LM4F232H5BB ROM

USER'S GUIDE



ROM-LM4F232H5BB-UG-730

Copyright © 2011-2013
Texas Instruments Incorporated

Copyright

Copyright © 2011-2013 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 730 of this document, last updated on January 4, 2013.

Table of Contents

Copyright **2**
Revision Information **2**
1 Introduction **5**
2 Boot Loader **7**
2.1 Introduction 7
2.2 Serial Interfaces 7
2.3 USB Interface 12
3 AES Data Tables **27**
3.1 Introduction 27
3.2 Data Structures 27
4 Analog Comparator **29**
4.1 Introduction 29
4.2 Functions 29
5 Analog to Digital Converter (ADC) **35**
5.1 Introduction 35
5.2 Functions 35
6 Controller Area Network (CAN) **53**
6.1 Introduction 53
6.2 Functions 54
7 CRC **69**
7.1 Introduction 69
7.2 Functions 69
8 Flash **73**
8.1 Introduction 73
8.2 Functions 73
9 Floating-Point Unit (FPU) **83**
9.1 Introduction 83
9.2 API Functions 84
10 GPIO **89**
10.1 Introduction 89
10.2 Functions 89
11 Hibernation Module **111**
11.1 Introduction 111
11.2 Functions 112
12 Inter-Integrated Circuit (I2C) **129**
12.1 Introduction 129
12.2 Functions 130
13 Interrupt Controller (NVIC) **151**
13.1 Introduction 151
13.2 Functions 151
14 Memory Protection Unit (MPU) **159**
14.1 Introduction 159
14.2 Functions 160
15 Pulse Width Modulator (PWM) **167**

15.1 Introduction . . . 167
15.2 Functions . . . 167
16 Quadrature Encoder (QEI) . . . 189
16.1 Introduction . . . 189
16.2 Functions . . . 189
17 Synchronous Serial Interface (SSI) . . . 199
17.1 Introduction . . . 199
17.2 Functions . . . 199
18 System Control . . . 211
18.1 Introduction . . . 211
18.2 Functions . . . 212
19 System Exception Module . . . 239
19.1 Introduction . . . 239
19.2 API Functions . . . 239
20 System Tick (SysTick) . . . 243
20.1 Introduction . . . 243
20.2 Functions . . . 243
21 Timer . . . 247
21.1 Introduction . . . 247
21.2 Functions . . . 247
22 UART . . . 265
22.1 Introduction . . . 265
22.2 Functions . . . 265
23 uDMA Controller . . . 287
23.1 Introduction . . . 287
23.2 Functions . . . 289
24 USB Controller . . . 313
24.1 Introduction . . . 313
24.2 Using USB with the uDMA Controller . . . 314
24.3 Functions . . . 318
25 Watchdog Timer . . . 359
25.1 Introduction . . . 359
25.2 Functions . . . 359
IMPORTANT NOTICE . . . 370

1 Introduction

The LM4F232H5BB 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:

ROM_APITABLE (at `0x0100.0010`)
[0] = ROM_VERSION
[1] = pointer to ROM_UARTTABLE
[2] = pointer to ROM_SSITABLE
[3] = pointer to ROM_I2CTABLE
[4] = pointer to ROM_GPIOTABLE
[5] = pointer to ROM_ADCTABLE
[6] = pointer to ROM_COMPARATORTABLE
[7] = pointer to ROM_FLASHTABLE
...

$\Longrightarrow$

ROM_GPIOTABLE
[0] = pointer to ROM_GPIOPinWrite
[1] = pointer to ROM_GPIODirModeSet
[2] = pointer to ROM_GPIODirModeGet
...

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_BLIZZARD_RA1
#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 25.

2 Boot Loader

Introduction 7
Serial Interfaces 7
USB 12

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
- USB using DFU protocol

The USB portion of the boot loader must run with the main PLL enabled which requires not only that the boot loader detect if a crystal is present, but also determine the frequency of the attached crystal to properly configure the main PLL. This crystal detection algorithm is only run when the boot loader does not find a valid application and is not run when the boot loader is called from an application. The device is configured to run from the PLL when there is a valid external crystal for USB operation as specified in the data sheet.

The LM Flash Programmer GUI can be used to download an application via the boot loader over the UART or USB interface on a PC. The LM Flash Programmer utility is available for download from `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. Because the system clock must be at least 32 times the baud rate, the maximum baud rate that can be used is 500 Kbaud (which is 16 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 1.3 MHz (which is 16 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 Serial Protocol

The boot loader uses well-defined packets on the serial interfaces 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`.

2.3 USB Interface

The USB boot loader allows programming the flash over USB using the standard USB DFU update method. To use the USB boot loader, the system running the boot loader must be capable of acting as a USB device and have the proper USB B or USB AB connector. The boot loader does not support firmware update of systems which operate solely as USB hosts. The USB update mechanism allows the boot loader to either be called from a functioning application or from startup when no firmware has been downloaded to the microcontroller. When the USB boot loader is called from an application, the application can customize some of the descriptors that are used for enumeration. This customization option is not available when the microcontroller is being programmed and there is no application present on the microcontroller.

When the USB boot loader is invoked from a running application, the boot loader reconfigures the USB controller to publish the required descriptor set for a Device Firmware Upgrade (DFU) class device and merge in any custom values that the application provides. Before calling the boot loader from an application, the main application is responsible for the following: enable the main PLL and configure it as the system clock, enable the USB PLL, enable the USB controller, and enable the USB D+ and D- pins on the microcontroller. If the main application is already acting as a USB device, then the application must remove the device from the bus by calling the DriverLib function `ROM_USBDevDisconnect()` prior to entering the boot loader.

2.3.1 USB Device Firmware Upgrade Overview

The USB boot loader enumerates as a Device Firmware Upgrade (DFU) class device. This standard device class specifies a set of class-specific requests and a state machine that can be used to download and flash firmware images to a device and, op-

tionally, upload the existing firmware image to the USB host. The full specification for the device class can be downloaded from the USB Implementer's Forum web site at http://www.usb.org/developers/devclass_docs#approved.

All communication with the DFU device takes place using the USB control endpoint, endpoint 0. The device publishes a standard device descriptor with vendor, product, device revisions and strings. It also publishes a single configuration descriptor and a single interface descriptor where the interface class of 0xFE indicates an application-specific class and the subclass of 0x01 indicates "Device Firmware Upgrade". Attached to the interface descriptor is a DFU Functional Descriptor which provides information to the host on DFU-specific device capabilities such as whether the device can perform upload operations and what the maximum transfer size for upload and download operations is.

DFU functions are initiated by means of a set of class-specific requests. Each request, which follows the standard USB request format, performs some operation and moves the DFU device between a series of well-defined states. Additional requests allow the host to query the current state of the device to determine whether, for example, it is ready to receive the next block of download data.

A DFU device may operation in either “Run Time” mode or “DFU” mode. In “Run Time” mode, the device publishes the DFU interface and functional descriptors along with any other descriptors that the device requires for normal operation. However, it is not required to respond to any DFU class-specific requests other than DFU_DETACH which indicates that it should switch to “DFU” mode. The “Run Time” mode support is not part of the USB boot loader but can be added to a USB device application to provide simple switching to “DFU” mode. If a main application wishes to publish DFU descriptors and respond to the DFU_DETACH request, it can cause a switch to “DFU” mode on receiving a DFU_DETACH request by removing itself from the USB bus using a call to ROM_USBDevDisconnect() before transferring control to the USB boot loader by making a call to ROM_UpdateUSB(). Once in “DFU” mode in the boot loader, the device supports all DFU functionality and can perform upload and download operations as specified in its DFU functional descriptor.

2.3.2 DFU Requests

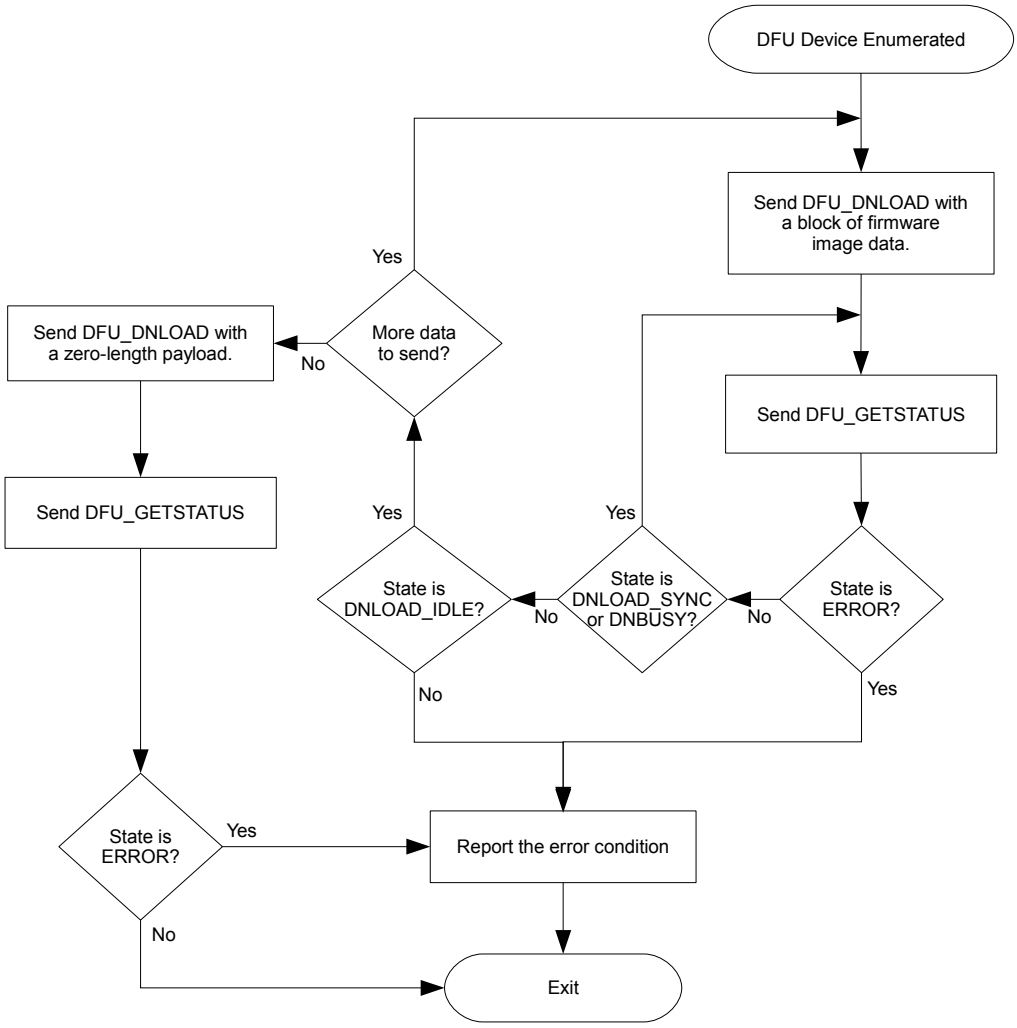
Requests supported by the USB boot loader are as follow:

DFU_DNLOAD — This OUT request is used to send a block of binary data to the device. The DFU class specification does not define the content and format of the binary data but typically it is either binary data to be written to some position in the device's flash memory or a device-specific command. The request payload size is constrained by the maximum packet size specified in the DFU functional descriptor. In this implementation, that maximum is set to 1024 bytes.
After sending a DFU_DNLOAD request, the host must poll the device status and wait until the state reverts to DNLOAD_IDLE before sending another request. If the host wishes to indicate that it has finished sending download data, it sends a DFU_DNLOAD request with a payload length of 0.

DFU_UPLOAD	This IN request is used to request a block of binary data from the device. The data returned is device-specific but is typically the contents of a region of the device's flash memory or a device-specific response to a command previously sent via a DFU_DNLOAD request. As with DFU_DNLOAD, the maximum amount of data that can be requested is governed by the maximum packet size specified in the DFU functional descriptor, here 1024 bytes.
DFU_GETSTATUS	This IN request allows the host to query the current status of the DFU device. It is typically used during download operations to determine when it is safe to send the next block of data. Depending upon the state of the DFU device, this request may also trigger a state change. During download, for example, the device enters DNLOAD_SYNC state after receiving a block of data and remains there until the data has been processed and a DFU_GETSTATUS request is received at which point the state changes to DNLOAD_IDLE.
DFU_CLRSTATUS	This request is used to reset any error condition reported by the DFU device. If an error is reported via the response to a DFU_GETSTATUS request, that error condition is cleared when this request is received and the device returns to IDLE state.
DFU_GETSTATE	This IN request is used to query the current state of the device without triggering any state change. The single byte of data returned indicates the current state of the DFU device.
DFU_ABORT	This request cancels any partially complete upload or download operation and returns the device to IDLE state in preparation for some other request.

2.3.3 Typical Firmware Download Sequence

The following flow chart illustrates a typical firmware image download sequence from the perspective of the host application.


DFU Device Enumerated
Send DFU_DNLOAD with a block of firmware image data.
Send DFU_GETSTATUS
State is ERROR?
Yes
State is DNLOAD_SYNC or DNBUSY?
Yes
No
State is DNLOAD_IDLE?
Yes
No
More data to send?
Yes
No
Send DFU_DNLOAD with a zero-length payload.
Send DFU_GETSTATUS
State is ERROR?
Yes
No
Report the error condition
Exit

2.3.4 Stellaris-Specific USB Download Commands

The DFU class specification provides the framework necessary to download and upload firmware files to the USB device but does not specify the actual format of the binary data that is transferred. As a result, different device implementations have used different methods to perform operations that are not defined in the standard such as:

- Setting the address to flash a downloaded binary.
- Setting the address and size of the area of flash with contents to be uploaded.
- Erasing sections of the flash.
- Querying the size of flash and writeable area addresses.

The USB boot loader implementation employs a small set of commands which can be sent to the DFU device via a `DFU_DNLOAD` request when the device is in IDLE state. Each command takes the form of an 8-byte structure that defines the operation to carry out and provides any required additional parameters.

To ensure that a host application that does not have explicit support for Stellaris-specific commands can still be used to download binary firmware images to the device, the protocol is defined such that only a single 8-byte header structure must be placed at the start of the binary image being downloaded. This header and the DFU-defined suffix structure can both be added using the “dfuwrap” command-line application supplied with StellarisWare, thus providing a single binary that can be sent to a device running the Stellaris USB boot loader using a standard sequence of `DFU_DNLOAD` requests with no other embedded commands or device-specific operations required. An application which does understand the Stellaris-specific commands may make use of them to offer additional functionality that would not otherwise be available.

Querying Command Support

Because the device-specific commands defined here are sent to the DFU device in `DFU_DNLOAD` requests, the possibility exists that sending them to a device which does not understand the protocol could result in corruption of that device’s firmware. To guard against this possibility, the Stellaris USB boot loader supports an additional USB request which is used to query the device capabilities and allow a host to determine whether or not the device supports the Stellaris commands. A device which does not support the commands either stalls the request or returns unexpected data.

To determine whether a target DFU device supports the Stellaris-specific DFU commands, send the following IN request to the DFU interface:

bmRequest-Type	bRequest	wValue	wIndex	wLength	Data
10100001b	0x42	0x23	Interface	4	Protocol Info

where the protocol information returned is a 4-byte structure, the first two bytes of which are 0x4D, 0x4C and the second two bytes indicates the protocol version supported, currently 0x01 and 0x00 respectively.

Download Command Definitions

The following commands may be sent to the USB boot loader as the first 8 bytes of the payload to a `DFU_DNLOAD` request. The boot loader expects any `DFU_DNLOAD` request received while in the

IDLE state to contain a command header but does not look for a command unless the state is IDLE. This protocol allows an application that is unaware of the command header to download a DFU-wrapped binary image using a standard sequence of multiple DFU_DNLOAD and DFU_GETSTATUS requests without the need to insert additional command headers during the download.

The commands defined here and their parameter block structures can be found in header file usbdfu.h provided with StellarisWare. In all cases where multi-byte numbers are specified, the numbers are stored in little-endian format with the least significant byte in the lowest addressed location. The following definitions specify the command byte ordering unambiguously but care must be taken to ensure correct byte swapping if using the command structure types defined in usbdfu.h on big-endian systems.

DFU_CMD_PROG

This command is used to provide the USB boot loader with the address at which the next download should be written and the total length of the firmware image that follows. This structure forms the header that is written to the DFU-wrapped file generated by the dfuwrap tool.
The start address is provided in terms of 1024-byte flash blocks. To convert a byte address to a block address, merely divide by 1024. The start address must always be on a 1024-byte boundary.
This command may be followed by up to 1016 bytes of firmware image data, this number being the maximum transfer size minus the 8 bytes of the command structure.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_PROG (0x01)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Image Size [7:0];
ucData[5] = Image Size [15:8];
ucData[6] = Image Size [23:16];
ucData[7] = Image Size [31:24];
```

DFU_CMD_READ

This command is used to set the address range of the content to be returned on subsequent DFU_UPLOAD requests from the host.
The start address is provided in terms of 1024-byte flash blocks. To convert a byte address to a block address, divide by 1024. The start address must always be on a 1024-byte boundary.
To read back a the contents of a region of flash, the host should send a DFU_DNLOAD request with command DFU_CMD_READ, start address set to the 1KB block start address and length set to the number of bytes to read. The host should then send one or more DFU_UPLOAD requests to receive the current flash contents from the configured addresses. Data returned includes an 8-byte DFU_CMD_PROG prefix structure unless the prefix has been disabled by sending a DFU_CMD_BIN command with the bBinary parameter set to 1. The host should, therefore, be prepared to read 8 bytes more than the length specified in the READ command if the prefix is enabled.
By default, the 8-byte prefix is enabled for all upload operations. This prefix is required by the DFU class specification which states that uploaded images must be formatted to allow them to be directly downloaded back to the device at a later time.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_READ (0x02)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Image Size [7:0];
ucData[5] = Image Size [15:8];
ucData[6] = Image Size [23:16];
ucData[7] = Image Size [31:24];
```

DFU_CMD_CHECK

This command is used to check a region of flash to ensure that it is completely erased.
The start address is provided in terms of 1024-byte flash blocks. To convert a byte address to a block address, divide by 1024. The start address must always be on a 1024-byte boundary. The length must also be a multiple of 4.
To check that a region of flash is erased, the DFU_CMD_CHECK command should be sent with the required start address and region length set then the host should issue a DFU_GETSTATUS request. If the erase check was successful, the returned bStatus value is OK (0x00), otherwise it is errCheckErased (0x05).

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_CHECK (0x03)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Region Size [7:0];
ucData[5] = Region Size [15:8];
ucData[6] = Region Size [23:16];
ucData[7] = Region Size [31:24];
```

DFU_CMD_ERASE

This command is used to erase a region of flash.
The start address is provided in terms of 1024-byte flash blocks. To convert a byte address to a block address, divide by 1024. The start address must always be on a 1024-byte boundary. The length must also be a multiple of 4.
The size of the region to erase is expressed in terms of flash blocks. The block size can be determined using the DFU_CMD_INFO command.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_ERASE (0x04)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Number of Blocks [7:0];
ucData[5] = Number of Blocks [15:8];
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
```

DFU_CMD_INFO

This command is used to query information relating to the target device and programmable region of flash. The device information structure, tDFUDeviceInfo, is returned on the next DFU_UPLOAD request following this command.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_INFO (0x05)
ucData[1] = Reserved - set to 0x00
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00

//*****************************************************************
//
// Payload returned in response to the DFU_CMD_INFO command.
//
// This is structure is returned in response to the first DFU_UPLOAD
// request following a DFU_CMD_INFO command.  Note that byte ordering
// of multi-byte fields is little-endian.
//
//*****************************************************************
typedef struct
{
    //
    // The size of a flash block in bytes.
    //
    unsigned short usFlashBlockSize;

    //
    // The number of blocks of flash in the  device.  Total
    // flash size is usNumFlashBlocks * usFlashBlockSize.
    //
    unsigned short usNumFlashBlocks;

    //
    // Information on the part number, family, version and
    // package as read from SYSCTL register DID1.
    //
    unsigned long ulPartInfo;

    //
    // Information on the part class and revision as read
    // from SYSCTL DID0.
    //
    unsigned long ulClassInfo;

    //
    // Address 1 byte above the highest location the boot
    // loader can access.
    //
    unsigned long ulFlashTop;

    //
    // Lowest address the boot loader can write or erase.
    //
    unsigned long ulAppStartAddr;
}
PACKED tDFUDeviceInfo;
```

DFU_CMD_BIN

By default, data returned in response to a DFU_UPLOAD request includes an 8-byte DFU_CMD_PROG prefix structure. This ensures that an uploaded image can be directly downloaded again without the need to further wrap it but, in cases where pure binary data is required, can be awkward. The DFU_CMD_BIN command allows the upload prefix to be disabled or enabled under host control.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_BIN (0x06)
ucData[1] = 0x01 to disable upload prefix, 0x00 to enable
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
```

DFU_CMD_RESET

This command may be sent to the USB boot loader to cause it to perform a soft reset of the board. This reboots the system and, assuming that the main application image is present, runs the main application. Note that a reboot also takes place if a firmware download operation completes and the host issues a USB reset to the DFU device.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_RESET (0x07)
ucData[1] = Reserved - set to 0x00
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
```

2.3.5 Customizing the USB Device Firmware Upgrade

The USB boot loader provides a method to pass a set of custom values to the boot loader when called from a running application. When the USB boot loader is entered because the flash was empty, the boot ROM offers no customization and provides generic Texas Instruments and Stellaris strings to the host as identifiers. The USB boot loader is called by an application via a call to ROM_UpdateUSB(), which takes a single pointer as an argument that is a pointer to an array containing the values that the boot loader uses for customization. This pointer can also be 0 to indicate that the application wishes to use the defaults provided by the ROM boot loader. The application can place the customization structure anywhere in flash or SRAM.

The actual values that can be customized are the following:

- Vendor ID (default 0x1CBE)
- Product ID (default 0x00FF)
- Bus/Self Power (default Self Powered)
- Max Power (default 0)
- Device Version (default 0x110)
- All strings in multiple languages

The defaults for the strings are a Manufacturer set to "Texas Instruments", Product set to "Device Firmware Update", and serial number set to "1.0".

If the application is providing a structure to the boot loader with custom values, then the first 12 bytes must contain valid data. The values in the structure that is passed to the boot loader have the following meaning and are merged directly to the device and configuration descriptor for the boot loader:

Byte	Value
0	VID bits 7:0
1	VID bits 15:8
2	PID bits 7:9
3	PID bits 15:8
4	Device Version(BCD) bits 7:9
5	Device Version(BCD) bits 15:8
6	Power in 2mA Increments(250 maximum)
7	USB Configuration Descriptor bmAttributes (Self powered 0xC0, Bus powered 0x80).
8-11	Pointer to the string table

The values for the custom strings are contained in a pointer at the end of the structure passed to the boot loader. If the application wants to use the default strings, this pointer should be set to 0. The string table provided supports custom strings for Manufacturer, Product, and Serial Number in multiple languages. The string table is optimized to not require that a string be provided for every language in case there is not a unique string for every language. This is most likely the case for the Serial Number string but it could be any string that is provided.

The string table has a header that describes the number of languages provided and the language identifiers for each followed by a set of strings for each language. Each entry in the table has a size header that specifies the size of the string in bytes plus a byte for the size itself and a one byte tag of `USB_DTYPE_STRING`(0x03). This size and ID padding is included for ease of string handling in the boot loader and is required. It is very important for the application to provide the correct size values because the boot loader uses these to skip through the array and find the requested strings requested by the USB host. If these sizes are incorrect, the boot loader may fail to enumerate, so care should be taken when building the string table. The application must provide all three strings in the first language because these strings are treated as the default if a string is left blank in other languages. In other languages, a string can be left blank for strings that do not change per language. A blank entry in the table consists of a two byte entry with a byte size value of 2 followed by the `USB_DTYPE_STRING` value.

The string table has the following format:

Byte	Description
0	Size of the header(Number of Languages * 2) + 2

1	USB_DTYPE_STRING
2	Language ID 0 bits[7:0]
3	Language ID 0 bits[15:8]
...	...
.	Language ID n bits[7:0]
.	Language ID n bits[15:8]
.	Language 0 Manufacturer string size in bytes + 2
.	USB_DTYPE_STRING
.	Language 0 Manufacturer Unicode String
.	Language 0 Product string size in bytes + 2
.	USB_DTYPE_STRING
.	Language 0 Product Unicode String
.	Language 0 Serial Number size in bytes + 2
.	USB_DTYPE_STRING
.	Language 0 Serial Number Unicode String
...	...
.	Language n Manufacturer string size in bytes + 2
.	USB_DTYPE_STRING
.	Language n Manufacturer Unicode String
.	Language n Product string size in bytes + 2
.	USB_DTYPE_STRING
.	Language n Product Unicode String
.	Language n Serial Number size in bytes + 2
.	USB_DTYPE_STRING
.	Language n Serial Number Unicode String

The next three examples show an example customization structure for a no string case, a single language case and a multiple language case.

Example: No strings but custom descriptor.

```
unsigned char pucUSBBootROMInfo[]=
{
    0xbe, 0x1c,    // Stellaris VID
    0xff, 0x00,    // Stellaris DFU PID
    0x00, 0x02,    // USB version 2.0
    0x00,          // 0mA of Bus power
    0xC0,          // Self powered using no bus power
    0              // Address of the string table
}

//
// Call to ROM USB boot loader.
//
ROM_UpdateUSB(pucUSBBootROMInfo);
```

Example: Single US English string set.

```
unsigned char pucStrings[]=
```

```
{
    (1 * 2) + 2,    // One Language (1 * 2) + 2
    USB_DTYPE_STRING,
    0x09, 0x04,     // Language code for US English.

    (17 * 2) + 2,   // Size of Manufacturer String.
                    // "Texas Instruments"
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0,
    's', 0, 't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0,
    's', 0,

    (23 * 2) + 2,   // Size of Product String.
    USB_DTYPE_STRING,
                    // "Device Firmware Upgrade"
    'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0, 'e', 0, ' ', 0, 'F', 0,
    'i', 0, 'r', 0, 'm', 0, 'w', 0, 'a', 0, 'r', 0, 'e', 0, ' ', 0,
    'U', 0, 'p', 0, 'g', 0, 'r', 0, 'a', 0, 'd', 0, 'e', 0,

    (3 * 2) + 2,    // Size of Serial Number.
    USB_DTYPE_STRING,
                    // "1.0"
    '1', 0, '.', 0, '0', 0
};

unsigned char pucUSBBootROMInfo[]=
{
    0xbe, 0x1c,     // Stellaris VID
    0xff, 0x00,     // Stellaris DFU PID
    0x00, 0x02,     // USB version 2.0
    0x00,           // 0mA of Bus power
    0xC0,           // Self powered using no bus power
    pucStrings      // Address of the string table
}

//
// Call to ROM USB boot loader.
//
ROM_UpdateUSB(pucUSBBootROMInfo);
```

Example: Two Languages with English(United States) and Spanish(Mexico) strings.

```
unsigned char pucStrings[]=
{
    (2 * 2) + 2,    // Two languages
    USB_DTYPE_STRING,
    0x09, 0x04,     // Language code for English(United States).
    0x0a, 0x08,     // Language code for Spanish(Mexico)

    (10 * 2) + 2,   // Size of Manufacturer String.
                    // "My Company"
    USB_DTYPE_STRING,
```

```
    'M', 0, 'y', 0, ' ', 0, 'C', 0, 'o', 0, 'm', 0, 'p', 0, 'a', 0,
    'n', 0, 'y', 0

    (23 * 2) + 2,  // Size of Product String.
    USB_DTYPE_STRING,
                   // "Device Firmware Upgrade"
    'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0, 'e', 0, ' ', 0, 'F', 0,
    'i', 0, 'r', 0, 'm', 0, 'w', 0, 'a', 0, 'r', 0, 'e', 0, ' ', 0,
    'U', 0, 'p', 0, 'g', 0, 'r', 0, 'a', 0, 'd', 0, 'e', 0,

    (4 * 2) + 2,   // Size of Serial Number.
    USB_DTYPE_STRING,
                   // "1.01"
    '1', 0, '.', 0, '0', 0, '1', 0

    (10 * 2) + 2,  // Size of Manufacturer String.
    USB_DTYPE_STRING,
                   // "My Company"
    'M', 0, 'y', 0, ' ', 0, 'C', 0, 'o', 0, 'm', 0, 'p', 0, 'a', 0,
    'n', 0, 'y', 0
    's', 0,

    (25 * 2) + 2,  // Size of Product String.
    USB_DTYPE_STRING,
                   // "Actualizacion de Firmware"
    'A', 0, 'c', 0, 't', 0, 'u', 0, 'a', 0, 'l', 0, 'i', 0, 'z', 0,
    'a', 0, 'c', 0, 'i', 0, 'o', 0, 'n', 0, ' ', 0, 'd', 0, 'e', 0,
    ' ', 0, 'F', 0, 'i', 0, 'r', 0, 'm', 0, 'w', 0, 'a', 0, 'r', 0,
    'e', 0,

    2,             // Size of Serial Number, this will use the last
                   // serial number found since this represents a null
                   // string.
    USB_DTYPE_STRING,
};

unsigned char pucUSBBootROMInfo[]=
{
    0xbe, 0x1c,    // Stellaris VID
    0xff, 0x00,    // Stellaris DFU PID
    250,           // 500mA of Bus power
    0x80,          // Bus Powered
    0x00, 0x02,    // USB version 2.0
    pucStrings     // Address of the string table
};

//
// Call to ROM USB boot loader.
//
ROM_UpdateUSB(pucUSBBootROMInfo);
```

3 AES Data Tables

Introduction 27
Functions 27

3.1 Introduction

The Advanced Encryption Standard (AES) is a publicly defined encryption standard used by the U.S. Government. It is a strong encryption method with reasonable performance and size. AES is fast in both hardware and software, is fairly easy to implement, and requires little memory. AES is ideal for applications that can use pre-arranged keys, such as setup during manufacturing or configuration.

Four data tables used by the XySSL AES implementation are provided in the ROM. The first is the forward S-box substitution table, the second is the reverse S-box substitution table, the third is the forward polynomial table, and the final is the reverse polynomial table. The meanings of these tables and their use can be found in the AES code provided in StellarisWare.

3.2 Data Structures

Data Structures

- ROM_pvAESTable

3.2.1 Data Structure Documentation

3.2.1.1 ROM_pvAESTable

This structure describes the AES tables that are available in the ROM.

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SOFTWARETABLE` is an array of pointers located at `ROM_APITABLE[21]`.
`ROM_pvAESTable` is an array located at `&ROM_SOFTWARETABLE[7]`.

Definition:

```
typedef struct
{
    unsigned char ucForwardSBox[256];
    unsigned long ulForwardTable[256];
    unsigned char ucReverseSBox[256];
    unsigned long ulReverseTable[256];
}
ROM_pvAESTable
```

Members:

ucForwardSBox This table contains the forward S-Box, as defined by the AES standard.

ulForwardTable This table contains the forward polynomial table, as used by the XySSL AES implementation.

ucReverseSBox This table contains the reverse S-Box, as defined by the AES standard. This is simply the reverse of *ucForwardSBox*.

ulReverseTable This table contains the reverse polynomial table, as used by the XySSL AES implementation.

4 Analog Comparator

Introduction 29
Functions 29

4.1 Introduction

The comparator API provides a set of functions for dealing with the analog comparators. A comparator can compare a test voltage against individual external reference voltage, a shared single external reference voltage, or a shared internal reference voltage. It can provide its output to a device pin, acting as a replacement for an analog comparator on the board, or it can be used to signal the application via interrupts or triggers to the ADC to cause it to start capturing a sample sequence. The interrupt generation and ADC triggering logic is separate, so that an interrupt can be generated on a rising edge and the ADC triggered on a falling edge (for example).

4.2 Functions

Functions

- void ROM_ComparatorConfigure (unsigned long ulBase, unsigned long ulComp, unsigned long ulConfig)
- void ROM_ComparatorIntClear (unsigned long ulBase, unsigned long ulComp)
- void ROM_ComparatorIntDisable (unsigned long ulBase, unsigned long ulComp)
- void ROM_ComparatorIntEnable (unsigned long ulBase, unsigned long ulComp)
- tBoolean ROM_ComparatorIntStatus (unsigned long ulBase, unsigned long ulComp, tBoolean bMasked)
- void ROM_ComparatorRefSet (unsigned long ulBase, unsigned long ulRef)
- tBoolean ROM_ComparatorValueGet (unsigned long ulBase, unsigned long ulComp)

4.2.1 Function Documentation

4.2.1.1 ROM_ComparatorConfigure

Configures a comparator.

Prototype:

```
void
ROM_ComparatorConfigure(unsigned long ulBase,
                        unsigned long ulComp,
                        unsigned long ulConfig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_COMPARATORTABLE` is an array of pointers located at `ROM_APITABLE[6]`.
`ROM_ComparatorConfigure` is a function pointer located at `ROM_COMPARATORTABLE[1]`.

