

# LM3S5732 ROM


# USER'S GUIDE



---

# Copyright

Copyright © 2008-2011 Texas Instruments Incorporated. All rights reserved. Stellaris and StellarisWare 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  
108 Wild Basin, Suite 350  
Austin, TX 78746  
Main: +1-512-279-8800  
Fax: +1-512-279-8879  
<http://www.ti.com/stellaris>



## Revision Information

This is version 461 of this document, last updated on September 9, 2011.

# Table of Contents

<b>Copyright</b> .....	<b>2</b>
<b>Revision Information</b> .....	<b>2</b>
<b>1 Introduction</b> .....	<b>5</b>
<b>2 Boot Loader</b> .....	<b>7</b>
2.1 Introduction .....	7
2.2 Serial Interfaces .....	7
<b>3 Analog to Digital Converter (ADC)</b> .....	<b>13</b>
3.1 Introduction .....	13
3.2 Functions .....	13
<b>4 Flash</b> .....	<b>23</b>
4.1 Introduction .....	23
4.2 Functions .....	23
<b>5 GPIO</b> .....	<b>33</b>
5.1 Introduction .....	33
5.2 Functions .....	33
<b>6 Inter-Integrated Circuit (I2C)</b> .....	<b>49</b>
6.1 Introduction .....	49
6.2 Functions .....	50
<b>7 Interrupt Controller (NVIC)</b> .....	<b>65</b>
7.1 Introduction .....	65
7.2 Functions .....	65
<b>8 Synchronous Serial Interface (SSI)</b> .....	<b>69</b>
8.1 Introduction .....	69
8.2 Functions .....	69
<b>9 System Control</b> .....	<b>77</b>
9.1 Introduction .....	77
9.2 Functions .....	78
<b>10 System Tick (SysTick)</b> .....	<b>97</b>
10.1 Introduction .....	97
10.2 Functions .....	97
<b>11 Timer</b> .....	<b>101</b>
11.1 Introduction .....	101
11.2 Functions .....	101
<b>12 UART</b> .....	<b>113</b>
12.1 Introduction .....	113
12.2 Functions .....	113
<b>13 Watchdog Timer</b> .....	<b>127</b>
13.1 Introduction .....	127
13.2 Functions .....	127
<b>IMPORTANT NOTICE</b> .....	<b>136</b>

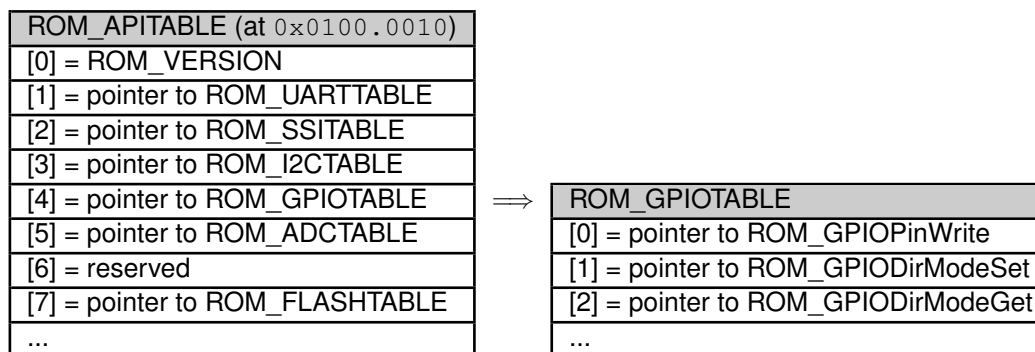


# 1 Introduction

The LM3S5732 ROM contains the Stellaris® Peripheral Driver Library and the Stellaris Boot Loader. The peripheral driver library can be utilized by applications to reduce their flash footprint, allowing the flash to be used for other purposes (such as additional features in the application). The boot loader is used as an initial program loader (when the flash is empty) as well as an application-initiated firmware upgrade mechanism (by calling back to the boot loader).

There is a table at the beginning of the ROM that points to the entry points for the APIs that are provided in the ROM. Accessing the API through these tables provides scalability; while the API locations may change in future versions of the ROM, the API tables will not. The tables are split into two levels; the main table contains one pointer per peripheral which points to a secondary table that contains one pointer per API that is associated with that peripheral. The main table is located at `0x0100.0010`, right after the Cortex-M3 vector table in the ROM.

The following table shows a small portion of the API tables in a graphical form that helps to illustrate the arrangement of the tables:



From this, the address of the ROM\_GPIOTABLE table is located in the memory location at `0x0100.0020`. The address of the [ROM\\_GPIODirModeSet\(\)](#) function is contained at offset `0x4` from that table. In the function documentation, this is represented as:

```
ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_GPIOTABLE is an array of pointers located at ROM_APITABLE[4].
ROM_GPIODirModeSet is a function pointer located at ROM_GPIOTABLE[1].
```

The Stellaris Peripheral Driver Library contains a file called `driverlib/rom.h` that assists with calling the peripheral driver library functions in the ROM. The naming conventions for the tables and APIs that are used in this document match those used in that file.

The following is an example of calling the [ROM\\_GPIODirModeSet\(\)](#) function:

```
#define TARGET_IS_DUSTDEVIL_RA0
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/rom.h"

int
main(void)
{
    // ...

    ROM_GPIODirModeSet(GPIO_PORTA_BASE, GPIO_PIN_0, GPIO_DIR_MODE_OUT);
```

```
} // ....
```

See the “Using the ROM” chapter of the *Stellaris Peripheral Driver Library User's Guide* for more details on calling the ROM functions and using `driverlib/rom.h`.

The API provided by the ROM can be utilized by any compiler so long as it complies with the Embedded Applications Binary Interface (EABI), which includes all recent compilers for the Stellaris microcontroller.

## Documentation Overview

The ROM-based Stellaris Boot Loader is described in chapter [2](#), and the ROM-based Stellaris Peripheral Driver Library is described in chapters [3](#) through [13](#).

---

## 2 Boot Loader

Introduction .....	7
Serial Interfaces .....	7

### 2.1 Introduction

The ROM-based boot loader is executed each time the device is reset when the flash is empty. The flash is assumed to be empty if the first two words are all ones (since the second word is the reset vector address, it must be programmed for an application in flash to execute). When run, it will allow the flash to be updated using one of the following interfaces:

- UART0 using a custom serial protocol
- SSI0 using a custom serial protocol
- I2C0 using a custom serial protocol

Since the boot loader has no knowledge of the frequency of the attached crystal, or in fact if one is even present, it operates entirely from the internal oscillator. This is a 12-MHz clock, with an accuracy of +/- 30%, meaning that the boot loader can expect to run from a clock as slow as 8.4 MHz or as fast as 15.6 MHz.

The LM Flash Programmer GUI can be used to download an application via the boot loader over the UART on a PC. The LM Flash Programmer utility is available for download from [www.ti.com/stellaris](http://www.ti.com/stellaris).

### 2.2 Serial Interfaces

The serial interfaces used to communicate with the boot loader share a common protocol and differ only in the physical connections and signaling used to transfer the bytes of the protocol.

#### 2.2.1 UART Interface

The UART pins **U0Tx** and **U0Rx** are used to communicate with the boot loader. The device communicating with the boot loader is responsible for driving the **U0Rx** pin on the Stellaris microcontroller, while the Stellaris microcontroller drives the **U0Tx** pin.

The serial data format is fixed at 8 data bits, no parity, and one stop bit. An auto-baud feature is used to determine the baud rate at which data is transmitted. Since the system clock must be at least 32 times the baud rate, the maximum baud rate that can be used is 262.5 Kbaud (which is 8.4 MHz divided by 32).

When an application calls back to the ROM-based boot loader to start an update over the UART port, the auto-baud feature is bypassed, along with UART configuration and pin configuration. Therefore, the UART must be configured and the UART pins switched to their hardware function before calling the boot loader.

## 2.2.2 SSI Interface

The SSI pins **SSIFss**, **SSIClk**, **SSITx**, and **SSIRx** are used to communicate with the boot loader. The device communicating with the boot loader is responsible for driving the **SSIRx**, **SSIClk**, and **SSIFss** pins, while the Stellaris microcontroller drives the **SSITx** pin.

The serial data format is fixed to the Motorola format with SPH set to 1 and SPO set to 1 (see the applicable Stellaris family data sheet for more information on this format). Since the system clock must be at least 12 times the serial clock rate, the maximum serial clock rate that can be used is 700 KHz (which is 8.4 MHz divided by 12).

When an application calls back to the ROM-based boot loader to start an update over the SSI port, the SSI configuration and pin configuration is bypassed. Therefore, the SSI port must be configured and the SSI pins switched to their hardware function before calling the boot loader.

## 2.2.3 I2C Interface

The I2C pins **I2CSCL** and **I2CSDA** are used to communicate with the boot loader. The device communicating with the boot loader must operate as the I2C master and provide the **I2CSCL** signal. The **I2CSDA** pin is open-drain and can be driven by either the master or the slave I2C device.

The I2C interface can run at up to 400 KHz, the maximum rate supported by the I2C protocol. The boot loader uses an I2C slave address of 0x42.

When an application calls back to the ROM-based boot loader to start an update over the I2C port, the I2C configuration and pin configuration is bypassed. Therefore, the I2C port must be configured, the I2C slave address set, and the I2C pins switched to their hardware function before calling the boot loader. Additionally, the I2C master must be enabled since it is used to detect start and stop conditions on the I2C bus.

## 2.2.4 Protocol

The boot loader uses well-defined packets to ensure reliable communications with the update program. The packets are always acknowledged or not acknowledged by the communicating devices. The packets use the same format for receiving and sending packets. This includes the method used to acknowledge successful or unsuccessful reception of a packet. While the actual signaling on the serial ports is different, the packet format remains independent of the method of transporting the data.

The following steps must be performed to successfully send a packet:

1. Send the size of the packet that will be sent to the device. The size is always the number of bytes of data + 2 bytes.
2. Send the checksum of the data buffer to help ensure proper transmission of the command. The checksum is simply a sum of the data bytes.
3. Send the actual data bytes.
4. Wait for a single-byte acknowledgment from the device that it either properly received the data or that it detected an error in the transmission.

The following steps must be performed to successfully receive a packet:



1. Wait for non-zero data to be returned from the device. This is important as the device may send zero bytes between a sent and received data packet. The first non-zero byte received will be the size of the packet that is being received.
2. Read the next byte which will be the checksum for the packet.
3. Read the data bytes from the device. There will be packet size - 2 bytes of data sent during the data phase. For example, if the packet size was 3, then there is only 1 byte of data to be received.
4. Calculate the checksum of the data bytes and ensure that it matches the checksum received in the packet.
5. Send an acknowledge (ACK) or not-acknowledge (NAK) to the device to indicate the successful or unsuccessful reception of the packet.

An acknowledge packet is sent whenever a packet is successfully received and verified by the boot loader. A not-acknowledge packet is sent whenever a sent packet is detected to have an error, usually as a result of a checksum error or just malformed data in the packet. This allows the sender to re-transmit the previous packet.

The following commands are used by the custom protocol:

COMMAND\_PING  
= 0x20

This command is used to receive an acknowledge from the boot loader indicating that communication has been established. This command is a single byte.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_PING;
```

COMMAND\_DOWNLOAD  
= 0x21

This command is sent to the boot loader to indicate where to store data and how many bytes will be sent by the COMMAND\_SEND\_DATA commands that follow. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address to start programming data into, while the second is the 32-bit size of the data that will be sent. This command also triggers a mass erase of the flash, which causes the command to take longer to send the ACK/NAK in response to the command. This command should be followed by a COMMAND\_GET\_STATUS to ensure that the program address and program size were valid for the microcontroller running the boot loader.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_DOWNLOAD;
ucCommand[1] = Program Address [31:24];
ucCommand[2] = Program Address [23:16];
ucCommand[3] = Program Address [15:8];
ucCommand[4] = Program Address [7:0];
ucCommand[5] = Program Size [31:24];
ucCommand[6] = Program Size [23:16];
ucCommand[7] = Program Size [15:8];
ucCommand[8] = Program Size [7:0];
```

COMMAND\_RUN  
= 0x22

This command is sent to the boot loader to transfer execution control to the specified address. The command is followed by a 32-bit value, transferred MSB first, that is the address to which execution control is transferred.

The format of the command is as follows:

```
unsigned char ucCommand[5];

ucCommand[0] = COMMAND_RUN;
ucCommand[1] = Run Address [31:24];
ucCommand[2] = Run Address [23:16];
ucCommand[3] = Run Address [15:8];
ucCommand[4] = Run Address [7:0];
```

COMMAND\_GET\_STATUS  
= 0x23

This command returns the status of the last command that was issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires one byte in the data of the packet and the boot loader should respond by sending a packet with one byte of data that contains the current status code.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_GET_STATUS;
```

The following are the definitions for the possible status values that can be returned from the boot loader when COMMAND\_GET\_STATUS is sent to the the microcontroller.

```
COMMAND_RET_SUCCESS
COMMAND_RET_UNKNOWN_CMD
COMMAND_RET_INVALID_CMD
COMMAND_RET_INVALID_ADD
COMMAND_RET_FLASH_FAIL
```

COMMAND\_SEND\_DATA  
= 0x24

This command should only follow a `COMMAND_DOWNLOAD` command or another `COMMAND_SEND_DATA` command, if more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited by the maximum size of a packet, which allows up to 252 data bytes to be transferred at a time. The command terminates programming once the number of bytes indicated by the `COMMAND_DOWNLOAD` command has been received. Each time this function is called, it should be followed by a `COMMAND_GET_STATUS` command to ensure that the data was successfully programmed into the flash. If the boot loader sends a NAK to this command, the boot loader will not increment the current address which allows for retransmission of the previous data.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_SEND_DATA
ucCommand[1] = Data[0];
ucCommand[2] = Data[1];
ucCommand[3] = Data[2];
ucCommand[4] = Data[3];
ucCommand[5] = Data[4];
ucCommand[6] = Data[5];
ucCommand[7] = Data[6];
ucCommand[8] = Data[7];
```

COMMAND\_RESET  
= 0x25

This command is used to tell the boot loader to reset. This is used after downloading a new image to the microcontroller to cause the new application to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. It can also be used to reset the boot loader if a critical error occurs and the host device wants to restart communication with the boot loader.

The boot loader responds with an ACK signal to the host device before actually executing the software reset on the microcontroller running the boot loader. This informs the updater application that the command was received successfully and the part will be reset.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_RESET;
```

The definitions for these commands are provided as part of the Stellaris Peripheral Driver Library, in `boot_loader/bl_commands.h`.



## 3 Analog to Digital Converter (ADC)

Introduction .....	13
Functions .....	13

### 3.1 Introduction

The analog to digital converter (ADC) API provides a set of functions for dealing with the ADC. Functions are provided to configure the sample sequencers, read the captured data, register a sample sequence interrupt handler, and handle interrupt masking/clearing.

The ADC supports six input channels plus an internal temperature sensor. Four sampling sequences, each with configurable trigger events, can be captured. The first sequence will capture up to eight samples, the second and third sequences will capture up to four samples, and the fourth sequence will capture a single sample. Each sample can be the same channel, different channels, or any combination in any order.

The sample sequences have configurable priorities that determine the order in which they are captured when multiple triggers occur simultaneously. The highest priority sequence that is currently triggered will be sampled. Care must be taken with triggers that occur frequently (such as the “always” trigger); if their priority is too high it is possible to starve the lower priority sequences.

Hardware oversampling of the ADC data is available for improved accuracy. An oversampling factor of 2x, 4x, 8x, 16x, 32x, and 64x is supported, but reduces the throughput of the ADC by a corresponding factor. Hardware oversampling is applied uniformly across all sample sequences.

### 3.2 Functions

#### Functions

- void [ROM\\_ADCHardwareOversampleConfigure](#) (unsigned long ulBase, unsigned long ulFactor)
- void [ROM\\_ADCIntClear](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCIntDisable](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCIntEnable](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- unsigned long [ROM\\_ADCIntStatus](#) (unsigned long ulBase, unsigned long ulSequenceNum, tBoolean bMasked)
- void [ROM\\_ADCProcessorTrigger](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCSequenceConfigure](#) (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)
- long [ROM\\_ADCSequenceDataGet](#) (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long \*pulBuffer)
- void [ROM\\_ADCSequenceDisable](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCSequenceEnable](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- long [ROM\\_ADCSequenceOverflow](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCSequenceOverflowClear](#) (unsigned long ulBase, unsigned long ulSequenceNum)

- void [ROM\\_ADCSequenceStepConfigure](#) (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
- long [ROM\\_ADCSequenceUnderflow](#) (unsigned long ulBase, unsigned long ulSequenceNum)
- void [ROM\\_ADCSequenceUnderflowClear](#) (unsigned long ulBase, unsigned long ulSequenceNum)

## 3.2.1 Function Documentation

### 3.2.1.1 ROM\_ADCHardwareOversampleConfigure

Configures the hardware oversampling factor of the ADC.

**Prototype:**

```
void  
ROM_ADCHardwareOversampleConfigure(unsigned long ulBase,  
                                   unsigned long ulFactor)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCHardwareOversampleConfigure is a function pointer located at ROM\_ADCTABLE[14].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulFactor** is the number of samples to be averaged.

**Description:**

This function configures the hardware oversampling for the ADC, which can be used to provide better resolution on the sampled data. Oversampling is accomplished by averaging multiple samples from the same analog input. Six different oversampling rates are supported; 2x, 4x, 8x, 16x, 32x, and 64x. Specifying an oversampling factor of zero will disable the hardware oversampler.

Hardware oversampling applies uniformly to all sample sequencers. It does not reduce the depth of the sample sequencers like the software oversampling APIs; each sample written into the sample sequence FIFO is a fully oversampled analog input reading.

Enabling hardware averaging increases the precision of the ADC at the cost of throughput. For example, enabling 4x oversampling reduces the throughput of a 250 KSps ADC to 62.5 KSps.

**Returns:**

None.

### 3.2.1.2 ROM\_ADCIntClear

Clears sample sequence interrupt source.

**Prototype:**

```
void  
ROM_ADCIntClear(unsigned long ulBase,  
                unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCIntClear is a function pointer located at ROM\_ADCTABLE[4].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

The specified sample sequence interrupt is cleared, 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:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 3.2.1.3 ROM\_ADCIntDisable

Disables a sample sequence interrupt.

**Prototype:**

```
void
ROM_ADCIntDisable(unsigned long ulBase,
                  unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCIntDisable is a function pointer located at ROM\_ADCTABLE[1].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

This function disables the requested sample sequence interrupt.

**Returns:**

None.

### 3.2.1.4 ROM\_ADCIntEnable

Enables a sample sequence interrupt.

**Prototype:**

```
void  
ROM_ADCIntEnable(unsigned long ulBase,  
                 unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCIntEnable is a function pointer located at ROM\_ADCTABLE[2].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

This function enables the requested sample sequence interrupt. Any outstanding interrupts are cleared before enabling the sample sequence interrupt.

**Returns:**

None.

### 3.2.1.5 ROM\_ADCIntStatus

Gets the current interrupt status.

**Prototype:**

```
unsigned long  
ROM_ADCIntStatus(unsigned long ulBase,  
                 unsigned long ulSequenceNum,  
                 tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCIntStatus is a function pointer located at ROM\_ADCTABLE[3].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.  
**bMasked** is false if the raw interrupt status is required and true if the masked interrupt status is required.

**Description:**

This returns the interrupt status for the specified sample sequence. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

**Returns:**

The current raw or masked interrupt status.



### 3.2.1.6 ROM\_ADCProcessorTrigger

Causes a processor trigger for a sample sequence.

**Prototype:**

```
void
ROM_ADCProcessorTrigger(unsigned long ulBase,
                        unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCProcessorTrigger is a function pointer located at ROM\_ADCTABLE[13].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to ADC\_TRIGGER\_PROCESSOR.

**Returns:**

None.

### 3.2.1.7 ROM\_ADCSequenceConfigure

Configures the trigger source and priority of a sample sequence.

**Prototype:**

```
void
ROM_ADCSequenceConfigure(unsigned long ulBase,
                        unsigned long ulSequenceNum,
                        unsigned long ulTrigger,
                        unsigned long ulPriority)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCSequenceConfigure is a function pointer located at ROM\_ADCTABLE[7].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.  
**ulTrigger** is the trigger source that initiates the sample sequence; must be one of the ADC\_TRIGGER\_\* values.  
**ulPriority** is the relative priority of the sample sequence with respect to the other sample sequences.

**Description:**

This function configures the initiation criteria for a sample sequence. Valid sample sequences range from zero to three; sequence zero will capture up to eight samples, sequences one and

two will capture up to four samples, and sequence three will capture a single sample. The trigger condition and priority (with respect to other sample sequence execution) is set.

