://www.ti.com/wirelessconnectivity">www.ti.com/wirelessconnectivity</a>

### Applications

Automotive and Transportation	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Communications and Telecom	<a href="http://www.ti.com/communications">www.ti.com/communications</a>
Computers and Peripherals	<a href="http://www.ti.com/computers">www.ti.com/computers</a>
Consumer Electronics	<a href="http://www.ti.com/consumer-apps">www.ti.com/consumer-apps</a>
Energy and Lighting	<a href="http://www.ti.com/energy">www.ti.com/energy</a>
Industrial	<a href="http://www.ti.com/industrial">www.ti.com/industrial</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Space, Avionics and Defense	<a href="http://www.ti.com/space-avionics-defense">www.ti.com/space-avionics-defense</a>
Video and Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>

### TI E2E Community

[e2e.ti.com](http://e2e.ti.com)