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator to configure.

ulConfig is the configuration of the comparator.

Description:

This function configures a comparator. The *ulConfig* parameter is the result of a logical OR operation between the **COMP_TRIG_xxx**, **COMP_INT_xxx**, **COMP_ASRCP_xxx**, and **COMP_OUTPUT_xxx** values.

The **COMP_TRIG_xxx** term can take on the following values:

- **COMP_TRIG_NONE** to have no trigger to the ADC.
- **COMP_TRIG_HIGH** to trigger the ADC when the comparator output is high.
- **COMP_TRIG_LOW** to trigger the ADC when the comparator output is low.
- **COMP_TRIG_FALL** to trigger the ADC when the comparator output goes low.
- **COMP_TRIG_RISE** to trigger the ADC when the comparator output goes high.
- **COMP_TRIG_BOTH** to trigger the ADC when the comparator output goes low or high.

The **COMP_INT_xxx** term can take on the following values:

- **COMP_INT_HIGH** to generate an interrupt when the comparator output is high.
- **COMP_INT_LOW** to generate an interrupt when the comparator output is low.
- **COMP_INT_FALL** to generate an interrupt when the comparator output goes low.
- **COMP_INT_RISE** to generate an interrupt when the comparator output goes high.
- **COMP_INT_BOTH** to generate an interrupt when the comparator output goes low or high.

The **COMP_ASRCP_xxx** term can take on the following values:

- **COMP_ASRCP_PIN** to use the dedicated Comp+ pin as the reference voltage.
- **COMP_ASRCP_PIN0** to use the Comp0+ pin as the reference voltage (this the same as **COMP_ASRCP_PIN** for the comparator 0).
- **COMP_ASRCP_REF** to use the internally generated voltage as the reference voltage.

The **COMP_OUTPUT_xxx** term can take on the following values:

- **COMP_OUTPUT_NORMAL** to enable a non-inverted output from the comparator to a device pin.
- **COMP_OUTPUT_INVERT** to enable an inverted output from the comparator to a device pin.

Returns:

None.

4.2.1.2 ROM_ComparatorIntClear

Clears a comparator interrupt.

Prototype:

```
void
ROM_ComparatorIntClear(unsigned long ulBase,
                       unsigned long ulComp)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_COMPARATORTABLE is an array of pointers located at ROM_APITABLE[6].
ROM_ComparatorIntClear is a function pointer located at ROM_COMPARATORTABLE[0].

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

The comparator interrupt is cleared, so that it no longer asserts. This function must be called in the interrupt handler to keep the handler from being called again immediately upon exit. Note that for a level-triggered interrupt, the interrupt cannot be cleared until it stops asserting.

Note:

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

Returns:

None.

4.2.1.3 ROM_ComparatorIntDisable

Disables the comparator interrupt.

Prototype:

```
void
ROM_ComparatorIntDisable(unsigned long ulBase,
                         unsigned long ulComp)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_COMPARATORTABLE is an array of pointers located at ROM_APITABLE[6].
ROM_ComparatorIntDisable is a function pointer located at ROM_COMPARATORTABLE[5].

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

This function disables generation of an interrupt from the specified comparator. Only comparators whose interrupts are enabled can be reflected to the processor.

Returns:

None.

4.2.1.4 ROM_ComparatorIntEnable

Enables the comparator interrupt.

Prototype:

```
void
ROM_ComparatorIntEnable(unsigned long ulBase,
                        unsigned long ulComp)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_COMPARATORTABLE` is an array of pointers located at `ROM_APITABLE[6]`.
`ROM_ComparatorIntEnable` is a function pointer located at `ROM_COMPARATORTABLE[4]`.

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

This function enables generation of an interrupt from the specified comparator. Only comparators whose interrupts are enabled can be reflected to the processor.

Returns:

None.

4.2.1.5 ROM_ComparatorIntStatus

Gets the current interrupt status.

Prototype:

```
tBoolean
ROM_ComparatorIntStatus(unsigned long ulBase,
                        unsigned long ulComp,
                        tBoolean bMasked)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_COMPARATORTABLE` is an array of pointers located at `ROM_APITABLE[6]`.
`ROM_ComparatorIntStatus` is a function pointer located at `ROM_COMPARATORTABLE[6]`.

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

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 comparator. Either the raw or the masked interrupt status can be returned.

Returns:

true if the interrupt is asserted and **false** if it is not asserted.

4.2.1.6 ROM_ComparatorRefSet

Sets the internal reference voltage.

Prototype:

```
void
ROM_ComparatorRefSet(unsigned long ulBase,
                     unsigned long ulRef)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_COMPARATORTABLE` is an array of pointers located at `ROM_APITABLE[6]`.
`ROM_ComparatorRefSet` is a function pointer located at `ROM_COMPARATORTABLE[2]`.

Parameters:

ulBase is the base address of the comparator module.

ulRef is the desired reference voltage.

Description:

This function sets the internal reference voltage value. The voltage is specified as one of the following values:

- **COMP_REF_OFF** to turn off the reference voltage
- **COMP_REF_0V** to set the reference voltage to 0 V
- **COMP_REF_0_1375V** to set the reference voltage to 0.1375 V
- **COMP_REF_0_275V** to set the reference voltage to 0.275 V
- **COMP_REF_0_4125V** to set the reference voltage to 0.4125 V
- **COMP_REF_0_55V** to set the reference voltage to 0.55 V
- **COMP_REF_0_6875V** to set the reference voltage to 0.6875 V
- **COMP_REF_0_825V** to set the reference voltage to 0.825 V
- **COMP_REF_0_928125V** to set the reference voltage to 0.928125 V
- **COMP_REF_0_9625V** to set the reference voltage to 0.9625 V
- **COMP_REF_1_03125V** to set the reference voltage to 1.03125 V
- **COMP_REF_1_134375V** to set the reference voltage to 1.134375 V
- **COMP_REF_1_1V** to set the reference voltage to 1.1 V
- **COMP_REF_1_2375V** to set the reference voltage to 1.2375 V
- **COMP_REF_1_340625V** to set the reference voltage to 1.340625 V
- **COMP_REF_1_375V** to set the reference voltage to 1.375 V
- **COMP_REF_1_44375V** to set the reference voltage to 1.44375 V
- **COMP_REF_1_5125V** to set the reference voltage to 1.5125 V
- **COMP_REF_1_546875V** to set the reference voltage to 1.546875 V
- **COMP_REF_1_65V** to set the reference voltage to 1.65 V
- **COMP_REF_1_753125V** to set the reference voltage to 1.753125 V
- **COMP_REF_1_7875V** to set the reference voltage to 1.7875 V
- **COMP_REF_1_85625V** to set the reference voltage to 1.85625 V
- **COMP_REF_1_925V** to set the reference voltage to 1.925 V
- **COMP_REF_1_959375V** to set the reference voltage to 1.959375 V
- **COMP_REF_2_0625V** to set the reference voltage to 2.0625 V
- **COMP_REF_2_165625V** to set the reference voltage to 2.165625 V

- **COMP_REF_2_26875V** to set the reference voltage to 2.26875 V
- **COMP_REF_2_371875V** to set the reference voltage to 2.371875 V

Returns:
None.

4.2.1.7 ROM_ComparatorValueGet

Gets the current comparator output value.

Prototype:
```
tBoolean
ROM_ComparatorValueGet(unsigned long ulBase,
                       unsigned long ulComp)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_COMPARATORTABLE` is an array of pointers located at `ROM_APITABLE[6]`.
`ROM_ComparatorValueGet` is a function pointer located at `ROM_COMPARATORTABLE[3]`.

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

Description:
This function retrieves the current value of the comparator output.

Returns:
Returns **true** if the comparator output is high and **false** if the comparator output is low.

5 Analog to Digital Converter (ADC)

Introduction 35
Functions 35

5.1 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 twenty-four 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.

5.2 Functions

Functions

- void ROM_ADCComparatorConfigure (unsigned long ulBase, unsigned long ulComp, unsigned long ulConfig)
- void ROM_ADCComparatorIntClear (unsigned long ulBase, unsigned long ulStatus)
- void ROM_ADCComparatorIntDisable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ROM_ADCComparatorIntEnable (unsigned long ulBase, unsigned long ulSequenceNum)
- unsigned long ROM_ADCComparatorIntStatus (unsigned long ulBase)
- void ROM_ADCComparatorRegionSet (unsigned long ulBase, unsigned long ulComp, unsigned long ulLowRef, unsigned long ulHighRef)
- void ROM_ADCComparatorReset (unsigned long ulBase, unsigned long ulComp, tBoolean bTrigger, tBoolean bInterrupt)
- 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)
- unsigned long ROM_ADCPhaseDelayGet (unsigned long ulBase)
- void ROM_ADCPhaseDelaySet (unsigned long ulBase, unsigned long ulPhase)
- void ROM_ADCProcessorTrigger (unsigned long ulBase, unsigned long ulSequenceNum)
- unsigned long ROM_ADCReferenceGet (unsigned long ulBase)
- void ROM_ADCReferenceSet (unsigned long ulBase, unsigned long ulRef)
- 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)

5.2.1 Function Documentation

5.2.1.1 ROM_ADCComparatorConfigure

Configures an ADC digital comparator.

Prototype:

```
void
ROM_ADCComparatorConfigure(unsigned long ulBase,
                           unsigned long ulComp,
                           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_ADCComparatorConfigure` is a function pointer located at `ROM_ADCTABLE[15]`.

Parameters:

ulBase is the base address of the ADC module.

ulComp is the index of the comparator to configure.

ulConfig is the configuration of the comparator.

Description:

This function will configure a comparator. The *ulConfig* parameter is the result of a logical OR operation between the **ADC_COMP_TRIG_xxx**, and **ADC_COMP_INT_xxx** values.

The **ADC_COMP_TRIG_xxx** term can take on the following values:

- **ADC_COMP_TRIG_NONE** to never trigger PWM fault condition.
- **ADC_COMP_TRIG_LOW_ALWAYS** to always trigger PWM fault condition when ADC output is in the low-band.
- **ADC_COMP_TRIG_LOW_ONCE** to trigger PWM fault condition once when ADC output transitions into the low-band.
- **ADC_COMP_TRIG_LOW_HALWAYS** to always trigger PWM fault condition when ADC output is in the low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_TRIG_LOW_HONCE** to trigger PWM fault condition once when ADC output transitions into low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_TRIG_MID_ALWAYS** to always trigger PWM fault condition when ADC output is in the mid-band.
- **ADC_COMP_TRIG_MID_ONCE** to trigger PWM fault condition once when ADC output transitions into the mid-band.
- **ADC_COMP_TRIG_HIGH_ALWAYS** to always trigger PWM fault condition when ADC output is in the high-band.
- **ADC_COMP_TRIG_HIGH_ONCE** to trigger PWM fault condition once when ADC output transitions into the high-band.
- **ADC_COMP_TRIG_HIGH_HALWAYS** to always trigger PWM fault condition when ADC output is in the high-band only if ADC output has been in the low-band since the last trigger output.
- **ADC_COMP_TRIG_HIGH_HONCE** to trigger PWM fault condition once when ADC output transitions into high-band only if ADC output has been in the low-band since the last trigger output.

The **ADC_COMP_INT_xxx** term can take on the following values:

- **ADC_COMP_INT_NONE** to never generate ADC interrupt.
- **ADC_COMP_INT_LOW_ALWAYS** to always generate ADC interrupt when ADC output is in the low-band.
- **ADC_COMP_INT_LOW_ONCE** to generate ADC interrupt once when ADC output transitions into the low-band.
- **ADC_COMP__INT_LOW_HALWAYS** to always generate ADC interrupt when ADC output is in the low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_INT_LOW_HONCE** to generate ADC interrupt once when ADC output transitions into low-band only if ADC output has been in the high-band since the last trigger output.
- **ADC_COMP_INT_MID_ALWAYS** to always generate ADC interrupt when ADC output is in the mid-band.
- **ADC_COMP_INT_MID_ONCE** to generate ADC interrupt once when ADC output transitions into the mid-band.
- **ADC_COMP_INT_HIGH_ALWAYS** to always generate ADC interrupt when ADC output is in the high-band.
- **ADC_COMP_INT_HIGH_ONCE** to generate ADC interrupt once when ADC output transitions into the high-band.
- **ADC_COMP_INT_HIGH_HALWAYS** to always generate ADC interrupt when ADC output is in the high-band only if ADC output has been in the low-band since the last trigger output.
- **ADC_COMP_INT_HIGH_HONCE** to generate ADC interrupt once when ADC output transitions into high-band only if ADC output has been in the low-band since the last trigger output.

Returns:
None.

5.2.1.2 ROM_ADCComparatorIntClear

Clears sample sequence comparator interrupt source.

Prototype:
```
void
ROM_ADCComparatorIntClear(unsigned long ulBase,
                          unsigned long ulStatus)
```

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_ADCComparatorIntClear` is a function pointer located at `ROM_ADCTABLE[21]`.

Parameters:
ulBase is the base address of the ADC module.
ulStatus is the bit-mapped interrupts status to clear.

Description:
The specified interrupt status is cleared.

Returns:
None.

5.2.1.3 ROM_ADCComparatorIntDisable

Disables a sample sequence comparator interrupt.

Prototype:
```
void
ROM_ADCComparatorIntDisable(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_ADCComparatorIntDisable` is a function pointer located at `ROM_ADCTABLE[18]`.

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

Description:
This function disables the requested sample sequence comparator interrupt.

Returns:
None.

5.2.1.4 ROM_ADCComparatorIntEnable

Enables a sample sequence comparator interrupt.

Prototype:

```
void
ROM_ADCComparatorIntEnable(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_ADCComparatorIntEnable` is a function pointer located at `ROM_ADCTABLE[19]`.

Parameters:

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

Description:

This function enables the requested sample sequence comparator interrupt.

Returns:

None.

5.2.1.5 ROM_ADCComparatorIntStatus

Gets the current comparator interrupt status.

Prototype:

```
unsigned long
ROM_ADCComparatorIntStatus(unsigned long ulBase)
```

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_ADCComparatorIntStatus` is a function pointer located at `ROM_ADCTABLE[20]`.

Parameters:

ulBase is the base address of the ADC module.

Description:

This returns the digitial comparator interrupt status bits. This status is sequence agnostic.

Returns:

The current comparator interrupt status.

5.2.1.6 ROM_ADCComparatorRegionSet

Defines the ADC digital comparator regions.

Prototype:

```
void
ROM_ADCComparatorRegionSet(unsigned long ulBase,
                           unsigned long ulComp,
                           unsigned long ulLowRef,
                           unsigned long ulHighRef)
```

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_ADCComparatorRegionSet` is a function pointer located at `ROM_ADCTABLE[16]`.

Parameters:

ulBase is the base address of the ADC module.

ulComp is the index of the comparator to configure.

ulLowRef is the reference point for the low/mid band threshold.

ulHighRef is the reference point for the mid/high band threshold.

Description:

The ADC digital comparator operation is based on three ADC value regions:

- **low-band** is defined as any ADC value less than or equal to the *ulLowRef* value.
- **mid-band** is defined as any ADC value greater than the *ulLowRef* value but less than or equal to the *ulHighRef* value.
- **high-band** is defined as any ADC value greater than the *ulHighRef* value.

Returns:

None.

5.2.1.7 ROM_ADCComparatorReset

Resets the current ADC digital comparator conditions.

Prototype:

```
void
ROM_ADCComparatorReset(unsigned long ulBase,
                       unsigned long ulComp,
                       tBoolean bTrigger,
                       tBoolean bInterrupt)
```

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_ADCComparatorReset` is a function pointer located at `ROM_ADCTABLE[17]`.

Parameters:

ulBase is the base address of the ADC module.

ulComp is the index of the comparator.

bTrigger is the flag to indicate reset of Trigger conditions.

bInterrupt is the flag to indicate reset of Interrupt conditions.

Description:

Because the digital comparator uses current and previous ADC values, this function is provide to allow the comparator to be reset to its initial value to prevent stale data from being used when a sequence is enabled.

Returns:

None.

5.2.1.8 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 hardware oversampling.

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.

Note:

Hardware oversampling is available beginning with Rev C0 of the Stellaris microcontroller.

Returns:

None.

5.2.1.9 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:

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

Returns:

None.

5.2.1.10 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.

5.2.1.11 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.

5.2.1.12 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.

5.2.1.13 ROM_ADCPhaseDelayGet

Gets the phase delay between a trigger and the start of a sequence.

Prototype:

```
unsigned long
ROM_ADCPhaseDelayGet(unsigned long ulBase)
```

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_ADCPhaseDelayGet` is a function pointer located at `ROM_ADCTABLE[25]`.

Parameters:

ulBase is the base address of the ADC module.

Description:

This function gets the current phase delay between the detection of an ADC trigger event and the start of the sample sequence.

Returns:

Returns the phase delay, specified as one of **ADC_PHASE_0**, **ADC_PHASE_22_5**, **ADC_PHASE_45**, **ADC_PHASE_67_5**, **ADC_PHASE_90**, **ADC_PHASE_112_5**, **ADC_PHASE_135**, **ADC_PHASE_157_5**, **ADC_PHASE_180**, **ADC_PHASE_202_5**, **ADC_PHASE_225**, **ADC_PHASE_247_5**, **ADC_PHASE_270**, **ADC_PHASE_292_5**, **ADC_PHASE_315**, or **ADC_PHASE_337_5**.

5.2.1.14 ROM_ADCPhaseDelaySet

Sets the phase delay between a trigger and the start of a sequence.

Prototype:

```
void
ROM_ADCPhaseDelaySet(unsigned long ulBase,
                     unsigned long ulPhase)
```

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_ADCPhaseDelaySet` is a function pointer located at `ROM_ADCTABLE[24]`.

Parameters:

ulBase is the base address of the ADC module.

ulPhase is the phase delay, specified as one of **ADC_PHASE_0**, **ADC_PHASE_22_5**, **ADC_PHASE_45**, **ADC_PHASE_67_5**, **ADC_PHASE_90**, **ADC_PHASE_112_5**, **ADC_PHASE_135**, **ADC_PHASE_157_5**, **ADC_PHASE_180**, **ADC_PHASE_202_5**, **ADC_PHASE_225**, **ADC_PHASE_247_5**, **ADC_PHASE_270**, **ADC_PHASE_292_5**, **ADC_PHASE_315**, or **ADC_PHASE_337_5**.

Description:

This function sets the phase delay between the detection of an ADC trigger event and the start of the sample sequence. By selecting a different phase delay for a pair of ADC modules (such

as **ADC_PHASE_0** and **ADC_PHASE_180**) and having each ADC module sample the same analog input, it is possible to increase the sampling rate of the analog input (with samples N, N+2, N+4, and so on, coming from the first ADC and samples N+1, N+3, N+5, and so on, coming from the second ADC). The ADC module has a single phase delay that is applied to all sample sequences within that module.

Returns:
None.

5.2.1.15 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, with **ADC_TRIGGER_WAIT** or **ADC_TRIGGER_SIGNAL** optionally ORed into it.

Description:
This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to **ADC_TRIGGER_PROCESSOR**. If **ADC_TRIGGER_WAIT** is ORed into the sequence number, the processor-initiated trigger is delayed until a later processor-initiated trigger to a different ADC module that specifies **ADC_TRIGGER_SIGNAL**, allowing multiple ADCs to start from a processor-initiated trigger in a synchronous manner.

Returns:
None.

5.2.1.16 ROM_ADCReferenceGet

Returns the current setting of the ADC reference.

Prototype:
```
unsigned long
ROM_ADCReferenceGet(unsigned long ulBase)
```

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_ADCReferenceGet` is a function pointer located at `ROM_ADCTABLE[23]`.

Parameters:

ulBase is the base address of the ADC module.

Description:

Returns the value of the ADC reference setting. The returned value is one of **ADC_REF_INT** or **ADC_REF_EXT_3V**.

Returns:

The current setting of the ADC reference.

5.2.1.17 ROM_ADCReferenceSet

Selects the ADC reference.

Prototype:

```
void
ROM_ADCReferenceSet(unsigned long ulBase,
                    unsigned long ulRef)
```

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_ADCReferenceSet` is a function pointer located at `ROM_ADCTABLE[22]`.

Parameters:

ulBase is the base address of the ADC module.

ulRef is the reference to use.

Description:

The ADC reference is set as specified by *ulRef*. It must be one of **ADC_REF_INT** or **ADC_REF_EXT_3V**, for internal or external reference. If **ADC_REF_INT** is chosen, then an internal 3V reference is used and no external reference is needed. If **ADC_REF_EXT_3V** is chosen, then a 3V reference must be supplied to the AVREF pin.

Returns:

None.

5.2.1.18 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_COMP0** - A trigger generated by the first analog comparator; configured with ROM_ComparatorConfigure().
- **ADC_TRIGGER_COMP1** - A trigger generated by the second analog comparator; configured with ROM_ComparatorConfigure().
- **ADC_TRIGGER_COMP2** - A trigger generated by the third analog comparator; configured with ROM_ComparatorConfigure().
- **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_PWM0** - A trigger generated by the first PWM generator; configured with ROM_PWMGenIntTrigEnable().
- **ADC_TRIGGER_PWM1** - A trigger generated by the second PWM generator; configured with ROM_PWMGenIntTrigEnable().
- **ADC_TRIGGER_PWM2** - A trigger generated by the third PWM generator; configured with ROM_PWMGenIntTrigEnable().
- **ADC_TRIGGER_PWM3** - A trigger generated by the fourth PWM generator; configured with ROM_PWMGenIntTrigEnable().
- **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).

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.

5.2.1.19 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.

5.2.1.20 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.

5.2.1.21 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.

5.2.1.22 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.

5.2.1.23 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.

5.2.1.24 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**, one of the input channel selects (**ADC_CTL_CH0** through **ADC_CTL_CH23**), and one of the digital comparator selects (**ADC_CTL_CMP0** through **ADC_CTL_CMP7**).

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_CH23** 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). If the digital comparators are present on the device, this step may also be configured to send the ADC sample to the selected comparator using **ADC_CTL_CMP0** through **ADC_CTL_CMP7**. The configuration is used by the ADC at the appropriate time when the trigger for this sequence occurs.

Note:

If the Digitial Comparator is present and enabled using the **ADC_CTL_CMP0** through **ADC_CTL_CMP7** selects, the ADC sample will NOT be written into the ADC sequence data FIFO.

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 (for example, 0 and 1). The channel select must be the number of the channel pair to sample (for example, **ADC_CTL_CH0** for 0 and 1, or **ADC_CTL_CH1** for 2 and 3) or undefined results are returned by the ADC. Additionally, if differential mode is selected when the temperature sensor is being sampled, undefined results are 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.

5.2.1.25 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.

5.2.1.26 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.

6 Controller Area Network (CAN)

Introduction 53
Functions 54

6.1 Introduction

The Controller Area Network (CAN) APIs provide a set of functions for accessing the Stellaris CAN modules. Functions are provided to configure the CAN controllers, configure message objects, and manage CAN interrupts.

The Stellaris CAN module provides hardware processing of the CAN data link layer. It can be configured with message filters and preloaded message data so that it can autonomously send and receive messages on the bus, and notify the application accordingly. It automatically handles generation and checking of CRCs, error processing, and retransmission of CAN messages.

The message objects are stored in the CAN controller and provide the main interface for the CAN module on the CAN bus. There are 32 message objects that can each be programmed to handle a separate message ID, or can be chained together for a sequence of frames with the same ID. The message identifier filters provide masking that can be programmed to match any or all of the message ID bits, and frame types.

The CAN module is disabled by default, so the the ROM_CANInit() function must be called before any other CAN functions are called. This call initializes the message objects to a safe state prior to enabling the controller on the CAN bus. Also, the bit timing values must be programmed prior to enabling the CAN controller. The ROM_CANBitTimingSet() function should be called with the appropriate bit timing values for the CAN bus. Once these two functions have been called, a CAN controller can be enabled using the ROM_CANEnable(), and later disabled using ROM_CANDisable() if needed. Calling ROM_CANDisable() does not reinitialize a CAN controller, so it can be used to temporarily remove a CAN controller from the bus.

The CAN controller is highly configurable and contains 32 message objects that can be programmed to automatically transmit and receive CAN messages under certain conditions. Message objects allow the application to perform some actions automatically without interaction from the microcontroller. Some examples of these actions are the following:

- Send a data frame immediately
- Send a data frame when a matching remote frame is seen on the CAN bus
- Receive a specific data frame
- Receive data frames that match a certain identifier pattern

To configure message objects to perform any of these actions, the application must first set up one of the 32 message objects using ROM_CANMessageSet(). This function must be used to configure a message object to send data, or to configure a message object to receive data. Each message object can be configured to generate interrupts on transmission or reception of CAN messages.

When data is received from the CAN bus, the application can use the ROM_CANMessageGet() function to read the received message. This function can also be used to read a message object that is already configured in order to populate a message structure prior to making changes to the

configuration of a message object. Reading the message object using this function will also clear any pending interrupt on the message object.

Once a message object has been configured using ROM_CANMessageSet(), it has allocated the message object and will continue to perform its programmed function unless it is released with a call to ROM_CANMessageClear(). The application is not required to clear out a message object before setting it with a new configuration, because each time ROM_CANMessageSet() is called, it will overwrite any previously programmed configuration.

The 32 message objects are identical except for priority. The lowest numbered message objects have the highest priority. Priority affects operation in two ways. First, if multiple actions are ready at the same time, the one with the highest priority message object will occur first. And second, when multiple message objects have interrupts pending, the highest priority will be presented first when reading the interrupt status. It is up to the application to manage the 32 message objects as a resource, and determine the best method for allocating and releasing them.

The CAN controller can generate interrupts on several conditions:

- When any message object transmits a message
- When any message object receives a message
- On warning conditions such as an error counter reaching a limit or occurrence of various bus errors
- On controller error conditions such as entering the bus-off state

Once CAN interrupts are enabled, the handler will be invoked whenever a CAN interrupt is triggered. The handler can determine which condition caused the interrupt by using the ROM_CANIntStatus() function. Multiple conditions can be pending when an interrupt occurs, so the handler must be designed to process all pending interrupt conditions before exiting. Each interrupt condition must be cleared before exiting the handler. There are two ways to do this. The ROM_CANIntClear() function will clear a specific interrupt condition without further action required by the handler. However, the handler can also clear the condition by performing certain actions. If the interrupt is a status interrupt, the interrupt can be cleared by reading the status register with ROM_CANStatusGet(). If the interrupt is caused by one of the message objects, then it can be cleared by reading the message object using ROM_CANMessageGet().

There are several status registers that can be used to help the application manage the controller. The status registers are read using the ROM_CANStatusGet() function. There is a controller status register that provides general status information such as error or warning conditions. There are also several status registers that provide information about all of the message objects at once using a 32-bit bit map of the status, with one bit representing each message object. These status registers can be used to determine:

- Which message objects have unprocessed received data
- Which message objects have pending transmission requests
- Which message objects are allocated for use

6.2 Functions

Functions

- unsigned long ROM_CANBitRateSet (unsigned long ulBase, unsigned long ulSourceClock, unsigned long ulBitRate)

- void ROM_CANBitTimingGet (unsigned long ulBase, tCANBitClkParms *pClkParms)
- void ROM_CANBitTimingSet (unsigned long ulBase, tCANBitClkParms *pClkParms)
- void ROM_CANDisable (unsigned long ulBase)
- void ROM_CANEnable (unsigned long ulBase)
- tBoolean ROM_CANErrCntrGet (unsigned long ulBase, unsigned long *pulRxCount, unsigned long *pulTxCount)
- void ROM_CANInit (unsigned long ulBase)
- void ROM_CANIntClear (unsigned long ulBase, unsigned long ulIntClr)
- void ROM_CANIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_CANIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long ROM_CANIntStatus (unsigned long ulBase, tCANIntStsReg eIntStsReg)
- void ROM_CANMessageClear (unsigned long ulBase, unsigned long ulObjID)
- void ROM_CANMessageGet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject *pMsgObject, tBoolean bClrPendingInt)
- void ROM_CANMessageSet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject *pMsgObject, tMsgObjType eMsgType)
- tBoolean ROM_CANRetryGet (unsigned long ulBase)
- void ROM_CANRetrySet (unsigned long ulBase, tBoolean bAutoRetry)
- unsigned long ROM_CANStatusGet (unsigned long ulBase, tCANStsReg eStatusReg)

6.2.1 Function Documentation

6.2.1.1 ROM_CANBitRateSet

This function is used to set the CAN bit timing values to a nominal setting based on a desired bit rate.

Prototype:

```
unsigned long
ROM_CANBitRateSet(unsigned long ulBase,
                  unsigned long ulSourceClock,
                  unsigned long ulBitRate)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANBitRateSet` is a function pointer located at `ROM_CANTABLE[16]`.