The *ulTrigger* parameter can take on the following values:

- **ADC\_TRIGGER\_PROCESSOR** - A trigger generated by the processor, via the [ROM\\_ADCProcessorTrigger\(\)](#) function.
- **ADC\_TRIGGER\_EXTERNAL** - A trigger generated by an input from the Port B4 pin.
- **ADC\_TRIGGER\_TIMER** - A trigger generated by a timer; configured with [ROM\\_TimerControlTrigger\(\)](#).
- **ADC\_TRIGGER\_ALWAYS** - A trigger that is always asserted, causing the sample sequence to capture repeatedly (so long as there is not a higher priority source active).

Note that not all trigger sources are available on all Stellaris family members; consult the data sheet for the device in question to determine the availability of triggers.

The *ulPriority* parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest. Note that when programming the priority among a set of sample sequences, each must have unique priority; it is up to the caller to guarantee the uniqueness of the priorities.

**Returns:**

None.

### 3.2.1.8 ROM\_ADCSequenceDataGet

Gets the captured data for a sample sequence.

**Prototype:**

```
long
ROM_ADCSequenceDataGet (unsigned long ulBase,
                        unsigned long ulSequenceNum,
                        unsigned long *pulBuffer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCSequenceDataGet is a function pointer located at ROM\_ADCTABLE[0].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.  
**pulBuffer** is the address where the data is stored.

**Description:**

This function copies data from the specified sample sequence output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples. This will only return the samples that are presently available, which may not be the entire sample sequence if it is in the process of being executed.

**Returns:**

Returns the number of samples copied to the buffer.

### 3.2.1.9 ROM\_ADCSequenceDisable

Disables a sample sequence.

**Prototype:**

```
void  
ROM_ADCSequenceDisable(unsigned long ulBase,  
                        unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].

ROM\_ADCSequenceDisable is a function pointer located at ROM\_ADCTABLE[6].

**Parameters:**

**ulBase** is the base address of the ADC module.

**ulSequenceNum** is the sample sequence number.

**Description:**

Prevents the specified sample sequence from being captured when its trigger is detected. A sample sequence should be disabled before it is configured.

**Returns:**

None.

### 3.2.1.10 ROM\_ADCSequenceEnable

Enables a sample sequence.

**Prototype:**

```
void  
ROM_ADCSequenceEnable(unsigned long ulBase,  
                      unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].

ROM\_ADCSequenceEnable is a function pointer located at ROM\_ADCTABLE[5].

**Parameters:**

**ulBase** is the base address of the ADC module.

**ulSequenceNum** is the sample sequence number.

**Description:**

Allows the specified sample sequence to be captured when its trigger is detected. A sample sequence must be configured before it is enabled.

**Returns:**

None.

### 3.2.1.11 ROM\_ADCSequenceOverflow

Determines if a sample sequence overflow occurred.

**Prototype:**

```
long  
ROM_ADCSequenceOverflow(unsigned long ulBase,  
                        unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCSequenceOverflow is a function pointer located at ROM\_ADCTABLE[9].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

This determines if a sample sequence overflow has occurred. This will happen if the captured samples are not read from the FIFO before the next trigger occurs.

**Returns:**

Returns zero if there was not an overflow, and non-zero if there was.

### 3.2.1.12 ROM\_ADCSequenceOverflowClear

Clears the overflow condition on a sample sequence.

**Prototype:**

```
void  
ROM_ADCSequenceOverflowClear(unsigned long ulBase,  
                             unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCSequenceOverflowClear is a function pointer located at ROM\_ADCTABLE[10].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.

**Description:**

This will clear an overflow condition on one of the sample sequences. The overflow condition must be cleared in order to detect a subsequent overflow condition (it otherwise causes no harm).

**Returns:**

None.

### 3.2.1.13 ROM\_ADCSequenceStepConfigure

Configure a step of the sample sequencer.

**Prototype:**

```
void
ROM_ADCSequenceStepConfigure(unsigned long ulBase,
                             unsigned long ulSequenceNum,
                             unsigned long ulStep,
                             unsigned long ulConfig)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
 ROM\_ADCSequenceStepConfigure is a function pointer located at ROM\_ADCTABLE[8].

**Parameters:**

**ulBase** is the base address of the ADC module.  
**ulSequenceNum** is the sample sequence number.  
**ulStep** is the step to be configured.  
**ulConfig** is the configuration of this step; must be a logical OR of **ADC\_CTL\_TS**, **ADC\_CTL\_IE**, **ADC\_CTL\_END**, **ADC\_CTL\_D**, and one of the input channel selects (**ADC\_CTL\_CH0** through **ADC\_CTL\_CH7**).

**Description:**

This function will set the configuration of the ADC for one step of a sample sequence. The ADC can be configured for single-ended or differential operation (the **ADC\_CTL\_D** bit selects differential operation when set), the channel to be sampled can be chosen (the **ADC\_CTL\_CH0** through **ADC\_CTL\_CH7** values), and the internal temperature sensor can be selected (the **ADC\_CTL\_TS** bit). Additionally, this step can be defined as the last in the sequence (the **ADC\_CTL\_END** bit) and it can be configured to cause an interrupt when the step is complete (the **ADC\_CTL\_IE** bit). The configuration is used by the ADC at the appropriate time when the trigger for this sequence occurs.

The *ulStep* parameter determines the order in which the samples are captured by the ADC when the trigger occurs. It can range from zero to seven for the first sample sequence, from zero to three for the second and third sample sequence, and can only be zero for the fourth sample sequence.

Differential mode only works with adjacent channel pairs (e.g. 0 and 1). The channel select must be the number of the channel pair to sample (e.g. **ADC\_CTL\_CH0** for 0 and 1, or **ADC\_CTL\_CH1** for 2 and 3) or undefined results will be returned by the ADC. Additionally, if differential mode is selected when the temperature sensor is being sampled, undefined results will be returned by the ADC.

It is the responsibility of the caller to ensure that a valid configuration is specified; this function does not check the validity of the specified configuration.

**Returns:**

None.

### 3.2.1.14 ROM\_ADCSequenceUnderflow

Determines if a sample sequence underflow occurred.

**Prototype:**

```
long  
ROM_ADCSequenceUnderflow(unsigned long ulBase,  
                           unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCSequenceUnderflow is a function pointer located at ROM\_ADCTABLE[11].

**Parameters:**

***ulBase*** is the base address of the ADC module.  
***ulSequenceNum*** is the sample sequence number.

**Description:**

This determines if a sample sequence underflow has occurred. This will happen if too many samples are read from the FIFO.

**Returns:**

Returns zero if there was not an underflow, and non-zero if there was.

### 3.2.1.15 ROM\_ADCSequenceUnderflowClear

Clears the underflow condition on a sample sequence.

**Prototype:**

```
void  
ROM_ADCSequenceUnderflowClear(unsigned long ulBase,  
                               unsigned long ulSequenceNum)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_ADCTABLE is an array of pointers located at ROM\_APITABLE[5].  
ROM\_ADCSequenceUnderflowClear is a function pointer located at ROM\_ADCTABLE[12].

**Parameters:**

***ulBase*** is the base address of the ADC module.  
***ulSequenceNum*** is the sample sequence number.

**Description:**

This will clear an underflow condition on one of the sample sequences. The underflow condition must be cleared in order to detect a subsequent underflow condition (it otherwise causes no harm).

**Returns:**

None.

## 4 Flash

Introduction .....	23
Functions .....	23

### 4.1 Introduction

The flash API provides a set of functions for dealing with the on-chip flash. Functions are provided to program and erase the flash, configure the flash protection, and handle the flash interrupt.

The flash is organized as a set of 1 kB blocks that can be individually erased. Erasing a block causes the entire contents of the block to be reset to all ones. These blocks are paired into a set of 2 kB blocks that can be individually protected. The blocks can be marked as read-only or execute-only, providing differing levels of code protection. Read-only blocks cannot be erased or programmed, protecting the contents of those blocks from being modified. Execute-only blocks cannot be erased or programmed, and can only be read by the processor instruction fetch mechanism, protecting the contents of those blocks from being read by either the processor or by debuggers.

The flash can be programmed on a word-by-word basis. Programming causes 1 bits to become 0 bits (where appropriate); because of this, a word can be repeatedly programmed so long as each programming operation only requires changing 1 bits to 0 bits.

The timing for the flash is automatically handled by the flash controller. In order to do this, the flash controller must know the clock rate of the system in order to be able to time the number of micro-seconds certain signals are asserted. The number of clock cycles per micro-second must be provided to the flash controller for it to accomplish this timing.

The flash controller has the ability to generate an interrupt when an invalid access is attempted (such as reading from execute-only flash). This can be used to validate the operation of a program; the interrupt will keep invalid accesses from being silently ignored, hiding potential bugs. The flash protection can be applied without being permanently enabled; this, along with the interrupt, allows the program to be debugged before the flash protection is permanently applied to the device (which is a non-reversible operation). An interrupt can also be generated when an erase or programming operation has completed.

### 4.2 Functions

#### Functions

- long [ROM\\_FlashErase](#) (unsigned long ulAddress)
- void [ROM\\_FlashIntClear](#) (unsigned long ullIntFlags)
- void [ROM\\_FlashIntDisable](#) (unsigned long ullIntFlags)
- void [ROM\\_FlashIntEnable](#) (unsigned long ullIntFlags)
- unsigned long [ROM\\_FlashIntStatus](#) (tBoolean bMasked)
- long [ROM\\_FlashProgram](#) (unsigned long \*pulData, unsigned long ulAddress, unsigned long ulCount)
- tFlashProtection [ROM\\_FlashProtectGet](#) (unsigned long ulAddress)
- long [ROM\\_FlashProtectSave](#) (void)

- long [ROM\\_FlashProtectSet](#) (unsigned long ulAddress, tFlashProtection eProtect)
- unsigned long [ROM\\_FlashUsecGet](#) (void)
- void [ROM\\_FlashUsecSet](#) (unsigned long ulClocks)
- long [ROM\\_FlashUserGet](#) (unsigned long \*pulUser0, unsigned long \*pulUser1)
- long [ROM\\_FlashUserSave](#) (void)
- long [ROM\\_FlashUserSet](#) (unsigned long ulUser0, unsigned long ulUser1)

## 4.2.1 Function Documentation

### 4.2.1.1 ROM\_FlashErase

Erases a block of flash.

**Prototype:**

```
long  
ROM_FlashErase(unsigned long ulAddress)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_FLASHTABLE` is an array of pointers located at `ROM_APITABLE[7]`.  
`ROM_FlashErase` is a function pointer located at `ROM_FLASHTABLE[3]`.

**Parameters:**

***ulAddress*** is the start address of the flash block to be erased.

**Description:**

This function will erase a 1 kB block of the on-chip flash. After erasing, the block will be filled with 0xFF bytes. Read-only and execute-only blocks cannot be erased.

This function will not return until the block has been erased.

**Returns:**

Returns 0 on success, or -1 if an invalid block address was specified or the block is write-protected.

### 4.2.1.2 ROM\_FlashIntClear

Clears flash controller interrupt sources.

**Prototype:**

```
void  
ROM_FlashIntClear(unsigned long ulIntFlags)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_FLASHTABLE` is an array of pointers located at `ROM_APITABLE[7]`.  
`ROM_FlashIntClear` is a function pointer located at `ROM_FLASHTABLE[13]`.

**Parameters:**

***ulIntFlags*** is the bit mask of the interrupt sources to be cleared. Can be any of the `FLASH_FCMISC_PROGRAM` or `FLASH_FCMISC_ACCESS` values.



**Description:**

The specified flash controller interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

#### 4.2.1.3 ROM\_FlashIntDisable

Disables individual flash controller interrupt sources.

**Prototype:**

```
void  
ROM_FlashIntDisable(unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashIntDisable is a function pointer located at ROM\_FLASHTABLE[11].

**Parameters:**

**ulIntFlags** is a bit mask of the interrupt sources to be disabled. Can be any of the **FLASH\_FCIM\_PROGRAM** or **FLASH\_FCIM\_ACCESS** values.

**Description:**

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

**Returns:**

None.

#### 4.2.1.4 ROM\_FlashIntEnable

Enables individual flash controller interrupt sources.

**Prototype:**

```
void  
ROM_FlashIntEnable(unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashIntEnable is a function pointer located at ROM\_FLASHTABLE[10].

**Parameters:**

***ullntFlags*** is a bit mask of the interrupt sources to be enabled. Can be any of the **FLASH\_FCIM\_PROGRAM** or **FLASH\_FCIM\_ACCESS** values.

**Description:**

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

**Returns:**

None.

#### 4.2.1.5 ROM\_FlashIntStatus

Gets the current interrupt status.

**Prototype:**

```
unsigned long  
ROM_FlashIntStatus(tBoolean bMasked)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.

**ROM\_FLASHTABLE** is an array of pointers located at **ROM\_APITABLE**[7].

**ROM\_FlashIntStatus** is a function pointer located at **ROM\_FLASHTABLE**[12].

**Parameters:**

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

**Description:**

This returns the interrupt status for the flash controller. 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 **FLASH\_FCMISC\_PROGRAM** and **FLASH\_FCMISC\_ACCESS**.

#### 4.2.1.6 ROM\_FlashProgram

Programs flash.

**Prototype:**

```
long  
ROM_FlashProgram(unsigned long *pulData,  
                 unsigned long ulAddress,  
                 unsigned long ulCount)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.

**ROM\_FLASHTABLE** is an array of pointers located at **ROM\_APITABLE**[7].

**ROM\_FlashProgram** is a function pointer located at **ROM\_FLASHTABLE**[0].

**Parameters:**

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

***ulAddress*** is the starting address in flash to be programmed. Must be a multiple of four.

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

**Description:**

This function will program 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.

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

This function will not return until the data has been programmed.

**Returns:**

Returns 0 on success, or -1 if a programming error is encountered.

#### 4.2.1.7 ROM\_FlashProtectGet

Gets the protection setting for a block of flash.

**Prototype:**

```
tFlashProtection  
ROM_FlashProtectGet(unsigned long ulAddress)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].

ROM\_FlashProtectGet is a function pointer located at ROM\_FLASHTABLE[4].

**Parameters:**

***ulAddress*** is the start address of the flash block to be queried.

**Description:**

This function will get the current protection for the specified 2 kB block of flash. Each block can be read/write, read-only, or execute-only. Read/write blocks can be read, executed, erased, and programmed. Read-only blocks can be read and executed. Execute-only blocks can only be executed; processor and debugger data reads are not allowed.

**Returns:**

Returns the protection setting for this block. See [ROM\\_FlashProtectSet\(\)](#) for possible values.

#### 4.2.1.8 ROM\_FlashProtectSave

Saves the flash protection settings.

**Prototype:**

```
long  
ROM_FlashProtectSave(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashProtectSave is a function pointer located at ROM\_FLASHTABLE[6].

**Description:**

This function will make the currently programmed flash protection settings permanent. This is a non-reversible operation; a chip reset or power cycle will not change the flash protection.

This function will not return until the protection has been saved.

**Returns:**

Returns 0 on success, or -1 if a hardware error is encountered.

#### 4.2.1.9 ROM\_FlashProtectSet

Sets the protection setting for a block of flash.

**Prototype:**

```
long  
ROM_FlashProtectSet(unsigned long ulAddress,  
                    tFlashProtection eProtect)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashProtectSet is a function pointer located at ROM\_FLASHTABLE[5].

**Parameters:**

**ulAddress** is the start address of the flash block to be protected.

**eProtect** is the protection to be applied to the block. Can be one of **FlashReadWrite**, **FlashReadOnly**, or **FlashExecuteOnly**.

**Description:**

This function will set the protection for the specified 2 kB block of flash. Blocks which are read/write can be made read-only or execute-only. Blocks which are read-only can be made execute-only. Blocks which are execute-only cannot have their protection modified. Attempts to make the block protection less stringent (i.e. read-only to read/write) will result in a failure (and be prevented by the hardware).

Changes to the flash protection are maintained only until the next reset. This allows the application to be executed in the desired flash protection environment to check for inappropriate flash access (via the flash interrupt). To make the flash protection permanent, use the [ROM\\_FlashProtectSave\(\)](#) function.

**Returns:**

Returns 0 on success, or -1 if an invalid address or an invalid protection was specified.

#### 4.2.1.10 ROM\_FlashUsecGet

Gets the number of processor clocks per micro-second.

**Prototype:**

```
unsigned long  
ROM_FlashUsecGet (void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashUsecGet is a function pointer located at ROM\_FLASHTABLE[1].

**Description:**

This function returns the number of clocks per micro-second, as presently known by the flash controller.

**Returns:**

Returns the number of processor clocks per micro-second.

#### 4.2.1.11 ROM\_FlashUsecSet

Sets the number of processor clocks per micro-second.

**Prototype:**

```
void  
ROM_FlashUsecSet (unsigned long ulClocks)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashUsecSet is a function pointer located at ROM\_FLASHTABLE[2].

**Parameters:**

*ulClocks* is the number of processor clocks per micro-second.

**Description:**

This function is used to tell the flash controller the number of processor clocks per micro-second. This value must be programmed correctly or the flash most likely will not program correctly; it has no affect on reading flash.

**Returns:**

None.

#### 4.2.1.12 ROM\_FlashUserGet

Gets the User Registers

**Prototype:**

```
long  
ROM_FlashUserGet (unsigned long *pulUser0,  
                  unsigned long *pulUser1)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashUserGet is a function pointer located at ROM\_FLASHTABLE[7].

**Parameters:**

***pulUser0*** is a pointer to the location to store USER Register 0.  
***pulUser1*** is a pointer to the location to store USER Register 1.

**Description:**

This function will read the contents of User Registers (0 and 1), and store them in the specified locations.

**Returns:**

Returns 0 on success, or -1 if a hardware error is encountered.

#### 4.2.1.13 ROM\_FlashUserSave

Saves the User Registers

**Prototype:**

```
long  
ROM_FlashUserSave(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashUserSave is a function pointer located at ROM\_FLASHTABLE[9].

**Description:**

This function will make the currently programmed User register settings permanent. This is a non-reversible operation; a chip reset or power cycle will not change this setting.

This function will not return until the protection has been saved.

**Returns:**

Returns 0 on success, or -1 if a hardware error is encountered.

#### 4.2.1.14 ROM\_FlashUserSet

Sets the User Registers

**Prototype:**

```
long  
ROM_FlashUserSet(unsigned long ulUser0,  
                 unsigned long ulUser1)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_FLASHTABLE is an array of pointers located at ROM\_APITABLE[7].  
ROM\_FlashUserSet is a function pointer located at ROM\_FLASHTABLE[8].

**Parameters:**

*ulUser0* is the value to store in USER Register 0.

*ulUser1* is the value to store in USER Register 1.

**Description:**

This function will set the contents of the User Registers (0 and 1) to the specified values.

**Returns:**

Returns 0 on success, or -1 if a hardware error is encountered.





## 5 GPIO

Introduction .....	33
Functions .....	33

### 5.1 Introduction

The GPIO module provides control for up to eight independent GPIO pins (the actual number present depend upon the GPIO port and part number). Each pin has the following capabilities:

- Can be configured as an input or an output. On reset, they default to being an input.
- In input mode, can generate interrupts on high level, low level, rising edge, falling edge, or both edges.
- In output mode, can be configured for 2 mA, 4 mA, or 8 mA drive strength. The 8 mA drive strength configuration has optional slew rate control to limit the rise and fall times of the signal. On reset, they default to 2 mA drive strength.
- Optional weak pull-up or pull-down resistors. On reset, they default to no pull-up or pull-down resistors.
- Optional open-drain operation. On reset, they default to standard push/pull operation.
- Can be configured to be a GPIO or a peripheral pin. On reset, they default to being GPIOs. Note that not all pins on all parts have peripheral functions, in which case the pin is only useful as a GPIO (that is, when configured for peripheral function the pin will not do anything useful).

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

This is most useful for the [ROM\\_GPIOPinRead\(\)](#) and [ROM\\_GPIOPinWrite\(\)](#) functions; a read will return only the value of the requested pins (with the other pin values masked out) and a write will affect 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 *ucPin* (singular) parameter, only a single pin is affected by the function. In this case, this value specifies the pin number (that is, 0 through 7).