Parameters:

ulBase is the base address of the CAN controller.

ulSourceClock is the system clock for the device in Hz.

ulBitRate is the desired bit rate.

Description:

This function will set the CAN bit timing for the bit rate passed in the *ulBitRate* parameter based on the *ulSourceClock* parameter. Since the CAN clock is based off of the system clock the calling function should pass in the source clock rate either by retrieving it from ROM_SysCtlClockGet() or using a specific value in Hz. The CAN bit timing is calculated assuming a minimal amount of propagation delay, which will work for most cases where the

network length is short. If tighter timing requirements or longer network lengths are needed, then the ROM_CANBitTimingSet() function is available for full customization of all of the CAN bit timing values. Since not all bit rates can be matched exactly, the bit rate is set to the value closest to the desired bit rate without being higher than the *ulBitRate* value.

Returns:

This function returns the bit rate that the CAN controller was configured to use or it returns 0 to indicate that the bit rate was not changed because the requested bit rate was not valid.

6.2.1.2 ROM_CANBitTimingGet

Reads the current settings for the CAN controller bit timing.

Prototype:

```
void
ROM_CANBitTimingGet(unsigned long ulBase,
                    tCANBitClkParms *pClkParms)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANBitTimingGet` is a function pointer located at `ROM_CANTABLE[5]`.

Parameters:

ulBase is the base address of the CAN controller.

pClkParms is a pointer to a structure to hold the timing parameters.

Description:

This function reads the current configuration of the CAN controller bit clock timing, and stores the resulting information in the structure supplied by the caller. Refer to ROM_CANBitTimingSet() for the meaning of the values that are returned in the structure pointed to by *pClkParms*.

Returns:

None.

6.2.1.3 ROM_CANBitTimingSet

Configures the CAN controller bit timing.

Prototype:

```
void
ROM_CANBitTimingSet(unsigned long ulBase,
                    tCANBitClkParms *pClkParms)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANBitTimingSet` is a function pointer located at `ROM_CANTABLE[4]`.

Parameters:

ulBase is the base address of the CAN controller.

pClkParms points to the structure with the clock parameters.

Description:

Configures the various timing parameters for the CAN bus bit timing: Propagation segment, Phase Buffer 1 segment, Phase Buffer 2 segment, and the Synchronization Jump Width. The values for Propagation and Phase Buffer 1 segments are derived from the combination *pClkParms->uSyncPropPhase1Seg* parameter. Phase Buffer 2 is determined from the *pClkParms->uPhase2Seg* parameter. These two parameters, along with *pClkParms->uSJW* are based in units of bit time quanta. The actual quantum time is determined by the *pClkParms->uQuantumPrescaler* value, which specifies the divisor for the CAN module clock.

The total bit time, in quanta, is the sum of the two Seg parameters, as follows:

bit_time_q = uSyncPropPhase1Seg + uPhase2Seg + 1

Note that the Sync_Seg is always one quantum in duration, and is added to derive the correct duration of Prop_Seg and Phase1_Seg.

The equation to determine the actual bit rate is as follows:

CAN Clock / ((*uSyncPropPhase1Seg* + *uPhase2Seg* + 1) $*$ (*uQuantumPrescaler*))

This means that with *uSyncPropPhase1Seg* = 4, *uPhase2Seg* = 1, *uQuantumPrescaler* = 2 and an 8 MHz CAN clock, that the bit rate is (8 MHz) / ((5 + 2 + 1) $*$ 2) or 500 Kbit/sec.

Returns:

None.

6.2.1.4 ROM_CANDisable

Disables the CAN controller.

Prototype:

```
void
ROM_CANDisable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANDisable` is a function pointer located at `ROM_CANTABLE[3]`.

Parameters:

ulBase is the base address of the CAN controller to disable.

Description:

Disables the CAN controller for message processing. When disabled, the controller will no longer automatically process data on the CAN bus. The controller can be restarted by calling ROM_CANEnable(). The state of the CAN controller and the message objects in the controller are left as they were before this call was made.

Returns:

None.

6.2.1.5 ROM_CANEnable

Enables the CAN controller.

Prototype:

```
void
ROM_CANEnable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANEnable` is a function pointer located at `ROM_CANTABLE[2]`.

Parameters:

ulBase is the base address of the CAN controller to enable.

Description:

Enables the CAN controller for message processing. Once enabled, the controller will automatically transmit any pending frames, and process any received frames. The controller can be stopped by calling ROM_CANDisable(). Prior to calling ROM_CANEnable(), ROM_CANInit() should have been called to initialize the controller and the CAN bus clock should be configured by calling ROM_CANBitTimingSet().

Returns:

None.

6.2.1.6 ROM_CANErrCntrGet

Reads the CAN controller error counter register.

Prototype:

```
tBoolean
ROM_CANErrCntrGet(unsigned long ulBase,
                  unsigned long *pulRxCount,
                  unsigned long *pulTxCount)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANErrCntrGet` is a function pointer located at `ROM_CANTABLE[15]`.

Parameters:

ulBase is the base address of the CAN controller.
pulRxCount is a pointer to storage for the receive error counter.
pulTxCount is a pointer to storage for the transmit error counter.

Description:

Reads the error counter register and returns the transmit and receive error counts to the caller along with a flag indicating if the controller receive counter has reached the error passive limit. The values of the receive and transmit error counters are returned through the pointers provided as parameters.

After this call, $*$*pulRxCount* will hold the current receive error count and $*$*pulTxCount* will hold the current transmit error count.

Returns:

Returns **true** if the receive error count has reached the error passive limit, and **false** if the error count is below the error passive limit.

6.2.1.7 ROM_CANInit

Initializes the CAN controller after reset.

Prototype:

```
void
ROM_CANInit(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANInit` is a function pointer located at `ROM_CANTABLE[1]`.

Parameters:

ulBase is the base address of the CAN controller.

Description:

After reset, the CAN controller is left in the disabled state. However, the memory used for message objects contains undefined values and must be cleared prior to enabling the CAN controller the first time. This prevents unwanted transmission or reception of data before the message objects are configured. This function must be called before enabling the controller the first time.

Returns:

None.

6.2.1.8 ROM_CANIntClear

Clears a CAN interrupt source.

Prototype:

```
void
ROM_CANIntClear(unsigned long ulBase,
                unsigned long ulIntClr)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANIntClear` is a function pointer located at `ROM_CANTABLE[0]`.

Parameters:

ulBase is the base address of the CAN controller.

ulIntClr is a value indicating which interrupt source to clear.

Description:

This function can be used to clear a specific interrupt source. The *ulIntClr* parameter should be one of the following values:

- **CAN_INT_INTID_STATUS** - Clears a status interrupt.
- 1-32 - Clears the specified message object interrupt

It is not necessary to use this function to clear an interrupt. This should only be used if the application wants to clear an interrupt source without taking the normal interrupt action.

Normally, the status interrupt is cleared by reading the controller status using ROM_CANStatusGet(). A specific message object interrupt is normally cleared by reading the message object using ROM_CANMessageGet().

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

Returns:
None.

6.2.1.9 ROM_CANIntDisable

Disables individual CAN controller interrupt sources.

Prototype:
```
void
ROM_CANIntDisable(unsigned long ulBase,
                  unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANIntDisable` is a function pointer located at `ROM_CANTABLE[11]`.

Parameters:
ulBase is the base address of the CAN controller.
ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:
Disables the specified CAN controller interrupt sources. Only enabled interrupt sources can cause a processor interrupt.

The *ulIntFlags* parameter has the same definition as in the ROM_CANIntEnable() function.

Returns:
None.

6.2.1.10 ROM_CANIntEnable

Enables individual CAN controller interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANIntEnable` is a function pointer located at `ROM_CANTABLE[10]`.

Parameters:

ulBase is the base address of the CAN controller.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Enables specific interrupt sources of the CAN controller. Only enabled sources will cause a processor interrupt.

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

- **CAN_INT_ERROR** - a controller error condition has occurred
- **CAN_INT_STATUS** - a message transfer has completed, or a bus error has been detected
- **CAN_INT_MASTER** - allow CAN controller to generate interrupts

In order to generate any interrupts, **CAN_INT_MASTER** must be enabled. Further, for any particular transaction from a message object to generate an interrupt, that message object must have interrupts enabled (see ROM_CANMessageSet()). **CAN_INT_ERROR** will generate an interrupt if the controller enters the “bus off” condition, or if the error counters reach a limit. **CAN_INT_STATUS** will generate an interrupt under quite a few status conditions and may provide more interrupts than the application needs to handle. When an interrupt occurs, use ROM_CANIntStatus() to determine the cause.

Returns:

None.

6.2.1.11 ROM_CANIntStatus

Returns the current CAN controller interrupt status.

Prototype:

```
unsigned long
ROM_CANIntStatus(unsigned long ulBase,
                 tCANIntStsReg eIntStsReg)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANIntStatus` is a function pointer located at `ROM_CANTABLE[12]`.

Parameters:

ulBase is the base address of the CAN controller.

eIntStsReg indicates which interrupt status register to read

Description:

Returns the value of one of two interrupt status registers. The interrupt status register read is determined by the *eIntStsReg* parameter, which can have one of the following values:

- **CAN_INT_STS_CAUSE** - indicates the cause of the interrupt
- **CAN_INT_STS_OBJECT** - indicates pending interrupts of all message objects

CAN_INT_STS_CAUSE returns the value of the controller interrupt register and indicates the cause of the interrupt. It is a value of **CAN_INT_INTID_STATUS** if the cause is a status interrupt. In this case, the status register should be read with the ROM_CANStatusGet() function. Calling this function to read the status will also clear the status interrupt. If the value of the interrupt register is in the range 1-32, then this indicates the number of the highest priority message object that has an interrupt pending. The message object interrupt can be cleared by using the ROM_CANIntClear() function, or by reading the message using ROM_CANMessageGet() in the case of a received message. The interrupt handler can read the interrupt status again to make sure all pending interrupts are cleared before returning from the interrupt.

CAN_INT_STS_OBJECT returns a bit mask indicating which message objects have pending interrupts. This can be used to discover all of the pending interrupts at once, as opposed to repeatedly reading the interrupt register by using **CAN_INT_STS_CAUSE**.

Returns:

Returns the value of one of the interrupt status registers.

6.2.1.12 ROM_CANMessageClear

Clears a message object so that it is no longer used.

Prototype:

```
void
ROM_CANMessageClear(unsigned long ulBase,
                    unsigned long ulObjID)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANMessageClear` is a function pointer located at `ROM_CANTABLE[9]`.

Parameters:

ulBase is the base address of the CAN controller.
ulObjID is the message object number to disable (1-32).

Description:

This function frees the specified message object from use. Once a message object has been "cleared," it will no longer automatically send or receive messages, or generate interrupts.

Returns:

None.

6.2.1.13 ROM_CANMessageGet

Reads a CAN message from one of the message object buffers.

Prototype:

```
void
ROM_CANMessageGet(unsigned long ulBase,
                  unsigned long ulObjID,
                  tCANMsgObject *pMsgObject,
                  tBoolean bClrPendingInt)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANMessageGet` is a function pointer located at `ROM_CANTABLE[7]`.

Parameters:

ulBase is the base address of the CAN controller.

ulObjID is the object number to read (1-32).

pMsgObject points to a structure containing message object fields.

bClrPendingInt indicates whether an associated interrupt should be cleared.

Description:

This function is used to read the contents of one of the 32 message objects in the CAN controller, and return it to the caller. The data returned is stored in the fields of the caller-supplied structure pointed to by *pMsgObject*. The data consists of all of the parts of a CAN message, plus some control and status information.

Normally this is used to read a message object that has received and stored a CAN message with a certain identifier. However, this could also be used to read the contents of a message object in order to load the fields of the structure in case only part of the structure needs to be changed from a previous setting.

When using CANMessageGet, all of the same fields of the structure are populated in the same way as when the ROM_CANMessageSet() function is used, with the following exceptions:

pMsgObject->ulFlags:

- **MSG_OBJ_NEW_DATA** indicates if this is new data since the last time it was read
- **MSG_OBJ_DATA_LOST** indicates that at least one message was received on this message object, and not read by the host before being overwritten.

Returns:

None.

6.2.1.14 ROM_CANMessageSet

Configures a message object in the CAN controller.

Prototype:

```
void
ROM_CANMessageSet(unsigned long ulBase,
                  unsigned long ulObjID,
                  tCANMsgObject *pMsgObject,
                  tMsgObjType eMsgType)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANMessageSet` is a function pointer located at `ROM_CANTABLE[6]`.

Parameters:

ulBase is the base address of the CAN controller.

ulObjID is the object number to configure (1-32).

pMsgObject is a pointer to a structure containing message object settings.

eMsgType indicates the type of message for this object.

Description:

This function is used to configure any one of the 32 message objects in the CAN controller. A message object can be configured as any type of CAN message object as well as several options for automatic transmission and reception. This call also allows the message object to be configured to generate interrupts on completion of message receipt or transmission. The message object can also be configured with a filter/mask so that actions are only taken when a message that meets certain parameters is seen on the CAN bus.

The *eMsgType* parameter must be one of the following values:

- **MSG_OBJ_TYPE_TX** - CAN transmit message object.
- **MSG_OBJ_TYPE_TX_REMOTE** - CAN transmit remote request message object.
- **MSG_OBJ_TYPE_RX** - CAN receive message object.
- **MSG_OBJ_TYPE_RX_REMOTE** - CAN receive remote request message object.
- **MSG_OBJ_TYPE_RXTX_REMOTE** - CAN remote frame receive remote, then transmit message object.

The message object pointed to by *pMsgObject* must be populated by the caller, as follows:

- *ulMsgID* - contains the message ID, either 11 or 29 bits.
- *ulMsgIDMask* - mask of bits from *ulMsgID* that must match if identifier filtering is enabled.
- *ulFlags*
  - Set **MSG_OBJ_TX_INT_ENABLE** flag to enable interrupt on transmission.
  - Set **MSG_OBJ_RX_INT_ENABLE** flag to enable interrupt on receipt.
  - Set **MSG_OBJ_USE_ID_FILTER** flag to enable filtering based on the identifier mask specified by *ulMsgIDMask*.
- *ulMsgLen* - the number of bytes in the message data. This should be non-zero even for a remote frame; it should match the expected bytes of the data responding data frame.
- *pucMsgData* - points to a buffer containing up to 8 bytes of data for a data frame.

Example: To send a data frame or remote frame(in response to a remote request), take the following steps:

1. Set *eMsgType* to **MSG_OBJ_TYPE_TX**.
2. Set *pMsgObject->ulMsgID* to the message ID.
3. Set *pMsgObject->ulFlags*. Make sure to set **MSG_OBJ_TX_INT_ENABLE** to allow an interrupt to be generated when the message is sent.
4. Set *pMsgObject->ulMsgLen* to the number of bytes in the data frame.
5. Set *pMsgObject->pucMsgData* to point to an array containing the bytes to send in the message.
6. Call this function with *ulObjID* set to one of the 32 object buffers.

Example: To receive a specific data frame, take the following steps:

1. Set *eMsgObjType* to **MSG_OBJ_TYPE_RX**.
2. Set *pMsgObject->ulMsgID* to the full message ID, or a partial mask to use partial ID matching.
3. Set *pMsgObject->ulMsgIDMask* bits that should be used for masking during comparison.
4. Set *pMsgObject->ulFlags* as follows:
   - Set **MSG_OBJ_RX_INT_ENABLE** flag to be interrupted when the data frame is received.
   - Set **MSG_OBJ_USE_ID_FILTER** flag to enable identifier based filtering.
5. Set *pMsgObject->ulMsgLen* to the number of bytes in the expected data frame.
6. The buffer pointed to by *pMsgObject->pucMsgData* is not used by this call as no data is present at the time of the call.
7. Call this function with *ulObjID* set to one of the 32 object buffers.

If you specify a message object buffer that already contains a message definition, it is overwritten.

Returns:
None.

6.2.1.15 ROM_CANRetryGet

Returns the current setting for automatic retransmission.

Prototype:
```
tBoolean
ROM_CANRetryGet(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_CANTABLE` is an array of pointers located at `ROM_APITABLE[18]`.
`ROM_CANRetryGet` is a function pointer located at `ROM_CANTABLE[13]`.

Parameters:
ulBase is the base address of the CAN controller.

Description:
Reads the current setting for the automatic retransmission in the CAN controller and returns it to the caller.

Returns:
Returns **true** if automatic retransmission is enabled, **false** otherwise.

6.2.1.16 ROM_CANRetrySet

Sets the CAN controller automatic retransmission behavior.

Prototype:
```
void
ROM_CANRetrySet(unsigned long ulBase,
                tBoolean bAutoRetry)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_CANTABLE is an array of pointers located at ROM_APITABLE[18].
ROM_CANRetrySet is a function pointer located at ROM_CANTABLE[14].

Parameters:

ulBase is the base address of the CAN controller.

bAutoRetry enables automatic retransmission.

Description:

Enables or disables automatic retransmission of messages with detected errors. If *bAutoRetry* is **true**, then automatic retransmission is enabled, otherwise it is disabled.

Returns:

None.

6.2.1.17 ROM_CANStatusGet

Reads one of the controller status registers.

Prototype:

```
unsigned long
ROM_CANStatusGet(unsigned long ulBase,
                 tCANStsReg eStatusReg)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_CANTABLE is an array of pointers located at ROM_APITABLE[18].
ROM_CANStatusGet is a function pointer located at ROM_CANTABLE[8].

Parameters:

ulBase is the base address of the CAN controller.

eStatusReg is the status register to read.

Description:

Reads a status register of the CAN controller and returns it to the caller. The different status registers are:

- **CAN_STS_CONTROL** - the main controller status
- **CAN_STS_TXREQUEST** - bit mask of objects pending transmission
- **CAN_STS_NEWDAT** - bit mask of objects with new data
- **CAN_STS_MSGVAL** - bit mask of objects with valid configuration

When reading the main controller status register, a pending status interrupt is cleared. This should be used in the interrupt handler for the CAN controller if the cause is a status interrupt. The controller status register fields are as follows:

- **CAN_STATUS_BUS_OFF** - controller is in bus-off condition
- **CAN_STATUS_EWARN** - an error counter has reached a limit of at least 96
- **CAN_STATUS_EPASS** - CAN controller is in the error passive state
- **CAN_STATUS_RXOK** - a message was received successfully (independent of any message filtering).

- **CAN_STATUS_TXOK** - a message was successfully transmitted
- **CAN_STATUS_LEC_MSK** - mask of last error code bits (3 bits)
- **CAN_STATUS_LEC_NONE** - no error
- **CAN_STATUS_LEC_STUFF** - stuffing error detected
- **CAN_STATUS_LEC_FORM** - a format error occurred in the fixed format part of a message
- **CAN_STATUS_LEC_ACK** - a transmitted message was not acknowledged
- **CAN_STATUS_LEC_BIT1** - dominant level detected when trying to send in recessive mode
- **CAN_STATUS_LEC_BIT0** - recessive level detected when trying to send in dominant mode
- **CAN_STATUS_LEC_CRC** - CRC error in received message

The remaining status registers are 32-bit bit maps to the message objects. They can be used to quickly obtain information about the status of all the message objects without needing to query each one. They contain the following information:

- **CAN_STS_TXREQUEST** - if a message object's TxRequest bit is set, that means that a transmission is pending on that object. The application can use this to determine which objects are still waiting to send a message.
- **CAN_STS_NEWDAT** - if a message object's NewDat bit is set, that means that a new message has been received in that object, and has not yet been picked up by the host application
- **CAN_STS_MSGVAL** - if a message object's MsgVal bit is set, that means it has a valid configuration programmed. The host application can use this to determine which message objects are empty/unused.

Returns:

Returns the value of the status register.

7 CRC

Introduction 69
Functions 69

7.1 Introduction

CRC (Cyclic Redundancy Check) is a technique to validate a span of data has the same contents as when previously checked. This technique can be used to validate correct receipt of messages (nothing lost or modified in transit), to validate data after decompression, to validate that Flash memory contents have not been changed, and for other cases where the data must be validated. A CRC is preferred over a simple checksum (for example, XOR all bits) because it catches changes more readily.

The CRC API provides functions to compute the CRC-8-CCITT and CRC-16 of a buffer of data. Support is provided for computing a running CRC, where a partial CRC is computed on one portion of the data, and then continued at a later time on another portion of the data. This is useful when computing the CRC on a stream of data that is coming in via a serial link (for example).

The CRC-16 APIs implement the standard CRC-16 polynomial (also known as CRC-16-IBM):

$$x^{16} + x^{15} + x^2 + 1$$

The CRC-8-CCITT API implements the standard CRC-8-CCITT polynomial:

$$x^8 + x^2 + x + 1$$

The ROM_Crc16Array3() function fperforms three separate CRC-16 calculations; one across all bytes in the input data array, one across the even bytes, and one across the odd bytes. The ability of a CRC to detect errors decreases as the size of the data array increases. The triple CRC-16 function tries to slow this decrease in error detection rate as it is more difficult for a data error (or errors) to result in all three CRC-16 calculations being correct.

7.2 Functions

Functions

- unsigned short ROM_Crc16 (unsigned short usCrc, const unsigned char *pucData, unsigned long ulCount)
- unsigned short ROM_Crc16Array (unsigned long ulWordLen, unsigned long *pulData)
- void ROM_Crc16Array3 (unsigned long ulWordLen, unsigned long *pulData, unsigned short *pusCrc3)
- unsigned char ROM_Crc8CCITT (unsigned char ucCrc, const unsigned char *pucData, unsigned long ulCount)

7.2.1 Function Documentation

7.2.1.1 ROM_Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

```
unsigned short
ROM_Crc16(unsigned short usCrc,
          const unsigned char *pucData,
          unsigned long ulCount)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_SOFTWARETABLE is an array of pointers located at ROM_APITABLE[21].
ROM_Crc16 is a function pointer located at ROM_SOFTWARETABLE[3].

Parameters:

usCrc is the starting CRC-16 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-16 of the input buffer. The CRC-16 is computed in a running fashion, meaning that the entire data block that is to have its CRC-16 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **usCrc** should be set to 0. If, however, the entire block of data is not available, then **usCrc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **usCrc** for the next portion of the data.

For example, to compute the CRC-16 of a block that has been split into three pieces, use the following:

```
usCrc = ROM_Crc16(0, pucData1, ulLen1);
usCrc = ROM_Crc16(usCrc, pucData2, ulLen2);
usCrc = ROM_Crc16(usCrc, pucData3, ulLen3);
```

Computing a CRC-16 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The CRC-16 of the input data.

7.2.1.2 ROM_Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

```
unsigned short
ROM_Crc16Array(unsigned long ulWordLen,
               unsigned long *pulData)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SOFTWARETABLE` is an array of pointers located at `ROM_APITABLE[21]`.
`ROM_Crc16Array` is a function pointer located at `ROM_SOFTWARETABLE[1]`.

Parameters:

ulWordLen is the length of the array in words.

pulData is a pointer to the array of words.

Description:

This function is used to calculate a standard CRC-16 cyclical redundancy check on the data passed to it. The length of the data only matters in terms of the "strength" of the CRC (likelihood of catching errors). The longer the data, the more likely it will not catch some errors.

Returns:

Returns the calculated CRC-16.

7.2.1.3 ROM_Crc16Array3

Calculates three CRC-16s of an array of words.

Prototype:

```
void
ROM_Crc16Array3(unsigned long ulWordLen,
                unsigned long *pulData,
                unsigned short *pusCrc3)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SOFTWARETABLE` is an array of pointers located at `ROM_APITABLE[21]`.
`ROM_Crc16Array3` is a function pointer located at `ROM_SOFTWARETABLE[2]`.

Parameters:

ulWordLen is the length of the array in words.

pulData is a pointer to the array of words.

pusCrc3 is a pointer to an array into which the three CRC values are to be placed.

Description:

This function is used to calculate three CRC-16s from the same array. This computes the CRC-16 on all of the bytes (same as ROM_Crc16Array()), on the even bytes, and on the odd bytes. This calculation of three CRC-16s increases the chance of detecting errors because it is much harder for a set of errors to end up being correct for all three CRC-16s.

Returns:

None

7.2.1.4 ROM_Crc8CCITT

Calculates the CRC-8-CCITT of an array of bytes.

Prototype:

```
unsigned char
ROM_Crc8CCITT(unsigned char ucCrc,
              const unsigned char *pucData,
              unsigned long ulCount)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SOFTWARETABLE` is an array of pointers located at `ROM_APITABLE[21]`.
`ROM_Crc8CCITT` is a function pointer located at `ROM_SOFTWARETABLE[4]`.

Parameters:

ucCrc is the starting CRC-8-CCITT value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-8-CCITT of the input buffer. The CRC-8-CCITT is computed in a running fashion, meaning that the entire data block that is to have its CRC-8-CCITT computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ucCrc** should be set to 0. If, however, the entire block of data is not available, then **ucCrc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **ucCrc** for the next portion of the data.

For example, to compute the CRC-8-CCITT of a block that has been split into three pieces, use the following:

```
ucCrc = ROM_Crc8CCITT(0, pucData1, ulLen1);
ucCrc = ROM_Crc8CCITT(ucCrc, pucData2, ulLen2);
ucCrc = ROM_Crc8CCITT(ucCrc, pucData3, ulLen3);
```

Computing a CRC-8-CCITT in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The CRC-8-CCITT of the input data.

8 Flash

Introduction 73
Functions 73

8.1 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.

8.2 Functions

Functions

- long ROM_FlashErase (unsigned long ulAddress)
- void ROM_FlashIntClear (unsigned long ulIntFlags)
- void ROM_FlashIntDisable (unsigned long ulIntFlags)
- void ROM_FlashIntEnable (unsigned long ulIntFlags)
- 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)

8.2.1 Function Documentation

8.2.1.1 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 is 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.

8.2.1.2 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_INT_PROGRAM** or **FLASH_INT_AMISC** 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:

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

Returns:

None.

8.2.1.3 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_INT_PROGRAM** or **FLASH_INT_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.

8.2.1.4 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:

ulIntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **FLASH_INT_PROGRAM** or **FLASH_INT_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.

8.2.1.5 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_INT_PROGRAM** and **FLASH_INT_ACCESS**.

8.2.1.6 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. Each word in a page of flash can only be programmed one time between an erase of that page; programming a word multiple times will result in an unpredictable value in that word of flash.

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.

8.2.1.7 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.

8.2.1.8 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.

8.2.1.9 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 (that is, 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.

8.2.1.10 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.

8.2.1.11 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.

8.2.1.12 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.

8.2.1.13 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.

8.2.1.14 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.

9 Floating-Point Unit (FPU)

Introduction 83
API Functions 84

9.1 Introduction

The floating-point unit (FPU) driver provides methods for manipulating the behavior of the floating-point unit in the Cortex-M processor. By default, the floating-point is disabled and must be enabled prior to the execution of any floating-point instructions. If a floating-point instruction is executed when the floating-point unit is disabled, a NOCP usage fault is generated. This feature can be used by an RTOS, for example, to keep track of which tasks actually use the floating-point unit, and therefore only perform floating-point context save/restore on task switches that involve those tasks.

There are three methods of handling the floating-point context when the processor executes an interrupt handler: it can do nothing with the floating-point context, it can always save the floating-point context, or it can perform a lazy save/restore of the floating-point context. If nothing is done with the floating-point context, the interrupt stack frame is identical to a Cortex-M processor that does not have a floating-point unit, containing only the volatile registers of the integer unit. This method is useful for applications where the floating-point unit is used by the main thread of execution, but not in any of the interrupt handlers. By not saving the floating-point context, stack usage is reduced and interrupt latency is kept to a minimum.

Alternatively, the floating-point context can always be saved onto the stack. This method allows floating-point operations to be performed inside interrupt handlers without any special precautions, at the expense of increased stack usage (for the floating-point context) and increased interrupt latency (due to the additional writes to the stack). The advantage to this method is that the stack frame always contains the floating-point context when inside an interrupt handler.

The default handling of the floating-point context is to perform a lazy save/restore. When an interrupt is taken, space is reserved on the stack for the floating-point context but the context is not written. This method keeps the interrupt latency to a minimum because only the integer state is written to the stack. Then, if a floating-point instruction is executed from within the interrupt handler, the floating-point context is written to the stack prior to the execution of the floating-point instruction. Finally, upon return from the interrupt, the floating-point context is restored from the stack only if it was written. Using lazy save/restore provides a blend between fast interrupt response and the ability to use floating-point instructions in the interrupt handler.

The floating-point unit can generate an interrupt when one of several exceptions occur. The exceptions are underflow, overflow, divide by zero, invalid operation, input denormal, and inexact exception. The application can optionally choose to enable one or more of these interrupts and use the interrupt handler to decide upon a course of action to be taken in each case.

The behavior of the floating-point unit can also be adjusted, specifying the format of half-precision floating-point values, the handle of NaN values, the flush-to-zero mode (which sacrifices full IEEE compliance for execution speed), and the rounding mode for results.

9.2 API Functions

Functions

- void ROM_FPUDisable (void)
- void ROM_FPUEnable (void)
- void ROM_FPUFlushToZeroModeSet (unsigned long ulMode)
- void ROM_FPUHalfPrecisionModeSet (unsigned long ulMode)
- void ROM_FPULazyStackingEnable (void)
- void ROM_FPUNaNModeSet (unsigned long ulMode)
- void ROM_FPURoundingModeSet (unsigned long ulMode)
- void ROM_FPUStackingDisable (void)
- void ROM_FPUStackingEnable (void)

9.2.1 Function Documentation

9.2.1.1 ROM_FPUDisable

Disables the floating-point unit.

Prototype:

```
void
ROM_FPUDisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUDisable` is a function pointer located at `ROM_FPUTABLE[1]`.