### 5.2 Functions

#### Functions

- unsigned long [ROM\\_GPIODirModeGet](#) (unsigned long ulPort, unsigned char ucPin)
- void [ROM\\_GPIODirModeSet](#) (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)

- unsigned long [ROM\\_GPIOIntTypeGet](#) (unsigned long ulPort, unsigned char ucPin)
- void [ROM\\_GPIOIntTypeSet](#) (unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)
- void [ROM\\_GPIOPadConfigGet](#) (unsigned long ulPort, unsigned char ucPin, unsigned long \*pulStrength, unsigned long \*pulPinType)
- void [ROM\\_GPIOPadConfigSet](#) (unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPinType)
- void [ROM\\_GPIOPinIntClear](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinIntDisable](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinIntEnable](#) (unsigned long ulPort, unsigned char ucPins)
- long [ROM\\_GPIOPinIntStatus](#) (unsigned long ulPort, tBoolean bMasked)
- long [ROM\\_GPIOPinRead](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeCAN](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeGPIOInput](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeGPIOOutput](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeGPIOOutputOD](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeI2C](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeSSI](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeTimer](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinTypeUART](#) (unsigned long ulPort, unsigned char ucPins)
- void [ROM\\_GPIOPinWrite](#) (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)

## 5.2.1 Function Documentation

### 5.2.1.1 ROM\_GPIODirModeGet

Gets the direction and mode of a pin.

**Prototype:**

```
unsigned long
ROM_GPIODirModeGet(unsigned long ulPort,
                   unsigned char ucPin)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.  
`ROM_GPIODirModeGet` is a function pointer located at `ROM_GPIOTABLE[2]`.

**Parameters:**

***ulPort*** is the base address of the GPIO port.  
***ucPin*** 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 [ROM\\_GPIODirModeSet\(\)](#).

### 5.2.1.2 ROM\_GPIODirModeSet

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

**Prototype:**

```
void
ROM_GPIODirModeSet(unsigned long ulPort,
                    unsigned char ucPins,
                    unsigned long ulPinIO)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIODirModeSet is a function pointer located at ROM\_GPIOTABLE[1].

**Parameters:**

**ulPort** is the base address of the GPIO port  
**ucPins** is the bit-packed representation of the pin(s).  
**ulPinIO** is the pin direction and/or mode.

**Description:**

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

The *ulPinIO* parameter 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, etc.

**Returns:**

None.

### 5.2.1.3 ROM\_GPIOIntTypeGet

Gets the interrupt type for a pin.

**Prototype:**

```
unsigned long
ROM_GPIOIntTypeGet(unsigned long ulPort,
                   unsigned char ucPin)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOIntTypeGet is a function pointer located at ROM\_GPIOTABLE[4].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPin** 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 [ROM\\_GPIOIntTypeSet\(\)](#).

### 5.2.1.4 ROM\_GPIOIntTypeSet

Sets the interrupt type for the specified pin(s).

**Prototype:**

```
void  
ROM_GPIOIntTypeSet(unsigned long ulPort,  
                   unsigned char ucPins,  
                   unsigned long ulIntType)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOIntTypeSet is a function pointer located at ROM\_GPIOTABLE[3].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** is the bit-packed representation of the pin(s).  
**ullIntType** 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 *ullIntType* parameter 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, etc.

**Note:**

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

**Returns:**

None.

### 5.2.1.5 ROM\_GPIOPadConfigGet

Gets the pad configuration for a pin.

**Prototype:**

```
void
ROM_GPIOPadConfigGet (unsigned long ulPort,
                      unsigned char ucPin,
                      unsigned long *pulStrength,
                      unsigned long *pulPinType)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPadConfigGet is a function pointer located at ROM\_GPIOTABLE[6].

**Parameters:**

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

**ucPin** is the pin number.

**pulStrength** is a pointer to storage for the output drive strength.

**pulPinType** is a pointer to storage for the output drive type.

**Description:**

This function gets the pad configuration for a specified pin on the selected GPIO port. The values returned in *eStrength* and *eOutType* correspond to the values used in [ROM\\_GPIOPadConfigSet\(\)](#). This function also works for pin(s) configured as input pin(s); however, the only meaningful data returned is whether the pin is terminated with a pull-up or down resistor.

**Returns:**

None

### 5.2.1.6 ROM\_GPIOPadConfigSet

Sets the pad configuration for the specified pin(s).

**Prototype:**

```
void
ROM_GPIOPadConfigSet (unsigned long ulPort,
```

```
    unsigned char ucPins,  
    unsigned long ulStrength,  
    unsigned long ulPinType)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPadConfigSet is a function pointer located at ROM\_GPIOTABLE[5].

**Parameters:**

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

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

**ulStrength** specifies the output drive strength.

**ulPinType** specifies the pin type.

**Description:**

This function sets the drive strength and type for the specified pin(s) on the selected GPIO port. For pin(s) configured as input ports, the pad is configured as requested, but the only real effect on the input is the configuration of the pull-up or pull-down termination.

The *ulStrength* parameter can be one of the following values:

- GPIO\_STRENGTH\_2MA
- GPIO\_STRENGTH\_4MA
- GPIO\_STRENGTH\_8MA
- GPIO\_STRENGTH\_8MA\_SC

where **GPIO\_STRENGTH\_xMA** specifies either 2, 4, or 8 mA output drive strength, and **GPIO\_OUT\_STRENGTH\_8MA\_SC** specifies 8 mA output drive with slew control.

The *ulPinType* parameter can be one of the following values:

- GPIO\_PIN\_TYPE\_STD
- GPIO\_PIN\_TYPE\_STD\_WPU
- GPIO\_PIN\_TYPE\_STD\_WPD
- GPIO\_PIN\_TYPE\_OD
- GPIO\_PIN\_TYPE\_OD\_WPU
- GPIO\_PIN\_TYPE\_OD\_WPD
- GPIO\_PIN\_TYPE\_ANALOG

where **GPIO\_PIN\_TYPE\_STD\*** specifies a push-pull pin, **GPIO\_PIN\_TYPE\_OD\*** specifies an open-drain pin, **\*\_WPU** specifies a weak pull-up, **\*\_WPD** specifies a weak pull-down, and **GPIO\_PIN\_TYPE\_ANALOG** specifies an analog input.

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, etc.

**Returns:**

None.

### 5.2.1.7 ROM\_GPIOPinIntClear

Clears the interrupt for the specified pin(s).

**Prototype:**

```
void
ROM_GPIOPinIntClear(unsigned long ulPort,
                    unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinIntClear is a function pointer located at ROM\_GPIOTABLE[10].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** 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, etc.

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 5.2.1.8 ROM\_GPIOPinIntDisable

Disables interrupts for the specified pin(s).

**Prototype:**

```
void
ROM_GPIOPinIntDisable(unsigned long ulPort,
                      unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinIntDisable is a function pointer located at ROM\_GPIOTABLE[8].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** 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, etc.

**Returns:**

None.

### 5.2.1.9 ROM\_GPIOPinIntEnable

Enables interrupts for the specified pin(s).

**Prototype:**

```
void  
ROM_GPIOPinIntEnable(unsigned long ulPort,  
                     unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinIntEnable is a function pointer located at ROM\_GPIOTABLE[7].

**Parameters:**

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

**ucPins** 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, etc.

**Returns:**

None.

### 5.2.1.10 ROM\_GPIOPinIntStatus

Gets interrupt status for the specified GPIO port.

**Prototype:**

```
long  
ROM_GPIOPinIntStatus(unsigned long ulPort,  
                    tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinIntStatus is a function pointer located at ROM\_GPIOTABLE[9].



**Parameters:**

*ulPort* 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 will be 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, etc. Bits 31:8 should be ignored.

### 5.2.1.11 ROM\_GPIOPinRead

Reads the values present of the specified pin(s).

**Prototype:**

```
long
ROM_GPIOPinRead(unsigned long ulPort,
                 unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinRead is a function pointer located at ROM\_GPIOTABLE[11].

**Parameters:**

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

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

**Description:**

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

**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, etc. Any bit that is not specified by *ucPins* is returned as a 0. Bits 31:8 should be ignored.

### 5.2.1.12 ROM\_GPIOPinTypeCAN

Configures pin(s) for use as a CAN device.

**Prototype:**

```
void
ROM_GPIOPinTypeCAN(unsigned long ulPort,
                    unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinTypeCAN is a function pointer located at ROM\_GPIOTABLE[12].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** is the bit-packed representation of the pin(s).

**Description:**

The CAN pins must be properly configured for the CAN peripherals to function correctly. This function provides a typical configuration for those pin(s); other configurations may 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, etc.

**Note:**

This cannot be used to turn any pin into a CAN pin(s); it only configures a CAN pin(s) for proper operation.

**Returns:**

None.

### 5.2.1.13 ROM\_GPIOPinTypeGPIOInput

Configures pin(s) for use as GPIO inputs.

**Prototype:**

```
void
ROM_GPIOPinTypeGPIOInput(unsigned long ulPort,
                          unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinTypeGPIOInput is a function pointer located at ROM\_GPIOTABLE[14].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** 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 is especially true of Fury-class devices where the digital input enable is turned off by default. 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, etc.

**Returns:**

None.

#### 5.2.1.14 ROM\_GPIOPinTypeGPIOOutput

Configures pin(s) for use as GPIO outputs.

**Prototype:**

```
void
ROM_GPIOPinTypeGPIOOutput(unsigned long ulPort,
                           unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinTypeGPIOOutput is a function pointer located at ROM\_GPIOTABLE[15].

**Parameters:***ulPort* is the base address of the GPIO port.*ucPins* is the bit-packed representation of the pin(s).**Description:**

The GPIO pins must be properly configured in order to function correctly as GPIO outputs; this is especially true of Fury-class devices where the digital input enable is turned off by default. 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, etc.

**Returns:**

None.

#### 5.2.1.15 ROM\_GPIOPinTypeGPIOOutputOD

Configures pin(s) for use as GPIO open drain outputs.

**Prototype:**

```
void
ROM_GPIOPinTypeGPIOOutputOD(unsigned long ulPort,
                             unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinTypeGPIOOutputOD is a function pointer located at ROM\_GPIOTABLE[22].

**Parameters:**

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

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

**Description:**

The GPIO pins must be properly configured in order to function correctly as GPIO outputs; this is especially true of Fury-class devices where the digital input enable is turned off by default. 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, etc.

**Returns:**

None.

### 5.2.1.16 ROM\_GPIOPinTypeI2C

Configures pin(s) for use by the I2C peripheral.

**Prototype:**

```
void
ROM_GPIOPinTypeI2C(unsigned long ulPort,
                   unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].

ROM\_GPIOPinTypeI2C is a function pointer located at ROM\_GPIOTABLE[16].

**Parameters:**

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

*ucPins* 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, etc.

**Note:**

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

**Returns:**

None.

### 5.2.1.17 ROM\_GPIOPinTypeSSI

Configures pin(s) for use by the SSI peripheral.

**Prototype:**

```
void
ROM_GPIOPinTypeSSI(unsigned long ulPort,
                   unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinTypeSSI is a function pointer located at ROM\_GPIOTABLE[19].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** 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 may 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, etc.

**Note:**

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

**Returns:**

None.

### 5.2.1.18 ROM\_GPIOPinTypeTimer

Configures pin(s) for use by the Timer peripheral.

**Prototype:**

```
void
ROM_GPIOPinTypeTimer(unsigned long ulPort,
                    unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinTypeTimer is a function pointer located at ROM\_GPIOTABLE[20].

**Parameters:**

**ulPort** is the base address of the GPIO port.  
**ucPins** 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 may 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, etc.

**Note:**

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

**Returns:**

None.

### 5.2.1.19 ROM\_GPIOPinTypeUART

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

**Prototype:**

```
void  
ROM_GPIOPinTypeUART(unsigned long ulPort,  
                     unsigned char ucPins)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinTypeUART is a function pointer located at ROM\_GPIOTABLE[21].

**Parameters:**

*ulPort* is the base address of the GPIO port.  
*ucPins* is the bit-packed representation of the pin(s).

**Description:**

The UART pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may 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, etc.

**Note:**

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

**Returns:**

None.

### 5.2.1.20 ROM\_GPIOPinWrite

Writes a value to the specified pin(s).

**Prototype:**

```
void  
ROM_GPIOPinWrite(unsigned long ulPort,  
                 unsigned char ucPins,  
                 unsigned char ucVal)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_GPIOTABLE is an array of pointers located at ROM\_APITABLE[4].  
ROM\_GPIOPinWrite is a function pointer located at ROM\_GPIOTABLE[0].

**Parameters:**

***ulPort*** is the base address of the GPIO port.  
***ucPins*** is the bit-packed representation of the pin(s).  
***ucVal*** is the value to write to the pin(s).

**Description:**

Writes the corresponding bit values to the output pin(s) specified by *ucPins*. 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, etc.

**Returns:**

None.





## 6 Inter-Integrated Circuit (I2C)

Introduction .....	49
Functions .....	50

### 6.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the Stellaris I2C master and slave modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules.

The I2C 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 Stellaris 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 Stellaris I2C modules can operate at two speeds: Standard (100 kb/s) and Fast (400 kb/s).

Both the master and slave I2C modules can generate interrupts. The I2C master module will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The I2C slave module will generate interrupts when data has been sent or requested by a master.

#### 6.1.1 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 [ROM\\_I2CMasterInitExpClk\(\)](#). That function will set the bus speed and enable 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 [ROM\\_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 Stellaris I2C master must first call [ROM\\_I2CMasterBusBusy\(\)](#) before attempting to initiate the desired transaction. After determining that the bus is not busy, if trying to send data, the user must call the [ROM\\_I2CMasterDataPut\(\)](#) function. The transaction can then be initiated on the bus by calling the [ROM\\_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 those commands will 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 will involve looping on the return from [ROM\\_I2CMasterBusy\(\)](#). Once that function indicates that the I2C master is no longer busy, the bus transaction has been completed and can be checked for errors

using [ROM\\_I2CMasterErr\(\)](#). If there are no errors, then the data has been sent or is ready to be read using [ROM\\_I2CMasterDataGet\(\)](#). For the burst send and receive cases, the polling method also involves calling the [ROM\\_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 [ROM\\_I2CMasterControl\(\)](#) function should be called using the appropriate stop command (**I2C\_MASTER\_CMD\_BURST\_SEND\_ERROR\_STOP** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_ERROR\_STOP**).

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

## 6.1.2 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 [ROM\\_I2CSlaveInit\(\)](#). This will enable the I2C slave module and initialize the slave's own address. After the initialization is complete, the user may poll the slave status using [ROM\\_I2CSlaveStatus\(\)](#) to determine if a master requested a send or receive operation. Depending on the type of operation requested, the user can call [ROM\\_I2CSlaveDataPut\(\)](#) or [ROM\\_I2CSlaveDataGet\(\)](#) to complete the transaction. Alternatively, the I2C slave can handle transactions using an interrupt handler.

## 6.2 Functions

### Functions

- tBoolean [ROM\\_I2CMasterBusBusy](#) (unsigned long ulBase)
- tBoolean [ROM\\_I2CMasterBusy](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterControl](#) (unsigned long ulBase, unsigned long ulCmd)
- unsigned long [ROM\\_I2CMasterDataGet](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterDataPut](#) (unsigned long ulBase, unsigned char ucData)
- void [ROM\\_I2CMasterDisable](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterEnable](#) (unsigned long ulBase)
- unsigned long [ROM\\_I2CMasterErr](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterInitExpCik](#) (unsigned long ulBase, unsigned long ullI2CCik, tBoolean bFast)
- void [ROM\\_I2CMasterIntClear](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterIntDisable](#) (unsigned long ulBase)
- void [ROM\\_I2CMasterIntEnable](#) (unsigned long ulBase)
- tBoolean [ROM\\_I2CMasterIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- void [ROM\\_I2CMasterSlaveAddrSet](#) (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)
- unsigned long [ROM\\_I2CSlaveDataGet](#) (unsigned long ulBase)
- void [ROM\\_I2CSlaveDataPut](#) (unsigned long ulBase, unsigned char ucData)

- void [ROM\\_I2CSlaveDisable](#) (unsigned long ulBase)
- void [ROM\\_I2CSlaveEnable](#) (unsigned long ulBase)
- void [ROM\\_I2CSlaveInit](#) (unsigned long ulBase, unsigned char ucSlaveAddr)
- void [ROM\\_I2CSlaveIntClear](#) (unsigned long ulBase)
- void [ROM\\_I2CSlaveIntDisable](#) (unsigned long ulBase)
- void [ROM\\_I2CSlaveIntEnable](#) (unsigned long ulBase)
- tBoolean [ROM\\_I2CSlaveIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- unsigned long [ROM\\_I2CSlaveStatus](#) (unsigned long ulBase)
- void [ROM\\_UpdateI2C](#) (void)

## 6.2.1 Function Documentation

### 6.2.1.1 ROM\_I2CMasterBusBusy

Indicates whether or not the I2C bus is busy.

**Prototype:**

```
tBoolean
ROM_I2CMasterBusBusy(unsigned long ulBase)
```

**ROM Location:**

```
ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_I2CTABLE is an array of pointers located at ROM_APITABLE[3].
ROM_I2CMasterBusBusy is a function pointer located at ROM_I2CTABLE[17].
```

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**Description:**

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

**Returns:**

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

### 6.2.1.2 ROM\_I2CMasterBusy

Indicates whether or not the I2C Master is busy.

**Prototype:**

```
tBoolean
ROM_I2CMasterBusy(unsigned long ulBase)
```

**ROM Location:**

```
ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_I2CTABLE is an array of pointers located at ROM_APITABLE[3].
ROM_I2CMasterBusy is a function pointer located at ROM_I2CTABLE[16].
```

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**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**.

### 6.2.1.3 ROM\_I2CMasterControl

Controls the state of the I2C Master module.

**Prototype:**

```
void  
ROM_I2CMasterControl(unsigned long ulBase,  
                     unsigned long ulCmd)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].

ROM\_I2CMasterControl is a function pointer located at ROM\_I2CTABLE[18].

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**ulCmd** 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 *ucCmd* 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.

### 6.2.1.4 ROM\_I2CMasterDataGet

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

**Prototype:**

```
unsigned long  
ROM_I2CMasterDataGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
 ROM\_I2CMasterDataGet is a function pointer located at ROM\_I2CTABLE[20].

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**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 unsigned long.

## 6.2.1.5 ROM\_I2CMasterDataPut

Transmits a byte from the I2C Master.

**Prototype:**

```
void
ROM_I2CMasterDataPut(unsigned long ulBase,
                     unsigned char ucData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
 ROM\_I2CMasterDataPut is a function pointer located at ROM\_I2CTABLE[0].

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**ucData** data to be transmitted from the I2C Master

**Description:**

This function will place the supplied data into I2C Master Data Register.

**Returns:**

None.

## 6.2.1.6 ROM\_I2CMasterDisable

Disables the I2C master block.

**Prototype:**

```
void
ROM_I2CMasterDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
 ROM\_I2CMasterDisable is a function pointer located at ROM\_I2CTABLE[5].

**Parameters:**

***ulBase*** is the base address of the I2C Master module.

**Description:**

This will disable operation of the I2C master block.

**Returns:**

None.

### 6.2.1.7 ROM\_I2CMasterEnable

Enables the I2C Master block.

**Prototype:**

```
void  
ROM_I2CMasterEnable(unsigned long ulBase)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.  
`ROM_I2CMasterEnable` is a function pointer located at `ROM_I2CTABLE[3]`.

**Parameters:**

***ulBase*** is the base address of the I2C Master module.

**Description:**

This will enable operation of the I2C Master block.

**Returns:**

None.

### 6.2.1.8 ROM\_I2CMasterErr

Gets the error status of the I2C Master module.

**Prototype:**

```
unsigned long  
ROM_I2CMasterErr(unsigned long ulBase)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.  
`ROM_I2CMasterErr` is a function pointer located at `ROM_I2CTABLE[19]`.

**Parameters:**

***ulBase*** is the base address of the I2C Master module.

**Description:**

This function is used to obtain the error status of the Master module send and receive operations. It returns one of the following values:

- **I2C\_MASTER\_ERR\_NONE**

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

**Returns:**

None.

## 6.2.1.9 ROM\_I2CMasterInitExpClk

Initializes the I2C Master block.

**Prototype:**

```
void
ROM_I2CMasterInitExpClk(unsigned long ulBase,
                        unsigned long ulI2CClk,
                        tBoolean bFast)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].

ROM\_I2CMasterInitExpClk is a function pointer located at ROM\_I2CTABLE[1].

**Parameters:****ulBase** is the base address of the I2C Master module.**ulI2CClk** 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 will have set the bus speed for the master, and will have enabled the I2C Master block.

If the *bFast* parameter 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 [ROM\\_SysCtlClockGet\(\)](#), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to [ROM\\_SysCtlClockGet\(\)](#)).

**Returns:**

None.

## 6.2.1.10 ROM\_I2CMasterIntClear

Clears I2C Master interrupt sources.

**Prototype:**

```
void
ROM_I2CMasterIntClear(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CMasterIntClear is a function pointer located at ROM\_I2CTABLE[13].

**Parameters:**

*ulBase* is the base address of the I2C Master module.