Description:

This function disables the floating-point unit, preventing floating-point instructions from executing (generating a NOCP usage fault instead).

Returns:

None.

9.2.1.2 ROM_FPUEnable

Enables the floating-point unit.

Prototype:

```
void
ROM_FPUEnable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUEnable` is a function pointer located at `ROM_FPUTABLE[0]`.

Description:
This function enables the floating-point unit, allowing the floating-point instructions to be executed. This function must be called prior to performing any hardware floating-point operations; failure to do so results in a NOCP usage fault.

Returns:
None.

9.2.1.3 ROM_FPUFlushToZeroModeSet

Selects the flush-to-zero mode.

Prototype:
```
void
ROM_FPUFlushToZeroModeSet(unsigned long ulMode)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUFlushToZeroModeSet` is a function pointer located at `ROM_FPUTABLE[2]`.

Parameters:
ulMode is the flush-to-zero mode; which is either **FPU_FLUSH_TO_ZERO_DIS** or **FPU_FLUSH_TO_ZERO_EN**.

Description:
This function enables or disables the flush-to-zero mode of the floating-point unit. When disabled (the default), the floating-point unit is fully IEEE compliant. When enabled, values close to zero are treated as zero, greatly improving the execution speed at the expense of some accuracy (as well as IEEE compliance).

Note:
Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:
None.

9.2.1.4 ROM_FPUHalfPrecisionModeSet

Selects the format of half-precision floating-point values.

Prototype:
```
void
ROM_FPUHalfPrecisionModeSet(unsigned long ulMode)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUHalfPrecisionModeSet` is a function pointer located at `ROM_FPUTABLE[3]`.

Parameters:

ulMode is the format for half-precision floating-point values; which is either **FPU_HALF_IEEE** or **FPU_HALF_ALTERNATE**.

Description:

This function selects between the IEEE half-precision floating-point representation and the Cortex-M processor alternative representation. The alternative representation has a larger range but does not have a way to encode infinity (positive or negative) or NaN (quiet or signaling). The default setting is the IEEE format.

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

9.2.1.5 ROM_FPULazyStackingEnable

Enables the lazy stacking of floating-point registers.

Prototype:

```
void
ROM_FPULazyStackingEnable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPULazyStackingEnable` is a function pointer located at `ROM_FPUTABLE[4]`.

Description:

This function enables the lazy stacking of floating-point registers s0-s15 when an interrupt is handled. When lazy stacking is enabled, space is reserved on the stack for the floating-point context, but the floating-point state is not saved. If a floating-point instruction is executed from within the interrupt context, the floating-point context is first saved into the space reserved on the stack. On completion of the interrupt handler, the floating-point context is only restored if it was saved (as the result of executing a floating-point instruction).

This provides a compromise between fast interrupt response (because the floating-point state is not saved on interrupt entry) and the ability to use floating-point in interrupt handlers (because the floating-point state is saved if floating-point instructions are used).

Returns:

None.

9.2.1.6 ROM_FPUNaNModeSet

Selects the NaN mode.

Prototype:

```
void
ROM_FPUNaNModeSet(unsigned long ulMode)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUNaNModeSet` is a function pointer located at `ROM_FPUTABLE[5]`.

Parameters:

ulMode is the mode for NaN results; which is either **FPU_NAN_PROPAGATE** or **FPU_NAN_DEFAULT**.

Description:

This function selects the handling of NaN results during floating-point computations. NaNs can either propagate (the default), or they can return the default NaN.

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

9.2.1.7 ROM_FPURoundingModeSet

Selects the rounding mode for floating-point results.

Prototype:

```
void
ROM_FPURoundingModeSet(unsigned long ulMode)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPURoundingModeSet` is a function pointer located at `ROM_FPUTABLE[6]`.

Parameters:

ulMode is the rounding mode.

Description:

This function selects the rounding mode for floating-point results. After a floating-point operation, the result is rounded toward the specified value. The default mode is **FPU_ROUND_NEAREST**.

The following rounding modes are available (as specified by *ulMode*):

- **FPU_ROUND_NEAREST** - round toward the nearest value
- **FPU_ROUND_POS_INF** - round toward positive infinity
- **FPU_ROUND_NEG_INF** - round toward negative infinity
- **FPU_ROUND_ZERO** - round toward zero

Note:

Unless this function is called prior to executing any floating-point instructions, the default mode is used.

Returns:

None.

9.2.1.8 ROM_FPUStackingDisable

Disables the stacking of floating-point registers.

Prototype:

```
void
ROM_FPUStackingDisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUStackingDisable` is a function pointer located at `ROM_FPUTABLE[7]`.

Description:

This function disables the stacking of floating-point registers s0-s15 when an interrupt is handled. When floating-point context stacking is disabled, floating-point operations performed in an interrupt handler destroy the floating-point context of the main thread of execution.

Returns:

None.

9.2.1.9 ROM_FPUStackingEnable

Enables the stacking of floating-point registers.

Prototype:

```
void
ROM_FPUStackingEnable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_FPUTABLE` is an array of pointers located at `ROM_APITABLE[26]`.
`ROM_FPUStackingEnable` is a function pointer located at `ROM_FPUTABLE[8]`.

Description:

This function enables the stacking of floating-point registers s0-s15 when an interrupt is handled. When enabled, space is reserved on the stack for the floating-point context and the floating-point state is saved into this stack space. Upon return from the interrupt, the floating-point context is restored.

If the floating-point registers are not stacked, floating-point instructions cannot be safely executed in an interrupt handler because the values of s0-s15 are not likely to be preserved for the interrupted code. On the other hand, stacking the floating-point registers increases the stacking operation from 8 words to 26 words, also increasing the interrupt response latency.

Returns:

None.

10 GPIO

Introduction 89
Functions 89

10.1 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).

10.2 Functions

Functions

- void GPIOADCTriggerDisable (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOADCTriggerEnable (unsigned long ulPort, unsigned char ucPins)

- unsigned long ROM_GPIODirModeGet (unsigned long ulPort, unsigned char ucPin)
- void ROM_GPIODirModeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)
- void ROM_GPIODMATriggerDisable (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIODMATriggerEnable (unsigned long ulPort, unsigned char ucPins)
- 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_GPIOPinConfigure (unsigned long ulPinConfig)
- 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_GPIOPinTypeADC (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypeCAN (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypeComparator (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_GPIOPinTypeI2CSCL (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypePWM (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypeQEI (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_GPIOPinTypeUSBAnalog (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypeUSBDigital (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinWrite (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)

10.2.1 Function Documentation

10.2.1.1 GPIOADCTriggerDisable

Disable a GPIO pin as a trigger to start an ADC capture.

Prototype:

```
void
GPIOADCTriggerDisable(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_GPIOADCTriggerDisable is a function pointer located at ROM_GPIOTABLE[34].

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

Description:
This function disables a GPIO pin to be used as a trigger to start an ADC sequence. This function can be used to disable this feature if it was enabled via a call to GPIOADCTriggerEnable().

Returns:
None.

10.2.1.2 ROM_GPIOADCTriggerEnable

Enables a GPIO pin as a trigger to start an ADC capture.

Prototype:
```
void
ROM_GPIOADCTriggerEnable(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_GPIOADCTriggerEnable is a function pointer located at ROM_GPIOTABLE[33].

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

Description:
This function enables a GPIO pin to be used as a trigger to start an ADC sequence. Any GPIO pin can be configured to be an external trigger for an ADC sequence. The GPIO pin will still generate interrupts if the interrupt is enabled for the selected pin.

Returns:
None.

10.2.1.3 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().

10.2.1.4 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 parameter *ulPinIO* 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 is programmed as a software controlled input, **GPIO_DIR_MODE_OUT** specifies that the pin is programmed as a software controlled output, and **GPIO_DIR_MODE_HW** specifies that the pin is placed under hardware control.

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

Note:
ROM_GPIOPadConfigSet() must also be used to configure the corresponding pad(s) in order for them to propagate the signal to/from the GPIO.

Returns:
None.

10.2.1.5 ROM_GPIODMATriggerDisable

Disables a GPIO pin as a trigger to start a DMA transaction.

Prototype:
```
void
ROM_GPIODMATriggerDisable(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_GPIODMATriggerDisable` is a function pointer located at `ROM_GPIOTABLE[32]`.

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

Description:
This function disables a GPIO pin to be used as a trigger to start a uDMA transaction. This function can be used to disable this feature if it was enabled via a call to GPIODMATriggerEnable().

Returns:
None.

10.2.1.6 ROM_GPIODMATriggerEnable

Enables a GPIO pin as a trigger to start a DMA transaction.

Prototype:
```
void
ROM_GPIODMATriggerEnable(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_GPIODMATriggerEnable` is a function pointer located at `ROM_GPIOTABLE[31]`.

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

Description:
This function enables a GPIO pin to be used as a trigger to start a uDMA transaction. Any GPIO pin can be configured to be an external trigger for the uDMA. The GPIO pin will still generate interrupts if the interrupt is enabled for the selected pin.

Returns:
None.

10.2.1.7 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().

10.2.1.8 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).

ulIntType specifies the type of interrupt trigger mechanism.

Description:

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

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

- **GPIO_FALLING_EDGE**
- **GPIO_RISING_EDGE**
- **GPIO_BOTH_EDGES**
- **GPIO_LOW_LEVEL**
- **GPIO_HIGH_LEVEL**

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

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

Note:

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.

10.2.1.9 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 *pulStrength* and *pulPinType* 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

10.2.1.10 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 parameter *ulStrength* 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 parameter *ulPinType* 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, and so on.

Returns:
None.

10.2.1.11 ROM_GPIOPinConfigure

Configures the alternate function of a GPIO pin.

Prototype:
```
void
ROM_GPIOPinConfigure(unsigned long ulPinConfig)
```

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_GPIOPinConfigure` is a function pointer located at `ROM_GPIOTABLE[26]`.

Parameters:
ulPinConfig is the pin configuration value, specified as only one of the **GPIO_P**??_??? values.

Description:
This function configures the pin mux that selects the peripheral function associated with a particular GPIO pin. Only one peripheral function at a time can be associated with a GPIO pin, and each peripheral function should only be associated with a single GPIO pin at a time (despite the fact that many of them can be associated with more than one GPIO pin).

Returns:
None.

10.2.1.12 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, and so on.

Note:

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

Returns:

None.

10.2.1.13 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, and so on.

Returns:

None.

10.2.1.14 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, and so on.

Returns:

None.

10.2.1.15 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 is returned.

Returns:

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

10.2.1.16 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, and so on.

Returns:

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

10.2.1.17 ROM_GPIOPinTypeADC

Configures pin(s) for use as analog-to-digital converter inputs.

Prototype:

```
void
ROM_GPIOPinTypeADC(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_GPIOPinTypeADC` is a function pointer located at `ROM_GPIOTABLE[23]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

The analog-to-digital converter input pins must be properly configured to function correctly. This function provides the proper configuration for those pin(s).

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

Note:
This cannot be used to turn any pin into an ADC input; it only configures an ADC input pin for proper operation.

Returns:
None.

10.2.1.18 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, and so on.

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

Returns:
None.

10.2.1.19 ROM_GPIOPinTypeComparator

Configures pin(s) for use as an analog comparator input.

Prototype:

```
void
ROM_GPIOPinTypeComparator(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_GPIOPinTypeComparator` is a function pointer located at `ROM_GPIOTABLE[13]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

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

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

Note:

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

Returns:

None.

10.2.1.20 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 function provides the proper configuration for those pin(s).

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

Returns:
None.

10.2.1.21 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 function provides the proper configuration for those pin(s).

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

Returns:
None.

10.2.1.22 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 function provides the proper configuration for those pin(s).

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

Returns:

None.

10.2.1.23 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, and so on.

Note:

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

Returns:

None.

10.2.1.24 ROM_GPIOPinTypeI2CSCL

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

Prototype:

```
void
ROM_GPIOPinTypeI2CSCL(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_GPIOPinTypeI2CSCL is a function pointer located at ROM_GPIOTABLE[39].

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 the SCL pin(s).

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

Note:

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

Returns:

None.

10.2.1.25 ROM_GPIOPinTypePWM

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

Prototype:

```
void
ROM_GPIOPinTypePWM(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_GPIOPinTypePWM is a function pointer located at ROM_GPIOTABLE[17].

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

The PWM pins must be properly configured for the PWM 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, and so on.

Note:

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

Returns:
None.

10.2.1.26 ROM_GPIOPinTypeQEI

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

Prototype:
```
void
ROM_GPIOPinTypeQEI(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_GPIOPinTypeQEI` is a function pointer located at `ROM_GPIOTABLE[18]`.

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

Description:
The QEI pins must be properly configured for the QEI 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, not using the on-chip pull-ups).

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

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

Returns:
None.

10.2.1.27 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, and so on.

Note:

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

Returns:

None.

10.2.1.28 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, and so on.

Note:

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

Returns:

None.

10.2.1.29 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, and so on.

Note:

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

Returns:

None.

10.2.1.30 ROM_GPIOPinTypeUSBAnalog

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

Prototype:

```
void
ROM_GPIOPinTypeUSBAnalog(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_GPIOPinTypeUSBAnalog` is a function pointer located at `ROM_GPIOTABLE[28]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

Some USB analog pins must be properly configured for the USB peripheral to function correctly. This function provides the proper configuration for any USB pin(s). This can also be used to configure the EPEN and PFAULT pins so that they are no longer used by the USB controller.

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

Note:

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

Returns:

None.

10.2.1.31 ROM_GPIOPinTypeUSBDigital

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

Prototype:

```
void
ROM_GPIOPinTypeUSBDigital(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_GPIOPinTypeUSBDigital` is a function pointer located at `ROM_GPIOTABLE[24]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

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

This function should only be used with EPEN and PFAULT pins as all other USB pins are analog in nature or are not used in devices without OTG functionality.

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

Note:

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

Returns:

None.

10.2.1.32 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, and so on.

Returns:

None.

11 Hibernation Module

Introduction 111
Functions 112

11.1 Introduction

The Hibernate API provides a set of functions for using the Hibernation module on the Stellaris microcontroller. The Hibernation module allows the software application to cause power to be removed from the microcontroller, and then be powered on later based on specific time or a signal on the external **WAKE** pin. The API provides functions to configure wake conditions, manage interrupts, read status, save and restore program state information, and request hibernation mode.

Some of the features of the Hibernation module are:

- 32-bit real time clock
- Trim register for fine tuning the RTC rate
- Two RTC match registers for generating RTC events
- External **WAKE** pin to initiate a wake-up
- Low-battery detection
- 64 32-bit words of non-volatile memory
- Programmable interrupts for hibernation events

The Hibernation module must be enabled before it can be used. Use the ROM_HibernateEnableExpClk() function to enable it. If a crystal is used for the clock source, then the initializing code must allow time for the crystal to stabilize after calling the ROM_HibernateEnableExpClk() function. Refer to the device data sheet for information about crystal stabilization time. If an oscillator is used, then no delay is necessary. After the module is enabled, the clock source must be configured by calling ROM_HibernateClockSelect().

In order to use the RTC feature of the Hibernation module, the RTC must be enabled by calling ROM_HibernateRTCEnable(). It can be later disabled by calling ROM_HibernateRTCDisable(). These functions can be called at any time to start and stop the RTC. The RTC value can be read or set by using the ROM_HibernateRTCGet() and ROM_HibernateRTCSet() functions. The two match registers can be read and set by using the ROM_HibernateRTCMatch0Get(), ROM_HibernateRTCMatch0Set(), ROM_HibernateRTCMatch1Get(), and ROM_HibernateRTCMatch1Set() functions. The real-time clock rate can be adjusted by using the trim register. Use the ROM_HibernateRTCTrimGet() and ROM_HibernateRTCTrimSet() functions for this purpose.

Application state information can be stored in the non-volatile memory of the Hibernation module when the processor is powered off. Use the ROM_HibernateDataSet() and ROM_HibernateDataGet() functions to access the non-volatile memory area.

The module can be configured to wake when the external **WAKE** pin is asserted, or when an RTC match occurs, or both. Use the ROM_HibernateWakeSet() function to configure the wake conditions. The present configuration can be read by calling ROM_HibernateWakeGet().

The Hibernation module can detect a low battery and signal the processor. It can also be configured to abort a hibernation request if the battery voltage is too low. Use the ROM_HibernateLowBatSet() and ROM_HibernateLowBatGet() functions to configure this feature.

Several functions are provided for managing interrupts. Use the ROM_HibernateIntEnable() and ROM_HibernateIntDisable() functions to enable and disable specific interrupt sources. The present interrupt status can be found by calling ROM_HibernateIntStatus(). In the interrupt handler, all pending interrupts must be cleared. Use the ROM_HibernateIntClear() function to clear pending interrupts.

Finally, once the module is appropriately configured, the state saved, and the software application is ready to hibernate, call the ROM_HibernateRequest() function. This will initiate the sequence to remove power from the processor. At a power-on reset, the software application can use the ROM_HibernateIsActive() function to determine if the Hibernation module is already active and therefore does not need to be enabled. This can provide a hint to the software that the processor is waking from hibernation instead of a cold start. The software can then use the ROM_HibernateIntStatus() and ROM_HibernateDataGet() functions to discover the cause of the wake and to get the saved system state.

11.2 Functions

Functions

- unsigned long ROM_HibernateBatCheckDone (void)
- void ROM_HibernateBatCheckStart (void)
- void ROM_HibernateClockConfig (unsigned long ulConfig)
- void ROM_HibernateClockSelect (unsigned long ulClockInput)
- void ROM_HibernateDataGet (unsigned long *pulData, unsigned long ulCount)
- void ROM_HibernateDataSet (unsigned long *pulData, unsigned long ulCount)
- void ROM_HibernateDisable (void)
- void ROM_HibernateEnableExpClk (unsigned long ulHibClk)
- void ROM_HibernateIntClear (unsigned long ulIntFlags)
- void ROM_HibernateIntDisable (unsigned long ulIntFlags)
- void ROM_HibernateIntEnable (unsigned long ulIntFlags)
- unsigned long ROM_HibernateIntStatus (tBoolean bMasked)
- unsigned int ROM_HibernateIsActive (void)
- unsigned long ROM_HibernateLowBatGet (void)
- void ROM_HibernateLowBatSet (unsigned long ulLowBatFlags)
- void ROM_HibernateRequest (void)
- void ROM_HibernateRTCDisable (void)
- void ROM_HibernateRTCEnable (void)
- unsigned long ROM_HibernateRTCGet (void)
- unsigned long ROM_HibernateRTCMatch0Get (void)
- void ROM_HibernateRTCMatch0Set (unsigned long ulMatch)
- void ROM_HibernateRTCSet (unsigned long ulRTCValue)
- unsigned long ROM_HibernateRTCSSGet (void)
- unsigned long ROM_HibernateRTCSSMatch0Get (void)

- void ROM_HibernateRTCSSMatch0Set (unsigned long ulMatch)
- unsigned long ROM_HibernateRTCTrimGet (void)
- void ROM_HibernateRTCTrimSet (unsigned long ulTrim)
- unsigned long ROM_HibernateWakeGet (void)
- void ROM_HibernateWakeSet (unsigned long ulWakeFlags)

11.2.1 Function Documentation

11.2.1.1 ROM_HibernateBatCheckDone

Returns if a forced battery check has completed.

Prototype:

```
unsigned long
ROM_HibernateBatCheckDone(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateBatCheckDone` is a function pointer located at `ROM_HIBERNATETABLE[30]`.

Description:

This function returns if the forced battery check initiated by a call to the HibernateBatCheckStart() function has completed. This function will return a non-zero value until the battery level check has completed. Once this function returns a value of zero, the hibernation module has completed the battery check and the HibernateIntStatus() function can be used to check if the battery was low by checking if the value returned has the **HIBERNATE_INT_LOW_BAT** set.

Returns:

The value is zero when the battery level check has completed or non-zero if the check is still in process.

11.2.1.2 ROM_HibernateBatCheckStart

Forces the Hibernation module to initiate a check of the battery voltage.

Prototype:

```
void
ROM_HibernateBatCheckStart(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateBatCheckStart` is a function pointer located at `ROM_HIBERNATETABLE[29]`.

Description:

This function forces the Hibernation module to initiate a check of the battery voltage immediately rather than waiting for the next check interval to pass. After calling this function, the

application should call the () function and wait for the function to return a zero value before calling the HibernateIntStatus() to check if the return code has the **HIBERNATE_INT_LOW_BAT** set. If **HIBERNATE_INT_LOW_BAT** is set this indicates that battery level is low. The application can also enable the **HIBERNATE_INT_LOW_BAT** interrupt and wait for an interrupt to indicate that the battery level is low.

Note:
A hibernation request is held off if a battery check is in progress.

Returns:
None.

11.2.1.3 ROM_HibernateClockConfig

Configures the clock input for the Hibernation module.

Prototype:
```
void
ROM_HibernateClockConfig(unsigned long ulConfig)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateClockConfig` is a function pointer located at `ROM_HIBERNATETABLE[28]`.

Parameters:
ulConfig is one of the possible configuration options for the clock input listed below.

Description:
This function is used to configure the clock input for the Hibernation module. The *ulConfig* parameter can be one of the following values:

- **HIBERNATE_OSC_DISABLE** specifies that the internal oscillator is powered off and either an externally supplied clock source or no clock source is being used.
- **HIBERNATE_OSC_HIGHDRIVE** specifies a higher drive strength when a 24pF filter capacitor is used with a crystal.
- **HIBERNATE_OSC_LOWDRIVE** specifies a lower drive strength when a 12pF filter capacitor is used with a crystal.

The **HIBERNATE_OSC_DISABLE** option is used to disable and power down the internal oscillator if an external clock source or no clock source is used instead of a 32.768 kHz crystal. In the case where an external crystal is used, either the **HIBERNATE_OSC_HIGHDRIVE** or **HIBERNATE_OSC_LOWDRIVE** is used. This optimizes the oscillator drive strength to match the size of the filter capacitor that is used with the external crystal circuit.

Returns:
None.

11.2.1.4 ROM_HibernateClockSelect

Selects the clock input for the Hibernation module.

Prototype:

```
void
ROM_HibernateClockSelect(unsigned long ulClockInput)
```

ROM Location:

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

Parameters:

ulClockInput specifies the clock input.

Description:

Configures the clock input for the Hibernation module. The configuration option chosen depends entirely on hardware design. The clock input for the module will either be a 32.768 kHz oscillator or a 4.194304 MHz crystal. The *ulClockFlags* parameter must be one of the following:

- **HIBERNATE_CLOCK_SEL_RAW** - use the raw signal from a 32.768 kHz oscillator.
- **HIBERNATE_CLOCK_SEL_DIV128** - use the crystal input, divided by 128.

Returns:

None.

11.2.1.5 ROM_HibernateDataGet

Reads a set of data from the non-volatile memory of the Hibernation module.

Prototype:

```
void
ROM_HibernateDataGet(unsigned long *pulData,
                     unsigned long ulCount)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateDataGet` is a function pointer located at `ROM_HIBERNATETABLE[19]`.

Parameters:

pulData points to a location where the data that is read from the Hibernation module will be stored.

ulCount is the count of 32-bit words to read.

Description:

Retrieves a set of data from the Hibernation module non-volatile memory that was previously stored with the ROM_HibernateDataSet() function. The caller must ensure that *pulData* points to a large enough memory block to hold all the data that is read from the non-volatile memory.

Returns:

None.

11.2.1.6 ROM_HibernateDataSet

Stores data in the non-volatile memory of the Hibernation module.

Prototype:

```
void
ROM_HibernateDataSet(unsigned long *pulData,
                     unsigned long ulCount)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateDataSet` is a function pointer located at `ROM_HIBERNATETABLE[18]`.

Parameters:

pulData points to the data that the caller wants to store in the memory of the Hibernation module.

ulCount is the count of 32-bit words to store.

Description:

Stores a set of data in the Hibernation module non-volatile memory. This memory is preserved when the power to the processor is turned off, and can be used to store application state information which will be available when the processor wakes. Up to 64 32-bit words can be stored in the non-volatile memory. The data can be restored by calling the ROM_HibernateDataGet() function.

Returns:

None.

11.2.1.7 ROM_HibernateDisable

Disables the Hibernation module for operation.

Prototype:

```
void
ROM_HibernateDisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateDisable` is a function pointer located at `ROM_HIBERNATETABLE[2]`.

Description:

Disables the Hibernation module for operation. After this function is called, none of the Hibernation module features are available.

Returns:

None.

11.2.1.8 ROM_HibernateEnableExpClk

Enables the Hibernation module for operation.

Prototype:

```
void
ROM_HibernateEnableExpClk(unsigned long ulHibClk)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateEnableExpClk` is a function pointer located at `ROM_HIBERNATETABLE[1]`.

Parameters:

ulHibClk is the rate of the clock supplied to the Hibernation module.

Description:

Enables the Hibernation module for operation. This function should be called before any of the Hibernation module features are used.

The peripheral clock is the same as the processor clock. This is 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.

11.2.1.9 ROM_HibernateIntClear

Clears pending interrupts from the Hibernation module.

Prototype:

```
void
ROM_HibernateIntClear(unsigned long ulIntFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateIntClear` is a function pointer located at `ROM_HIBERNATETABLE[0]`.

Parameters:

ulIntFlags is the bit mask of the interrupts to be cleared.

Description:

Clears the specified interrupt sources. This must be done from within the interrupt handler or else the handler is called again upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to the ROM_HibernateIntEnable() function.

Note:

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt

source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:
None.

11.2.1.10 ROM_HibernateIntDisable

Disables interrupts for the Hibernation module.

Prototype:
```
void
ROM_HibernateIntDisable(unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateIntDisable` is a function pointer located at `ROM_HIBERNATETABLE[22]`.

Parameters:
ulIntFlags is the bit mask of the interrupts to be disabled.

Description:
Disables the specified interrupt sources from the Hibernation module.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to the ROM_HibernateIntEnable() function.

Returns:
None.

11.2.1.11 ROM_HibernateIntEnable

Enables interrupts for the Hibernation module.

Prototype:
```
void
ROM_HibernateIntEnable(unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateIntEnable` is a function pointer located at `ROM_HIBERNATETABLE[21]`.

Parameters:
ulIntFlags is the bit mask of the interrupts to be enabled.

Description:
Enables the specified interrupt sources from the Hibernation module.

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

- **HIBERNATE_INT_PIN_WAKE** - wake from pin interrupt
- **HIBERNATE_INT_LOW_BAT** - low battery interrupt
- **HIBERNATE_INT_RTC_MATCH_0** - RTC match 0 interrupt
- **HIBERNATE_INT_RTC_MATCH_1** - RTC match 1 interrupt

Returns:
None.

11.2.1.12 ROM_HibernateIntStatus

Gets the current interrupt status of the Hibernation module.

Prototype:
```
unsigned long
ROM_HibernateIntStatus(tBoolean bMasked)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateIntStatus` is a function pointer located at `ROM_HIBERNATETABLE[23]`.

Parameters:
bMasked is false to retrieve the raw interrupt status, and true to retrieve the masked interrupt status.

Description:
Returns the interrupt status of the Hibernation module. The caller can use this to determine the cause of a hibernation interrupt. Either the masked or raw interrupt status can be returned.

Returns:
Returns the interrupt status as a bit field with the values as described in the ROM_HibernateIntEnable() function.

11.2.1.13 ROM_HibernateIsActive

Checks to see if the Hibernation module is already powered up.

Prototype:
```
unsigned int
ROM_HibernateIsActive(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateIsActive` is a function pointer located at `ROM_HIBERNATETABLE[24]`.

Description:
This function queries the control register to determine if the module is already active. This function can be called at a power-on reset to help determine if the reset is due to a wake from hibernation or a cold start. If the Hibernation module is already active, then it does not need to be re-enabled and its status can be queried immediately.

The software application should also use the ROM_HibernateIntStatus() function to read the raw interrupt status to determine the cause of the wake. The ROM_HibernateDataGet() function can be used to restore state. These combinations of functions can be used by the software to determine if the processor is waking from hibernation and the appropriate action to take as a result.

Returns:
Returns **true** if the module is already active, and **false** if not.

11.2.1.14 ROM_HibernateLowBatGet

Gets the currently configured low battery detection behavior.

Prototype:
```
unsigned long
ROM_HibernateLowBatGet(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateLowBatGet` is a function pointer located at `ROM_HIBERNATETABLE[9]`.

Description:
Returns a value representing the currently configured low battery detection behavior. The return value is one of the following:

- **HIBERNATE_LOW_BAT_DETECT** - detect a low battery condition.
- **HIBERNATE_LOW_BAT_ABORT** - detect a low battery condition, and abort hibernation if low battery is detected.

Returns:
Returns a value indicating the configured low battery detection.

11.2.1.15 ROM_HibernateLowBatSet

Configures the low battery detection.

Prototype:
```
void
ROM_HibernateLowBatSet(unsigned long ulLowBatFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateLowBatSet` is a function pointer located at `ROM_HIBERNATETABLE[8]`.

Parameters:
ulLowBatFlags specifies behavior of low battery detection.

Description:

Enables the low battery detection and whether hibernation is allowed if a low battery is detected. If low battery detection is enabled, then a low battery condition is indicated in the raw interrupt status register, and can also trigger an interrupt. Optionally, hibernation can be aborted if a low battery is detected.

The *ulLowBatFlags* parameter is one of the following values:

- **HIBERNATE_LOW_BAT_DETECT** - detect a low battery condition.
- **HIBERNATE_LOW_BAT_ABORT** - detect a low battery condition, and abort hibernation if low battery is detected.

Returns:

None.

11.2.1.16 ROM_HibernateRequest

Requests hibernation mode.

Prototype:

```
void
ROM_HibernateRequest(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRequest` is a function pointer located at `ROM_HIBERNATETABLE[20]`.

Description:

This function requests the Hibernation module to disable the external regulator, thus removing power from the processor and all peripherals. The Hibernation module will remain powered from the battery or auxiliary power supply.