**Description:**

The I2C Master interrupt source is cleared, 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:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 6.2.1.11 ROM\_I2CMasterIntDisable

Disables the I2C Master interrupt.

**Prototype:**

```
void  
ROM_I2CMasterIntDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CMasterIntDisable is a function pointer located at ROM\_I2CTABLE[9].

**Parameters:**

*ulBase* is the base address of the I2C Master module.

**Description:**

Disables the I2C Master interrupt source.

**Returns:**

None.

### 6.2.1.12 ROM\_I2CMasterIntEnable

Enables the I2C Master interrupt.

**Prototype:**

```
void  
ROM_I2CMasterIntEnable(unsigned long ulBase)
```



**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
 ROM\_I2CMasterIntEnable is a function pointer located at ROM\_I2CTABLE[7].

**Parameters:**

**ulBase** is the base address of the I2C Master module.

**Description:**

Enables the I2C Master interrupt source.

**Returns:**

None.

## 6.2.1.13 ROM\_I2CMasterIntStatus

Gets the current I2C Master interrupt status.

**Prototype:**

```
tBoolean
ROM_I2CMasterIntStatus(unsigned long ulBase,
                       tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
 ROM\_I2CMasterIntStatus is a function pointer located at ROM\_I2CTABLE[11].

**Parameters:**

**ulBase** is the base address of the I2C Master module.

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

**Description:**

This 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:**

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

## 6.2.1.14 ROM\_I2CMasterSlaveAddrSet

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

**Prototype:**

```
void
ROM_I2CMasterSlaveAddrSet(unsigned long ulBase,
                          unsigned char ucSlaveAddr,
                          tBoolean bReceive)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CMasterSlaveAddrSet is a function pointer located at ROM\_I2CTABLE[15].

**Parameters:**

**ulBase** is the base address of the I2C Master module.  
**ucSlaveAddr** 7-bit slave address  
**bReceive** flag indicating the type of communication with the slave

**Description:**

This function will set the address that the I2C Master will place on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address will indicate that the I2C Master is initiating a read from the slave; otherwise the address will indicate that the I2C Master is initiating a write to the slave.

**Returns:**

None.

### 6.2.1.15 ROM\_I2CSlaveDataGet

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

**Prototype:**

```
unsigned long  
ROM_I2CSlaveDataGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveDataGet is a function pointer located at ROM\_I2CTABLE[23].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**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 unsigned long.

### 6.2.1.16 ROM\_I2CSlaveDataPut

Transmits a byte from the I2C Slave.

**Prototype:**

```
void  
ROM_I2CSlaveDataPut(unsigned long ulBase,  
                    unsigned char ucData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveDataPut is a function pointer located at ROM\_I2CTABLE[22].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.  
**ucData** data to be transmitted from the I2C Slave

**Description:**

This function will place the supplied data into I2C Slave Data Register.

**Returns:**

None.

### 6.2.1.17 ROM\_I2CSlaveDisable

Disables the I2C slave block.

**Prototype:**

```
void  
ROM_I2CSlaveDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveDisable is a function pointer located at ROM\_I2CTABLE[6].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

This will disable operation of the I2C slave block.

**Returns:**

None.

### 6.2.1.18 ROM\_I2CSlaveEnable

Enables the I2C Slave block.

**Prototype:**

```
void  
ROM_I2CSlaveEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveEnable is a function pointer located at ROM\_I2CTABLE[4].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

This will enable operation of the I2C Slave block.

**Returns:**

None.

### 6.2.1.19 ROM\_I2CSlaveInit

Initializes the I2C Slave block.

**Prototype:**

```
void  
ROM_I2CSlaveInit(unsigned long ulBase,  
                 unsigned char ucSlaveAddr)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveInit is a function pointer located at ROM\_I2CTABLE[2].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.  
**ucSlaveAddr** 7-bit slave address

**Description:**

This function initializes operation of the I2C Slave block. Upon successful initialization of the I2C blocks, this function will have set the slave address and have enabled the I2C Slave block.

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

**Returns:**

None.

### 6.2.1.20 ROM\_I2CSlaveIntClear

Clears I2C Slave interrupt sources.

**Prototype:**

```
void  
ROM_I2CSlaveIntClear(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveIntClear is a function pointer located at ROM\_I2CTABLE[14].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

The I2C Slave interrupt source is cleared, 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:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 6.2.1.21 ROM\_I2CSlaveIntDisable

Disables the I2C Slave interrupt.

**Prototype:**

```
void  
ROM_I2CSlaveIntDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveIntDisable is a function pointer located at ROM\_I2CTABLE[10].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

Disables the I2C Slave interrupt source.

**Returns:**

None.

### 6.2.1.22 ROM\_I2CSlaveIntEnable

Enables the I2C Slave interrupt.

**Prototype:**

```
void  
ROM_I2CSlaveIntEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveIntEnable is a function pointer located at ROM\_I2CTABLE[8].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

Enables the I2C Slave interrupt source.

**Returns:**  
None.

### 6.2.1.23 ROM\_I2CSlaveIntStatus

Gets the current I2C Slave interrupt status.

**Prototype:**

```
tBoolean  
ROM_I2CSlaveIntStatus(unsigned long ulBase,  
                      tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveIntStatus is a function pointer located at ROM\_I2CTABLE[12].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.  
**bMasked** is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

**Description:**

This 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:**

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

### 6.2.1.24 ROM\_I2CSlaveStatus

Gets the I2C Slave module status

**Prototype:**

```
unsigned long  
ROM_I2CSlaveStatus(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_I2CSlaveStatus is a function pointer located at ROM\_I2CTABLE[21].

**Parameters:**

**ulBase** is the base address of the I2C Slave module.

**Description:**

This function will return 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 slave's own address has been received.

**6.2.1.25 ROM\_UpdateI2C**

Starts an update over the I2C0 interface.

**Prototype:**

```
void  
ROM_UpdateI2C(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_I2CTABLE is an array of pointers located at ROM\_APITABLE[3].  
ROM\_UpdateI2C is a function pointer located at ROM\_I2CTABLE[24].

**Description:**

Calling this function commences an update of the firmware via the I2C0 interface. This function assumes that the I2C0 interface has already been configured and is currently operational. The I2C0 slave is used for data transfer, and the I2C0 master is used to monitor bus busy conditions (therefore, both must be enabled).

**Returns:**

Never returns.





# 7 Interrupt Controller (NVIC)

Introduction .....	65
Functions .....	65

## 7.1 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. This version of the Stellaris family supports thirty-two interrupt sources and eight 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 NVIC is tightly coupled with the Cortex-M3 microprocessor. When the processor responds to an interrupt, NVIC will supply the address of the function to handle the interrupt directly to the processor. This 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, reducing interrupt response time.

The interrupt prioritization in the NVIC allows higher priority interrupts to be handled 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).

Sub-prioritization is also possible; instead of having N bits of preemptable prioritization, NVIC can be configured (via 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 will not cause a preemption; tail chaining will instead be 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 will be processed first. 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.

## 7.2 Functions

### Functions

- void [ROM\\_IntDisable](#) (unsigned long ulInterrupt)
- void [ROM\\_IntEnable](#) (unsigned long ulInterrupt)
- long [ROM\\_IntPriorityGet](#) (unsigned long ulInterrupt)
- unsigned long [ROM\\_IntPriorityGroupingGet](#) (void)
- void [ROM\\_IntPriorityGroupingSet](#) (unsigned long ulBits)
- void [ROM\\_IntPrioritySet](#) (unsigned long ulInterrupt, unsigned char ucPriority)

## 7.2.1 Function Documentation

### 7.2.1.1 ROM\_IntDisable

Disables an interrupt.

**Prototype:**

```
void  
ROM_IntDisable(unsigned long ulInterrupt)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntDisable is a function pointer located at ROM\_INTERRUPTTABLE[3].

**Parameters:**

*ulInterrupt* specifies the interrupt to be disabled.

**Description:**

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

**Returns:**

None.

### 7.2.1.2 ROM\_IntEnable

Enables an interrupt.

**Prototype:**

```
void  
ROM_IntEnable(unsigned long ulInterrupt)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntEnable is a function pointer located at ROM\_INTERRUPTTABLE[0].

**Parameters:**

*ulInterrupt* specifies the interrupt to be enabled.

**Description:**

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

**Returns:**

None.

### 7.2.1.3 ROM\_IntPriorityGet

Gets the priority of an interrupt.

**Prototype:**

```
long  
ROM_IntPriorityGet(unsigned long ulInterrupt)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntPriorityGet is a function pointer located at ROM\_INTERRUPTTABLE[7].

**Parameters:**

*ulInterrupt* specifies the interrupt in question.

**Description:**

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

**Returns:**

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

### 7.2.1.4 ROM\_IntPriorityGroupingGet

Gets the priority grouping of the interrupt controller.

**Prototype:**

```
unsigned long  
ROM_IntPriorityGroupingGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntPriorityGroupingGet is a function pointer located at ROM\_INTERRUPTTABLE[5].

**Description:**

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

**Returns:**

The number of bits of preemptable priority.

### 7.2.1.5 ROM\_IntPriorityGroupingSet

Sets the priority grouping of the interrupt controller.

**Prototype:**

```
void  
ROM_IntPriorityGroupingSet(unsigned long ulBits)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntPriorityGroupingSet is a function pointer located at ROM\_INTERRUPTTABLE[4].

**Parameters:**

**ulBits** 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 are dependent upon the hardware implementation; on the Stellaris family, three bits are available for hardware interrupt prioritization and therefore priority grouping values of three through seven have the same effect.

**Returns:**

None.

### 7.2.1.6 ROM\_IntPrioritySet

Sets the priority of an interrupt.

**Prototype:**

```
void  
ROM_IntPrioritySet(unsigned long ulInterrupt,  
                  unsigned char ucPriority)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_INTERRUPTTABLE is an array of pointers located at ROM\_APITABLE[14].  
ROM\_IntPrioritySet is a function pointer located at ROM\_INTERRUPTTABLE[6].

**Parameters:**

**ulInterrupt** specifies the interrupt in question.  
**ucPriority** specifies the priority of the interrupt.

**Description:**

This function is used to set the priority of an interrupt. When multiple interrupts are asserted simultaneously, the ones 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 only look at the upper N bits of the priority level (where N is 3 for the Stellaris family), so any prioritization must be performed in those bits. The remaining bits can be used to sub-prioritize 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.

**Returns:**

None.

## 8 Synchronous Serial Interface (SSI)

Introduction .....	69
Functions .....	69

### 8.1 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 to be between 4 and 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 SSI module's input clock. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

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

### 8.2 Functions

#### Functions

- void [ROM\\_SSIConfigSetExpClk](#) (unsigned long ulBase, unsigned long ulSSIClk, unsigned long ulProtocol, unsigned long ulMode, unsigned long ulBitRate, unsigned long ulDataWidth)
- void [ROM\\_SSIDataGet](#) (unsigned long ulBase, unsigned long \*pulData)
- long [ROM\\_SSIDataGetNonBlocking](#) (unsigned long ulBase, unsigned long \*pulData)
- void [ROM\\_SSIDataPut](#) (unsigned long ulBase, unsigned long ulData)
- long [ROM\\_SSIDataPutNonBlocking](#) (unsigned long ulBase, unsigned long ulData)
- void [ROM\\_SSIDisable](#) (unsigned long ulBase)
- void [ROM\\_SSIEnable](#) (unsigned long ulBase)
- void [ROM\\_SSIIntClear](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_SSIIntDisable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_SSIIntEnable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long [ROM\\_SSIIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- void [ROM\\_UpdateSSI](#) (void)

## 8.2.1 Function Documentation

### 8.2.1.1 ROM\_SSIConfigSetExpClk

Configures the synchronous serial interface.

**Prototype:**

```
void  
ROM_SSIConfigSetExpClk(unsigned long ulBase,  
                        unsigned long ulSSIClk,  
                        unsigned long ulProtocol,  
                        unsigned long ulMode,  
                        unsigned long ulBitRate,  
                        unsigned long ulDataWidth)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIConfigSetExpClk is a function pointer located at ROM\_SSITABLE[1].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ulSSIClk** is the rate of the clock supplied to the SSI module.  
**ulProtocol** specifies the data transfer protocol.  
**ulMode** specifies the mode of operation.  
**ulBitRate** specifies the clock rate.  
**ulDataWidth** 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 *ulProtocol* parameter defines the data frame format. The *ulProtocol* 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 *ulMode* 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 *ulMode* parameter can be one of the following values: **SSI\_MODE\_MASTER**, **SSI\_MODE\_SLAVE**, or **SSI\_MODE\_SLAVE\_OD**.

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

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

where FSSI is the frequency of the clock supplied to the SSI module.

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

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

**Returns:**

None.

### 8.2.1.2 ROM\_SSIDataGet

Gets a data element from the SSI receive FIFO.

**Prototype:**

```
void  
ROM_SSIDataGet(unsigned long ulBase,  
               unsigned long *pulData)
```

**ROM Location:**

[ROM\\_APITABLE](#) is an array of pointers located at 0x0100.0010.  
[ROM\\_SSITABLE](#) is an array of pointers located at [ROM\\_APITABLE](#)[2].  
[ROM\\_SSIDataGet](#) is a function pointer located at [ROM\\_SSITABLE](#)[9].

**Parameters:**

*ulBase* specifies the SSI module base address.

*pulData* pointer to a storage location for data that was received over the SSI interface.

**Description:**

This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *pulData* parameter.

**Note:**

Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by [ROM\\_SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8 bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

**Returns:**

None.

### 8.2.1.3 ROM\_SSIDataGetNonBlocking

Gets a data element from the SSI receive FIFO.

**Prototype:**

```
long  
ROM_SSIDataGetNonBlocking(unsigned long ulBase,  
                           unsigned long *pulData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIDataGetNonBlocking is a function pointer located at ROM\_SSITABLE[10].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**pulData** pointer to a storage location for data that was received over the SSI interface.

**Description:**

This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *ulData* parameter. If there is no data in the FIFO, then this function will return a zero.

**Note:**

Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by [ROM\\_SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8 bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

**Returns:**

Returns the number of elements read from the SSI receive FIFO.

#### 8.2.1.4 ROM\_SSIDataPut

Puts a data element into the SSI transmit FIFO.

**Prototype:**

```
void  
ROM_SSIDataPut(unsigned long ulBase,  
               unsigned long ulData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIDataPut is a function pointer located at ROM\_SSITABLE[0].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ulData** data to be transmitted over the SSI interface.

**Description:**

This function will place the supplied data into the transmit FIFO of the specified SSI module.

**Note:**

The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by [ROM\\_SSIConfigSetExpClk\(\)](#). For example, if the interface is configured for 8 bit data width, the upper 24 bits of *ulData* will be discarded.

**Returns:**

None.



### 8.2.1.5 ROM\_SSIDataPutNonBlocking

Puts a data element into the SSI transmit FIFO.

**Prototype:**

```
long  
ROM_SSIDataPutNonBlocking(unsigned long ulBase,  
                           unsigned long ulData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIDataPutNonBlocking is a function pointer located at ROM\_SSITABLE[8].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ulData** data to be transmitted over the SSI interface.

**Description:**

This function will place the supplied data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function will return a zero.

**Note:**

The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by [ROM\\_SSICfgSetExpClk\(\)](#). For example, if the interface is configured for 8 bit data width, the upper 24 bits of *ulData* will be discarded.

**Returns:**

Returns the number of elements written to the SSI transmit FIFO.

### 8.2.1.6 ROM\_SSIDisable

Disables the synchronous serial interface.

**Prototype:**

```
void  
ROM_SSIDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIDisable is a function pointer located at ROM\_SSITABLE[3].

**Parameters:**

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

**Description:**

This will disable operation of the synchronous serial interface.

**Returns:**

None.

### 8.2.1.7 ROM\_SSIEnable

Enables the synchronous serial interface.

**Prototype:**

```
void  
ROM_SSIEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIEnable is a function pointer located at ROM\_SSITABLE[2].

**Parameters:**

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

**Description:**

This will enable operation of the synchronous serial interface. It must be configured before it is enabled.

**Returns:**

None.

### 8.2.1.8 ROM\_SSIIntClear

Clears SSI interrupt sources.

**Prototype:**

```
void  
ROM_SSIIntClear(unsigned long ulBase,  
                unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIIntClear is a function pointer located at ROM\_SSITABLE[7].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ulIntFlags** 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 must be done in the interrupt handler to keep it from being called again immediately upon exit. The *ulIntFlags* parameter can consist of either or both the **SSI\_RXTO** and **SSI\_RXOR** values.

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**  
None.

### 8.2.1.9 ROM\_SSIIntDisable

Disables individual SSI interrupt sources.

**Prototype:**

```
void  
ROM_SSIIntDisable(unsigned long ulBase,  
                  unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIIntDisable is a function pointer located at ROM\_SSITABLE[5].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ullntFlags** is a bit mask of the interrupt sources to be disabled.

**Description:**

Disables the indicated SSI interrupt sources. The *ullntFlags* parameter can be any of the **SSI\_TXFF**, **SSI\_RXFF**, **SSI\_RXTO**, or **SSI\_RXOR** values.

**Returns:**  
None.

### 8.2.1.10 ROM\_SSIIntEnable

Enables individual SSI interrupt sources.

**Prototype:**

```
void  
ROM_SSIIntEnable(unsigned long ulBase,  
                 unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIIntEnable is a function pointer located at ROM\_SSITABLE[4].

**Parameters:**

**ulBase** specifies the SSI module base address.  
**ullntFlags** 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 *ullntFlags* parameter can be any of the **SSI\_TXFF**, **SSI\_RXFF**, **SSI\_RXTO**, or **SSI\_RXOR** values.

**Returns:**  
None.

### 8.2.1.11 ROM\_SSIIntStatus

Gets the current interrupt status.

**Prototype:**  
unsigned long  
ROM\_SSIIntStatus(unsigned long ulBase,  
tBoolean bMasked)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_SSIIntStatus is a function pointer located at ROM\_SSITABLE[6].

**Parameters:**  
**ulBase** specifies the SSI module base address.  
**bMasked** is false if the raw interrupt status is required and true if the masked interrupt status is required.

**Description:**  
This 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\_RXTO**, and **SSI\_RXOR**.

### 8.2.1.12 ROM\_UpdateSSI

Starts an update over the SSI0 interface.

**Prototype:**  
void  
ROM\_UpdateSSI(void)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SSITABLE is an array of pointers located at ROM\_APITABLE[2].  
ROM\_UpdateSSI is a function pointer located at ROM\_SSITABLE[11].

**Description:**  
Calling this function commences an update of the firmware via the SSI0 interface. This function assumes that the SSI0 interface has already been configured and is currently operational.

**Returns:**  
Never returns.

## 9 System Control

Introduction .....	77
Functions .....	78

### 9.1 Introduction

System control determines the overall operation of the device. It controls the clocking of the device, the set of peripherals that are enabled, configuration of the device and its resets, and provides information about the device.

The members of the Stellaris family have a varying peripheral set and memory sizes. The device has a set of read-only registers that indicate the size of the memories, the peripherals that are present, and the pins that are present for peripherals that have a varying number of pins. This information can be used to write adaptive software that will run on more than one member of the Stellaris family.

The device can be clocked from one of five sources: an external oscillator, the main oscillator, the internal oscillator, the internal oscillator divided by four, or the PLL. The PLL can use any of the four oscillators as its input. Since the internal oscillator has a very wide error range (+/- 50%), it cannot be used for applications that require specific timing; its real use is for detecting failures of the main oscillator and the PLL, and for applications that strictly respond to external events and do not use time-based peripherals (such as a timer). When using the PLL, the input clock frequency is constrained to specific frequencies between 3.579545 MHz and 16.384 MHz (that is, the standard crystal frequencies in that range). When direct clocking with an external oscillator or the main oscillator, the frequency is constrained to between 0 Hz and 50 MHz (depending on the device). The internal oscillator is 12 MHz, +/- 30%; its frequency will vary by device, with voltage, and with temperature. The internal oscillator provides no tuning or frequency measurement mechanism; its frequency is not adjustable.

Three modes of operation are supported by the Stellaris family: run mode, sleep mode, and 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 may change (depending upon the run mode clock configuration) and the processor no longer executes code (and is no longer clocked). An interrupt will return the device to run mode from one of the sleep modes; the sleep modes are entered upon request from the code.

The device has an internal LDO for generating the on-chip 2.5 V power supply; the output voltage of the LDO can be adjusted between 2.25 V and 2.75 V. Depending upon the application, lower voltage may be advantageous for its power savings, or higher voltage may be advantageous for its improved performance. The default setting of 2.5 V is a good compromise between the two, and should not be changed without careful consideration and evaluation.

There are several system events that, when detected, will cause system control to reset the device. These events are the input voltage dropping too low, the LDO voltage dropping too low, an external reset, a software reset request, and a watchdog timeout. The properties of some of these events can be configured, and the reason for a reset can be determined from system control.

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. Care must be taken with deep-sleep

mode, though, since in this mode the PLL is no longer used and the system is clocked by the input crystal. Peripherals that depend upon a particular input clock rate (such as a timer) will not operate as expected in deep-sleep mode due to the clock rate change; these peripherals must either be reconfigured upon entry to and exit from deep-sleep mode, or simply not enabled in deep-sleep mode.

There are various system events that, when detected, will cause system control to generate a processor interrupt. These events are the PLL achieving lock, the internal LDO current limit being exceeded, the internal oscillator failing, the main oscillator failing, the input voltage dropping too low, the internal LDO voltage dropping too low, and the PLL failing. Each of these interrupts can be individually enabled or disabled, and the sources must be cleared by the interrupt handler when they occur.

## 9.2 Functions

### Functions

- unsigned long [ROM\\_SysCtlADCSpeedGet](#) (void)
- void [ROM\\_SysCtlADCSpeedSet](#) (unsigned long ulSpeed)
- unsigned long [ROM\\_SysCtlClockGet](#) (void)
- void [ROM\\_SysCtlClockSet](#) (unsigned long ulConfig)
- void [ROM\\_SysCtlDeepSleep](#) (void)
- unsigned long [ROM\\_SysCtlFlashSizeGet](#) (void)
- void [ROM\\_SysCtlGPIOAHBDisable](#) (unsigned long ulGPIOPeripheral)
- void [ROM\\_SysCtlGPIOAHBEnable](#) (unsigned long ulGPIOPeripheral)
- void [ROM\\_SysCtlIntClear](#) (unsigned long ulInts)
- void [ROM\\_SysCtlIntDisable](#) (unsigned long ulInts)
- void [ROM\\_SysCtlIntEnable](#) (unsigned long ulInts)
- unsigned long [ROM\\_SysCtlIntStatus](#) (tBoolean bMasked)
- unsigned long [ROM\\_SysCtlLDOGet](#) (void)
- void [ROM\\_SysCtlLDOSet](#) (unsigned long ulVoltage)
- void [ROM\\_SysCtlPeripheralClockGating](#) (tBoolean bEnable)
- void [ROM\\_SysCtlPeripheralDeepSleepDisable](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralDeepSleepEnable](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralDisable](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralEnable](#) (unsigned long ulPeripheral)
- tBoolean [ROM\\_SysCtlPeripheralPresent](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralReset](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralSleepDisable](#) (unsigned long ulPeripheral)
- void [ROM\\_SysCtlPeripheralSleepEnable](#) (unsigned long ulPeripheral)
- tBoolean [ROM\\_SysCtlPinPresent](#) (unsigned long ulPin)
- void [ROM\\_SysCtlReset](#) (void)
- void [ROM\\_SysCtlResetCauseClear](#) (unsigned long ulCauses)
- unsigned long [ROM\\_SysCtlResetCauseGet](#) (void)
- void [ROM\\_SysCtlSleep](#) (void)
- unsigned long [ROM\\_SysCtlSRAMSizeGet](#) (void)

## 9.2.1 Function Documentation

### 9.2.1.1 ROM\_SysCtlADCSpeedGet

Gets the sample rate of the ADC.

**Prototype:**