The Hibernation module will re-enable the external regulator when one of the configured wake conditions occurs (such as RTC match or external **WAKE** pin). When the power is restored the processor will go through a normal power-on reset. The processor can retrieve saved state information with the ROM_HibernateDataGet() function. Prior to calling the function to request hibernation mode, the conditions for waking must have already been set by using the ROM_HibernateWakeSet() function.

Note that this function may return because some time may elapse before the power is actually removed, or it may not be removed at all. For this reason, the processor will continue to execute instructions for some time and the caller should be prepared for this function to return. There are various reasons why the power may not be removed. For example, if the ROM_HibernateLowBatSet() function was used to configure an abort if low battery is detected, then the power will not be removed if the battery voltage is too low. There may be other reasons, related to the external circuit design, that a request for hibernation may not actually occur.

For all these reasons, the caller must be prepared for this function to return. The simplest way to handle it is to just enter an infinite loop and wait for the power to be removed.

Returns:

None.

11.2.1.17 ROM_HibernateRTCDisable

Disables the RTC feature of the Hibernation module.

Prototype:

```
void
ROM_HibernateRTCDisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCDisable` is a function pointer located at `ROM_HIBERNATETABLE[5]`.

Description:

Disables the RTC in the Hibernation module. After calling this function the RTC features of the Hibernation module will not be available.

Returns:

None.

11.2.1.18 ROM_HibernateRTCEnable

Enables the RTC feature of the Hibernation module.

Prototype:

```
void
ROM_HibernateRTCEnable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCEnable` is a function pointer located at `ROM_HIBERNATETABLE[4]`.

Description:

Enables the RTC in the Hibernation module. The RTC can be used to wake the processor from hibernation at a certain time, or to generate interrupts at certain times. This function must be called before using any of the RTC features of the Hibernation module.

Returns:

None.

11.2.1.19 ROM_HibernateRTCGet

Gets the value of the real time clock (RTC) counter.

Prototype:

```
unsigned long
ROM_HibernateRTCGet(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCGet` is a function pointer located at `ROM_HIBERNATETABLE[11]`.

Description:
Gets the value of the RTC and returns it to the caller.

Returns:
Returns the value of the RTC.

11.2.1.20 ROM_HibernateRTCMatch0Get

Gets the value of the RTC match 0 register.

Prototype:
```
unsigned long
ROM_HibernateRTCMatch0Get(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCMatch0Get` is a function pointer located at `ROM_HIBERNATETABLE[13]`.

Description:
Gets the value of the match 0 register for the RTC.

Returns:
Returns the value of the match register.

11.2.1.21 ROM_HibernateRTCMatch0Set

Sets the value of the RTC match 0 register.

Prototype:
```
void
ROM_HibernateRTCMatch0Set(unsigned long ulMatch)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCMatch0Set` is a function pointer located at `ROM_HIBERNATETABLE[12]`.

Parameters:
ulMatch is the value for the match register.

Description:
Sets the match 0 register for the RTC. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match register.

Returns:
None.

11.2.1.22 ROM_HibernateRTCSet

Sets the value of the real time clock (RTC) counter.

Prototype:

```
void
ROM_HibernateRTCSet(unsigned long ulRTCValue)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCSet` is a function pointer located at `ROM_HIBERNATETABLE[10]`.

Parameters:

ulRTCValue is the new value for the RTC.

Description:

Sets the value of the RTC. The RTC will count seconds if the hardware is configured correctly. The RTC must be enabled by calling ROM_HibernateRTCEnable() before calling this function.

Returns:

None.

11.2.1.23 ROM_HibernateRTCSSGet

Returns the current value of the RTC sub second count.

Prototype:

```
unsigned long
ROM_HibernateRTCSSGet(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCSSGet` is a function pointer located at `ROM_HIBERNATETABLE[27]`.

Description:

This function will return the current value of the sub second count for the for the RTC in 1/32768 of a second increments.

Returns:

The current RTC sub second count in 1/32768 seconds.

11.2.1.24 ROM_HibernateRTCSSMatch0Get

Returns the value of the RTC sub second match 0 register.

Prototype:

```
unsigned long
ROM_HibernateRTCSSMatch0Get(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCSSMatch0Get` is a function pointer located at `ROM_HIBERNATETABLE[26]`.

Description:
This function returns the current value of the sub second match 0 register for the RTC. The value returned is in 1/32768 second increments.

Returns:
Returns the value of the sub section match register.

11.2.1.25 ROM_HibernateRTCSSMatch0Set

Sets the value of the RTC sub second match 0 register.

Prototype:
```
void
ROM_HibernateRTCSSMatch0Set(unsigned long ulMatch)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCSSMatch0Set` is a function pointer located at `ROM_HIBERNATETABLE[25]`.

Parameters:
ulMatch is the value for the sub second match register.

Description:
Sets the sub second match 0 register for the RTC in 1/32768 of a second increments. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match combined with the sub second match register.

Returns:
None.

11.2.1.26 ROM_HibernateRTCTrimGet

Gets the value of the RTC predivider trim register.

Prototype:
```
unsigned long
ROM_HibernateRTCTrimGet(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCTrimGet` is a function pointer located at `ROM_HIBERNATETABLE[17]`.

Description:
Gets the value of the pre-divider trim register. This function can be used to get the current value of the trim register prior to making an adjustment by using the ROM_HibernateRTCTrimSet() function.

Returns:
None.

11.2.1.27 ROM_HibernateRTCTrimSet

Sets the value of the RTC predivider trim register.

Prototype:
```
void
ROM_HibernateRTCTrimSet(unsigned long ulTrim)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateRTCTrimSet` is a function pointer located at `ROM_HIBERNATETABLE[16]`.

Parameters:
ulTrim is the new value for the pre-divider trim register.

Description:
Sets the value of the pre-divider trim register. The input time source is divided by the pre-divider to achieve a one-second clock rate. Once every 64 seconds, the value of the pre-divider trim register is applied to the predivider to allow fine-tuning of the RTC rate, in order to make corrections to the rate. The software application can make adjustments to the predivider trim register to account for variations in the accuracy of the input time source. The nominal value is 0x7FFF, and it can be adjusted up or down in order to fine-tune the RTC rate.

Returns:
None.

11.2.1.28 ROM_HibernateWakeGet

Gets the currently configured wake conditions for the Hibernation module.

Prototype:
```
unsigned long
ROM_HibernateWakeGet(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateWakeGet` is a function pointer located at `ROM_HIBERNATETABLE[7]`.

Description:
Returns the flags representing the wake configuration for the Hibernation module. The return value is a combination of the following flags:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted.
- **HIBERNATE_WAKE_RTC** - wake when one of the RTC matches occurs.

Returns:
Returns flags indicating the configured wake conditions.

11.2.1.29 ROM_HibernateWakeSet

Configures the wake conditions for the Hibernation module.

Prototype:
```
void
ROM_HibernateWakeSet(unsigned long ulWakeFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_HIBERNATETABLE` is an array of pointers located at `ROM_APITABLE[19]`.
`ROM_HibernateWakeSet` is a function pointer located at `ROM_HIBERNATETABLE[6]`.

Parameters:
ulWakeFlags specifies which conditions should be used for waking.

Description:
Enables the conditions under which the Hibernation module will wake. The *ulWakeFlags* parameter is the logical OR of any combination of the following:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted.
- **HIBERNATE_WAKE_RTC** - wake when one of the RTC matches occurs.

Returns:
None.

12 Inter-Integrated Circuit (I2C)

Introduction 129
Functions 130

12.1 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.

12.1.1 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.

12.1.2 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.

12.2 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_I2CMasterInitExpClk (unsigned long ulBase, unsigned long ulI2CClk, tBoolean bFast)
- void ROM_I2CMasterIntClear (unsigned long ulBase)
- void ROM_I2CMasterIntClearEx (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2CMasterIntDisable (unsigned long ulBase)
- void ROM_I2CMasterIntDisableEx (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2CMasterIntEnable (unsigned long ulBase)
- void ROM_I2CMasterIntEnableEx (unsigned long ulBase, unsigned long ulIntFlags)
- tBoolean ROM_I2CMasterIntStatus (unsigned long ulBase, tBoolean bMasked)
- unsigned long ROM_I2CMasterIntStatusEx (unsigned long ulBase, tBoolean bMasked)

- unsigned long ROM_I2CMasterLineStateGet (unsigned long ulBase)
- void ROM_I2CMasterSlaveAddrSet (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)
- void ROM_I2CMasterTimeoutSet (unsigned long ulBase, unsigned long ulValue)
- void ROM_I2CSlaveACKOverride (unsigned long ulBase, tBoolean bEnable)
- void ROM_I2CSlaveACKValueSet (unsigned long ulBase, tBoolean bACK)
- void ROM_I2CSlaveAddressSet (unsigned long ulBase, unsigned char ucAddrNum, unsigned char ucSlaveAddr)
- 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_I2CSlaveIntClearEx (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2CSlaveIntDisable (unsigned long ulBase)
- void ROM_I2CSlaveIntDisableEx (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2CSlaveIntEnable (unsigned long ulBase)
- void ROM_I2CSlaveIntEnableEx (unsigned long ulBase, unsigned long ulIntFlags)
- tBoolean ROM_I2CSlaveIntStatus (unsigned long ulBase, tBoolean bMasked)
- unsigned long ROM_I2CSlaveIntStatusEx (unsigned long ulBase, tBoolean bMasked)
- unsigned long ROM_I2CSlaveStatus (unsigned long ulBase)
- void ROM_UpdateI2C (void)

12.2.1 Function Documentation

12.2.1.1 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**.

12.2.1.2 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**.

12.2.1.3 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.

12.2.1.4 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.

12.2.1.5 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.

12.2.1.6 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.

12.2.1.7 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.

12.2.1.8 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.

Returns:

Returns the error status, as one of **I2C_MASTER_ERR_NONE**, **I2C_MASTER_ERR_ADDR_ACK**, **I2C_MASTER_ERR_DATA_ACK**, or **I2C_MASTER_ERR_ARB_LOST**.

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

The peripheral clock is the same as the processor clock. This is 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.

12.2.1.10 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:

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

Returns:

None.

12.2.1.11 ROM_I2CMasterIntClearEx

Clears I2C Master interrupt sources.

Prototype:

```
void
ROM_I2CMasterIntClearEx(unsigned long ulBase,
                        unsigned long ulIntFlags)
```

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_I2CMasterIntClearEx` is a function pointer located at `ROM_I2CTABLE[32]`.

Parameters:

ulBase is the base address of the I2C Master module.

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

Description:

The specified I2C Master 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 has the same definition as the *ulIntFlags* parameter to I2CMasterIntEnableEx().

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

Returns:
None.

12.2.1.12 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.

12.2.1.13 ROM_I2CMasterIntDisableEx

Disables individual I2C Master interrupt sources.

Prototype:
```
void
ROM_I2CMasterIntDisableEx(unsigned long ulBase,
                          unsigned long ulIntFlags)
```

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_I2CMasterIntDisableEx` is a function pointer located at `ROM_I2CTABLE[30]`.

Parameters:
ulBase is the base address of the I2C Master module.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

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

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to I2CMasterIntEnableEx().

Returns:

None.

12.2.1.14 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.

12.2.1.15 ROM_I2CMasterIntEnableEx

Enables individual I2C Master interrupt sources.

Prototype:

```
void
ROM_I2CMasterIntEnableEx(unsigned long ulBase,
                         unsigned long ulIntFlags)
```

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_I2CMasterIntEnableEx` is a function pointer located at `ROM_I2CTABLE[29]`.

Parameters:

ulBase is the base address of the I2C Master module.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

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

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

- **I2C_MASTER_INT_TIMEOUT** - Clock Timeout interrupt
- **I2C_MASTER_INT_DATA** - Data interrupt

Returns:
None.

12.2.1.16 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.

12.2.1.17 ROM_I2CMasterIntStatusEx

Gets the current I2C Master interrupt status.

Prototype:
```
unsigned long
ROM_I2CMasterIntStatusEx(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_I2CMasterIntStatusEx` is a function pointer located at `ROM_I2CTABLE[31]`.

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:

Returns the current interrupt status, enumerated as a bit field of values described in I2CMasterIntEnableEx().

12.2.1.18 ROM_I2CMasterLineStateGet

Reads the state of the SDA and SCL pins.

Prototype:

```
unsigned long
ROM_I2CMasterLineStateGet(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_I2CMasterLineStateGet` is a function pointer located at `ROM_I2CTABLE[38]`.

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This function returns the state of the I2C bus by providing the real time values of the SDA and SCL pins.

Returns:

Returns the state of the bus with SDA in bit position 1 and SCL in bit position 0.

12.2.1.19 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.

12.2.1.20 ROM_I2CMasterTimeoutSet

Sets the Master clock timeout value.

Prototype:

```
void
ROM_I2CMasterTimeoutSet(unsigned long ulBase,
                        unsigned long ulValue)
```

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_I2CMasterTimeoutSet` is a function pointer located at `ROM_I2CTABLE[33]`.

Parameters:

ulBase is the base address of the I2C Master module.

ulValue is the number of I2C clocks before the timeout is asserted.

Description:

This function enables and configures the clock low timeout feature in the I2C peripheral. This feature is implemented as a 12-bit counter, with the upper 8-bits being programmable. For example, to program a timeout of 20ms with a 100kHz SCL frequency, *ulValue* would be 0x7d.

Returns:

None.

12.2.1.21 ROM_I2CSlaveACKOverride

Configures ACK override behavior of the I2C Slave.

Prototype:

```
void
ROM_I2CSlaveACKOverride(unsigned long ulBase,
                        tBoolean bEnable)
```

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_I2CSlaveACKOverride is a function pointer located at ROM_I2CTABLE[34].

Parameters:
ulBase is the base address of the I2C Slave module.
bEnable enables or disables ACK override.

Description:
This function enables or disables ACK override, allowing the user application to drive the value on SDA during the ACK cycle.

Returns:
None.

12.2.1.22 ROM_I2CSlaveACKValueSet

Writes the ACK value.

Prototype:
```
void
ROM_I2CSlaveACKValueSet(unsigned long ulBase,
                        tBoolean bACK)
```

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_I2CSlaveACKValueSet is a function pointer located at ROM_I2CTABLE[35].

Parameters:
ulBase is the base address of the I2C Slave module.
bACK chooses whether to ACK (true) or NACK (false) the transfer.

Description:
This function puts the desired ACK value on SDA during the ACK cycle. The value written is only valid when ACK override is enabled using ROM_I2CSlaveACKOverride().

Returns:
None.

12.2.1.23 ROM_I2CSlaveAddressSet

Sets the I2C slave address.

Prototype:
```
void
ROM_I2CSlaveAddressSet(unsigned long ulBase,
                       unsigned char ucAddrNum,
                       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_I2CSlaveAddressSet is a function pointer located at ROM_I2CTABLE[37].

Parameters:
ulBase is the base address of the I2C Slave module.
ucAddrNum determines which slave address is set.
ucSlaveAddr 7-bit slave address

Description:
This function writes the specified slave address. The *ulAddrNum* field dictates which slave address is configured. For example, a value of 0 configures the primary address and a value of 1 the secondary.

Returns:
None.

12.2.1.24 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.

12.2.1.25 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.

12.2.1.26 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.

12.2.1.27 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.

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

Returns:
None.

12.2.1.29 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:
Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:
None.

12.2.1.30 ROM_I2CSlaveIntClearEx

Clears I2C Slave interrupt sources.

Prototype:
```
void
ROM_I2CSlaveIntClearEx(unsigned long ulBase,
                       unsigned long ulIntFlags)
```

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_I2CSlaveIntClearEx` is a function pointer located at `ROM_I2CTABLE[28]`.

Parameters:
ulBase is the base address of the I2C Slave module.
ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:
The specified I2C Slave 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 has the same definition as the *ulIntFlags* parameter to ROM_I2CSlaveIntEnableEx().

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

Returns:
None.

12.2.1.31 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.

12.2.1.32 ROM_I2CSlaveIntDisableEx

Disables individual I2C Slave interrupt sources.

Prototype:

```
void
ROM_I2CSlaveIntDisableEx(unsigned long ulBase,
                         unsigned long ulIntFlags)
```

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_I2CSlaveIntDisableEx` is a function pointer located at `ROM_I2CTABLE[26]`.

Parameters:

ulBase is the base address of the I2C Slave module.
ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

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

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to ROM_I2CSlaveIntEnableEx().

Returns:

None.

12.2.1.33 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.

12.2.1.34 ROM_I2CSlaveIntEnableEx

Enables individual I2C Slave interrupt sources.

Prototype:
```
void
ROM_I2CSlaveIntEnableEx(unsigned long ulBase,
                        unsigned long ulIntFlags)
```

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_I2CSlaveIntEnableEx` is a function pointer located at `ROM_I2CTABLE[25]`.

Parameters:
ulBase is the base address of the I2C Slave module.
ulIntFlags is the bit mask of the interrupt sources to be enabled.

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

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

- **I2C_SLAVE_INT_STOP** - Stop condition detected interrupt
- **I2C_SLAVE_INT_START** - Start condition detected interrupt
- **I2C_SLAVE_INT_DATA** - Data interrupt

Returns:
None.

12.2.1.35 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.

12.2.1.36 ROM_I2CSlaveIntStatusEx

Gets the current I2C Slave interrupt status.

Prototype:

```
unsigned long
ROM_I2CSlaveIntStatusEx(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_I2CSlaveIntStatusEx` is a function pointer located at `ROM_I2CTABLE[27]`.

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:

Returns the current interrupt status, enumerated as a bit field of values described in ROM_I2CSlaveIntEnableEx().

12.2.1.37 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**
- **I2C_SLAVE_ACT_OWN2SEL**

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.

12.2.1.38 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.

13 Interrupt Controller (NVIC)

Introduction 151
Functions 151

13.1 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.

13.2 Functions

Functions

- void ROM_IntDisable (unsigned long ulInterrupt)
- void ROM_IntEnable (unsigned long ulInterrupt)
- tBoolean ROM_IntMasterDisable (void)
- tBoolean ROM_IntMasterEnable (void)
- void ROM_IntPendClear (unsigned long ulInterrupt)
- void ROM_IntPendSet (unsigned long ulInterrupt)

- long ROM_IntPriorityGet (unsigned long ulInterrupt)
- unsigned long ROM_IntPriorityGroupingGet (void)
- void ROM_IntPriorityGroupingSet (unsigned long ulBits)
- unsigned long ROM_IntPriorityMaskGet (void)
- void ROM_IntPriorityMaskSet (unsigned long ulPriorityMask)
- void ROM_IntPrioritySet (unsigned long ulInterrupt, unsigned char ucPriority)

13.2.1 Function Documentation

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 ROM_IntMasterDisable

Disables the processor interrupt.

Prototype:
```
tBoolean
ROM_IntMasterDisable(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_IntMasterDisable` is a function pointer located at `ROM_INTERRUPTTABLE[2]`.

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

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

13.2.1.4 ROM_IntMasterEnable

Enables the processor interrupt.

Prototype:
```
tBoolean
ROM_IntMasterEnable(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_IntMasterEnable` is a function pointer located at `ROM_INTERRUPTTABLE[1]`.

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

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

13.2.1.5 ROM_IntPendClear

Unpends an interrupt.

Prototype:

```
void
ROM_IntPendClear(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_IntPendClear` is a function pointer located at `ROM_INTERRUPTTABLE[9]`.

Parameters:

ulInterrupt specifies the interrupt to be unpended.

Description:

The specified interrupt is unpended in the interrupt controller. This will cause any previously generated interrupts that have not been handled yet (due to higher priority interrupts or the interrupt no having been enabled yet) to be discarded.

Returns:

None.

13.2.1.6 ROM_IntPendSet

Pends an interrupt.

Prototype:

```
void
ROM_IntPendSet(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_IntPendSet` is a function pointer located at `ROM_INTERRUPTTABLE[8]`.

Parameters:

ulInterrupt specifies the interrupt to be pended.

Description:

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

Returns:

None.

13.2.1.7 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.

13.2.1.8 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.

13.2.1.9 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.

13.2.1.10 ROM_IntPriorityMaskGet

Gets the priority masking level

Prototype:

```
unsigned long
ROM_IntPriorityMaskGet(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_IntPriorityMaskGet is a function pointer located at ROM_INTERRUPTTABLE[11].

Description:

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

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 will allow interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater is blocked.

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.

Returns:

Returns the value of the interrupt priority level mask.

13.2.1.11 ROM_IntPriorityMaskSet

Sets the priority masking level

Prototype:

```
void
ROM_IntPriorityMaskSet(unsigned long ulPriorityMask)
```

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_IntPriorityMaskSet` is a function pointer located at `ROM_INTERRUPTTABLE[10]`.

Parameters:

ulPriorityMask is the priority level that is masked.

Description:

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

Smaller numbers correspond to higher interrupt priorities. So for example a priority level mask of 4 will allow interrupts of priority level 0-3, and interrupts with a numerical priority of 4 and greater is blocked.

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.

Returns:

None.

13.2.1.12 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.

14 Memory Protection Unit (MPU)

Introduction 159
Functions 160

14.1 Introduction

The Memory Protection Unit (MPU) API provides functions to configure the MPU. The MPU is tightly coupled to the Cortex-M3 processor core and provides a means to establish access permissions on regions of memory.

Up to eight memory regions can be defined. Each region has a base address and a size. The size is specified as a power of 2 between 32 bytes and 4 GB, inclusive. The region's base address must be aligned to the size of the region. Each region also has access permissions. Code execution can be allowed or disallowed for a region. A region can be set for read-only access, read/write access, or no access for both privileged and user modes. This can be used to set up an environment where only kernel or system code can access certain hardware registers or sections of code.

The MPU creates 8 sub-regions within each region. Any sub-region or combination of sub-regions can be disabled, allowing creation of "holes" or complex overlaying regions with different permissions. The sub-regions can also be used to create an unaligned beginning or ending of a region by disabling one or more of the leading or trailing sub-regions.

Once the regions are defined and the MPU is enabled, any access violation of a region will cause a memory management fault, and the fault handler will be activated.

Generally, the memory protection regions should be defined before enabling the MPU. The regions can be configured by calling ROM_MPURegionSet() once for each region to be configured.

A region that is defined by ROM_MPURegionSet() can be initially enabled or disabled. If the region is not initially enabled, it can be enabled later by calling ROM_MPURegionEnable(). An enabled region can be disabled by calling ROM_MPURegionDisable(). When a region is disabled, its configuration is preserved as long as it is not overwritten. In this case it can be enabled again with ROM_MPURegionEnable() without the need to reconfigure the region.

Care must be taken when setting up a protection region using ROM_MPURegionSet(). The function will write to multiple registers and is not protected from interrupts. Therefore, it is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to protect against this is to make sure that a region is always disabled before making any changes. Otherwise, it is up to the caller to ensure that ROM_MPURegionSet() is always called from within code that cannot be interrupted, or from code that will not be affected if an interrupt occurs while the region attributes are being changed.

The attributes of a region that has already been programmed can be retrieved and saved using the ROM_MPURegionGet() function. This function is intended to save the attributes in a format that can be used later to reload the region using the ROM_MPURegionSet() function. Note that the enable state of the region is saved with the attributes and will take effect when the region is reloaded.

When one or more regions are defined, the MPU can be enabled by calling ROM_MPUEnable(). This turns on the MPU and also defines the behavior in privileged mode and in the Hard Fault and NMI fault handlers. The MPU can be configured so that when in privileged mode and no regions are

enabled, a default memory map is applied. If this feature is not enabled, then a memory management fault is generated if the MPU is enabled and no regions are configured and enabled. The MPU can also be set to use a default memory map when in the Hard Fault or NMI handlers, instead of using the configured regions. All of these features are selected when calling ROM_MPUEnable(). When the MPU is enabled, it can be disabled by calling ROM_MPUDisable().

14.2 Functions

Functions

- void ROM_MPUDisable (void)
- void ROM_MPUEnable (unsigned long ulMPUConfig)
- unsigned long ROM_MPURegionCountGet (void)
- void ROM_MPURegionDisable (unsigned long ulRegion)
- void ROM_MPURegionEnable (unsigned long ulRegion)
- void ROM_MPURegionGet (unsigned long ulRegion, unsigned long *pulAddr, unsigned long *pulFlags)
- void ROM_MPURegionSet (unsigned long ulRegion, unsigned long ulAddr, unsigned long ulFlags)

14.2.1 Function Documentation

14.2.1.1 ROM_MPUDisable

Disables the MPU for use.

Prototype:

```
void
ROM_MPUDisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPUDisable` is a function pointer located at `ROM_MPUTABLE[1]`.

Description:

This function disables the Cortex-M3 memory protection unit. When the MPU is disabled, the default memory map is used and memory management faults are not generated.

Returns:

None.

14.2.1.2 ROM_MPUEnable

Enables and configures the MPU for use.

Prototype:

```
void
ROM_MPUEnable(unsigned long ulMPUConfig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPUEnable` is a function pointer located at `ROM_MPUTABLE[0]`.

Parameters:

ulMPUConfig is the logical OR of the possible configurations.

Description:

This function enables the Cortex-M3 memory protection unit. It also configures the default behavior when in privileged mode and while handling a hard fault or NMI. Prior to enabling the MPU, at least one region must be set by calling ROM_MPURegionSet() or else by enabling the default region for privileged mode by passing the **MPU_CONFIG_PRIV_DEFAULT** flag to ROM_MPUEnable(). Once the MPU is enabled, a memory management fault is generated for any memory access violations.

The *ulMPUConfig* parameter should be the logical OR of any of the following:

- **MPU_CONFIG_PRIV_DEFAULT** enables the default memory map when in privileged mode and when no other regions are defined. If this option is not enabled, then there must be at least one valid region already defined when the MPU is enabled.
- **MPU_CONFIG_HARDFLT_NMI** enables the MPU while in a hard fault or NMI exception handler. If this option is not enabled, then the MPU is disabled while in one of these exception handlers and the default memory map is applied.
- **MPU_CONFIG_NONE** chooses none of the above options. In this case, no default memory map is provided in privileged mode, and the MPU will not be enabled in the fault handlers.

Returns:

None.

14.2.1.3 ROM_MPURegionCountGet

Gets the count of regions supported by the MPU.

Prototype:

```
unsigned long
ROM_MPURegionCountGet(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPURegionCountGet` is a function pointer located at `ROM_MPUTABLE[2]`.

Description:

This function is used to get the number of regions that are supported by the MPU. This is the total number that are supported, including regions that are already programmed.

Returns:
The number of memory protection regions that are available for programming using ROM_MPURegionSet().

14.2.1.4 ROM_MPURegionDisable

Disables a specific region.

Prototype:
```
void
ROM_MPURegionDisable(unsigned long ulRegion)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPURegionDisable` is a function pointer located at `ROM_MPUTABLE[4]`.

Parameters:
ulRegion is the region number to disable.

Description:
This function is used to disable a previously enabled memory protection region. The region will remain configured if it is not overwritten with another call to ROM_MPURegionSet(), and can be enabled again by calling ROM_MPURegionEnable().

Returns:
None.

14.2.1.5 ROM_MPURegionEnable

Enables a specific region.

Prototype:
```
void
ROM_MPURegionEnable(unsigned long ulRegion)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPURegionEnable` is a function pointer located at `ROM_MPUTABLE[3]`.

Parameters:
ulRegion is the region number to enable.

Description:
This function is used to enable a memory protection region. The region should already be set up with the ROM_MPURegionSet() function. Once enabled, the memory protection rules of the region are applied and access violations will cause a memory management fault.

Returns:
None.

14.2.1.6 ROM_MPURegionGet

Gets the current settings for a specific region.

Prototype:

```
void
ROM_MPURegionGet(unsigned long ulRegion,
                 unsigned long *pulAddr,
                 unsigned long *pulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPURegionGet` is a function pointer located at `ROM_MPUTABLE[6]`.

Parameters:

ulRegion is the region number to get.

pulAddr points to storage for the base address of the region.

pulFlags points to the attribute flags for the region.

Description:

This function retrieves the configuration of a specific region. The meanings and format of the parameters is the same as that of the ROM_MPURegionSet() function.

This function can be used to save the configuration of a region for later use with the ROM_MPURegionSet() function. The region's enable state is preserved in the attributes that are saved.

Returns:

None.

14.2.1.7 ROM_MPURegionSet

Sets up the access rules for a specific region.