```
unsigned long  
ROM_SysCtlADCSpeedGet (void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlADCSpeedGet is a function pointer located at ROM\_SYSCCTLTABLE[28].

**Description:**

This function gets the current sample rate of the ADC.

**Returns:**

Returns the current ADC sample rate; will be one of **SYSCCTL\_ADCSPEED\_1MSPS**, **SYSCCTL\_ADCSPEED\_500KSPS**, **SYSCCTL\_ADCSPEED\_250KSPS**, or **SYSCCTL\_ADCSPEED\_125KSPS**.

### 9.2.1.2 ROM\_SysCtlADCSpeedSet

Sets the sample rate of the ADC.

**Prototype:**

```
void  
ROM_SysCtlADCSpeedSet (unsigned long ulSpeed)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlADCSpeedSet is a function pointer located at ROM\_SYSCCTLTABLE[27].

**Parameters:**

**ulSpeed** is the desired sample rate of the ADC; must be one of **SYSCCTL\_ADCSPEED\_1MSPS**, **SYSCCTL\_ADCSPEED\_500KSPS**, **SYSCCTL\_ADCSPEED\_250KSPS**, or **SYSCCTL\_ADCSPEED\_125KSPS**.

**Description:**

This function sets the rate at which the ADC samples are captured by the ADC block. The sampling speed may be limited by the hardware, so the sample rate may end up being slower than requested. [ROM\\_SysCtlADCSpeedGet\(\)](#) will return the actual speed in use.

**Returns:**

None.

### 9.2.1.3 ROM\_SysCtlClockGet

Gets the processor clock rate.

**Prototype:**

```
unsigned long  
ROM_SysCtlClockGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlClockGet is a function pointer located at ROM\_SYSCCTLTABLE[24].

**Description:**

This function determines the clock rate of the processor clock. This is also the clock rate of all the peripheral modules.

**Note:**

This will not return accurate results if ROM\_SysCtlClockSet() has not been called to configure the clocking of the device, or if the device is directly clocked from a crystal (or a clock source) that is not one of the supported crystal frequencies. In the later case, this function should not be used to determine the system clock rate.

**Returns:**

The processor clock rate.

### 9.2.1.4 ROM\_SysCtlClockSet

Sets the clocking of the device.

**Prototype:**

```
void  
ROM_SysCtlClockSet(unsigned long ulConfig)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlClockSet is a function pointer located at ROM\_SYSCCTLTABLE[23].

**Parameters:**

*ulConfig* is the required configuration of the device clocking.

**Description:**

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The *ulConfig* parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen.

The system clock divider is chosen with one of the following values: **SYSCCTL\_SYSDIV\_1**, **SYSCCTL\_SYSDIV\_2**, **SYSCCTL\_SYSDIV\_3**, ... **SYSCCTL\_SYSDIV\_64**.

The use of the PLL is chosen with either **SYSCCTL\_USE\_PLL** or **SYSCCTL\_USE\_OSC**.



The external crystal frequency is chosen with one of the following values: **SYSCTL\_XTAL\_1MHZ**, **SYSCTL\_XTAL\_1\_84MHZ**, **SYSCTL\_XTAL\_2MHZ**, **SYSCTL\_XTAL\_2\_45MHZ**, **SYSCTL\_XTAL\_3\_57MHZ**, **SYSCTL\_XTAL\_3\_68MHZ**, **SYSCTL\_XTAL\_4MHZ**, **SYSCTL\_XTAL\_4\_09MHZ**, **SYSCTL\_XTAL\_4\_91MHZ**, **SYSCTL\_XTAL\_5MHZ**, **SYSCTL\_XTAL\_5\_12MHZ**, **SYSCTL\_XTAL\_6MHZ**, **SYSCTL\_XTAL\_6\_14MHZ**, **SYSCTL\_XTAL\_7\_37MHZ**, **SYSCTL\_XTAL\_8MHZ**, **SYSCTL\_XTAL\_8\_19MHZ**, **SYSCTL\_XTAL\_10MHZ**, **SYSCTL\_XTAL\_12MHZ**, **SYSCTL\_XTAL\_12\_2MHZ**, **SYSCTL\_XTAL\_13\_5MHZ**, **SYSCTL\_XTAL\_14\_3MHZ**, **SYSCTL\_XTAL\_16MHZ**, or **SYSCTL\_XTAL\_16\_3MHZ**. Values below **SYSCTL\_XTAL\_3\_57MHZ** are not valid when the PLL is in operation.

The oscillator source is chosen with one of the following values: **SYSCTL\_OSC\_MAIN**, **SYSCTL\_OSC\_INT**, **SYSCTL\_OSC\_INT4**, **SYSCTL\_OSC\_EXT32**, or **SYSCTL\_OSC\_INT30**. **SYSCTL\_OSC\_EXT32** is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

The internal and main oscillators are disabled with the **SYSCTL\_INT\_OSC\_DIS** and **SYSCTL\_MAIN\_OSC\_DIS** flags, respectively. The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device will be prevented by the hardware.

To clock the system from an external source (such as an external crystal oscillator), use **SYSCTL\_USE\_OSC | SYSCTL\_OSC\_MAIN**. To clock the system from the main oscillator, use **SYSCTL\_USE\_OSC | SYSCTL\_OSC\_MAIN**. To clock the system from the PLL, use **SYSCTL\_USE\_PLL | SYSCTL\_OSC\_MAIN**, and select the appropriate crystal with one of the **SYSCTL\_XTAL\_xxx** values.

**Note:**

If selecting the PLL as the system clock source (i.e. via **SYSCTL\_USE\_PLL**), this function will poll the PLL lock interrupt to determine when the PLL has locked. If an interrupt handler for the system control interrupt is in place, and it responds to and clears the PLL lock interrupt, this function will delay until its timeout has occurred instead of completing as soon as PLL lock is achieved.

**Returns:**

None.

### 9.2.1.5 ROM\_SysCtlDeepSleep

Puts the processor into deep-sleep mode.

**Prototype:**

```
void
ROM_SysCtlDeepSleep(void)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at `0x0100.0010`.  
**ROM\_SYSCTLTABLE** is an array of pointers located at `ROM_APITABLE[13]`.  
**ROM\_SysCtlDeepSleep** is a function pointer located at `ROM_SYSCTLTABLE[20]`.

**Description:**

This function places the processor into deep-sleep mode; it will not return until the processor returns to run mode. The peripherals that are enabled via

[ROM\\_SysCtlPeripheralDeepSleepEnable\(\)](#) continue to operate and can wake up the processor (if automatic clock gating is enabled with [ROM\\_SysCtlPeripheralClockGating\(\)](#), otherwise all peripherals continue to operate).

**Returns:**

None.

### 9.2.1.6 ROM\_SysCtlFlashSizeGet

Gets the size of the flash.

**Prototype:**

```
unsigned long  
ROM_SysCtlFlashSizeGet(void)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_SYSCCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.  
`ROM_SysCtlFlashSizeGet` is a function pointer located at `ROM_SYSCCTLTABLE[2]`.

**Description:**

This function determines the size of the flash on the Stellaris device.

**Returns:**

The total number of bytes of flash.

### 9.2.1.7 ROM\_SysCtlGPIOAHBDisable

Disables a GPIO peripheral for access from the high speed bus.

**Prototype:**

```
void  
ROM_SysCtlGPIOAHBDisable(unsigned long ulGPIOPeripheral)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_SYSCCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.  
`ROM_SysCtlGPIOAHBDisable` is a function pointer located at `ROM_SYSCCTLTABLE[30]`.

**Parameters:**

***ulGPIOPeripheral*** is the GPIO peripheral to disable.

**Description:**

This function will disable the specified GPIOP peripherals for access from the high speed bus. Once disabled, the GPIO peripheral is accessed from the peripheral bus.

The `ulGPIOPeripheral` argument must be only one of the following values:  
**SYSCTL\_PERIPH\_GPIOA, SYSCTL\_PERIPH\_GPIOB, SYSCTL\_PERIPH\_GPIOC,**  
**SYSCTL\_PERIPH\_GPIOD, SYSCTL\_PERIPH\_GPIOE, SYSCTL\_PERIPH\_GPIOF,**  
**SYSCTL\_PERIPH\_GPIOG, or SYSCTL\_PERIPH\_GPIOH.**

**Returns:**

None.

### 9.2.1.8 ROM\_SysCtlGPIOAHBEnable

Enables a GPIO peripheral for access from the high speed bus.

**Prototype:**

```
void  
ROM_SysCtlGPIOAHBEnable(unsigned long ulGPIOPeripheral)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlGPIOAHBEnable is a function pointer located at ROM\_SYSCCTLTABLE[29].

**Parameters:**

*ulGPIOPeripheral* is the GPIO peripheral to enable.

**Description:**

This function is used to enable the specified GPIO peripherals to be accessed from the high speed bus instead of the peripheral bus. When a GPIO peripheral is enabled for high speed access, the `_AHB_BASE` form of the base address should be used for GPIO functions. For example, instead of using `GPIO_PORTA_BASE` as the base address for GPIO functions, use `GPIO_PORTA_AHB_BASE` instead.

The `ulGPIOPeripheral` argument must be only one of the following values: `SYSCTL_PERIPH_GPIOA`, `SYSCTL_PERIPH_GPIOB`, `SYSCTL_PERIPH_GPIOC`, `SYSCTL_PERIPH_GPIOD`, `SYSCTL_PERIPH_GPIOE`, `SYSCTL_PERIPH_GPIOF`, `SYSCTL_PERIPH_GPIOG`, or `SYSCTL_PERIPH_GPIOH`.

**Returns:**

None.

### 9.2.1.9 ROM\_SysCtlIntClear

Clears system control interrupt sources.

**Prototype:**

```
void  
ROM_SysCtlIntClear(unsigned long ulInts)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlIntClear is a function pointer located at ROM\_SYSCCTLTABLE[15].

**Parameters:**

*ulInts* is a bit mask of the interrupt sources to be cleared. Must be a logical OR of `SYSCTL_INT_MOSC_PUP`, `SYSCTL_INT_USBPLL_LOCK`, `SYSCTL_INT_PLL_LOCK`, and/or `SYSCTL_INT_BOR`.

**Description:**

The specified system control interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 9.2.1.10 ROM\_SysCtlIntDisable

Disables individual system control interrupt sources.

**Prototype:**

```
void  
ROM_SysCtlIntDisable(unsigned long ulInts)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCITLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlIntDisable is a function pointer located at ROM\_SYSCITLTABLE[14].

**Parameters:**

**ulInts** is a bit mask of the interrupt sources to be disabled. Must be a logical OR of **SYSCITL\_INT\_MOSC\_PUP**, **SYSCITL\_INT\_USBPLL\_LOCK**, **SYSCITL\_INT\_PLL\_LOCK**, and/or **SYSCITL\_INT\_BOR**.

**Description:**

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

**Returns:**

None.

### 9.2.1.11 ROM\_SysCtlIntEnable

Enables individual system control interrupt sources.

**Prototype:**

```
void  
ROM_SysCtlIntEnable(unsigned long ulInts)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCITLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlIntEnable is a function pointer located at ROM\_SYSCITLTABLE[13].

**Parameters:**

*ullnts* is a bit mask of the interrupt sources to be enabled. Must be a logical OR of **SYSCTL\_INT\_MOSC\_PUP**, **SYSCTL\_INT\_USBPLL\_LOCK**, **SYSCTL\_INT\_PLL\_LOCK**, and/or **SYSCTL\_INT\_BOR**.

**Description:**

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

**Returns:**

None.

### 9.2.1.12 ROM\_SysCtlIntStatus

Gets the current interrupt status.

**Prototype:**

```
unsigned long  
ROM_SysCtlIntStatus(tBoolean bMasked)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.  
**ROM\_SYSCTLTABLE** is an array of pointers located at **ROM\_APITABLE**[13].  
**ROM\_SysCtlIntStatus** is a function pointer located at **ROM\_SYSCTLTABLE**[16].

**Parameters:**

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

**Description:**

This returns the interrupt status for the system controller. 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 **SYSCTL\_INT\_MOSC\_PUP**, **SYSCTL\_INT\_USBPLL\_LOCK**, **SYSCTL\_INT\_PLL\_LOCK**, and/or **SYSCTL\_INT\_BOR**.

### 9.2.1.13 ROM\_SysCtlLDOGet

Gets the output voltage of the LDO.

**Prototype:**

```
unsigned long  
ROM_SysCtlLDOGet(void)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.  
**ROM\_SYSCTLTABLE** is an array of pointers located at **ROM\_APITABLE**[13].  
**ROM\_SysCtlLDOGet** is a function pointer located at **ROM\_SYSCTLTABLE**[18].

**Description:**

This function determines the output voltage of the LDO, as specified by the control register.

**Returns:**

Returns the current voltage of the LDO; will be one of **SYSCTL\_LDO\_2\_25V**, **SYSCTL\_LDO\_2\_30V**, **SYSCTL\_LDO\_2\_35V**, **SYSCTL\_LDO\_2\_40V**, **SYSCTL\_LDO\_2\_45V**, **SYSCTL\_LDO\_2\_50V**, **SYSCTL\_LDO\_2\_55V**, **SYSCTL\_LDO\_2\_60V**, **SYSCTL\_LDO\_2\_65V**, **SYSCTL\_LDO\_2\_70V**, or **SYSCTL\_LDO\_2\_75V**.

### 9.2.1.14 ROM\_SysCtlLDOSet

Sets the output voltage of the LDO.

**Prototype:**

```
void  
ROM_SysCtlLDOSet(unsigned long ulVoltage)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at `0x0100.0010`.  
**ROM\_SYSCTLTABLE** is an array of pointers located at `ROM_APITABLE[13]`.  
**ROM\_SysCtlLDOSet** is a function pointer located at `ROM_SYSCTLTABLE[17]`.

**Parameters:**

**ulVoltage** is the required output voltage from the LDO. Must be one of **SYSCTL\_LDO\_2\_25V**, **SYSCTL\_LDO\_2\_30V**, **SYSCTL\_LDO\_2\_35V**, **SYSCTL\_LDO\_2\_40V**, **SYSCTL\_LDO\_2\_45V**, **SYSCTL\_LDO\_2\_50V**, **SYSCTL\_LDO\_2\_55V**, **SYSCTL\_LDO\_2\_60V**, **SYSCTL\_LDO\_2\_65V**, **SYSCTL\_LDO\_2\_70V**, or **SYSCTL\_LDO\_2\_75V**.

**Description:**

This function sets the output voltage of the LDO. The default voltage is 2.5 V; it can be adjusted +/- 10%.

**Returns:**

None.

### 9.2.1.15 ROM\_SysCtlPeripheralClockGating

Controls peripheral clock gating in sleep and deep-sleep mode.

**Prototype:**

```
void  
ROM_SysCtlPeripheralClockGating(tBoolean bEnable)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at `0x0100.0010`.  
**ROM\_SYSCTLTABLE** is an array of pointers located at `ROM_APITABLE[13]`.  
**ROM\_SysCtlPeripheralClockGating** is a function pointer located at `ROM_SYSCTLTABLE[12]`.

**Parameters:**

**bEnable** is a boolean that is **true** if the sleep and deep-sleep peripheral configuration should be used and **false** if not.

**Description:**

This function controls how peripherals are clocked when the processor goes into sleep or deep-sleep mode. By default, the peripherals are clocked the same as in run mode; if peripheral clock gating is enabled they are clocked according to the configuration set by [ROM\\_SysCtlPeripheralSleepEnable\(\)](#), [ROM\\_SysCtlPeripheralSleepDisable\(\)](#), [ROM\\_SysCtlPeripheralDeepSleepEnable\(\)](#), and [ROM\\_SysCtlPeripheralDeepSleepDisable\(\)](#).

**Returns:**

None.

### 9.2.1.16 ROM\_SysCtlPeripheralDeepSleepDisable

Disables a peripheral in deep-sleep mode.

**Prototype:**

```
void
ROM_SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_SYSCCTLTABLE is an array of pointers located at ROM\_APITABLE[13].

ROM\_SysCtlPeripheralDeepSleepDisable is a function pointer located at ROM\_SYSCCTLTABLE[11].

**Parameters:**

*ulPeripheral* 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 (via [ROM\\_SysCtlPeripheralEnable\(\)](#)), the peripheral will automatically resume operation when the processor leaves deep-sleep mode, maintaining its entire state from before deep-sleep mode was entered.

Deep-sleep mode clocking of peripherals must be enabled via [ROM\\_SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ulPeripheral* argument must be one of the following values: **SYSCCTL\_PERIPH\_ADC0**, **SYSCCTL\_PERIPH\_CAN0**, **SYSCCTL\_PERIPH\_CAN1**, **SYSCCTL\_PERIPH\_GPIOA**, **SYSCCTL\_PERIPH\_GPIOB**, **SYSCCTL\_PERIPH\_GPIOC**, **SYSCCTL\_PERIPH\_GPIOD**, **SYSCCTL\_PERIPH\_GPIOE**, **SYSCCTL\_PERIPH\_GPIOF**, **SYSCCTL\_PERIPH\_GPIOG**, **SYSCCTL\_PERIPH\_GPIOH**, **SYSCCTL\_PERIPH\_HIBERNATE**, **SYSCCTL\_PERIPH\_I2C0**, **SYSCCTL\_PERIPH\_I2C1**, **SYSCCTL\_PERIPH\_SSI0**, **SYSCCTL\_PERIPH\_SSI1**, **SYSCCTL\_PERIPH\_TIMER0**, **SYSCCTL\_PERIPH\_TIMER1**, **SYSCCTL\_PERIPH\_TIMER2**, **SYSCCTL\_PERIPH\_TIMER3**, **SYSCCTL\_PERIPH\_UART0**, **SYSCCTL\_PERIPH\_UART1**, **SYSCCTL\_PERIPH\_UART2**, **SYSCCTL\_PERIPH\_UDMA**, **SYSCCTL\_PERIPH\_USB0**, or **SYSCCTL\_PERIPH\_WDOG0**.

**Returns:**

None.

### 9.2.1.17 ROM\_SysCtlPeripheralDeepSleepEnable

Enables a peripheral in deep-sleep mode.

**Prototype:**

```
void  
ROM_SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlPeripheralDeepSleepEnable is a function pointer located at ROM\_SYSCTLTABLE[10].