Prototype:

```
void
ROM_MPURegionSet(unsigned long ulRegion,
                 unsigned long ulAddr,
                 unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_MPUTABLE` is an array of pointers located at `ROM_APITABLE[20]`.
`ROM_MPURegionSet` is a function pointer located at `ROM_MPUTABLE[5]`.

Parameters:

ulRegion is the region number to set up.

ulAddr is the base address of the region. It must be aligned according to the size of the region specified in ulFlags.

ulFlags is a set of flags to define the attributes of the region.

Description:

This function sets up the protection rules for a region. The region has a base address and a set of attributes including the size, which must be a power of 2. The base address parameter, *ulAddr*, must be aligned according to the size.

The *ulFlags* parameter is the logical OR of all of the attributes of the region. It is a combination of choices for region size, execute permission, read/write permissions, disabled sub-regions, and a flag to determine if the region is enabled.

The size flag determines the size of a region, and must be one of the following:

- **MPU_RGN_SIZE_32B**
- **MPU_RGN_SIZE_64B**
- **MPU_RGN_SIZE_128B**
- **MPU_RGN_SIZE_256B**
- **MPU_RGN_SIZE_512B**
- **MPU_RGN_SIZE_1K**
- **MPU_RGN_SIZE_2K**
- **MPU_RGN_SIZE_4K**
- **MPU_RGN_SIZE_8K**
- **MPU_RGN_SIZE_16K**
- **MPU_RGN_SIZE_32K**
- **MPU_RGN_SIZE_64K**
- **MPU_RGN_SIZE_128K**
- **MPU_RGN_SIZE_256K**
- **MPU_RGN_SIZE_512K**
- **MPU_RGN_SIZE_1M**
- **MPU_RGN_SIZE_2M**
- **MPU_RGN_SIZE_4M**
- **MPU_RGN_SIZE_8M**
- **MPU_RGN_SIZE_16M**
- **MPU_RGN_SIZE_32M**
- **MPU_RGN_SIZE_64M**
- **MPU_RGN_SIZE_128M**
- **MPU_RGN_SIZE_256M**
- **MPU_RGN_SIZE_512M**
- **MPU_RGN_SIZE_1G**
- **MPU_RGN_SIZE_2G**
- **MPU_RGN_SIZE_4G**

The execute permission flag must be one of the following:

- **MPU_RGN_PERM_EXEC** enables the region for execution of code
- **MPU_RGN_PERM_NOEXEC** disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

- **MPU_RGN_PERM_PRV_NO_USR_NO** - no access in privileged or user mode
- **MPU_RGN_PERM_PRV_RW_USR_NO** - privileged read/write, user no access
- **MPU_RGN_PERM_PRV_RW_USR_RO** - privileged read/write, user read-only

- **MPU_RGN_PERM_PRV_RW_USR_RW** - privileged read/write, user read/write
- **MPU_RGN_PERM_PRV_RO_USR_NO** - privileged read-only, user no access
- **MPU_RGN_PERM_PRV_RO_USR_RO** - privileged read-only, user read-only

The region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled. This allows for creation of "holes" in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

- **MPU_SUB_RGN_DISABLE_0**
- **MPU_SUB_RGN_DISABLE_1**
- **MPU_SUB_RGN_DISABLE_2**
- **MPU_SUB_RGN_DISABLE_3**
- **MPU_SUB_RGN_DISABLE_4**
- **MPU_SUB_RGN_DISABLE_5**
- **MPU_SUB_RGN_DISABLE_6**
- **MPU_SUB_RGN_DISABLE_7**

Finally, the region can be initially enabled or disabled with one of the following flags:

- **MPU_RGN_ENABLE**
- **MPU_RGN_DISABLE**

As an example, to set a region with the following attributes: size of 32 KB, execution enabled, read-only for both privileged and user, one sub-region disabled, and initially enabled; the *ulFlags* parameter would have the following value:

```
(MPU_RG_SIZE_32K | MPU_RGN_PERM_EXEC | MPU_RGN_PERM_PRV_RO_USR_RO |
MPU_SUB_RGN_DISABLE_2 | MPU_RGN_ENABLE)
```

Note:

This function will write to multiple registers and is not protected from interrupts. It is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to handle this is to disable a region before changing it. Refer to the discussion of this in the API Detailed Description section.

Returns:

None.

15 Pulse Width Modulator (PWM)

Introduction 167
Functions 167

15.1 Introduction

The PWM module provides up to four instances of a PWM generator block, and an output control block. Each generator block has two PWM output signals, which can be operated independently, or as a pair of signals with dead band delays inserted. Each generator block also has an interrupt output and a trigger output. The control block determines the polarity of the PWM signals, and which signals are passed through to the pins.

Some of the features of the PWM module are:

- Up to four generator blocks, each containing:
  - One 16-bit down or up/down counter
  - Two comparators
  - PWM generator
  - Dead band generator
- Control block
  - PWM output enable
  - Output polarity control
  - Synchronization
  - Fault handling
  - Interrupt status

When discussing the various components of the PWM module, the following conventions are used:

- The four generator blocks are called **Gen0**, **Gen1**, **Gen2**, and **Gen3**.
- The two PWM output signals associated with each generator block are called **OutA** and **OutB**.
- The eight output signals are called **PWM0**, **PWM1**, **PWM2**, **PWM3**, **PWM4**, **PWM5**, **PWM6**, and **PWM7**.
- **PWM0** and **PWM1** are associated with **Gen0**, **PWM2** and **PWM3** are associated with **Gen1**, **PWM4** and **PWM5** are associated with **Gen2**, and **PWM6** and **PWM7** are associated with **Gen3**.

Also, as a simplifying assumption for this API, comparator A for each generator block is used exclusively to adjust the pulse width of the even numbered PWM outputs (**PWM0**, **PWM2**, **PWM4**, and **PWM6**). In addition, comparator B is used exclusively for the odd numbered PWM outputs (**PWM1**, **PWM3**, **PWM5**, and **PWM7**).

15.2 Functions

Functions

- void ROM_PWMDeadBandDisable (unsigned long ulBase, unsigned long ulGen)

- void ROM_PWMDeadBandEnable (unsigned long ulBase, unsigned long ulGen, unsigned short usRise, unsigned short usFall)
- void ROM_PWMFaultIntClear (unsigned long ulBase)
- void ROM_PWMFaultIntClearExt (unsigned long ulBase, unsigned long ulFaultInts)
- void ROM_PWMGenConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulConfig)
- void ROM_PWMGenDisable (unsigned long ulBase, unsigned long ulGen)
- void ROM_PWMGenEnable (unsigned long ulBase, unsigned long ulGen)
- void ROM_PWMGenFaultClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup, unsigned long ulFaultTriggers)
- void ROM_PWMGenFaultConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulMinFaultPeriod, unsigned long ulFaultSenses)
- unsigned long ROM_PWMGenFaultStatus (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)
- unsigned long ROM_PWMGenFaultTriggerGet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)
- void ROM_PWMGenFaultTriggerSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup, unsigned long ulFaultTriggers)
- void ROM_PWMGenIntClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulInts)
- unsigned long ROM_PWMGenIntStatus (unsigned long ulBase, unsigned long ulGen, tBoolean bMasked)
- void ROM_PWMGenIntTrigDisable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- void ROM_PWMGenIntTrigEnable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- unsigned long ROM_PWMGenPeriodGet (unsigned long ulBase, unsigned long ulGen)
- void ROM_PWMGenPeriodSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulPeriod)
- void ROM_PWMIntDisable (unsigned long ulBase, unsigned long ulGenFault)
- void ROM_PWMIntEnable (unsigned long ulBase, unsigned long ulGenFault)
- unsigned long ROM_PWMIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_PWMOutputFault (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bFaultSuppress)
- void ROM_PWMOutputFaultLevel (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bDriveHigh)
- void ROM_PWMOutputInvert (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bInvert)
- void ROM_PWMOutputState (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bEnable)
- unsigned long ROM_PWMPulseWidthGet (unsigned long ulBase, unsigned long ulPWMOut)
- void ROM_PWMPulseWidthSet (unsigned long ulBase, unsigned long ulPWMOut, unsigned long ulWidth)
- void ROM_PWMSyncTimeBase (unsigned long ulBase, unsigned long ulGenBits)
- void ROM_PWMSyncUpdate (unsigned long ulBase, unsigned long ulGenBits)

15.2.1 Function Documentation

15.2.1.1 ROM_PWMDeadBandDisable

Disables the PWM dead band output.

Prototype:

```
void
ROM_PWMDeadBandDisable(unsigned long ulBase,
                       unsigned long ulGen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMDeadBandDisable` is a function pointer located at `ROM_PWMTABLE[8]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function disables the dead band mode for the specified PWM generator. Doing so decouples the **OutA** and **OutB** signals.

Returns:

None.

15.2.1.2 ROM_PWMDeadBandEnable

Enables the PWM dead band output, and sets the dead band delays.

Prototype:

```
void
ROM_PWMDeadBandEnable(unsigned long ulBase,
                      unsigned long ulGen,
                      unsigned short usRise,
                      unsigned short usFall)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMDeadBandEnable` is a function pointer located at `ROM_PWMTABLE[7]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

usRise specifies the width of delay from the rising edge.

usFall specifies the width of delay from the falling edge.

Description:

This function sets the dead bands for the specified PWM generator, where the dead bands are defined as the number of **PWM** clock ticks from the rising or falling edge of the generator's **OutA** signal. Note that this function causes the coupling of **OutB** to **OutA**.

Returns:

None.

15.2.1.3 ROM_PWMFaultIntClear

Clears the fault interrupt for a PWM module.

Prototype:

```
void
ROM_PWMFaultIntClear(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMFaultIntClear` is a function pointer located at `ROM_PWMTABLE[20]`.

Parameters:

ulBase is the base address of the PWM module.

Description:

Clears the fault interrupt by writing to the appropriate bit of the interrupt status register for the selected PWM module.

This function clears only the FAULT0 interrupt and is retained for backwards compatibility. It is recommended that ROM_PWMFaultIntClearExt() be used instead since it supports all fault interrupts supported on devices with and without extended PWM fault handling support.

Note:

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

Returns:

None.

15.2.1.4 ROM_PWMFaultIntClearExt

Clears the fault interrupt for a PWM module.

Prototype:

```
void
ROM_PWMFaultIntClearExt(unsigned long ulBase,
                        unsigned long ulFaultInts)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMFaultIntClearExt is a function pointer located at ROM_PWMTABLE[23].

Parameters:

ulBase is the base address of the PWM module.

ulFaultInts specifies the fault interrupts to clear.

Description:

Clears one or more fault interrupts by writing to the appropriate bit of the PWM interrupt status register. The parameter *ulFaultInts* must be the logical OR of any of **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

When running on a device supporting extended PWM fault handling, the fault interrupts are derived by performing a logical OR of each of the configured fault trigger signals for a given generator. Therefore, these interrupts are not directly related to the four possible FAULTn inputs to the device but indicate that a fault has been signaled to one of the four possible PWM generators. On a device without extended PWM fault handling, the interrupt is directly related to the state of the single FAULT pin.

Note:

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

Returns:

None.

15.2.1.5 ROM_PWMGenConfigure

Configures a PWM generator.

Prototype:

```
void
ROM_PWMGenConfigure(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulConfig)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMGenConfigure is a function pointer located at ROM_PWMTABLE[1].

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to configure. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulConfig is the configuration for the PWM generator.

Description:

This function is used to set the mode of operation for a PWM generator. The counting mode, synchronization mode, and debug behavior are all configured. After configuration, the generator is left in the disabled state.

A PWM generator can count in two different modes: count down mode or count up/down mode. In count down mode, it will count from a value down to zero, and then reset to the preset value. This will produce left-aligned PWM signals (that is the rising edge of the two PWM signals produced by the generator will occur at the same time). In count up/down mode, it will count up from zero to the preset value, count back down to zero, and then repeat the process. This will produce center-aligned PWM signals (that is, the middle of the high/low period of the PWM signals produced by the generator will occur at the same time).

When the PWM generator parameters (period and pulse width) are modified, their affect on the output PWM signals can be delayed. In synchronous mode, the parameter updates are not applied until a synchronization event occurs. This allows multiple parameters to be modified and take affect simultaneously, instead of one at a time. Additionally, parameters to multiple PWM generators in synchronous mode can be updated simultaneously, allowing them to be treated as if they were a unified generator. In non-synchronous mode, the parameter updates are not delayed until a synchronization event. In either mode, the parameter updates only occur when the counter is at zero to help prevent oddly formed PWM signals during the update (that is, a PWM pulse that is too short or too long).

The PWM generator can either pause or continue running when the processor is stopped via the debugger. If configured to pause, it will continue to count until it reaches zero, at which point it will pause until the processor is restarted. If configured to continue running, it will keep counting as if nothing had happened.

The *ulConfig* parameter contains the desired configuration. It is the logical OR of the following:

- **PWM_GEN_MODE_DOWN** or **PWM_GEN_MODE_UP_DOWN** to specify the counting mode
- **PWM_GEN_MODE_SYNC** or **PWM_GEN_MODE_NO_SYNC** to specify the counter load and comparator update synchronization mode
- **PWM_GEN_MODE_DBG_RUN** or **PWM_GEN_MODE_DBG_STOP** to specify the debug behavior
- **PWM_GEN_MODE_GEN_NO_SYNC**, **PWM_GEN_MODE_GEN_SYNC_LOCAL**, or **PWM_GEN_MODE_GEN_SYNC_GLOBAL** to specify the update synchronization mode for generator counting mode changes
- **PWM_GEN_MODE_DB_NO_SYNC**, **PWM_GEN_MODE_DB_SYNC_LOCAL**, or **PWM_GEN_MODE_DB_SYNC_GLOBAL** to specify the deadband parameter synchronization mode
- **PWM_GEN_MODE_FAULT_LATCHED** or **PWM_GEN_MODE_FAULT_UNLATCHED** to specify whether fault conditions are latched or not
- **PWM_GEN_MODE_FAULT_MINPER** or **PWM_GEN_MODE_FAULT_NO_MINPER** to specify whether minimum fault period support is required
- **PWM_GEN_MODE_FAULT_EXT** or **PWM_GEN_MODE_FAULT_LEGACY** to specify whether extended fault source selection support is enabled or not

Setting **PWM_GEN_MODE_FAULT_MINPER** allows an application to set the minimum duration of a PWM fault signal. Faults will be signaled for at least this time even if the external fault pin deasserts earlier. Care should be taken when using this mode since during the fault signal period, the fault interrupt from the PWM generator will remain asserted. The fault interrupt handler may, therefore, reenter immediately if it exits prior to expiration of the fault timer.

Note:
Changes to the counter mode will affect the period of the PWM signals produced. ROM_PWMGenPeriodSet() and ROM_PWMPulseWidthSet() should be called after any changes to the counter mode of a generator.

Returns:
None.

15.2.1.6 ROM_PWMGenDisable

Disables the timer/counter for a PWM generator block.

Prototype:
```
void
ROM_PWMGenDisable(unsigned long ulBase,
                  unsigned long ulGen)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenDisable` is a function pointer located at `ROM_PWMTABLE[5]`.

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to be disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:
This function blocks the PWM clock from driving the timer/counter for the specified generator block.

Returns:
None.

15.2.1.7 ROM_PWMGenEnable

Enables the timer/counter for a PWM generator block.

Prototype:
```
void
ROM_PWMGenEnable(unsigned long ulBase,
                 unsigned long ulGen)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenEnable` is a function pointer located at `ROM_PWMTABLE[4]`.

Parameters:
ulBase is the base address of the PWM module.

ulGen is the PWM generator to be enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:
This function allows the PWM clock to drive the timer/counter for the specified generator block.

Returns:
None.

15.2.1.8 ROM_PWMGenFaultClear

Clears one or more latched fault triggers for a given PWM generator.

Prototype:
```
void
ROM_PWMGenFaultClear(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulGroup,
                     unsigned long ulFaultTriggers)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenFaultClear` is a function pointer located at `ROM_PWMTABLE[28]`.

Parameters:
ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault trigger states are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

ulFaultTriggers is the set of fault triggers which are to be cleared.

Description:
This function allows an application to clear the fault triggers for a given PWM generator. This is only required if ROM_PWMGenConfigure() has previously been called with flag **PWM_GEN_MODE_LATCH_FAULT** in parameter *ulConfig*.

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
None.

15.2.1.9 ROM_PWMGenFaultConfigure

Configures the minimum fault period and fault pin senses for a given PWM generator.

Prototype:

```
void
ROM_PWMGenFaultConfigure(unsigned long ulBase,
                         unsigned long ulGen,
                         unsigned long ulMinFaultPeriod,
                         unsigned long ulFaultSenses)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMGenFaultConfigure is a function pointer located at ROM_PWMTABLE[24].

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault configuration is being set. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulMinFaultPeriod is the minimum fault active period expressed in PWM clock cycles.

ulFaultSenses indicates which sense of each FAULT input should be considered the "asserted" state. Valid values are logical OR combinations of **PWM_FAULTn_SENSE_HIGH** and **PWM_FAULTn_SENSE_LOW**.

Description:

This function sets the minimum fault period for a given generator along with the sense of each of the 4 possible fault inputs. The minimum fault period is expressed in PWM clock cycles and takes effect only if ROM_PWMGenConfigure() is called with flag **PWM_GEN_MODE_FAULT_PER** set in the *ulConfig* parameter. When a fault input is asserted, the minimum fault period timer ensures that it remains asserted for at least the number of clock cycles specified.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

15.2.1.10 ROM_PWMGenFaultStatus

Returns the current state of the fault triggers for a given PWM generator.

Prototype:

```
unsigned long
ROM_PWMGenFaultStatus(unsigned long ulBase,
                      unsigned long ulGen,
                      unsigned long ulGroup)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMGenFaultStatus is a function pointer located at ROM_PWMTABLE[27].

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault trigger states are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

Description:

This function allows an application to query the current state of each of the fault trigger inputs to a given PWM generator. The current state of each fault trigger input is returned unless ROM_PWMGenConfigure() has previously been called with flag **PWM_GEN_MODE_LATCH_FAULT** in the *ulConfig* parameter in which case the returned status is the latched fault trigger status.

If latched faults are configured, the application must call ROM_PWMGenFaultClear() to clear each trigger.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

Returns the current state of the fault triggers for the given PWM generator. A set bit indicates that the associated trigger is active. For **PWM_FAULT_GROUP_0**, the returned value is a logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**. For **PWM_FAULT_GROUP_1**, the return value is the logical OR of **PWM_FAULT_DCMP0**, **PWM_FAULT_DCMP1**, **PWM_FAULT_DCMP2**, **PWM_FAULT_DCMP3**, **PWM_FAULT_DCMP4**, **PWM_FAULT_DCMP5**, **PWM_FAULT_DCMP6**, or **PWM_FAULT_DCMP7**.

15.2.1.11 ROM_PWMGenFaultTriggerGet

Returns the set of fault triggers currently configured for a given PWM generator.

Prototype:

```
unsigned long
ROM_PWMGenFaultTriggerGet(unsigned long ulBase,
                          unsigned long ulGen,
                          unsigned long ulGroup)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenFaultTriggerGet` is a function pointer located at `ROM_PWMTABLE[26]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault triggers are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.

Description:

This function allows an application to query the current set of inputs that contribute towards the generation of a fault condition to a given PWM generator.

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
Returns the current fault triggers configured for the fault group provided. For **PWM_FAULT_GROUP_0**, the returned value is a logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**. For **PWM_FAULT_GROUP_1**, the return value is the logical OR of **PWM_FAULT_DCMP0**, **PWM_FAULT_DCMP1**, **PWM_FAULT_DCMP2**, **PWM_FAULT_DCMP3**, **PWM_FAULT_DCMP4**, **PWM_FAULT_DCMP5**, **PWM_FAULT_DCMP6**, or **PWM_FAULT_DCMP7**.

15.2.1.12 ROM_PWMGenFaultTriggerSet

Configures the set of fault triggers for a given PWM generator.

Prototype:
```
void
ROM_PWMGenFaultTriggerSet(unsigned long ulBase,
                          unsigned long ulGen,
                          unsigned long ulGroup,
                          unsigned long ulFaultTriggers)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenFaultTriggerSet` is a function pointer located at `ROM_PWMTABLE[25]`.

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator whose fault triggers are being set. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.
ulGroup indicates the subset of possible faults that are to be configured. This must be **PWM_FAULT_GROUP_0** or **PWM_FAULT_GROUP_1**.
ulFaultTriggers defines the set of inputs that are to contribute towards generation of the fault signal to the given PWM generator. For **PWM_FAULT_GROUP_0**, this is the logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**. For **PWM_FAULT_GROUP_1**, this is the logical OR of **PWM_FAULT_DCMP0**, **PWM_FAULT_DCMP1**, **PWM_FAULT_DCMP2**, **PWM_FAULT_DCMP3**, **PWM_FAULT_DCMP4**, **PWM_FAULT_DCMP5**, **PWM_FAULT_DCMP6**, or **PWM_FAULT_DCMP7**.

Description:
This function allows selection of the set of fault inputs that is combined to generate a fault condition to a given PWM generator. By default, all generators use only FAULT0 (for backwards compatibility) but if ROM_PWMGenConfigure() is called with flag **PWM_GEN_MODE_FAULT_SRC** in the *ulConfig* parameter, extended fault handling is enabled and this function must be called to configure the fault triggers.

The fault signal to the PWM generator is generated by ORing together each of the signals whose inputs are specified in the *ulFaultTriggers* parameter after having adjusted the sense of each FAULTn input based on the configuration previously set using a call to ROM_PWMGenFaultConfigure().

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
None.

15.2.1.13 ROM_PWMGenIntClear

Clears the specified interrupt(s) for the specified PWM generator block.

Prototype:
```
void
ROM_PWMGenIntClear(unsigned long ulBase,
                   unsigned long ulGen,
                   unsigned long ulInts)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenIntClear` is a function pointer located at `ROM_PWMTABLE[17]`.

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.
ulInts specifies the interrupts to be cleared.

Description:
Clears the specified interrupt(s) by writing a 1 to the specified bits of the interrupt status register for the specified PWM generator. The *ulInts* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, or **PWM_INT_CNT_BD**.

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

Returns:
None.

15.2.1.14 ROM_PWMGenIntStatus

Gets interrupt status for the specified PWM generator block.

Prototype:
```
unsigned long
ROM_PWMGenIntStatus(unsigned long ulBase,
```

```
                        unsigned long ulGen,
                        tBoolean bMasked)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenIntStatus` is a function pointer located at `ROM_PWMTABLE[16]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

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

Returns:

Returns the contents of the interrupt status register, or the contents of the raw interrupt status register, for the specified PWM generator.

15.2.1.15 ROM_PWMGenIntTrigDisable

Disables interrupts for the specified PWM generator block.

Prototype:

```
void
ROM_PWMGenIntTrigDisable(unsigned long ulBase,
                         unsigned long ulGen,
                         unsigned long ulIntTrig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenIntTrigDisable` is a function pointer located at `ROM_PWMTABLE[15]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulIntTrig specifies the interrupts and triggers to be disabled.

Description:

Masks the specified interrupt(s) and trigger(s) by clearing the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ulIntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

15.2.1.16 ROM_PWMGenIntTrigEnable

Enables interrupts and triggers for the specified PWM generator block.

Prototype:

```
void
ROM_PWMGenIntTrigEnable(unsigned long ulBase,
                        unsigned long ulGen,
                        unsigned long ulIntTrig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenIntTrigEnable` is a function pointer located at `ROM_PWMTABLE[14]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulIntTrig specifies the interrupts and triggers to be enabled.

Description:

Unmasks the specified interrupt(s) and trigger(s) by setting the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ulIntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

15.2.1.17 ROM_PWMGenPeriodGet

Gets the period of a PWM generator block.

Prototype:

```
unsigned long
ROM_PWMGenPeriodGet(unsigned long ulBase,
                    unsigned long ulGen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenPeriodGet` is a function pointer located at `ROM_PWMTABLE[3]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function gets the period of the specified PWM generator block. The period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

If the update of the counter for the specified PWM generator has yet to be completed, the value returned may not be the active period. The value returned is the programmed period, measured in PWM clock ticks.

Returns:

Returns the programmed period of the specified generator block in PWM clock ticks.

15.2.1.18 ROM_PWMGenPeriodSet

Set the period of a PWM generator.

Prototype:

```
void
ROM_PWMGenPeriodSet(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulPeriod)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMGenPeriodSet` is a function pointer located at `ROM_PWMTABLE[2]`.

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to be modified. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulPeriod specifies the period of PWM generator output, measured in clock ticks.

Description:

This function sets the period of the specified PWM generator block, where the period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

Note:

Any subsequent calls made to this function before an update occurs will cause the previous values to be overwritten.

Returns:

None.

15.2.1.19 ROM_PWMIntDisable

Disables generator and fault interrupts for a PWM module.

Prototype:

```
void
ROM_PWMIntDisable(unsigned long ulBase,
                  unsigned long ulGenFault)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMIntDisable is a function pointer located at ROM_PWMTABLE[19].

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be disabled. Must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

Masks the specified interrupt(s) by clearing the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

15.2.1.20 ROM_PWMIntEnable

Enables generator and fault interrupts for a PWM module.

Prototype:

```
void
ROM_PWMIntEnable(unsigned long ulBase,
                 unsigned long ulGenFault)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_PWMTABLE is an array of pointers located at ROM_APITABLE[8].
ROM_PWMIntEnable is a function pointer located at ROM_PWMTABLE[18].

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be enabled. Must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

Unmasks the specified interrupt(s) by setting the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

15.2.1.21 ROM_PWMIntStatus

Gets the interrupt status for a PWM module.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMIntStatus` is a function pointer located at `ROM_PWMTABLE[21]`.

Parameters:

ulBase is the base address of the PWM module.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

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

Returns:

The current interrupt status, enumerated as a bit field of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, and **PWM_INT_FAULT3**.

15.2.1.22 ROM_PWMOutputFault

Specifies the state of PWM outputs in response to a fault condition.

Prototype:

```
void
ROM_PWMOutputFault(unsigned long ulBase,
                   unsigned long ulPWMOutBits,
                   tBoolean bFaultSuppress)
```

ROM Location:

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

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bFaultSuppress determines if the signal is suppressed or passed through during an active fault condition.

Description:

This function sets the fault handling characteristics of the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bFaultSuppress* determines the fault handling characteristics for the selected outputs. If *bFaultSuppress* is **true**, then the selected outputs are made inactive. If *bFaultSuppress* is **false**, then the selected outputs are unaffected by the detected fault.

On devices supporting extended PWM fault handling, the state the affected output pins are driven to can be configured with ROM_PWMOutputFaultLevel(). If not configured, or if the device does not support extended PWM fault handling, affected outputs are driven low on a fault condition.

Returns:
None.

15.2.1.23 ROM_PWMOutputFaultLevel

Specifies the level of PWM outputs suppressed in response to a fault condition.

Prototype:
```
void
ROM_PWMOutputFaultLevel(unsigned long ulBase,
                        unsigned long ulPWMOutBits,
                        tBoolean bDriveHigh)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMOutputFaultLevel` is a function pointer located at `ROM_PWMTABLE[22]`.

Parameters:
ulBase is the base address of the PWM module.
ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.
bDriveHigh determines if the signal is driven high or low during an active fault condition.

Description:
This function determines whether a PWM output pin that is suppressed in response to a fault condition is driven high or low. The affected outputs are selected using the parameter *ulPWMOutBits*. The parameter *bDriveHigh* determines the output level for the pins identified by *ulPWMOutBits*. If *bDriveHigh* is **true** then the selected outputs are driven high when a fault is detected. If it is *false*, the pins are driven low.

In a fault condition, pins which have not been configured to be suppressed via a call to ROM_PWMOutputFault() are unaffected by this function.

Note:
This function is available only on devices which support extended PWM fault handling.

Returns:
None.

15.2.1.24 ROM_PWMOutputInvert

Selects the inversion mode for PWM outputs.

Prototype:

```
void
ROM_PWMOutputInvert(unsigned long ulBase,
                    unsigned long ulPWMOutBits,
                    tBoolean bInvert)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMOutputInvert` is a function pointer located at `ROM_PWMTABLE[12]`.

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bInvert determines if the signal is inverted or passed through.

Description:

This function is used to select the inversion mode for the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bInvert* determines the inversion mode for the selected outputs. If *bInvert* is **true**, this function will cause the specified PWM output signals to be inverted, or made active low. If *bInvert* is **false**, the specified outputs are passed through as is, or be made active high.

Returns:

None.

15.2.1.25 ROM_PWMOutputState

Enables or disables PWM outputs.

Prototype:

```
void
ROM_PWMOutputState(unsigned long ulBase,
                   unsigned long ulPWMOutBits,
                   tBoolean bEnable)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMOutputState` is a function pointer located at `ROM_PWMTABLE[11]`.

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bEnable determines if the signal is enabled or disabled.

Description:

This function is used to enable or disable the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bEnable* determines the state of the selected outputs. If *bEnable* is **true**, then the selected PWM outputs are enabled, or placed in the active state. If *bEnable* is **false**, then the selected outputs are disabled, or placed in the inactive state.

Returns:

None.

15.2.1.26 ROM_PWMPulseWidthGet

Gets the pulse width of a PWM output.

Prototype:

```
unsigned long
ROM_PWMPulseWidthGet(unsigned long ulBase,
                     unsigned long ulPWMOut)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMPulseWidthGet` is a function pointer located at `ROM_PWMTABLE[6]`.

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to query. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

Description:

This function gets the currently programmed pulse width for the specified PWM output. If the update of the comparator for the specified output has yet to be completed, the value returned may not be the active pulse width. The value returned is the programmed pulse width, measured in PWM clock ticks.

Returns:

Returns the width of the pulse in PWM clock ticks.

15.2.1.27 ROM_PWMPulseWidthSet

Sets the pulse width for the specified PWM output.

Prototype:

```
void
ROM_PWMPulseWidthSet(unsigned long ulBase,
                     unsigned long ulPWMOut,
                     unsigned long ulWidth)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMPulseWidthSet` is a function pointer located at `ROM_PWMTABLE[0]`.

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to modify. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

ulWidth specifies the width of the positive portion of the pulse.

Description:

This function sets the pulse width for the specified PWM output, where the pulse width is defined as the number of PWM clock ticks.

Note:

Any subsequent calls made to this function before an update occurs will cause the previous values to be overwritten.

Returns:

None.

15.2.1.28 ROM_PWMSyncTimeBase

Synchronizes the counters in one or multiple PWM generator blocks.

Prototype:

```
void
ROM_PWMSyncTimeBase(unsigned long ulBase,
                    unsigned long ulGenBits)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMSyncTimeBase` is a function pointer located at `ROM_PWMTABLE[10]`.

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be synchronized. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM module, this function synchronizes the time base of the generator blocks by causing the specified generator counters to be reset to zero.

Returns:

None.

15.2.1.29 ROM_PWMSyncUpdate

Synchronizes all pending updates.

Prototype:

```
void
ROM_PWMSyncUpdate(unsigned long ulBase,
                  unsigned long ulGenBits)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_PWMTABLE` is an array of pointers located at `ROM_APITABLE[8]`.
`ROM_PWMSyncUpdate` is a function pointer located at `ROM_PWMTABLE[9]`.

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be updated. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM generators, this function causes all queued updates to the period or pulse width to be applied the next time the corresponding counter becomes zero.

Returns:

None.

16 Quadrature Encoder (QEI)

Introduction 189
Functions 189

16.1 Introduction

The quadrature encoder API provides a set of functions for dealing with the Quadrature Encoder with Index (QEI). Functions are provided to configure and read the position and velocity captures, register a QEI interrupt handler, and handle QEI interrupt masking/clearing.

The quadrature encoder module provides hardware encoding of the two channels and the index signal from a quadrature encoder device into an absolute or relative position. There is additional hardware for capturing a measure of the encoder velocity, which is simply a count of encoder pulses during a fixed time period; the number of pulses is directly proportional to the encoder speed. Note that the velocity capture can only operate when the position capture is enabled.

The QEI module supports two modes of operation: phase mode and clock/direction mode. In phase mode, the encoder produces two clocks that are 90 degrees out of phase; the edge relationship is used to determine the direction of rotation. In clock/direction mode, the encoder produces a clock signal to indicate steps and a direction signal to indicate the direction of rotation.

When in phase mode, edges on the first channel or edges on both channels can be counted; counting edges on both channels provides higher encoder resolution if required. In either mode, the input signals can be swapped before being processed; this allows wiring mistakes on the circuit board to be corrected without modifying the board.