**Parameters:**

*ulPeripheral* 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. Since the clocking configuration of the device may change, not all peripherals can safely continue operating while the processor is in sleep mode. Those that must run at a particular frequency (such as a timer) will not work as expected if the clock changes. It is the responsibility of the caller to make sensible choices.

Deep-sleep mode clocking of peripherals must be enabled via [ROM\\_SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ulPeripheral* argument must be one of the following values: **SYSCTL\_PERIPH\_ADC0**, **SYSCTL\_PERIPH\_CAN0**, **SYSCTL\_PERIPH\_CAN1**, **SYSCTL\_PERIPH\_GPIOA**, **SYSCTL\_PERIPH\_GPIOB**, **SYSCTL\_PERIPH\_GPIOC**, **SYSCTL\_PERIPH\_GPIOD**, **SYSCTL\_PERIPH\_GPIOE**, **SYSCTL\_PERIPH\_GPIOF**, **SYSCTL\_PERIPH\_GPIOG**, **SYSCTL\_PERIPH\_GPIOH**, **SYSCTL\_PERIPH\_HIBERNATE**, **SYSCTL\_PERIPH\_I2C0**, **SYSCTL\_PERIPH\_I2C1**, **SYSCTL\_PERIPH\_SSI0**, **SYSCTL\_PERIPH\_SSI1**, **SYSCTL\_PERIPH\_TIMER0**, **SYSCTL\_PERIPH\_TIMER1**, **SYSCTL\_PERIPH\_TIMER2**, **SYSCTL\_PERIPH\_TIMER3**, **SYSCTL\_PERIPH\_UART0**, **SYSCTL\_PERIPH\_UART1**, **SYSCTL\_PERIPH\_UART2**, **SYSCTL\_PERIPH\_UDMA**, **SYSCTL\_PERIPH\_USB0**, or **SYSCTL\_PERIPH\_WDOG0**.

**Returns:**

None.

### 9.2.1.18 ROM\_SysCtlPeripheralDisable

Disables a peripheral.

**Prototype:**

```
void  
ROM_SysCtlPeripheralDisable(unsigned long ulPeripheral)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlPeripheralDisable is a function pointer located at ROM\_SYSCTLTABLE[7].



**Parameters:**

*ulPeripheral* is the peripheral to disable.

**Description:**

Peripherals are disabled with this function. Once disabled, they will not operate or respond to register reads/writes.

The *ulPeripheral* argument must be only one of the following values:

<code>SYSCTL_PERIPH_ADC0,</code>	<code>SYSCTL_PERIPH_CAN0,</code>	<code>SYSCTL_PERIPH_CAN1,</code>
<code>SYSCTL_PERIPH_GPIOA,</code>	<code>SYSCTL_PERIPH_GPIOB,</code>	<code>SYSCTL_PERIPH_GPIOC,</code>
<code>SYSCTL_PERIPH_GPIOD,</code>	<code>SYSCTL_PERIPH_GPIOE,</code>	<code>SYSCTL_PERIPH_GPIOF,</code>
<code>SYSCTL_PERIPH_GPIOG,</code>	<code>SYSCTL_PERIPH_GPIOH,</code>	<code>SYSCTL_PERIPH_HIBERNATE,</code>
<code>SYSCTL_PERIPH_I2C0,</code>	<code>SYSCTL_PERIPH_I2C1,</code>	<code>SYSCTL_PERIPH_SSI0,</code>
<code>SYSCTL_PERIPH_SSI1,</code>	<code>SYSCTL_PERIPH_TIMER0,</code>	<code>SYSCTL_PERIPH_TIMER1,</code>
<code>SYSCTL_PERIPH_TIMER2,</code>	<code>SYSCTL_PERIPH_TIMER3,</code>	<code>SYSCTL_PERIPH_UART0,</code>
<code>SYSCTL_PERIPH_UART1,</code>	<code>SYSCTL_PERIPH_UART2,</code>	<code>SYSCTL_PERIPH_UDMA,</code>
<code>SYSCTL_PERIPH_USB0,</code>	or <code>SYSCTL_PERIPH_WDOG0.</code>	

**Returns:**

None.

### 9.2.1.19 ROM\_SysCtlPeripheralEnable

Enables a peripheral.

**Prototype:**

```
void
ROM_SysCtlPeripheralEnable(unsigned long ulPeripheral)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.

`ROM_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.

`ROM_SysCtlPeripheralEnable` is a function pointer located at `ROM_SYSCTLTABLE[6]`.

**Parameters:**

*ulPeripheral* is the peripheral to enable.

**Description:**

Peripherals are enabled with this function. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

The *ulPeripheral* argument must be only one of the following values:

<code>SYSCTL_PERIPH_ADC0,</code>	<code>SYSCTL_PERIPH_CAN0,</code>	<code>SYSCTL_PERIPH_CAN1,</code>
<code>SYSCTL_PERIPH_GPIOA,</code>	<code>SYSCTL_PERIPH_GPIOB,</code>	<code>SYSCTL_PERIPH_GPIOC,</code>
<code>SYSCTL_PERIPH_GPIOD,</code>	<code>SYSCTL_PERIPH_GPIOE,</code>	<code>SYSCTL_PERIPH_GPIOF,</code>
<code>SYSCTL_PERIPH_GPIOG,</code>	<code>SYSCTL_PERIPH_GPIOH,</code>	<code>SYSCTL_PERIPH_HIBERNATE,</code>
<code>SYSCTL_PERIPH_I2C0,</code>	<code>SYSCTL_PERIPH_I2C1,</code>	<code>SYSCTL_PERIPH_SSI0,</code>
<code>SYSCTL_PERIPH_SSI1,</code>	<code>SYSCTL_PERIPH_TIMER0,</code>	<code>SYSCTL_PERIPH_TIMER1,</code>
<code>SYSCTL_PERIPH_TIMER2,</code>	<code>SYSCTL_PERIPH_TIMER3,</code>	<code>SYSCTL_PERIPH_UART0,</code>
<code>SYSCTL_PERIPH_UART1,</code>	<code>SYSCTL_PERIPH_UART2,</code>	<code>SYSCTL_PERIPH_UDMA,</code>
<code>SYSCTL_PERIPH_USB0,</code>	or <code>SYSCTL_PERIPH_WDOG0.</code>	

**Note:**

It takes five clock cycles after the write to enable a peripheral before the the peripheral is actually enabled. During this time, attempts to access the peripheral will result in a bus fault. Care should be taken to ensure that the peripheral is not accessed during this brief time period.

**Returns:**

None.

### 9.2.1.20 ROM\_SysCtlPeripheralPresent

Determines if a peripheral is present.

**Prototype:**

```
tBoolean  
ROM_SysCtlPeripheralPresent(unsigned long ulPeripheral)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].

ROM\_SysCtlPeripheralPresent is a function pointer located at ROM\_SYSCTLTABLE[4].

**Parameters:**

*ulPeripheral* is the peripheral in question.

**Description:**

Determines if a particular peripheral is present in the device. Each member of the Stellaris family has a different peripheral set; this will determine which are present on this device.

The *ulPeripheral* argument must be only one of the following values:  
SYSCTL\_PERIPH\_ADC0,      SYSCTL\_PERIPH\_CAN0,      SYSCTL\_PERIPH\_CAN1,  
SYSCTL\_PERIPH\_COMP0,    SYSCTL\_PERIPH\_COMP1,    SYSCTL\_PERIPH\_COMP2,  
SYSCTL\_PERIPH\_GPIOA,    SYSCTL\_PERIPH\_GPIOB,    SYSCTL\_PERIPH\_GPIOC,  
SYSCTL\_PERIPH\_GPIOD,    SYSCTL\_PERIPH\_GPIOE,    SYSCTL\_PERIPH\_GPIOF,  
SYSCTL\_PERIPH\_GPIOG,    SYSCTL\_PERIPH\_GPIOH,    SYSCTL\_PERIPH\_HIBERNATE,  
SYSCTL\_PERIPH\_I2C0,      SYSCTL\_PERIPH\_I2C1,      SYSCTL\_PERIPH\_MPU,  
SYSCTL\_PERIPH\_PLL,      SYSCTL\_PERIPH\_PWM,      SYSCTL\_PERIPH\_QEI0,  
SYSCTL\_PERIPH\_SSI0,      SYSCTL\_PERIPH\_SSI1,      SYSCTL\_PERIPH\_TEMP,  
SYSCTL\_PERIPH\_TIMER0,    SYSCTL\_PERIPH\_TIMER1,    SYSCTL\_PERIPH\_TIMER2,  
SYSCTL\_PERIPH\_TIMER3,    SYSCTL\_PERIPH\_UART0,    SYSCTL\_PERIPH\_UART1,  
SYSCTL\_PERIPH\_UART2,    SYSCTL\_PERIPH\_UDMA,    SYSCTL\_PERIPH\_USB0, or  
SYSCTL\_PERIPH\_WDOG0.

**Returns:**

Returns **true** if the specified peripheral is present and **false** if it is not.

### 9.2.1.21 ROM\_SysCtlPeripheralReset

Performs a software reset of a peripheral.

**Prototype:**

```
void  
ROM_SysCtlPeripheralReset(unsigned long ulPeripheral)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.

`ROM_SYSCCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.

`ROM_SysCtlPeripheralReset` is a function pointer located at `ROM_SYSCCTLTABLE[5]`.

**Parameters:**

***ulPeripheral*** 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 ***ulPeripheral*** argument must be only one of the following values:  
**SYSCTL\_PERIPH\_ADC0**, **SYSCTL\_PERIPH\_CAN0**, **SYSCTL\_PERIPH\_CAN1**,  
**SYSCTL\_PERIPH\_GPIOA**, **SYSCTL\_PERIPH\_GPIOB**, **SYSCTL\_PERIPH\_GPIOC**,  
**SYSCTL\_PERIPH\_GPIOD**, **SYSCTL\_PERIPH\_GPIOE**, **SYSCTL\_PERIPH\_GPIOF**,  
**SYSCTL\_PERIPH\_GPIOG**, **SYSCTL\_PERIPH\_GPIOH**, **SYSCTL\_PERIPH\_HIBERNATE**,  
**SYSCTL\_PERIPH\_I2C0**, **SYSCTL\_PERIPH\_I2C1**, **SYSCTL\_PERIPH\_SSI0**,  
**SYSCTL\_PERIPH\_SSI1**, **SYSCTL\_PERIPH\_TIMER0**, **SYSCTL\_PERIPH\_TIMER1**,  
**SYSCTL\_PERIPH\_TIMER2**, **SYSCTL\_PERIPH\_TIMER3**, **SYSCTL\_PERIPH\_UART0**,  
**SYSCTL\_PERIPH\_UART1**, **SYSCTL\_PERIPH\_UART2**, **SYSCTL\_PERIPH\_UDMA**,  
**SYSCTL\_PERIPH\_USB0**, or **SYSCTL\_PERIPH\_WDOG0**.

**Returns:**

None.

9.2.1.22 `ROM_SysCtlPeripheralSleepDisable`

Disables a peripheral in sleep mode.

**Prototype:**

```
void
ROM_SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.

`ROM_SYSCCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.

`ROM_SysCtlPeripheralSleepDisable` is a function pointer located at `ROM_SYSCCTLTABLE[9]`.

**Parameters:**

***ulPeripheral*** 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 to lower the current draw of the device. If enabled (via [ROM\\_SysCtlPeripheralEnable\(\)](#)), the peripheral will automatically resume operation when the processor leaves sleep mode, maintaining its entire state from before sleep mode was entered.

Sleep mode clocking of peripherals must be enabled via [ROM\\_SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The `ulPeripheral` argument must be only one of the following values:  
**SYSCTL\_PERIPH\_ADC0,**      **SYSCTL\_PERIPH\_CAN0,**      **SYSCTL\_PERIPH\_CAN1,**  
**SYSCTL\_PERIPH\_GPIOA,**      **SYSCTL\_PERIPH\_GPIOB,**      **SYSCTL\_PERIPH\_GPIOC,**  
**SYSCTL\_PERIPH\_GPIOD,**      **SYSCTL\_PERIPH\_GPIOE,**      **SYSCTL\_PERIPH\_GPIOF,**  
**SYSCTL\_PERIPH\_GPIOG,**      **SYSCTL\_PERIPH\_GPIOH,**      **SYSCTL\_PERIPH\_HIBERNATE,**  
**SYSCTL\_PERIPH\_I2C0,**      **SYSCTL\_PERIPH\_I2C1,**      **SYSCTL\_PERIPH\_SSI0,**  
**SYSCTL\_PERIPH\_SSI1,**      **SYSCTL\_PERIPH\_TIMER0,**      **SYSCTL\_PERIPH\_TIMER1,**  
**SYSCTL\_PERIPH\_TIMER2,**      **SYSCTL\_PERIPH\_TIMER3,**      **SYSCTL\_PERIPH\_UART0,**  
**SYSCTL\_PERIPH\_UART1,**      **SYSCTL\_PERIPH\_UART2,**      **SYSCTL\_PERIPH\_UDMA,**  
**SYSCTL\_PERIPH\_USB0,** or **SYSCTL\_PERIPH\_WDOG0.**

**Returns:**  
None.

### 9.2.1.23 ROM\_SysCtlPeripheralSleepEnable

Enables a peripheral in sleep mode.

**Prototype:**

```
void
ROM_SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.  
`ROM_SysCtlPeripheralSleepEnable` is a function pointer located at `ROM_SYSCTLTABLE[8]`.

**Parameters:**

***ulPeripheral*** is the peripheral to enable in sleep mode.

**Description:**

This function allows a peripheral to continue operating when the processor goes into sleep mode. Since 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.

Sleep mode clocking of peripherals must be enabled via [ROM\\_SysCtlPeripheralClockGating\(\)](#); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The `ulPeripheral` argument must be only one of the following values:  
**SYSCTL\_PERIPH\_ADC0,**      **SYSCTL\_PERIPH\_CAN0,**      **SYSCTL\_PERIPH\_CAN1,**  
**SYSCTL\_PERIPH\_GPIOA,**      **SYSCTL\_PERIPH\_GPIOB,**      **SYSCTL\_PERIPH\_GPIOC,**  
**SYSCTL\_PERIPH\_GPIOD,**      **SYSCTL\_PERIPH\_GPIOE,**      **SYSCTL\_PERIPH\_GPIOF,**  
**SYSCTL\_PERIPH\_GPIOG,**      **SYSCTL\_PERIPH\_GPIOH,**      **SYSCTL\_PERIPH\_HIBERNATE,**  
**SYSCTL\_PERIPH\_I2C0,**      **SYSCTL\_PERIPH\_I2C1,**      **SYSCTL\_PERIPH\_SSI0,**  
**SYSCTL\_PERIPH\_SSI1,**      **SYSCTL\_PERIPH\_TIMER0,**      **SYSCTL\_PERIPH\_TIMER1,**  
**SYSCTL\_PERIPH\_TIMER2,**      **SYSCTL\_PERIPH\_TIMER3,**      **SYSCTL\_PERIPH\_UART0,**  
**SYSCTL\_PERIPH\_UART1,**      **SYSCTL\_PERIPH\_UART2,**      **SYSCTL\_PERIPH\_UDMA,**  
**SYSCTL\_PERIPH\_USB0,** or **SYSCTL\_PERIPH\_WDOG0.**

**Returns:**  
None.

### 9.2.1.24 ROM\_SysCtlPinPresent

Determines if a pin is present.

**Prototype:**

```
tBoolean  
ROM_SysCtlPinPresent(unsigned long ulPin)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlPinPresent is a function pointer located at ROM\_SYSCTLTABLE[3].

**Parameters:**

*ulPin* is the pin in question.

**Description:**

Determines if a particular pin is present in the device. Some peripherals have a varying number of pins across members of the Stellaris family; this will determine which are present on this device.

The *ulPin* argument must be only one of the following values: **SYSCTL\_PIN\_ADC0**, **SYSCTL\_PIN\_ADC1**, **SYSCTL\_PIN\_ADC2**, **SYSCTL\_PIN\_ADC3**, **SYSCTL\_PIN\_ADC4**, **SYSCTL\_PIN\_ADC5**, **SYSCTL\_PIN\_ADC6**, **SYSCTL\_PIN\_ADC7**, **SYSCTL\_PIN\_CCP0**, **SYSCTL\_PIN\_CCP1**, **SYSCTL\_PIN\_CCP2**, **SYSCTL\_PIN\_CCP3**, **SYSCTL\_PIN\_CCP4**, **SYSCTL\_PIN\_CCP5**, **SYSCTL\_PIN\_CCP6**, **SYSCTL\_PIN\_CCP7**, or **SYSCTL\_PIN\_32KHZ**.

**Returns:**

Returns **true** if the specified pin is present and **false** if it is not.

### 9.2.1.25 ROM\_SysCtlReset

Resets the device.

**Prototype:**

```
void  
ROM_SysCtlReset(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlReset is a function pointer located at ROM\_SYSCTLTABLE[19].

**Description:**

This function will perform a software reset of the entire device. The processor and all peripherals will be reset and all device registers will return to their default values (with the exception of the reset cause register, which will maintain its current value but have the software reset bit set as well).

**Returns:**

This function does not return.

### 9.2.1.26 ROM\_SysCtlResetCauseClear

Clears reset reasons.

**Prototype:**

```
void  
ROM_SysCtlResetCauseClear(unsigned long ulCauses)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlResetCauseClear is a function pointer located at ROM\_SYSCTLTABLE[22].

**Parameters:**

**ulCauses** are the reset causes to be cleared; must be a logical OR of **SYSCTL\_CAUSE\_SW**, **SYSCTL\_CAUSE\_WDOG**, **SYSCTL\_CAUSE\_BOR**, **SYSCTL\_CAUSE\_POR**, and/or **SYSCTL\_CAUSE\_EXT**.

**Description:**

This function clears the specified sticky reset reasons. Once cleared, another reset for the same reason can be detected, and a reset for a different reason can be distinguished (instead of having two reset causes set). If the reset reason is used by an application, all reset causes should be cleared after they are retrieved with [ROM\\_SysCtlResetCauseGet\(\)](#).

**Returns:**

None.

### 9.2.1.27 ROM\_SysCtlResetCauseGet

Gets the reason for a reset.

**Prototype:**

```
unsigned long  
ROM_SysCtlResetCauseGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlResetCauseGet is a function pointer located at ROM\_SYSCTLTABLE[21].

**Description:**

This function will return the reason(s) for a reset. Since the reset reasons are sticky until either cleared by software or an external reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason will be a logical OR of **SYSCTL\_CAUSE\_SW**, **SYSCTL\_CAUSE\_WDOG**, **SYSCTL\_CAUSE\_BOR**, **SYSCTL\_CAUSE\_POR**, and/or **SYSCTL\_CAUSE\_EXT**.

**Returns:**

The reason(s) for a reset.

### 9.2.1.28 ROM\_SysCtlSleep

Puts the processor into sleep mode.

**Prototype:**

```
void  
ROM_SysCtlSleep(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlSleep is a function pointer located at ROM\_SYSCTLTABLE[0].

**Description:**

This function places the processor into sleep mode; it will not return until the processor returns to run mode. The peripherals that are enabled via [ROM\\_SysCtlPeripheralSleepEnable\(\)](#) continue to operate and can wake up the processor (if automatic clock gating is enabled with [ROM\\_SysCtlPeripheralClockGating\(\)](#), otherwise all peripherals continue to operate).

**Returns:**

None.

### 9.2.1.29 ROM\_SysCtlSRAMSizeGet

Gets the size of the SRAM.

**Prototype:**

```
unsigned long  
ROM_SysCtlSRAMSizeGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSCTLTABLE is an array of pointers located at ROM\_APITABLE[13].  
ROM\_SysCtlSRAMSizeGet is a function pointer located at ROM\_SYSCTLTABLE[1].

**Description:**

This function determines the size of the SRAM on the Stellaris device.

**Returns:**

The total number of bytes of SRAM.





# 10 System Tick (SysTick)

Introduction .....	97
Functions .....	97

## 10.1 Introduction

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M3 microprocessor. Its intended purpose is to provide a periodic interrupt for a RTOS, but it can be used for other simple timing purposes.

The SysTick interrupt handler does not need to clear the SysTick interrupt source. This will be done automatically by NVIC when the SysTick interrupt handler is called.

## 10.2 Functions

### Functions

- void [ROM\\_SysTickDisable](#) (void)
- void [ROM\\_SysTickEnable](#) (void)
- void [ROM\\_SysTickIntDisable](#) (void)
- void [ROM\\_SysTickIntEnable](#) (void)
- unsigned long [ROM\\_SysTickPeriodGet](#) (void)
- void [ROM\\_SysTickPeriodSet](#) (unsigned long ulPeriod)
- unsigned long [ROM\\_SysTickValueGet](#) (void)

### 10.2.1 Function Documentation

#### 10.2.1.1 ROM\_SysTickDisable

Disables the SysTick counter.

**Prototype:**

```
void  
ROM_SysTickDisable(void)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_SYSTICKTABLE` is an array of pointers located at `ROM_APITABLE[10]`.  
`ROM_SysTickDisable` is a function pointer located at `ROM_SYSTICKTABLE[2]`.

**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.

### 10.2.1.2 ROM\_SysTickEnable

Enables the SysTick counter.

**Prototype:**  
void  
ROM\_SysTickEnable(void)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickEnable is a function pointer located at ROM\_SYSTICKTABLE[1].

**Description:**  
This will start the SysTick counter. If an interrupt handler has been registered, it will be called when the SysTick counter rolls over.

**Note:**  
Calling this function will cause 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 [ROM\\_SysTickPeriodSet\(\)](#). If an immediate reload is required, register NVIC\_ST\_CURRENT must be written to force this. Any write to this register clears the SysTick counter to 0 and will cause a reload with the supplied period on the next clock.

**Returns:**  
None.

### 10.2.1.3 ROM\_SysTickIntDisable

Disables the SysTick interrupt.

**Prototype:**  
void  
ROM\_SysTickIntDisable(void)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickIntDisable is a function pointer located at ROM\_SYSTICKTABLE[4].

**Description:**  
This function will disable the SysTick interrupt, preventing it from being reflected to the processor.

**Returns:**  
None.

#### 10.2.1.4 ROM\_SysTickIntEnable

Enables the SysTick interrupt.

**Prototype:**

```
void  
ROM_SysTickIntEnable(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickIntEnable is a function pointer located at ROM\_SYSTICKTABLE[3].

**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.

#### 10.2.1.5 ROM\_SysTickPeriodGet

Gets the period of the SysTick counter.

**Prototype:**

```
unsigned long  
ROM_SysTickPeriodGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickPeriodGet is a function pointer located at ROM\_SYSTICKTABLE[6].

**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.

#### 10.2.1.6 ROM\_SysTickPeriodSet

Sets the period of the SysTick counter.

**Prototype:**

```
void  
ROM_SysTickPeriodSet(unsigned long ulPeriod)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickPeriodSet is a function pointer located at ROM\_SYSTICKTABLE[5].

**Parameters:**

**ulPeriod** 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, register NVIC\_ST\_CURRENT must be written. Any write to this register clears the SysTick counter to 0 and will cause a reload with the **ulPeriod** supplied here on the next clock after the SysTick is enabled.

**Returns:**

None.

### 10.2.1.7 ROM\_SysTickValueGet

Gets the current value of the SysTick counter.

**Prototype:**

```
unsigned long  
ROM_SysTickValueGet(void)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_SYSTICKTABLE is an array of pointers located at ROM\_APITABLE[10].  
ROM\_SysTickValueGet is a function pointer located at ROM\_SYSTICKTABLE[0].

**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.

# 11 Timer

Introduction .....	101
Functions .....	101

## 11.1 Introduction

The timer API provides a set of functions for dealing with the timer module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

The timer module provides two 16-bit timer/counters that can be configured to operate independently as timers or event counters, or they can be configured to operate as one 32-bit timer or one 32-bit Real Time Clock (RTC). For the purpose of this API, the two timers provided by the timer are referred to as TimerA and TimerB.

When configured as either a 32-bit or 16-bit timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured as a one-shot timer, when it reaches zero the timer will cease counting. If configured as a continuous timer, when it reaches zero the timer will continue counting from a reloaded value. When configured as a 32-bit timer, the timer can also be configured to operate as an RTC. In that case, the timer expects to be driven by a 32 KHz external clock, which is divided down to produce 1 second clock ticks.

When in 16-bit 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 either count the time between events, or it can count 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 line used to capture events becomes an output line, and the timer is used to drive an edge-aligned pulse onto that line.

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 zero, or when the RTC matches a certain value.

## 11.2 Functions

### Functions

- void [ROM\\_TimerConfigure](#) (unsigned long ulBase, unsigned long ulConfig)
- void [ROM\\_TimerControlLevel](#) (unsigned long ulBase, unsigned long ulTimer, tBoolean blnVert)
- void [ROM\\_TimerControlStall](#) (unsigned long ulBase, unsigned long ulTimer, tBoolean bStall)
- void [ROM\\_TimerControlTrigger](#) (unsigned long ulBase, unsigned long ulTimer, tBoolean bEnable)
- void [ROM\\_TimerDisable](#) (unsigned long ulBase, unsigned long ulTimer)

- void [ROM\\_TimerEnable](#) (unsigned long ulBase, unsigned long ulTimer)
- void [ROM\\_TimerIntClear](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_TimerIntDisable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_TimerIntEnable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long [ROM\\_TimerIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- unsigned long [ROM\\_TimerLoadGet](#) (unsigned long ulBase, unsigned long ulTimer)
- void [ROM\\_TimerLoadSet](#) (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long [ROM\\_TimerMatchGet](#) (unsigned long ulBase, unsigned long ulTimer)
- void [ROM\\_TimerMatchSet](#) (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long [ROM\\_TimerPrescaleGet](#) (unsigned long ulBase, unsigned long ulTimer)
- void [ROM\\_TimerPrescaleSet](#) (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void [ROM\\_TimerRTCDisable](#) (unsigned long ulBase)
- void [ROM\\_TimerRTCEnable](#) (unsigned long ulBase)
- unsigned long [ROM\\_TimerValueGet](#) (unsigned long ulBase, unsigned long ulTimer)

## 11.2.1 Function Documentation

### 11.2.1.1 ROM\_TimerConfigure

Configures the timer(s).

**Prototype:**

```
void  
ROM_TimerConfigure(unsigned long ulBase,  
                   unsigned long ulConfig)
```

**ROM Location:**

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.  
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.  
`ROM_TimerConfigure` is a function pointer located at `ROM_TIMERTABLE[3]`.

**Parameters:**

***ulBase*** is the base address of the timer module.  
***ulConfig*** 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 configuration is specified in *ulConfig* as one of the following values:

- **TIMER\_CFG\_32\_BIT\_OS** - 32-bit one shot timer
- **TIMER\_CFG\_32\_BIT\_PER** - 32-bit periodic timer
- **TIMER\_CFG\_32\_RTC** - 32-bit real time clock timer
- **TIMER\_CFG\_16\_BIT\_PAIR** - Two 16-bit timers

When configured for a pair of 16-bit timers, each timer is separately configured. The first timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the following values and *ulConfig*:

- **TIMER\_CFG\_A\_ONE\_SHOT** - 16-bit one shot timer
- **TIMER\_CFG\_A\_PERIODIC** - 16-bit periodic timer
- **TIMER\_CFG\_A\_CAP\_COUNT** - 16-bit edge count capture
- **TIMER\_CFG\_A\_CAP\_TIME** - 16-bit edge time capture
- **TIMER\_CFG\_A\_PWM** - 16-bit PWM output

Similarly, the second timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the corresponding **TIMER\_CFG\_B\_\*** values and *ulConfig*.

**Returns:**

None.

### 11.2.1.2 ROM\_TimerControlLevel

Controls the output level.

**Prototype:**

```
void
ROM_TimerControlLevel(unsigned long ulBase,
                     unsigned long ulTimer,
                     tBoolean bInvert)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].

ROM\_TimerControlLevel is a function pointer located at ROM\_TIMERTABLE[4].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer(s) to adjust; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_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 will be made active low; otherwise, it will be made active high.

**Returns:**

None.

### 11.2.1.3 ROM\_TimerControlStall

Controls the stall handling.

**Prototype:**

```
void
ROM_TimerControlStall(unsigned long ulBase,
                      unsigned long ulTimer,
                      tBoolean bStall)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerControlStall is a function pointer located at ROM\_TIMERTABLE[7].

**Parameters:**

**ulBase** is the base address of the timer module.  
**ulTimer** specifies the timer(s) to be adjusted; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_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 will stop counting if the processor enters debug mode; otherwise the timer will keep running while in debug mode.

**Returns:**

None.

#### 11.2.1.4 ROM\_TimerControlTrigger

Enables or disables the trigger output.

**Prototype:**

```
void
ROM_TimerControlTrigger(unsigned long ulBase,
                       unsigned long ulTimer,
                       tBoolean bEnable)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerControlTrigger is a function pointer located at ROM\_TIMERTABLE[5].

**Parameters:**

**ulBase** is the base address of the timer module.  
**ulTimer** specifies the timer to adjust; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_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.



### 11.2.1.5 ROM\_TimerDisable

Disables the timer(s).

**Prototype:**

```
void  
ROM_TimerDisable(unsigned long ulBase,  
                 unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerDisable is a function pointer located at ROM\_TIMERTABLE[2].

**Parameters:**

*ulBase* is the base address of the timer module.

*ulTimer* specifies the timer(s) to disable; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_BOTH**.

**Description:**

This will disable operation of the timer module.

**Returns:**

None.

### 11.2.1.6 ROM\_TimerEnable

Enables the timer(s).

**Prototype:**

```
void  
ROM_TimerEnable(unsigned long ulBase,  
               unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerEnable is a function pointer located at ROM\_TIMERTABLE[1].

**Parameters:**

*ulBase* is the base address of the timer module.

*ulTimer* specifies the timer(s) to enable; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_BOTH**.

**Description:**

This will enable operation of the timer module. The timer must be configured before it is enabled.

**Returns:**

None.

### 11.2.1.7 ROM\_TimerIntClear

Clears timer interrupt sources.

**Prototype:**

```
void  
ROM_TimerIntClear(unsigned long ulBase,  
                  unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerIntClear is a function pointer located at ROM\_TIMERTABLE[0].

**Parameters:**

**ulBase** is the base address of the timer module.  
**ullntFlags** 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 must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ullntFlags* parameter has the same definition as the *ullntFlags* parameter to [ROM\\_TimerIntEnable\(\)](#).

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 11.2.1.8 ROM\_TimerIntDisable

Disables individual timer interrupt sources.

**Prototype:**

```
void  
ROM_TimerIntDisable(unsigned long ulBase,  
                    unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerIntDisable is a function pointer located at ROM\_TIMERTABLE[20].

**Parameters:**

**ulBase** is the base address of the timer module.  
**ullntFlags** 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 *ullntFlags* parameter has the same definition as the *ullntFlags* parameter to [ROM\\_TimerIntEnable\(\)](#).

**Returns:**

None.

### 11.2.1.9 ROM\_TimerIntEnable

Enables individual timer interrupt sources.

**Prototype:**

```
void  
ROM_TimerIntEnable(unsigned long ulBase,  
                   unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].

ROM\_TimerIntEnable is a function pointer located at ROM\_TIMERTABLE[19].

**Parameters:**

**ulBase** is the base address of the timer module.

**ullntFlags** 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 *ullntFlags* parameter must be the logical OR of any combination of the following:

- **TIMER\_CAPB\_EVENT** - Capture B event interrupt
- **TIMER\_CAPB\_MATCH** - Capture B match interrupt
- **TIMER\_TIMB\_TIMEOUT** - Timer B timeout interrupt
- **TIMER\_RTC\_MATCH** - RTC interrupt mask
- **TIMER\_CAPA\_EVENT** - Capture A event interrupt
- **TIMER\_CAPA\_MATCH** - Capture A match interrupt
- **TIMER\_TIMA\_TIMEOUT** - Timer A timeout interrupt

**Returns:**

None.

### 11.2.1.10 ROM\_TimerIntStatus

Gets the current interrupt status.

**Prototype:**

```
unsigned long
ROM_TimerIntStatus(unsigned long ulBase,
                   tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerIntStatus is a function pointer located at ROM\_TIMERTABLE[21].

**Parameters:**

**ulBase** 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 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 [ROM\\_TimerIntEnable\(\)](#).

### 11.2.1.11 ROM\_TimerLoadGet

Gets the timer load value.

**Prototype:**

```
unsigned long
ROM_TimerLoadGet(unsigned long ulBase,
                 unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerLoadGet is a function pointer located at ROM\_TIMERTABLE[15].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer; must be one of **TIMER\_A** or **TIMER\_B**. Only **TIMER\_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.

### 11.2.1.12 ROM\_TimerLoadSet

Sets the timer load value.

**Prototype:**

```
void
ROM_TimerLoadSet (unsigned long ulBase,
                  unsigned long ulTimer,
                  unsigned long ulValue)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerLoadSet is a function pointer located at ROM\_TIMERTABLE[14].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer(s) to adjust; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_BOTH**. Only **TIMER\_A** should be used when the timer is configured for 32-bit operation.

**ulValue** 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.

### 11.2.1.13 ROM\_TimerMatchGet

Gets the timer match value.

**Prototype:**

```
unsigned long
ROM_TimerMatchGet (unsigned long ulBase,
                  unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerMatchGet is a function pointer located at ROM\_TIMERTABLE[18].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer; must be one of **TIMER\_A** or **TIMER\_B**. Only **TIMER\_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.

### 11.2.1.14 ROM\_TimerMatchSet

Sets the timer match value.

**Prototype:**

```
void  
ROM_TimerMatchSet (unsigned long ulBase,  
                  unsigned long ulTimer,  
                  unsigned long ulValue)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerMatchSet is a function pointer located at ROM\_TIMERTABLE[17].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer(s) to adjust; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_BOTH**. Only **TIMER\_A** should be used when the timer is configured for 32-bit operation.

**ulValue** 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.

### 11.2.1.15 ROM\_TimerPrescaleGet

Get the timer prescale value.

**Prototype:**

```
unsigned long  
ROM_TimerPrescaleGet (unsigned long ulBase,  
                    unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerPrescaleGet is a function pointer located at ROM\_TIMERTABLE[11].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer; must be one of **TIMER\_A** or **TIMER\_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.

**Returns:**

The value of the timer prescaler.

### 11.2.1.16 ROM\_TimerPrescaleSet

Set the timer prescale value.

**Prototype:**

```
void  
ROM_TimerPrescaleSet(unsigned long ulBase,  
                     unsigned long ulTimer,  
                     unsigned long ulValue)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].

ROM\_TimerPrescaleSet is a function pointer located at ROM\_TIMERTABLE[10].

**Parameters:**

**ulBase** is the base address of the timer module.

**ulTimer** specifies the timer(s) to adjust; must be one of **TIMER\_A**, **TIMER\_B**, or **TIMER\_BOTH**.

**ulValue** 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.

**Returns:**

None.

### 11.2.1.17 ROM\_TimerRTCDisable

Disable RTC counting.

**Prototype:**

```
void  
ROM_TimerRTCDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.

ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].

ROM\_TimerRTCDisable is a function pointer located at ROM\_TIMERTABLE[9].

**Parameters:**

**ulBase** is the base address of the timer module.

**Description:**

This function causes the timer to stop counting when in RTC mode.

**Returns:**

None.

### 11.2.1.18 ROM\_TimerRTCEnable

Enable RTC counting.

**Prototype:**

```
void  
ROM_TimerRTCEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerRTCEnable is a function pointer located at ROM\_TIMERTABLE[8].

**Parameters:**

*ulBase* is the base address of the timer module.

**Description:**

This function causes the timer to start counting when in RTC mode. If not configured for RTC mode, this will do nothing.

**Returns:**

None.

### 11.2.1.19 ROM\_TimerValueGet

Gets the current timer value.

**Prototype:**

```
unsigned long  
ROM_TimerValueGet(unsigned long ulBase,  
                  unsigned long ulTimer)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_TIMERTABLE is an array of pointers located at ROM\_APITABLE[11].  
ROM\_TimerValueGet is a function pointer located at ROM\_TIMERTABLE[16].

**Parameters:**

*ulBase* is the base address of the timer module.

*ulTimer* specifies the timer; must be one of **TIMER\_A** or **TIMER\_B**. Only **TIMER\_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.



## 12 UART

Introduction .....	113
Functions .....	113

### 12.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Stellaris 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 Stellaris 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 Stellaris UART are:

- A 16x12 bit receive FIFO and a 16x8 bit transmit FIFO.
- 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
  - baud rate generation, from DC to processor clock/16
- IrDA serial-IR (SIR) encoder/decoder.
- DMA interface

### 12.2 Functions

#### Functions

- void [ROM\\_UARTBreakCtl](#) (unsigned long ulBase, tBoolean bBreakState)
- long [ROM\\_UARTCharGet](#) (unsigned long ulBase)
- long [ROM\\_UARTCharGetNonBlocking](#) (unsigned long ulBase)
- void [ROM\\_UARTCharPut](#) (unsigned long ulBase, unsigned char ucData)
- tBoolean [ROM\\_UARTCharPutNonBlocking](#) (unsigned long ulBase, unsigned char ucData)
- tBoolean [ROM\\_UARTCharsAvail](#) (unsigned long ulBase)
- void [ROM\\_UARTConfigGetExpClk](#) (unsigned long ulBase, unsigned long ulUARTClk, unsigned long \*pulBaud, unsigned long \*pulConfig)
- void [ROM\\_UARTConfigSetExpClk](#) (unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)
- void [ROM\\_UARTDisable](#) (unsigned long ulBase)
- void [ROM\\_UARTDisableSIR](#) (unsigned long ulBase)

- void [ROM\\_UARTEnable](#) (unsigned long ulBase)
- void [ROM\\_UARTEnableSIR](#) (unsigned long ulBase, tBoolean bLowPower)
- void [ROM\\_UARTFIFOLevelGet](#) (unsigned long ulBase, unsigned long \*pulTxLevel, unsigned long \*pulRxLevel)
- void [ROM\\_UARTFIFOLevelSet](#) (unsigned long ulBase, unsigned long ulTxLevel, unsigned long ulRxLevel)
- void [ROM\\_UARTIntClear](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_UARTIntDisable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- void [ROM\\_UARTIntEnable](#) (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long [ROM\\_UARTIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- unsigned long [ROM\\_UARTParityModeGet](#) (unsigned long ulBase)
- void [ROM\\_UARTParityModeSet](#) (unsigned long ulBase, unsigned long ulParity)
- tBoolean [ROM\\_UARTSpaceAvail](#) (unsigned long ulBase)
- void [ROM\\_UpdateUART](#) (void)

## 12.2.1 Function Documentation

### 12.2.1.1 ROM\_UARTBreakCtl

Causes a BREAK to be sent.

**Prototype:**

```
void  
ROM_UARTBreakCtl(unsigned long ulBase,  
                 tBoolean bBreakState)
```

**ROM Location:**

[ROM\\_APITABLE](#) is an array of pointers located at 0x0100.0010.  
[ROM\\_UARTTABLE](#) is an array of pointers located at [ROM\\_APITABLE](#)[1].  
[ROM\\_UARTBreakCtl](#) is a function pointer located at [ROM\\_UARTTABLE](#)[16].

**Parameters:**

***ulBase*** is the base address of the UART port.  
***bBreakState*** controls the output level.

**Description:**

Calling this function with *bBreakState* set to **true** will assert a break condition on the UART. Calling this function with *bBreakState* set to **false** will remove the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

**Returns:**

None.

### 12.2.1.2 ROM\_UARTCharGet

Waits for a character from the specified port.

**Prototype:**

```
long
ROM_UARTCharGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTCharGet is a function pointer located at ROM\_UARTTABLE[14].

**Parameters:**

**ulBase** is the base address of the UART port.

**Description:**

Gets a character from the receive FIFO for the specified port. If there are no characters available, this function will wait until a character is received before returning.

**Returns:**

Returns the character read from the specified port, cast as an *int*.

### 12.2.1.3 ROM\_UARTCharGetNonBlocking

Receives a character from the specified port.