The index pulse can be used to reset the position counter; this causes the position counter to maintain the absolute encoder position. Otherwise, the position counter maintains the relative position and is never reset.

The velocity capture has a timer to measure equal periods of time. The number of encoder pulses over each time period is accumulated as a measure of the encoder velocity. The running total for the current time period and the final count for the previous time period are available to be read. The final count for the previous time period is usually used as the velocity measure.

The QEI module will generate interrupts when the index pulse is detected, when the velocity timer expires, when the encoder direction changes, and when a phase signal error is detected. These interrupt sources can be individually masked so that only the events of interest cause a processor interrupt.

16.2 Functions

Functions

- void ROM_QEIConfigure (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMaxPosition)
- long ROM_QEIDirectionGet (unsigned long ulBase)
- void ROM_QEIDisable (unsigned long ulBase)

- void ROM_QEIEnable (unsigned long ulBase)
- tBoolean ROM_QEIErrorGet (unsigned long ulBase)
- void ROM_QEIIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_QEIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_QEIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long ROM_QEIIntStatus (unsigned long ulBase, tBoolean bMasked)
- unsigned long ROM_QEIPositionGet (unsigned long ulBase)
- void ROM_QEIPositionSet (unsigned long ulBase, unsigned long ulPosition)
- void ROM_QEIVelocityConfigure (unsigned long ulBase, unsigned long ulPreDiv, unsigned long ulPeriod)
- void ROM_QEIVelocityDisable (unsigned long ulBase)
- void ROM_QEIVelocityEnable (unsigned long ulBase)
- unsigned long ROM_QEIVelocityGet (unsigned long ulBase)

16.2.1 Function Documentation

16.2.1.1 ROM_QEIConfigure

Configures the quadrature encoder.

Prototype:

```
void
ROM_QEIConfigure(unsigned long ulBase,
                 unsigned long ulConfig,
                 unsigned long ulMaxPosition)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIConfigure` is a function pointer located at `ROM_QEITABLE[3]`.

Parameters:

ulBase is the base address of the quadrature encoder module.

ulConfig is the configuration for the quadrature encoder. See below for a description of this parameter.

ulMaxPosition specifies the maximum position value.

Description:

This will configure the operation of the quadrature encoder. The *ulConfig* parameter provides the configuration of the encoder and is the logical OR of several values:

- **QEI_CONFIG_CAPTURE_A** or **QEI_CONFIG_CAPTURE_A_B** to specify if edges on channel A or on both channels A and B should be counted by the position integrator and velocity accumulator.
- **QEI_CONFIG_NO_RESET** or **QEI_CONFIG_RESET_IDX** to specify if the position integrator should be reset when the index pulse is detected.
- **QEI_CONFIG_QUADRATURE** or **QEI_CONFIG_CLOCK_DIR** to specify if quadrature signals are being provided on ChA and ChB, or if a direction signal and a clock are being provided instead.

- **QEI_CONFIG_NO_SWAP** or **QEI_CONFIG_SWAP** to specify if the signals provided on ChA and ChB should be swapped before being processed.

ulMaxPosition is the maximum value of the position integrator, and is the value used to reset the position capture when in index reset mode and moving in the reverse (negative) direction.

Returns:
None.

16.2.1.2 ROM_QEIDirectionGet

Gets the current direction of rotation.

Prototype:
```
long
ROM_QEIDirectionGet(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIDirectionGet` is a function pointer located at `ROM_QEITABLE[5]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This returns the current direction of rotation. In this case, current means the most recently detected direction of the encoder; it may not be presently moving but this is the direction it last moved before it stopped.

Returns:
Returns 1 if moving in the forward direction or -1 if moving in the reverse direction.

16.2.1.3 ROM_QEIDisable

Disables the quadrature encoder.

Prototype:
```
void
ROM_QEIDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIDisable` is a function pointer located at `ROM_QEITABLE[2]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will disable operation of the quadrature encoder module.

Returns:
None.

16.2.1.4 ROM_QEIEnable

Enables the quadrature encoder.

Prototype:
```
void
ROM_QEIEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIEnable` is a function pointer located at `ROM_QEITABLE[1]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will enable operation of the quadrature encoder module. It must be configured before it is enabled.

See also:
ROM_QEIConfigure()

Returns:
None.

16.2.1.5 ROM_QEIErrorGet

Gets the encoder error indicator.

Prototype:
```
tBoolean
ROM_QEIErrorGet(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIErrorGet` is a function pointer located at `ROM_QEITABLE[6]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This returns the error indicator for the quadrature encoder. It is an error for both of the signals of the quadrature input to change at the same time.

Returns:
Returns **true** if an error has occurred and **false** otherwise.

16.2.1.6 ROM_QEIIntClear

Clears quadrature encoder interrupt sources.

Prototype:

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

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_QEITABLE is an array of pointers located at ROM_APITABLE[9].
ROM_QEIIntClear is a function pointer located at ROM_QEITABLE[14].

Parameters:

ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be cleared. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:

The specified quadrature encoder 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:

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

Returns:

None.

16.2.1.7 ROM_QEIIntDisable

Disables individual quadrature encoder interrupt sources.

Prototype:

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

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_QEITABLE is an array of pointers located at ROM_APITABLE[9].
ROM_QEIIntDisable is a function pointer located at ROM_QEITABLE[12].

Parameters:

ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be disabled. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:
Disables the indicated quadrature encoder 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.

16.2.1.8 ROM_QEIIntEnable

Enables individual quadrature encoder interrupt sources.

Prototype:
```
void
ROM_QEIIntEnable(unsigned long ulBase,
                 unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIIntEnable` is a function pointer located at `ROM_QEITABLE[11]`.

Parameters:
ulBase is the base address of the quadrature encoder module.
ulIntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:
Enables the indicated quadrature encoder 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.

16.2.1.9 ROM_QEIIntStatus

Gets the current interrupt status.

Prototype:
```
unsigned long
ROM_QEIIntStatus(unsigned long ulBase,
                 tBoolean bMasked)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIIntStatus` is a function pointer located at `ROM_QEITABLE[13]`.

Parameters:
ulBase is the base address of the quadrature encoder 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 quadrature encoder module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, and **QEI_INTINDEX**.

16.2.1.10 ROM_QEIPositionGet

Gets the current encoder position.

Prototype:

```
unsigned long
ROM_QEIPositionGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIPositionGet` is a function pointer located at `ROM_QEITABLE[0]`.

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This returns the current position of the encoder. Depending upon the configuration of the encoder, and the incident of an index pulse, this value may or may not contain the expected data (that is, if in reset on index mode, if an index pulse has not been encountered, the position counter will not be aligned with the index pulse yet).

Returns:

The current position of the encoder.

16.2.1.11 ROM_QEIPositionSet

Sets the current encoder position.

Prototype:

```
void
ROM_QEIPositionSet(unsigned long ulBase,
                   unsigned long ulPosition)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIPositionSet` is a function pointer located at `ROM_QEITABLE[4]`.

Parameters:

ulBase is the base address of the quadrature encoder module.

ulPosition is the new position for the encoder.

Description:
This sets the current position of the encoder; the encoder position will then be measured relative to this value.

Returns:
None.

16.2.1.12 ROM_QEIVelocityConfigure

Configures the velocity capture.

Prototype:
```
void
ROM_QEIVelocityConfigure(unsigned long ulBase,
                         unsigned long ulPreDiv,
                         unsigned long ulPeriod)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIVelocityConfigure` is a function pointer located at `ROM_QEITABLE[9]`.

Parameters:
ulBase is the base address of the quadrature encoder module.
ulPreDiv specifies the predivider applied to the input quadrature signal before it is counted; can be one of **QEI_VELDIV_1**, **QEI_VELDIV_2**, **QEI_VELDIV_4**, **QEI_VELDIV_8**, **QEI_VELDIV_16**, **QEI_VELDIV_32**, **QEI_VELDIV_64**, or **QEI_VELDIV_128**.
ulPeriod specifies the number of clock ticks over which to measure the velocity; must be non-zero.

Description:
This will configure the operation of the velocity capture portion of the quadrature encoder. The position increment signal is predivided as specified by *ulPreDiv* before being accumulated by the velocity capture. The divided signal is accumulated over *ulPeriod* system clock before being saved and resetting the accumulator.

Returns:
None.

16.2.1.13 ROM_QEIVelocityDisable

Disables the velocity capture.

Prototype:
```
void
ROM_QEIVelocityDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIVelocityDisable` is a function pointer located at `ROM_QEITABLE[8]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will disable operation of the velocity capture in the quadrature encoder module.

Returns:
None.

16.2.1.14 ROM_QEIVelocityEnable

Enables the velocity capture.

Prototype:
```
void
ROM_QEIVelocityEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIVelocityEnable` is a function pointer located at `ROM_QEITABLE[7]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will enable operation of the velocity capture in the quadrature encoder module. It must be configured before it is enabled. Velocity capture will not occur if the quadrature encoder is not enabled.

See also:
ROM_QEIVelocityConfigure() and ROM_QEIEnable()

Returns:
None.

16.2.1.15 ROM_QEIVelocityGet

Gets the current encoder speed.

Prototype:
```
unsigned long
ROM_QEIVelocityGet(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_QEITABLE` is an array of pointers located at `ROM_APITABLE[9]`.
`ROM_QEIVelocityGet` is a function pointer located at `ROM_QEITABLE[10]`.

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:

This returns the current speed of the encoder. The value returned is the number of pulses detected in the specified time period; this number can be multiplied by the number of time periods per second and divided by the number of pulses per revolution to obtain the number of revolutions per second.

Returns:

Returns the number of pulses captured in the given time period.

17 Synchronous Serial Interface (SSI)

Introduction 199
Functions 199

17.1 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.

17.2 Functions

Functions

- tBoolean ROM_SSIBusy (unsigned long ulBase)
- unsigned long ROM_SSIClockSourceGet (unsigned long ulBase)
- void ROM_SSIClockSourceSet (unsigned long ulBase, unsigned long ulSource)
- 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_SSIDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)
- void ROM_SSIDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)
- 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)

17.2.1 Function Documentation

17.2.1.1 ROM_SSIBusy

Determines whether the SSI transmitter is busy or not.

Prototype:

```
tBoolean
ROM_SSIBusy(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_SSIBusy` is a function pointer located at `ROM_SSITABLE[14]`.

Parameters:

ulBase is the base address of the SSI port.

Description:

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

Returns:

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

17.2.1.2 ROM_SSIClockSourceGet

Gets the data clock source for the specified SSI peripheral.

Prototype:

```
unsigned long
ROM_SSIClockSourceGet(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_SSIClockSourceGet` is a function pointer located at `ROM_SSITABLE[15]`.

Parameters:

ulBase is the base address of the SSI port.

Description:

This function returns the data clock source for the specified SSI. The possible data clock source are the system clock (**SSI_CLOCK_SYSTEM**) or the precision internal oscillator (**SSI_CLOCK_PIOSC**).

Returns:
None.

17.2.1.3 ROM_SSIClockSourceSet

Sets the data clock source for the specified SSI peripheral.

Prototype:
```
void
ROM_SSIClockSourceSet(unsigned long ulBase,
                      unsigned long ulSource)
```

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_SSIClockSourceSet` is a function pointer located at `ROM_SSITABLE[16]`.

Parameters:
ulBase is the base address of the SSI port.
ulSource is the baud clock source for the SSI.

Description:
This function allows the baud clock source for the SSI to be selected. The possible clock source are the system clock (**SSI_CLOCK_SYSTEM**) or the precision internal oscillator (**SSI_CLOCK_PIOSC**).

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

Returns:
None.

17.2.1.4 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 >= 2 * bit rate (master mode)
- FSSI >= 12 * 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, and can be a value between 4 and 16, inclusive.

The peripheral clock is the same as the processor clock. This is 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.

17.2.1.5 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 is a pointer to a storage location for data that was received over the SSI interface.

Description:

This function gets received data from the receive FIFO of the specified SSI module and places that data into the location specified by the *pulData* parameter.

Note:

Only the lower N bits of the value written to *pulData* 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* contain valid data.

Returns:

None.

17.2.1.6 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 is a pointer to a storage location for data that was received over the SSI interface.

Description:

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

Note:

Only the lower N bits of the value written to *pulData* 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* contain valid data.

Returns:

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

17.2.1.7 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 is the data to be transmitted over the SSI interface.

Description:

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

Note:

The upper 32 - N bits of the *ulData* are 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* are discarded.

Returns:

None.

17.2.1.8 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 is the data to be transmitted over the SSI interface.

Description:

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

Note:
The upper 32 - N bits of the *ulData* are 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* are discarded.

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

17.2.1.9 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 function disables operation of the synchronous serial interface.

Returns:
None.

17.2.1.10 ROM_SSIDMADisable

Disable SSI DMA operation.

Prototype:
```
void
ROM_SSIDMADisable(unsigned long ulBase,
                  unsigned long ulDMAFlags)
```

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_SSIDMADisable` is a function pointer located at `ROM_SSITABLE[13]`.

Parameters:
ulBase is the base address of the SSI port.
ulDMAFlags is a bit mask of the DMA features to disable.

Description:
This function is used to disable SSI DMA features that were enabled by ROM_SSIDMAEnable(). The specified SSI DMA features are disabled. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - disable DMA for receive
- SSI_DMA_TX - disable DMA for transmit

Returns:
None.

17.2.1.11 ROM_SSIDMAEnable

Enable SSI DMA operation.

Prototype:
```
void
ROM_SSIDMAEnable(unsigned long ulBase,
                 unsigned long ulDMAFlags)
```

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_SSIDMAEnable` is a function pointer located at `ROM_SSITABLE[12]`.

Parameters:
ulBase is the base address of the SSI port.
ulDMAFlags is a bit mask of the DMA features to enable.

Description:
The specified SSI DMA features are enabled. The SSI can be configured to use DMA for transmit and/or receive data transfers. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - enable DMA for receive
- SSI_DMA_TX - enable DMA for transmit

Note:
The uDMA controller must also be set up before DMA can be used with the SSI.

Returns:
None.

17.2.1.12 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 function enables operation of the synchronous serial interface. The synchronous serial interface must be configured before it is enabled.

Returns:

None.

17.2.1.13 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 function must be called in the interrupt handler to keep the interrupts from being recognized again immediately upon exit. The *ulIntFlags* parameter can consist of either or both the **SSI_RXTO** and **SSI_RXOR** values.

Note:

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

Returns:

None.

17.2.1.14 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.

ulIntFlags is a bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated SSI interrupt sources. The *ulIntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

17.2.1.15 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.

ulIntFlags 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 *ulIntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

17.2.1.16 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 or **true** if the masked interrupt status is required.

Description:

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

Returns:

The current interrupt status, enumerated as a bit field of **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, and **SSI_RXOR**.

17.2.1.17 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 oprational.

Returns:

Never returns.

18 System Control

Introduction 211
Functions 212

18.1 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. When using the PLL, the input clock frequency is constrained to specific frequencies between 5 MHz and 25 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 80 MHz (depending on the device). The internal oscillator is 16 MHz, +/- 1%; its frequency will vary by device, with voltage, and with temperature.

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.

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.

18.2 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)
- void ROM_SysCtlDeepSleepClockSet (unsigned long ulConfig)
- void ROM_SysCtlDelay (unsigned long ulCount)
- 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)
- void ROM_SysCtlMOSCConfigSet (unsigned long ulConfig)
- 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)
- void ROM_SysCtlPeripheralPowerOff (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralPowerOn (unsigned long ulPeripheral)
- tBoolean ROM_SysCtlPeripheralPresent (unsigned long ulPeripheral)
- tBoolean ROM_SysCtlPeripheralReady (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)
- unsigned long ROM_SysCtlPIOSCCalibrate (unsigned long ulType)
- unsigned long ROM_SysCtlPWMClockGet (void)
- void ROM_SysCtlPWMClockSet (unsigned long ulConfig)
- void ROM_SysCtlReset (void)
- void ROM_SysCtlResetCauseClear (unsigned long ulCauses)
- unsigned long ROM_SysCtlResetCauseGet (void)
- void ROM_SysCtlSleep (void)
- unsigned long ROM_SysCtlSRAMSizeGet (void)
- void ROM_SysCtlUSBPLLDisable (void)
- void ROM_SysCtlUSBPLLEnable (void)

18.2.1 Function Documentation

18.2.1.1 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlADCSpeedGet` is a function pointer located at `ROM_SYSCTLTABLE[28]`.

Description:

This function gets the current sample rate of the ADC.

Returns:

Returns the current ADC sample rate; is one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

18.2.1.2 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlADCSpeedSet` is a function pointer located at `ROM_SYSCTLTABLE[27]`.

Parameters:

ulSpeed is the desired sample rate of the ADC; must be one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_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.

18.2.1.3 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlClockGet` is a function pointer located at `ROM_SYSCTLTABLE[24]`.

Description:

This function determines the clock rate of the processor clock. This is also the clock rate of all the peripheral modules (with the exception of PWM, which has its own clock divider).

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 be modified to directly return the correct system clock rate.

Returns:

The processor clock rate.

18.2.1.4 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlClockSet` is a function pointer located at `ROM_SYSCTLTABLE[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: **SYSCTL_SYSDIV_1**, **SYSCTL_SYSDIV_2**, **SYSCTL_SYSDIV_3**, ... **SYSCTL_SYSDIV_64**.

The use of the PLL is chosen with either **SYSCTL_USE_PLL** or **SYSCTL_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 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 is 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 (that is, 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.

18.2.1.5 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.

18.2.1.6 ROM_SysCtlDeepSleepClockSet

Sets the clocking of the device while in deep-sleep mode.

Prototype:
```
void
ROM_SysCtlDeepSleepClockSet(unsigned long ulConfig)
```

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_SysCtlDeepSleepClockSet` is a function pointer located at `ROM_SYSCTLTABLE[46]`.

Parameters:
ulConfig is the required configuration of the device clocking while in deep-sleep mode.

Description:
This function configures the clocking of the device while in deep-sleep mode. The oscillator to be used and the system clock divider are configured with this function.

The *ulConfig* parameter is the logical OR of the following values:

The system clock divider is chosen with one of the following values: **SYSCTL_DSLP_DIV_1**, **SYSCTL_DSLP_DIV_2**, **SYSCTL_DSLP_DIV_3**, ... **SYSCTL_DSLP_DIV_64**.

The oscillator source is chosen with one of the following values: **SYSCTL_DSLP_OSC_MAIN**, **SYSCTL_DSLP_OSC_INT**, **SYSCTL_DSLP_OSC_INT30**, or **SYSCTL_DSLP_OSC_EXT32**. **SYSCTL_OSC_EXT32** is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

The precision internal oscillator can be powered down in deep-sleep mode by specifying **SYSCTL_DSLP_PIOSC_PD**. If it is required for operation while in deep-sleep (based on other configuration settings), it will not be powered down.

Returns:
None.

18.2.1.7 ROM_SysCtlDelay

Provides a small delay.

Prototype:
```
void
ROM_SysCtlDelay(unsigned long ulCount)
```

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_SysCtlDelay is a function pointer located at ROM_SYSCTLTABLE[34].

Parameters:
ulCount is the number of delay loop iterations to perform.

Description:
This function provides a means of generating a constant length delay. It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

Returns:
None.

18.2.1.8 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_SYSCTLTABLE is an array of pointers located at ROM_APITABLE[13].
ROM_SysCtlFlashSizeGet is a function pointer located at ROM_SYSCTLTABLE[2].

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

Returns:
The total number of bytes of flash.

18.2.1.9 ROM_SysCtlGPIOAHBDisable

Disables a GPIO peripheral for access from the AHB.

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

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_SysCtlGPIOAHBDisable is a function pointer located at ROM_SYSCTLTABLE[30].

Parameters:
ulGPIOPeripheral is the GPIO peripheral to disable.

Description:

This function disables the specified GPIO peripheral for access from the Advanced Host Bus (AHB). Once disabled, the GPIO peripheral is accessed from the legacy Advanced Peripheral Bus (AHB).

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.

18.2.1.10 ROM_SysCtlGPIOAHBEnable

Enables a GPIO peripheral for access from the AHB.

Prototype:

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

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_SysCtlGPIOAHBEnable` is a function pointer located at `ROM_SYSCTLTABLE[29]`.

Parameters:

ulGPIOPeripheral is the GPIO peripheral to enable.

Description:

This function is used to enable the specified GPIO peripheral to be accessed from the Advanced Host Bus (AHB) instead of the legacy Advanced Peripheral Bus (APB). When a GPIO peripheral is enabled for AHB 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.

18.2.1.11 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_SYSCTLTABLE is an array of pointers located at ROM_APITABLE[13].
ROM_SysCtlIntClear is a function pointer located at ROM_SYSCTLTABLE[15].

Parameters:

ulInts is a bit mask of the interrupt sources to be cleared. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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:

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

Returns:

None.

18.2.1.12 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_SYSCTLTABLE is an array of pointers located at ROM_APITABLE[13].
ROM_SysCtlIntDisable is a function pointer located at ROM_SYSCTLTABLE[14].

Parameters:

ulInts is a bit mask of the interrupt sources to be disabled. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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.

18.2.1.13 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlIntEnable` is a function pointer located at `ROM_SYSCTLTABLE[13]`.

Parameters:

ulInts is a bit mask of the interrupt sources to be enabled. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

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.

18.2.1.14 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_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and **SYSCTL_INT_PLL_FAIL**.

18.2.1.15 ROM_SysCtlMOSCConfigSet

Sets the configuration of the main oscillator (MOSC) control.

Prototype:

```
void
ROM_SysCtlMOSCConfigSet(unsigned long ulConfig)
```

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_SysCtlMOSCConfigSet` is a function pointer located at `ROM_SYSCTLTABLE[44]`.

Parameters:

ulConfig is the required configuration of the MOSC control.

Description:

This function configures the control of the main oscillator. The *ulConfig* is specified as follows:

- **SYSCTL_MOSC_VALIDATE** enables the MOSC verification circuit that detects a failure of the main oscillator (such as a loss of the clock).
- **SYSCTL_MOSC_INTERRUPT** indicates that a MOSC failure should generate an interrupt instead of resetting the processor.
- **SYSCTL_MOSC_NO_XTAL** indicates that there is no crystal connected to the OSC0/OSC1 pins, allowing power consumption to be reduced.

Returns:

None.

18.2.1.16 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.

18.2.1.17 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlPeripheralDeepSleepDisable` is a function pointer located at `ROM_SYSCTLTABLE[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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**,

SYSCTL_PERIPH_WDOG1, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:
None.

18.2.1.18 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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**,

SYSCTL_PERIPH_UART5, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:

None.

18.2.1.19 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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:
None.

18.2.1.20 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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

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.

18.2.1.21 ROM_SysCtlPeripheralPowerOff

Powers off a peripheral.

Prototype:
```
void
ROM_SysCtlPeripheralPowerOff(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_SysCtlPeripheralPowerOff` is a function pointer located at `ROM_SYSCTLTABLE[37]`.

Parameters:
ulPeripheral is the peripheral to be powered off.

Description:
This function allows the power to a peripheral to be turned off. The peripheral will continue to receive power when its clock is enabled, but the power will be removed when its clock is disabled.

The *ulPeripheral* paramter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:
None.

18.2.1.22 ROM_SysCtlPeripheralPowerOn

Powers on a peripheral.

Prototype:

```
void
ROM_SysCtlPeripheralPowerOn(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_SysCtlPeripheralPowerOn` is a function pointer located at `ROM_SYSCTLTABLE[36]`.

Parameters:

ulPeripheral is the peripheral to be powered on.

Description:

This function turns on the power to a peripheral. It will continue to receive power even when its clock is not enabled.

The *ulPeripheral* paramter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:

None.

18.2.1.23 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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EPI0**, **SYSCTL_PERIPH_ETH**, **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_GPIOJ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_IEEE1588**, **SYSCTL_PERIPH_MPU**, **SYSCTL_PERIPH_PLL**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TEMP**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:

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

18.2.1.24 ROM_SysCtlPeripheralReady

Determines if a peripheral is ready.

Prototype:

```
tBoolean
ROM_SysCtlPeripheralReady(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_SysCtlPeripheralReady` is a function pointer located at `ROM_SYSCTLTABLE[35]`.

Parameters:

ulPeripheral is the peripheral in question.

Description:

Determines if a particular peripheral is ready to be accessed. The peripheral may be in a non-ready state if it is not enabled, is being held in reset, or is in the process of becoming ready after be enabled or taken out of reset.

The *ulPeripheral* paramter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:

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

18.2.1.25 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlPeripheralReset` is a function pointer located at `ROM_SYSCTLTABLE[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, returning the internal state of the peripheral to its reset condition.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:

None.

18.2.1.26 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_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlPeripheralSleepDisable` is a function pointer located at `ROM_SYSCTLTABLE[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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:
None.

18.2.1.27 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* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC0**, **SYSCTL_PERIPH_ADC1**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_EEPROM0**, **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_GPIOJ**, **SYSCTL_PERIPH_GPIOK**, **SYSCTL_PERIPH_GPIOL**, **SYSCTL_PERIPH_GPIOM**, **SYSCTL_PERIPH_GPION**, **SYSCTL_PERIPH_GPIOP**, **SYSCTL_PERIPH_GPIOQ**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2C2**, **SYSCTL_PERIPH_I2C3**, **SYSCTL_PERIPH_I2C4**, **SYSCTL_PERIPH_I2C5**, **SYSCTL_PERIPH_PWM0**, **SYSCTL_PERIPH_PWM1**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_SSI2**, **SYSCTL_PERIPH_SSI3**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_TIMER4**, **SYSCTL_PERIPH_TIMER5**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UART3**, **SYSCTL_PERIPH_UART4**, **SYSCTL_PERIPH_UART5**, **SYSCTL_PERIPH_UART6**, **SYSCTL_PERIPH_UART7**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, **SYSCTL_PERIPH_WDOG1**, **SYSCTL_PERIPH_WTIMER0**, **SYSCTL_PERIPH_WTIMER1**, **SYSCTL_PERIPH_WTIMER2**, **SYSCTL_PERIPH_WTIMER3**, **SYSCTL_PERIPH_WTIMER4**, or **SYSCTL_PERIPH_WTIMER5**.

Returns:

None.

18.2.1.28 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. The PWM, analog comparators, ADC, and timers 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_PWM0**, **SYSCTL_PIN_PWM1**, **SYSCTL_PIN_PWM2**, **SYSCTL_PIN_PWM3**, **SYSCTL_PIN_PWM4**, **SYSCTL_PIN_PWM5**, **SYSCTL_PIN_MC_FAULT0**, **SYSCTL_PIN_C0MINUS**, **SYSCTL_PIN_C0PLUS**, **SYSCTL_PIN_C0O**, **SYSCTL_PIN_C1MINUS**, **SYSCTL_PIN_C1PLUS**, **SYSCTL_PIN_C1O**, **SYSCTL_PIN_C2MINUS**, **SYSCTL_PIN_C2PLUS**, **SYSCTL_PIN_C2O**, **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.

18.2.1.29 ROM_SysCtlPIOSCCalibrate

Calibrates the precision internal oscillator.

Prototype:

```
unsigned long
ROM_SysCtlPIOSCCalibrate(unsigned long ulType)
```

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_SysCtlPIOSCCalibrate` is a function pointer located at `ROM_SYSCTLTABLE[45]`.

Parameters:

ulType is the type of calibration to perform.

Description:

This function performs a calibration of the PIOSC. There are three types of calibration available; the desired calibration type as specified in *ulType* is one of:

- **SYSCTL_PIOSC_CAL_AUTO** to perform automatic calibration using the 32 kHz clock from the hibernate module as a reference. This is only possible on parts that have a hibernate module and then only if it is enabled and the hibernate module's RTC is also enabled.
- **SYSCTL_PIOSC_CAL_FACT** to reset the PIOSC calibration to the factory provided calibration.
- **SYSCTL_PIOSC_CAL_USER** to set the PIOSC calibration to a user-supplied value. The value to be used is ORed into the lower 7-bits of this value, with 0x40 being the "nominal" value (in other words, if everything were perfect, this would provide exactly 16 MHz). Values larger than 0x40 will slow down PIOSC, and values smaller than 0x40 will speed up PIOSC.

Returns:

None.

18.2.1.30 ROM_SysCtlPWMClockGet

Gets the current PWM clock configuration.

Prototype:

```
unsigned long
ROM_SysCtlPWMClockGet(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_SysCtlPWMClockGet` is a function pointer located at `ROM_SYSCTLTABLE[26]`.

Description:

This function returns the current PWM clock configuration.

Returns:

Returns the current PWM clock configuration; is one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

18.2.1.31 ROM_SysCtlPWMClockSet

Sets the PWM clock configuration.

Prototype:

```
void
ROM_SysCtlPWMClockSet(unsigned long ulConfig)
```

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_SysCtlPWMClockSet` is a function pointer located at `ROM_SYSCTLTABLE[25]`.

Parameters:

ulConfig is the configuration for the PWM clock; it must be one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

Description:

This function sets the rate of the clock provided to the PWM module as a ratio of the processor clock. This clock is used by the PWM module to generate PWM signals; its rate forms the basis for all PWM signals.

Note:

The clocking of the PWM is dependent upon the system clock rate as configured by ROM_SysCtlClockSet().

Returns:

None.

18.2.1.32 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 are 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.

18.2.1.33 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_LDO**, **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.

18.2.1.34 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 is a logical OR of **SYSCTL_CAUSE_LDO**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG**, **SYSCTL_CAUSE_BOR**, **SYSCTL_CAUSE_POR**, and/or **SYSCTL_CAUSE_EXT**.

Returns:

Returns the reason(s) for a reset.

18.2.1.35 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.

18.2.1.36 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.

18.2.1.37 ROM_SysCtlUSBPLLDisable

Powers down the USB PLL.

Prototype:
```
void
ROM_SysCtlUSBPLLDisable(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_SysCtlUSBPLLDisable is a function pointer located at ROM_SYSCTLTABLE[32].

Description:
This function will disable the USB controller's PLL which is used by it's physical layer. The USB registers are still accessible, but the physical layer will no longer function.

Returns:
None.

18.2.1.38 ROM_SysCtlUSBPLLEnable

Powers up the USB PLL.