**Prototype:**

```
long
ROM_UARTCharGetNonBlocking(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTCharGetNonBlocking is a function pointer located at ROM\_UARTTABLE[13].

**Parameters:**

**ulBase** is the base address of the UART port.

**Description:**

Gets a character from the receive FIFO for the specified port.

**Returns:**

Returns the character read from the specified port, cast as a *long*. A -1 will be returned if there are no characters present in the receive FIFO. The [ROM\\_UARTCharsAvail\(\)](#) function should be called before attempting to call this function.

### 12.2.1.4 ROM\_UARTCharPut

Waits to send a character from the specified port.

**Prototype:**

```
void
ROM_UARTCharPut(unsigned long ulBase,
                unsigned char ucData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTCharPut is a function pointer located at ROM\_UARTTABLE[0].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ucData** is the character to be transmitted.

**Description:**

Sends the character *ucData* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function will wait until there is space available before returning.

**Returns:**

None.

### 12.2.1.5 ROM\_UARTCharPutNonBlocking

Sends a character to the specified port.

**Prototype:**

```
tBoolean  
ROM_UARTCharPutNonBlocking(unsigned long ulBase,  
                             unsigned char ucData)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTCharPutNonBlocking is a function pointer located at ROM\_UARTTABLE[15].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ucData** is the character to be transmitted.

**Description:**

Writes the character *ucData* 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 will have to retry the function later.

**Returns:**

Returns **true** if the character was successfully placed in the transmit FIFO, and **false** if there was no space available in the transmit FIFO.

### 12.2.1.6 ROM\_UARTCharsAvail

Determines if there are any characters in the receive FIFO.

**Prototype:**

```
tBoolean  
ROM_UARTCharsAvail(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
 ROM\_UARTCharsAvail is a function pointer located at ROM\_UARTTABLE[11].

**Parameters:**

**ulBase** 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, and **false** if there is no data in the receive FIFO.

### 12.2.1.7 ROM\_UARTConfigGetExpClk

Gets the current configuration of a UART.

**Prototype:**

```
void
ROM_UARTConfigGetExpClk(unsigned long ulBase,
                        unsigned long ulUARTClk,
                        unsigned long *pulBaud,
                        unsigned long *pulConfig)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
 ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
 ROM\_UARTConfigGetExpClk is a function pointer located at ROM\_UARTTABLE[6].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ulUARTClk** is the rate of the clock supplied to the UART module.  
**pulBaud** is a pointer to storage for the baud rate.  
**pulConfig** 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 *pulConfig* is enumerated the same as the *ulConfig* parameter of [ROM\\_UARTConfigSetExpClk\(\)](#).

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

**Returns:**

None.

### 12.2.1.8 ROM\_UARTConfigSetExpClk

Sets the configuration of a UART.

**Prototype:**

```
void  
ROM_UARTConfigSetExpClk(unsigned long ulBase,  
                        unsigned long ulUARTClk,  
                        unsigned long ulBaud,  
                        unsigned long ulConfig)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTConfigSetExpClk is a function pointer located at ROM\_UARTTABLE[5].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ulUARTClk** is the rate of the clock supplied to the UART module.  
**ulBaud** is the desired baud rate.  
**ulConfig** is the data format for the port (number of data bits, number of stop bits, and parity).

**Description:**

This function will configure the UART for operation in the specified data format. The baud rate is provided in the *ulBaud* parameter and the data format in the *ulConfig* parameter.

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

**Returns:**

None.

### 12.2.1.9 ROM\_UARTDisable

Disables transmitting and receiving.

**Prototype:**

```
void  
ROM_UARTDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTDisable is a function pointer located at ROM\_UARTTABLE[8].

**Parameters:**

*ulBase* is the base address of the UART port.

**Description:**

Clears the UARTEN, TXE, and RXE bits, then waits for the end of transmission of the current character, and flushes the transmit FIFO.

**Returns:**

None.

### 12.2.1.10 ROM\_UARTDisableSIR

Disables SIR (IrDA) mode on the specified UART.

**Prototype:**

```
void  
ROM_UARTDisableSIR(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTDisableSIR is a function pointer located at ROM\_UARTTABLE[10].

**Parameters:**

*ulBase* is the base address of the UART port.

**Description:**

Clears the SIREN (IrDA) and SIRLP (Low Power) bits.

**Note:**

SIR (IrDA) operation is supported only on Fury-class devices.

**Returns:**

None.

### 12.2.1.11 ROM\_UARTEnable

Enables transmitting and receiving.

**Prototype:**

```
void  
ROM_UARTEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTEnable is a function pointer located at ROM\_UARTTABLE[7].

**Parameters:**

*ulBase* is the base address of the UART port.

**Description:**

Sets the UARTEN, TXE, and RXE bits, and enables the transmit and receive FIFOs.

**Returns:**

None.

### 12.2.1.12 ROM\_UARTEnableSIR

Enables SIR (IrDA) mode on specified UART.

**Prototype:**

```
void
ROM_UARTEnableSIR(unsigned long ulBase,
                  tBoolean bLowPower)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTEnableSIR is a function pointer located at ROM\_UARTTABLE[9].

**Parameters:**

**ulBase** is the base address of the UART port.  
**bLowPower** indicates if SIR Low Power Mode is to be used.

**Description:**

Enables the SIREN control bit for IrDA mode on the UART. If the *bLowPower* flag is set, then SIRLP bit will also be set.

**Note:**

SIR (IrDA) operation is supported only on Fury-class devices.

**Returns:**

None.

### 12.2.1.13 ROM\_UARTFIFOLevelGet

Gets the FIFO level at which interrupts are generated.

**Prototype:**

```
void
ROM_UARTFIFOLevelGet(unsigned long ulBase,
                    unsigned long *pulTxLevel,
                    unsigned long *pulRxLevel)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTFIFOLevelGet is a function pointer located at ROM\_UARTTABLE[4].

**Parameters:**

**ulBase** is the base address of the UART port.



***pulTxLevel*** 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**.

***pulRxLevel*** 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 will be generated.

**Returns:**

None.

### 12.2.1.14 ROM\_UARTFIFOLevelSet

Sets the FIFO level at which interrupts are generated.

**Prototype:**

```
void
ROM_UARTFIFOLevelSet (unsigned long ulBase,
                      unsigned long ulTxLevel,
                      unsigned long ulRxLevel)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.

**ROM\_UARTTABLE** is an array of pointers located at **ROM\_APITABLE**[1].

**ROM\_UARTFIFOLevelSet** is a function pointer located at **ROM\_UARTTABLE**[3].

**Parameters:**

***ulBase*** is the base address of the UART port.

***ulTxLevel*** 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**.

***ulRxLevel*** 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 will be generated.

**Returns:**

None.

### 12.2.1.15 ROM\_UARTIntClear

Clears UART interrupt sources.

**Prototype:**

```
void
ROM_UARTIntClear (unsigned long ulBase,
                  unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTIntClear is a function pointer located at ROM\_UARTTABLE[20].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ullntFlags** 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 must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ullntFlags* parameter has the same definition as the same parameter to [ROM\\_UARTIntEnable\(\)](#).

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**

None.

### 12.2.1.16 ROM\_UARTIntDisable

Disables individual UART interrupt sources.

**Prototype:**

```
void  
ROM_UARTIntDisable(unsigned long ulBase,  
                   unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTIntDisable is a function pointer located at ROM\_UARTTABLE[18].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ullntFlags** is the bit mask of the interrupt sources to be disabled.

**Description:**

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 *ullntFlags* parameter has the same definition as the same parameter to [ROM\\_UARTIntEnable\(\)](#).

**Returns:**

None.

### 12.2.1.17 ROM\_UARTIntEnable

Enables individual UART interrupt sources.

**Prototype:**

```
void  
ROM_UARTIntEnable(unsigned long ulBase,  
                  unsigned long ulIntFlags)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTIntEnable is a function pointer located at ROM\_UARTTABLE[17].

**Parameters:**

**ulBase** is the base address of the UART port.  
**ullntFlags** is the bit mask of the interrupt sources to be enabled.

**Description:**

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 *ullntFlags* parameter is the logical OR of any of the following:

- **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

**Returns:**

None.

### 12.2.1.18 ROM\_UARTIntStatus

Gets the current interrupt status.

**Prototype:**

```
unsigned long  
ROM_UARTIntStatus(unsigned long ulBase,  
                  tBoolean bMasked)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTIntStatus is a function pointer located at ROM\_UARTTABLE[19].

**Parameters:**

**ulBase** 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 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:**

The current interrupt status, enumerated as a bit field of values described in [ROM\\_UARTIntEnable\(\)](#).

### 12.2.1.19 ROM\_UARTParityModeGet

Gets the type of parity currently being used.

**Prototype:**

```
unsigned long  
ROM_UARTParityModeGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTParityModeGet is a function pointer located at ROM\_UARTTABLE[2].

**Parameters:**

***ulBase*** 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:**

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**.

### 12.2.1.20 ROM\_UARTParityModeSet

Sets the type of parity.

**Prototype:**

```
void  
ROM_UARTParityModeSet(unsigned long ulBase,  
                      unsigned long ulParity)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_UARTTABLE is an array of pointers located at ROM\_APITABLE[1].  
ROM\_UARTParityModeSet is a function pointer located at ROM\_UARTTABLE[1].

**Parameters:**

***ulBase*** is the base address of the UART port.

*ulParity* specifies the type of parity to use.

**Description:**

Sets the type of parity to use for transmitting and expect when receiving. The *ulParity* 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 will always be either be one or zero based on the mode.

**Returns:**

None.

### 12.2.1.21 ROM\_UARTSpaceAvail

Determines if there is any space in the transmit FIFO.

**Prototype:**

```
tBoolean  
ROM_UARTSpaceAvail(unsigned long ulBase)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.  
**ROM\_UARTTABLE** is an array of pointers located at ROM\_APITABLE[1].  
**ROM\_UARTSpaceAvail** is a function pointer located at ROM\_UARTTABLE[12].

**Parameters:**

*ulBase* 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, and **false** if there is no space available in the transmit FIFO.

### 12.2.1.22 ROM\_UpdateUART

Starts an update over the UART0 interface.

**Prototype:**

```
void  
ROM_UpdateUART(void)
```

**ROM Location:**

**ROM\_APITABLE** is an array of pointers located at 0x0100.0010.  
**ROM\_UARTTABLE** is an array of pointers located at ROM\_APITABLE[1].  
**ROM\_UpdateUART** is a function pointer located at ROM\_UARTTABLE[21].

**Description:**

Calling this function commences an update of the firmware via the UART0 interface. This function assumes that the UART0 interface has already been configured and is currently operational.

**Returns:**

Never returns.

# 13 Watchdog Timer

Introduction .....	127
Functions .....	127

## 13.1 Introduction

The watchdog timer API provides a set of functions for using the watchdog timer module. Functions are provided to deal with the watchdog timer interrupts, and to handle status and configuration of the watchdog timer.

The watchdog timer module's function is to prevent system hangs. The watchdog timer module consists of a 32-bit down counter, a programmable load register, interrupt generation logic, and a locking register. Once the watchdog timer has been configured, the lock register can be written to prevent the timer configuration from being inadvertently altered.

The watchdog timer can be configured to generate an interrupt to the processor upon its first timeout, and to generate a reset signal upon its second timeout. The watchdog timer module generates the first timeout signal when the 32-bit counter reaches the zero state after being enabled; enabling the counter also enables the watchdog timer interrupt. After the first timeout event, the 32-bit counter is reloaded with the value of the watchdog timer load register, and the timer resumes counting down from that value. If the timer counts down to its zero state again before the first timeout interrupt is cleared, and the reset signal has been enabled, the watchdog timer asserts its reset signal to the system. If the interrupt is cleared before the 32-bit counter reaches its second timeout, the 32-bit counter is loaded with the value in the load register, and counting resumes from that value. If the load register is written with a new value while the watchdog timer counter is counting, then the counter is loaded with the new value and continues counting.

## 13.2 Functions

### Functions

- void [ROM\\_WatchdogEnable](#) (unsigned long ulBase)
- void [ROM\\_WatchdogIntClear](#) (unsigned long ulBase)
- void [ROM\\_WatchdogIntEnable](#) (unsigned long ulBase)
- unsigned long [ROM\\_WatchdogIntStatus](#) (unsigned long ulBase, tBoolean bMasked)
- void [ROM\\_WatchdogLock](#) (unsigned long ulBase)
- tBoolean [ROM\\_WatchdogLockState](#) (unsigned long ulBase)
- unsigned long [ROM\\_WatchdogReloadGet](#) (unsigned long ulBase)
- void [ROM\\_WatchdogReloadSet](#) (unsigned long ulBase, unsigned long ulLoadVal)
- void [ROM\\_WatchdogResetDisable](#) (unsigned long ulBase)
- void [ROM\\_WatchdogResetEnable](#) (unsigned long ulBase)
- tBoolean [ROM\\_WatchdogRunning](#) (unsigned long ulBase)
- void [ROM\\_WatchdogStallEnable](#) (unsigned long ulBase)
- void [ROM\\_WatchdogUnlock](#) (unsigned long ulBase)
- unsigned long [ROM\\_WatchdogValueGet](#) (unsigned long ulBase)

## 13.2.1 Function Documentation

### 13.2.1.1 ROM\_WatchdogEnable

Enables the watchdog timer.

**Prototype:**

```
void  
ROM_WatchdogEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogEnable is a function pointer located at ROM\_WATCHDOGTABLE[2].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

This will enable the watchdog timer counter and interrupt.

**Note:**

This function will have no effect if the watchdog timer has been locked.

**See also:**

[ROM\\_WatchdogLock\(\)](#), [ROM\\_WatchdogUnlock\(\)](#)

**Returns:**

None.

### 13.2.1.2 ROM\_WatchdogIntClear

Clears the watchdog timer interrupt.

**Prototype:**

```
void  
ROM_WatchdogIntClear(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogIntClear is a function pointer located at ROM\_WATCHDOGTABLE[0].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

The watchdog timer interrupt source is cleared, so that it no longer asserts.

**Note:**

Since 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 (since NVIC still sees the interrupt source asserted).

**Returns:**  
None.

### 13.2.1.3 ROM\_WatchdogIntEnable

Enables the watchdog timer interrupt.

**Prototype:**  
void  
ROM\_WatchdogIntEnable(unsigned long ulBase)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogIntEnable is a function pointer located at ROM\_WATCHDOGTABLE[11].

**Parameters:**  
*ulBase* is the base address of the watchdog timer module.

**Description:**  
Enables the watchdog timer interrupt.

**Note:**  
This function will have no effect if the watchdog timer has been locked.

**See also:**  
[ROM\\_WatchdogLock\(\)](#), [ROM\\_WatchdogUnlock\(\)](#), [ROM\\_WatchdogEnable\(\)](#)

**Returns:**  
None.

### 13.2.1.4 ROM\_WatchdogIntStatus

Gets the current watchdog timer interrupt status.

**Prototype:**  
unsigned long  
ROM\_WatchdogIntStatus(unsigned long ulBase,  
tBoolean bMasked)

**ROM Location:**  
ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogIntStatus is a function pointer located at ROM\_WATCHDOGTABLE[12].

**Parameters:**  
*ulBase* is the base address of the watchdog timer module.

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

**Description:**

This returns the interrupt status for the watchdog timer module. Either the raw interrupt status or the status of interrupt that is allowed to reflect to the processor can be returned.

**Returns:**

The current interrupt status, where a 1 indicates that the watchdog interrupt is active, and a 0 indicates that it is not active.

### 13.2.1.5 ROM\_WatchdogLock

Enables the watchdog timer lock mechanism.

**Prototype:**

```
void  
ROM_WatchdogLock(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogLock is a function pointer located at ROM\_WATCHDOGTABLE[5].

**Parameters:**

***ulBase*** is the base address of the watchdog timer module.

**Description:**

Locks out write access to the watchdog timer configuration registers.

**Returns:**

None.

### 13.2.1.6 ROM\_WatchdogLockState

Gets the state of the watchdog timer lock mechanism.

**Prototype:**

```
tBoolean  
ROM_WatchdogLockState(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogLockState is a function pointer located at ROM\_WATCHDOGTABLE[7].

**Parameters:**

***ulBase*** is the base address of the watchdog timer module.

**Description:**

Returns the lock state of the watchdog timer registers.

**Returns:**

Returns **true** if the watchdog timer registers are locked, and **false** if they are not locked.

### 13.2.1.7 ROM\_WatchdogReloadGet

Gets the watchdog timer reload value.

**Prototype:**

```
unsigned long  
ROM_WatchdogReloadGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogReloadGet is a function pointer located at ROM\_WATCHDOGTABLE[9].

**Parameters:**

**ulBase** is the base address of the watchdog timer module.

**Description:**

This function gets the value that is loaded into the watchdog timer when the count reaches zero for the first time.

**See also:**

[ROM\\_WatchdogReloadSet\(\)](#)

**Returns:**

None.

### 13.2.1.8 ROM\_WatchdogReloadSet

Sets the watchdog timer reload value.

**Prototype:**

```
void  
ROM_WatchdogReloadSet(unsigned long ulBase,  
                      unsigned long ulLoadVal)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogReloadSet is a function pointer located at ROM\_WATCHDOGTABLE[8].

**Parameters:**

**ulBase** is the base address of the watchdog timer module.

**ulLoadVal** is the load value for the watchdog timer.

**Description:**

This function sets the value to load into the watchdog timer when the count reaches zero for the first time; if the watchdog timer is running when this function is called, then the value will be immediately loaded into the watchdog timer counter. If the *ulLoadVal* parameter is 0, then an interrupt is immediately generated.

**Note:**

This function will have no effect if the watchdog timer has been locked.

**See also:**

[ROM\\_WatchdogLock\(\)](#), [ROM\\_WatchdogUnlock\(\)](#), [ROM\\_WatchdogReloadGet\(\)](#)

**Returns:**

None.

### 13.2.1.9 ROM\_WatchdogResetDisable

Disables the watchdog timer reset.

**Prototype:**

```
void  
ROM_WatchdogResetDisable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogResetDisable is a function pointer located at ROM\_WATCHDOGTABLE[4].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

Disables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

**Note:**

This function will have no effect if the watchdog timer has been locked.

**See also:**

[ROM\\_WatchdogLock\(\)](#), [ROM\\_WatchdogUnlock\(\)](#)

**Returns:**

None.

### 13.2.1.10 ROM\_WatchdogResetEnable

Enables the watchdog timer reset.

**Prototype:**

```
void  
ROM_WatchdogResetEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogResetEnable is a function pointer located at ROM\_WATCHDOGTABLE[3].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

Enables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

**Note:**

This function will have no effect if the watchdog timer has been locked.

**See also:**

[ROM\\_WatchdogLock\(\)](#), [ROM\\_WatchdogUnlock\(\)](#)

**Returns:**

None.

### 13.2.1.11 ROM\_WatchdogRunning

Determines if the watchdog timer is enabled.

**Prototype:**

```
tBoolean  
ROM_WatchdogRunning(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogRunning is a function pointer located at ROM\_WATCHDOGTABLE[1].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

This will check to see if the watchdog timer is enabled.

**Returns:**

Returns **true** if the watchdog timer is enabled, and **false** if it is not.

### 13.2.1.12 ROM\_WatchdogStallEnable

Enables stalling of the watchdog timer during debug events.

**Prototype:**

```
void  
ROM_WatchdogStallEnable(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogStallEnable is a function pointer located at ROM\_WATCHDOGTABLE[13].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

This function allows the watchdog timer to stop counting when the processor is stopped by the debugger. By doing so, the watchdog is prevented from expiring (typically almost immediately from a human time perspective) and resetting the system (if reset is enabled). The watchdog will instead expired after the appropriate number of processor cycles have been executed while debugging (or at the appropriate time after the processor has been restarted).

**Returns:**

None.

### 13.2.1.13 ROM\_WatchdogUnlock

Disables the watchdog timer lock mechanism.

**Prototype:**

```
void  
ROM_WatchdogUnlock(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogUnlock is a function pointer located at ROM\_WATCHDOGTABLE[6].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

Enables write access to the watchdog timer configuration registers.

**Returns:**

None.

### 13.2.1.14 ROM\_WatchdogValueGet

Gets the current watchdog timer value.

**Prototype:**

```
unsigned long  
ROM_WatchdogValueGet(unsigned long ulBase)
```

**ROM Location:**

ROM\_APITABLE is an array of pointers located at 0x0100.0010.  
ROM\_WATCHDOGTABLE is an array of pointers located at ROM\_APITABLE[12].  
ROM\_WatchdogValueGet is a function pointer located at ROM\_WATCHDOGTABLE[10].

**Parameters:**

*ulBase* is the base address of the watchdog timer module.

**Description:**

This function reads the current value of the watchdog timer.

**Returns:**

Returns the current value of the watchdog timer.

---

# IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

## Products

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>
RF/IF and ZigBee® Solutions	<a href="http://www.ti.com/lprf">www.ti.com/lprf</a>

## Applications

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2008-2011, Texas Instruments Incorporated