Prototype:
```
void
ROM_SysCtlUSBPLLEnable(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_SysCtlUSBPLLEnable is a function pointer located at ROM_SYSCTLTABLE[31].

Description:
This function will enable the USB controller's PLL which is used by it's physical layer. This call is necessary before connecting to any external devices.

Returns:
None.

19 System Exception Module

Introduction 239
API Functions 239

19.1 Introduction

The system exception module provides an interrupt mechanism for handling system exceptions, such as errors from the floating-point unit..

19.2 API Functions

Functions

- void ROM_SysExcIntClear (unsigned long ulIntFlags)
- void ROM_SysExcIntDisable (unsigned long ulIntFlags)
- void ROM_SysExcIntEnable (unsigned long ulIntFlags)
- unsigned long ROM_SysExcIntStatus (tBoolean bMasked)

19.2.1 Function Documentation

19.2.1.1 ROM_SysExcIntClear

Clears system exception interrupt sources.

Prototype:

```
void
ROM_SysExcIntClear(unsigned long ulIntFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSEXCTABLE` is an array of pointers located at `ROM_APITABLE[30]`.
`ROM_SysExcIntClear` is a function pointer located at `ROM_SYSEXCTABLE[1]`.

Parameters:

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

Description:

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

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

- **SYSEXCP_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXCP_INT_FP_OFC** - Floating-point overflow exception interrupt

- **SYSEXCP_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXCP_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXCP_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXCP_INT_FP_IDC** - Floating-point input denormal exception interrupt

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

Returns:
None.

19.2.1.2 ROM_SysExcIntDisable

Disables individual system exception interrupt sources.

Prototype:
```
void
ROM_SysExcIntDisable(unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSEXCTABLE` is an array of pointers located at `ROM_APITABLE[30]`.
`ROM_SysExcIntDisable` is a function pointer located at `ROM_SYSEXCTABLE[2]`.

Parameters:
ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:
This function disables the indicated system exception interrupt sources. Only sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

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

- **SYSEXCP_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXCP_INT_FP_OFC** - Floating-point overflow exception interrupt
- **SYSEXCP_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXCP_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXCP_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXCP_INT_FP_IDC** - Floating-point input denormal exception interrupt

Returns:
None.

19.2.1.3 ROM_SysExcIntEnable

Enables individual system exception interrupt sources.

Prototype:

```
void
ROM_SysExcIntEnable(unsigned long ulIntFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSEXCTABLE` is an array of pointers located at `ROM_APITABLE[30]`.
`ROM_SysExcIntEnable` is a function pointer located at `ROM_SYSEXCTABLE[3]`.

Parameters:

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:

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

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

- **SYSEXCP_INT_FP_IXC** - Floating-point inexact exception interrupt
- **SYSEXCP_INT_FP_OFC** - Floating-point overflow exception interrupt
- **SYSEXCP_INT_FP_UFC** - Floating-point underflow exception interrupt
- **SYSEXCP_INT_FP_IOC** - Floating-point invalid operation interrupt
- **SYSEXCP_INT_FP_DZC** - Floating-point divide by zero exception interrupt
- **SYSEXCP_INT_FP_IDC** - Floating-point input denormal exception interrupt

Returns:

None.

19.2.1.4 ROM_SysExcIntStatus

Gets the current system exception interrupt status.

Prototype:

```
unsigned long
ROM_SysExcIntStatus(tBoolean bMasked)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSEXCTABLE` is an array of pointers located at `ROM_APITABLE[30]`.
`ROM_SysExcIntStatus` is a function pointer located at `ROM_SYSEXCTABLE[0]`.

Parameters:

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

Description:

This function returns the system exception interrupt status. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current system exception interrupt status, enumerated as the logical OR of **SYSEXCP_INT_FP_IXC**, **SYSEXCP_INT_FP_OFC**, **SYSEXCP_INT_FP_UFC**, **SYSEXCP_INT_FP_IOC**, **SYSEXCP_INT_FP_DZC**, and **SYSEXCP_INT_FP_IDC**.

20 System Tick (SysTick)

Introduction 243
Functions 243

20.1 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.

20.2 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)

20.2.1 Function Documentation

20.2.1.1 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.

20.2.1.2 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 is 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, the **NVIC_ST_CURRENT** register 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.

20.2.1.3 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.

20.2.1.4 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.

20.2.1.5 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.

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

Returns:

None.

20.2.1.7 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.

21 Timer

Introduction 247
Functions 247

21.1 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.

21.2 Functions

Functions

- void ROM_TimerConfigure (unsigned long ulBase, unsigned long ulConfig)
- void ROM_TimerControlEvent (unsigned long ulBase, unsigned long ulTimer, unsigned long ulEvent)
- void ROM_TimerControlLevel (unsigned long ulBase, unsigned long ulTimer, tBoolean bInvert)
- void ROM_TimerControlStall (unsigned long ulBase, unsigned long ulTimer, tBoolean bStall)

- void ROM_TimerControlTrigger (unsigned long ulBase, unsigned long ulTimer, tBoolean bEnable)
- void ROM_TimerControlWaitOnTrigger (unsigned long ulBase, unsigned long ulTimer, tBoolean bWait)
- 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)
- unsigned long long ROM_TimerLoadGet64 (unsigned long ulBase)
- void ROM_TimerLoadSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void ROM_TimerLoadSet64 (unsigned long ulBase, unsigned long long ullValue)
- unsigned long ROM_TimerMatchGet (unsigned long ulBase, unsigned long ulTimer)
- unsigned long long ROM_TimerMatchGet64 (unsigned long ulBase)
- void ROM_TimerMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void ROM_TimerMatchSet64 (unsigned long ulBase, unsigned long long ullValue)
- unsigned long ROM_TimerPrescaleGet (unsigned long ulBase, unsigned long ulTimer)
- unsigned long ROM_TimerPrescaleMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void ROM_TimerPrescaleMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- 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)
- unsigned long long ROM_TimerValueGet64 (unsigned long ulBase)

21.2.1 Function Documentation

21.2.1.1 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. There are two types of timers; a 16/32-bit variety and a 32/64-bit variety. The 16/32-bit variety is comprised of two 16-bit timers that can operate independently or be concatenated to form a 32-bit timer. Similarly, the 32/64-bit variety is comprised of two 32-bit timers that can operate independently or be concatenated to form a 64-bit timer.

The configuration is specified in *ulConfig* as one of the following values:

- **TIMER_CFG_ONE_SHOT** - Full-width one-shot timer
- **TIMER_CFG_ONE_SHOT_UP** - Full-width one-shot timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_PERIODIC** - Full-width periodic timer
- **TIMER_CFG_PERIODIC_UP** - Full-width periodic timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_RTC** - Full-width real time clock timer
- **TIMER_CFG_SPLIT_PAIR** - Two half-width timers

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

- **TIMER_CFG_A_ONE_SHOT** - Half-width one-shot timer
- **TIMER_CFG_A_ONE_SHOT_UP** - Half-width one-shot timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_PERIODIC** - Half-width periodic timer
- **TIMER_CFG_A_PERIODIC_UP** - Half-width periodic timer that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_CAP_COUNT** - Half-width edge count capture
- **TIMER_CFG_A_CAP_COUNT_UP** - Half-width edge count capture that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_CAP_TIME** - Half-width edge time capture
- **TIMER_CFG_A_CAP_TIME_UP** - Half-width edge time capture that counts up instead of down (not available on all parts)
- **TIMER_CFG_A_PWM** - Half-width 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.

21.2.1.2 ROM_TimerControlEvent

Controls the event type.

Prototype:

```
void
ROM_TimerControlEvent(unsigned long ulBase,
                      unsigned long ulTimer,
                      unsigned long ulEvent)
```

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_TimerControlEvent` is a function pointer located at `ROM_TIMERTABLE[6]`.

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**.

ulEvent specifies the type of event; must be one of **TIMER_EVENT_POS_EDGE**, **TIMER_EVENT_NEG_EDGE**, or **TIMER_EVENT_BOTH_EDGES**.

Description:

This function sets the signal edge(s) that triggers the timer when in capture mode.

Returns:

None.

21.2.1.3 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 is made active low; otherwise, it is made active high.

Returns:

None.

21.2.1.4 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.

21.2.1.5 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.

21.2.1.6 ROM_TimerControlWaitOnTrigger

Controls the wait on trigger handling.

Prototype:
```
void
ROM_TimerControlWaitOnTrigger(unsigned long ulBase,
                              unsigned long ulTimer,
                              tBoolean bWait)
```

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_TimerControlWaitOnTrigger` is a function pointer located at `ROM_TIMERTABLE[22]`.

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**.
bWait specifies if the timer should wait for a trigger input.

Description:
This function controls whether or not a timer waits for a trigger input to start counting. When enabled, the previous timer in the trigger chain must count to its timeout in order for this timer to start counting. Refer to the data sheet for a description of the trigger chain.

Returns:
None.

21.2.1.7 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.

21.2.1.8 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.

21.2.1.9 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.
ulIntFlags 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 *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to ROM_TimerIntEnable().

Note:

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

Returns:

None.

21.2.1.10 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.

ulIntFlags 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 *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to ROM_TimerIntEnable().

Returns:

None.

21.2.1.11 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.

ulIntFlags 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 *ulIntFlags* 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.

21.2.1.12 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().

21.2.1.13 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 full-width operation.

Description:
This function gets the currently programmed interval load value for the specified timer.

Note:
This function can be used for both full- and half-width modes of 16/32-bit timers, and for half-width modes of 32/64-bit timers. Use ROM_TimerLoadGet64() for full-width modes of 32/64-bit timers.

Returns:
Returns the load value for the timer.

21.2.1.14 ROM_TimerLoadGet64

Gets the timer load value for a 64-bit timer.

Prototype:
```
unsigned long long
ROM_TimerLoadGet64(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_TimerLoadGet64` is a function pointer located at `ROM_TIMERTABLE[24]`.

Parameters:
ulBase is the base address of the timer module.

Description:
This function gets the currently programmed interval load value for the specified 64-bit timer.

Returns:
Returns the load value for the timer.

21.2.1.15 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 full-width operation.
ulValue is the load value.

Description:
This function sets the timer load value; if the timer is running then the value is immediately loaded into the timer.

Note:
This function can be used for both full- and half-width modes of 16/32-bit timers, and for half-width modes of 32/64-bit timers. Use ROM_TimerLoadSet64() for full-width modes of 32/64-bit timers.

Returns:
None.

21.2.1.16 ROM_TimerLoadSet64

Sets the timer load value for a 64-bit timer.

Prototype:
```
void
ROM_TimerLoadSet64(unsigned long ulBase,
                   unsigned long long ullValue)
```

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_TimerLoadSet64` is a function pointer located at `ROM_TIMERTABLE[23]`.

Parameters:

ulBase is the base address of the timer module.

ullValue is the load value.

Description:

This function sets the timer load value for a 64-bit timer; if the timer is running then the value is immediately loaded into the timer.

Returns:

None.

21.2.1.17 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 full-width operation.

Description:

This function gets the match value for the specified timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers, and for half-width modes of 32/64-bit timers. Use ROM_TimerMatchGet64() for full-width modes of 32/64-bit timers.

Returns:

Returns the match value for the timer.

21.2.1.18 ROM_TimerMatchGet64

Gets the timer match value for a 64-bit timer.

Prototype:

```
unsigned long long
ROM_TimerMatchGet64(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_TimerMatchGet64` is a function pointer located at `ROM_TIMERTABLE[27]`.

Parameters:
ulBase is the base address of the timer module.

Description:
This function gets the match value for the specified timer.

Returns:
Returns the match value for the timer.

21.2.1.19 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 full-width operation.
ulValue is the match value.

Description:
This function sets the match value for a timer. This value 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.

Note:
This function can be used for both full- and half-width modes of 16/32-bit timers, and for half-width modes of 32/64-bit timers. Use ROM_TimerMatchSet64() for full-width modes of 32/64-bit timers.

Returns:
None.

21.2.1.20 ROM_TimerMatchSet64

Sets the timer match value for a 64-bit timer.

Prototype:
```
void
ROM_TimerMatchSet64(unsigned long ulBase,
                    unsigned long long ullValue)
```

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_TimerMatchSet64 is a function pointer located at ROM_TIMERTABLE[26].

Parameters:
ulBase is the base address of the timer module.
ullValue is the match value.

Description:
This function sets the match value for a timer. This value 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.

21.2.1.21 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 half-width mode and is used to extend the range of the half-width timer modes.

Returns:
The value of the timer prescaler.

21.2.1.22 ROM_TimerPrescaleMatchGet

Get the timer prescale match value.

Prototype:
```
unsigned long
ROM_TimerPrescaleMatchGet(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_TimerPrescaleMatchGet` is a function pointer located at `ROM_TIMERTABLE[13]`.

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 match value. When in a half-width mode that uses the counter match and prescaler, the prescale match effectively extends the range of the match.

Returns:

The value of the timer prescale match.

21.2.1.23 ROM_TimerPrescaleMatchSet

Set the timer prescale match value.

Prototype:

```
void
ROM_TimerPrescaleMatchSet(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_TimerPrescaleMatchSet` is a function pointer located at `ROM_TIMERTABLE[12]`.

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 match value which must be between 0 and 255 (inclusive) for 16/32-bit timers and between 0 and 65535 (inclusive) for 32/64-bit timers.

Description:

This function sets the value of the input clock prescaler match value. When in a half-width mode that uses the counter match and the prescaler, the prescale match effectively extends the range of the match.

Returns:

None.

21.2.1.24 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 which must be between 0 and 255 (inclusive) for 16/32-bit timers and between 0 and 65535 (inclusive) for 32/64-bit timers.

Description:

This function sets the value of the input clock prescaler. The prescaler is only operational when in half-width mode and is used to extend the range of the half-width timer modes.

Returns:

None.

21.2.1.25 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.

21.2.1.26 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.

21.2.1.27 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 full-width operation.

Description:

This function reads the current value of the specified timer.

Note:

This function can be used for both full- and half-width modes of 16/32-bit timers, and for half-width modes of 32/64-bit timers. Use ROM_TimerValueGet64() for full-width modes of 32/64-bit timers.

Returns:

Returns the current value of the timer.

21.2.1.28 ROM_TimerValueGet64

Gets the current 64-bit timer value.

Prototype:

```
unsigned long long
ROM_TimerValueGet64(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_TimerValueGet64` is a function pointer located at `ROM_TIMERTABLE[25]`.

Parameters:

ulBase is the base address of the timer module.

Description:

This function reads the current value of the specified timer.

Returns:

Returns the current value of the timer.

22 UART

Introduction 265
Functions 265

22.1 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

22.2 Functions

Functions

- void ROM_UART9BitAddrSend (unsigned long ulBase, unsigned char ucAddr)
- void ROM_UART9BitAddrSet (unsigned long ulBase, unsigned char ucAddr, unsigned char ucMask)
- void ROM_UART9BitDisable (unsigned long ulBase)
- void ROM_UART9BitEnable (unsigned long ulBase)
- void ROM_UARTBreakCtl (unsigned long ulBase, tBoolean bBreakState)
- tBoolean ROM_UARTBusy (unsigned long ulBase)
- 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)

- unsigned long ROM_UARTClockSourceGet (unsigned long ulBase)
- void ROM_UARTClockSourceSet (unsigned long ulBase, unsigned long ulSource)
- 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_UARTDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)
- void ROM_UARTDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)
- void ROM_UARTEnable (unsigned long ulBase)
- void ROM_UARTEnableSIR (unsigned long ulBase, tBoolean bLowPower)
- void ROM_UARTFIFODisable (unsigned long ulBase)
- void ROM_UARTFIFOEnable (unsigned long ulBase)
- 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)
- void ROM_UARTRxErrorClear (unsigned long ulBase)
- unsigned long ROM_UARTRxErrorGet (unsigned long ulBase)
- tBoolean ROM_UARTSpaceAvail (unsigned long ulBase)
- unsigned long ROM_UARTTxIntModeGet (unsigned long ulBase)
- void ROM_UARTTxIntModeSet (unsigned long ulBase, unsigned long ulMode)
- void ROM_UpdateUART (void)

22.2.1 Function Documentation

22.2.1.1 ROM_UART9BitAddrSend

Sends an address character from the specified port when operating in 9-bit mode.

Prototype:

```
void
ROM_UART9BitAddrSend(unsigned long ulBase,
                     unsigned char ucAddr)
```

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_UART9BitAddrSend` is a function pointer located at `ROM_UARTTABLE[36]`.

Parameters:

ulBase is the base address of the UART port.

ucAddr is the address to be transmitted.

Description:

This function waits until all data has been sent from the specified port and then sends the given address as an address byte. It then waits until the address byte has been transmitted before returning.

The normal data functions (ROM_UARTCharPut(), ROM_UARTCharPutNonBlocking(), ROM_UARTCharGet(), and ROM_UARTCharGetNonBlocking()) are used to send and receive data characters in 9-bit mode.

Returns:

None.

22.2.1.2 ROM_UART9BitAddrSet

Sets the device address(es) for 9-bit mode.

Prototype:

```
void
ROM_UART9BitAddrSet(unsigned long ulBase,
                    unsigned char ucAddr,
                    unsigned char ucMask)
```

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_UART9BitAddrSet` is a function pointer located at `ROM_UARTTABLE[35]`.

Parameters:

ulBase is the base address of the UART port.

ucAddr is the device address.

ucMask is the device address mask.

Description:

This function sets the device address, or range of device addresses, that respond to requests on the 9-bit UART port. The received address is masked with the mask and then compared against the given address, allowing either a single address (if **ucMask** is 0xff) or a set of addresses to be matched.

Returns:

None.

22.2.1.3 ROM_UART9BitDisable

Disables 9-bit mode on the specified UART.

Prototype:

```
void
ROM_UART9BitDisable(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_UART9BitDisable is a function pointer located at ROM_UARTTABLE[34].

Parameters:
ulBase is the base address of the UART port.

Description:
This function disables the 9-bit operational mode of the UART.

Returns:
None.

22.2.1.4 ROM_UART9BitEnable

Enables 9-bit mode on the specified UART.

Prototype:
```
void
ROM_UART9BitEnable(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_UART9BitEnable is a function pointer located at ROM_UARTTABLE[33].

Parameters:
ulBase is the base address of the UART port.

Description:
This function enables the 9-bit operational mode of the UART.

Returns:
None.

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

Returns:
None.

22.2.1.6 ROM_UARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:
```
tBoolean
ROM_UARTBusy(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_UARTBusy` is a function pointer located at `ROM_UARTTABLE[26]`.

Parameters:
ulBase is the base address of the UART port.

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

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

22.2.1.7 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 waits until a character is received before returning.

Returns:
Returns the character read from the specified port, cast as a *long*.

22.2.1.8 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** is returned if there are no characters present in the receive FIFO. The ROM_UARTCharsAvail() function should be called before attempting to call this function.

22.2.1.9 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 waits until there is space available before returning.

Returns:
None.

22.2.1.10 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 must retry the function later.

Returns:
Returns **true** if the character was successfully placed in the transmit FIFO or **false** if there was no space available in the transmit FIFO.

22.2.1.11 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 or **false** if there is no data in the receive FIFO.

22.2.1.12 ROM_UARTClockSourceGet

Gets the baud clock source for the specified UART.

Prototype:

```
unsigned long
ROM_UARTClockSourceGet(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_UARTClockSourceGet` is a function pointer located at `ROM_UARTTABLE[32]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns the baud clock source for the specified UART. The possible baud clock source are the system clock (**UART_CLOCK_SYSTEM**) or the precision internal oscillator (**UART_CLOCK_PIOSC**).

Returns:

None.

22.2.1.13 ROM_UARTClockSourceSet

Sets the baud clock source for the specified UART.

Prototype:

```
void
ROM_UARTClockSourceSet(unsigned long ulBase,
                       unsigned long ulSource)
```

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_UARTClockSourceSet` is a function pointer located at `ROM_UARTTABLE[31]`.

Parameters:

ulBase is the base address of the UART port.

ulSource is the baud clock source for the UART.

Description:

This function allows the baud clock source for the UART to be selected. The possible clock source are the system clock (**UART_CLOCK_SYSTEM**) or the precision internal oscillator (**UART_CLOCK_PIOSC**).

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

Returns:

None.

22.2.1.14 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 is the same as the processor clock. This is 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()).

If the peripheral clock has been changed to PIOSC (via ROM_UARTClockSourceSet()), the peripheral clock should be specified as 16,000,000 (the nominal rate of PIOSC).

Returns:

None.

22.2.1.15 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 configures 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 is the same as the processor clock. This is 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()).

If the peripheral clock has been changed to PIOSC (via ROM_UARTClockSourceSet()), the peripheral clock should be specified as 16,000,000 (the nominal rate of PIOSC).

Returns:

None.

22.2.1.16 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.

22.2.1.17 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.

Returns:

None.

22.2.1.18 ROM_UARTDMADisable

Disable UART DMA operation.

Prototype:

```
void
ROM_UARTDMADisable(unsigned long ulBase,
                   unsigned long ulDMAFlags)
```

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_UARTDMADisable` is a function pointer located at `ROM_UARTTABLE[23]`.

Parameters:

ulBase is the base address of the UART port.
ulDMAFlags is a bit mask of the DMA features to disable.

Description:

This function is used to disable UART DMA features that were enabled by ROM_UARTDMAEnable(). The specified UART DMA features are disabled. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- UART_DMA_RX - disable DMA for receive
- UART_DMA_TX - disable DMA for transmit
- UART_DMA_ERR_RXSTOP - do not disable DMA receive on UART error

Returns:

None.

22.2.1.19 ROM_UARTDMAEnable

Enable UART DMA operation.

Prototype:

```
void
ROM_UARTDMAEnable(unsigned long ulBase,
                  unsigned long ulDMAFlags)
```

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_UARTDMAEnable` is a function pointer located at `ROM_UARTTABLE[22]`.

Parameters:

ulBase is the base address of the UART port.
ulDMAFlags is a bit mask of the DMA features to enable.

Description:

The specified UART DMA features are enabled. The UART can be configured to use DMA for transmit or receive, and to disable receive if an error occurs. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- UART_DMA_RX - enable DMA for receive
- UART_DMA_TX - enable DMA for transmit
- UART_DMA_ERR_RXSTOP - disable DMA receive on UART error

Note:

The uDMA controller must also be set up before DMA can be used with the UART.

Returns:

None.

22.2.1.20 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.

22.2.1.21 ROM_UARTEnableSIR

Enables SIR (IrDA) mode on the 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.

Returns:

None.

22.2.1.22 ROM_UARTFIFODisable

Disables the transmit and receive FIFOs.

Prototype:

```
void
ROM_UARTFIFODisable(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_UARTFIFODisable` is a function pointer located at `ROM_UARTTABLE[25]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This functions disables the transmit and receive FIFOs in the UART.

Returns:

None.

22.2.1.23 ROM_UARTFIFOEnable

Enables the transmit and receive FIFOs.

Prototype:

```
void
ROM_UARTFIFOEnable(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_UARTFIFOEnable` is a function pointer located at `ROM_UARTTABLE[24]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This functions enables the transmit and receive FIFOs in the UART.

Returns:

None.

22.2.1.24 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 are generated.

Returns:

None.

22.2.1.25 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 are generated.

Returns:

None.

22.2.1.26 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.

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

Description:

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

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to ROM_UARTIntEnable().

Note:

Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt

source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:
None.

22.2.1.27 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.
ulIntFlags 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 *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to ROM_UARTIntEnable().

Returns:
None.

22.2.1.28 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.

ulIntFlags 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 *ulIntFlags* parameter is the logical OR of any of the following:

- **UART_INT_9BIT** - 9-bit address match interrupt
- **UART_INT_OE** - Overrun Error interrupt
- **UART_INT_BE** - Break Error interrupt
- **UART_INT_PE** - Parity Error interrupt
- **UART_INT_FE** - Framing Error interrupt
- **UART_INT_RT** - Receive Timeout interrupt
- **UART_INT_TX** - Transmit interrupt
- **UART_INT_RX** - Receive interrupt
- **UART_INT_DSR** - DSR interrupt
- **UART_INT_DCD** - DCD interrupt
- **UART_INT_CTS** - CTS interrupt
- **UART_INT_RI** - RI interrupt

Returns:

None.

22.2.1.29 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:

Returns the current interrupt status, enumerated as a bit field of values described in ROM_UARTIntEnable().

22.2.1.30 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:

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**.

22.2.1.31 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 is always either one or zero based on the mode.

Returns:

None.

22.2.1.32 ROM_UARTRxErrorClear

Clears all reported receiver errors.

Prototype:

```
void
ROM_UARTRxErrorClear(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_UARTRxErrorClear` is a function pointer located at `ROM_UARTTABLE[30]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This function is used to clear all receiver error conditions reported via ROM_UARTRxErrorGet(). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

Returns:

None.

22.2.1.33 ROM_UARTRxErrorGet

Gets current receiver errors.

Prototype:

```
unsigned long
ROM_UARTRxErrorGet(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_UARTRxErrorGet` is a function pointer located at `ROM_UARTTABLE[29]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to ROM_UARTCharGet() or ROM_UARTCharGetNonBlocking() with the exception that the overrun error is set immediately the overrun occurs rather than when a character is next read.

Returns:

Returns a logical OR combination of the receiver error flags, **UART_RXERROR_FRAMING**, **UART_RXERROR_PARITY**, **UART_RXERROR_BREAK** and **UART_RXERROR_OVERRUN**.

22.2.1.34 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 or **false** if there is no space available in the transmit FIFO.

22.2.1.35 ROM_UARTTxIntModeGet

Returns the current operating mode for the UART transmit interrupt.

Prototype:

```
unsigned long
ROM_UARTTxIntModeGet(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_UARTTxIntModeGet` is a function pointer located at `ROM_UARTTABLE[28]`.

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns the current operating mode for the UART transmit interrupt. The return value is **UART_TXINT_MODE_EOT** if the transmit interrupt is currently set to be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter. The return value is **UART_TXINT_MODE_FIFO** if the interrupt is set to be asserted based upon the level of the transmit FIFO.

Returns:

Returns **UART_TXINT_MODE_FIFO** or **UART_TXINT_MODE_EOT**.

22.2.1.36 ROM_UARTTxIntModeSet

Sets the operating mode for the UART transmit interrupt.

Prototype:

```
void
ROM_UARTTxIntModeSet(unsigned long ulBase,
                     unsigned long ulMode)
```

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_UARTTxIntModeSet` is a function pointer located at `ROM_UARTTABLE[27]`.

Parameters:

ulBase is the base address of the UART port.

ulMode is the operating mode for the transmit interrupt. It may be **UART_TXINT_MODE_EOT** to trigger interrupts when the transmitter is idle or **UART_TXINT_MODE_FIFO** to trigger based on the current transmit FIFO level.

Description:

This function allows the mode of the UART transmit interrupt to be set. By default, the transmit interrupt is asserted when the FIFO level falls past a threshold set via a call to ROM_UARTFIFOLevelSet(). Alternatively, if this function is called with *ulMode* set to **UART_TXINT_MODE_EOT**, the transmit interrupt will only be asserted once the transmitter is completely idle - the transmit FIFO is empty and all bits, including any stop bits, have cleared the transmitter.

Returns:

None.

22.2.1.37 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.

23 uDMA Controller

Introduction 287
Functions 289

23.1 Introduction

The microDMA (uDMA) API provides functions to configure the Stellaris uDMA (Direct Memory Access) controller. The uDMA controller is designed to work with the the ARM Cortex-M3 processor and provides an efficient and low-overhead means of transferring blocks of data in the system.

The uDMA controller has the following features:

- dedicated channels for supported peripherals
- one channel each for receive and transmit for devices with receive and transmit paths
- dedicated channel for software initiated data transfers
- channels can be independently configured and operated
- an arbitration scheme that is configurable per channel
- two levels of priority
- subordinate to Cortex-M3 processor bus usage
- data sizes of 8, 16, or 32 bits
- address increment of byte, half-word, word, or none
- maskable device requests
- optional software initiated transfers on any channel
- interrupt on transfer completion

The uDMA controller supports several different transfer modes, allowing for complex transfer schemes. The following transfer modes are provided:

- **Basic** mode performs a simple transfer when request is asserted by a device. This is appropriate to use with peripherals where the peripheral asserts the request line whenever data should be transferred. The transfer will stop if request is de-asserted, even if the transfer is not complete.

- **Auto-request** mode performs a simple transfer that is started by a request, but will always complete the entire transfer, even if request is de-asserted. This is appropriate to use with software initiated transfers.

- **Ping-Pong** mode is used to transfer data to or from two buffers, switching from one buffer to the other as each buffer fills. This mode is appropriate to use with peripherals as a way to ensure a continuous flow of data to or from the peripheral. However, it is more complex to set up and requires code to manage the ping-pong buffers in the interrupt handler.

- **Memory scatter/gather** mode is a complex mode that provides a way to set up a list of transfer “tasks” for the uDMA controller. Blocks of data can be transferred to and from arbitrary locations in memory.

- **Peripheral scatter/gather** mode is similar to memory scatter/gather mode except that it is controlled by a peripheral request.

Detailed explanation of the various transfer modes is beyond the scope of this document. Please refer to the device data sheet for more information on the operation of the uDMA controller.

The naming convention for the microDMA controller is to use the Greek letter “mu” to represent “micro”. For the purposes of this document, and in the software library function names, a lower case “u” will be used in place of “mu” when the controller is referred to as “uDMA”.

The general order of function calls to set up and perform a uDMA transfer is the following:

- ROM_uDMAEnable() is called once to enable the controller.
- ROM_uDMAControlBaseSet() is called once to set the channel control table.
- ROM_uDMAChannelAttributeEnable() is called once or infrequently to configure the behavior of the channel.
- ROM_uDMAChannelControlSet() is used to set up characteristics of the data transfer. It only needs to be called once if the nature of the data transfer does not change.
- ROM_uDMAChannelTransferSet() is used to set the buffer pointers and size for a transfer. It is called before each new transfer.
- ROM_uDMAChannelEnable() enables a channel to perform data transfers.
- ROM_uDMAChannelRequest() is used to initiate a software based transfer. This is normally not used for peripheral based transfers.

In order to use the uDMA controller, you must first enable it by calling ROM_uDMAEnable(). You can later disable it, if no longer needed, by calling ROM_uDMADisable().

Once the uDMA controller is enabled, you must tell it where to find the channel control structures in system memory. This is done by using the function ROM_uDMAControlBaseSet() and passing a pointer to the base of the channel control structure. The control structure must be allocated by the application. One way to do this is to declare an array of data type `char` or `unsigned char`. In order to support all channels and transfer modes, the control table array should be 1024 bytes, but it can be fewer depending on transfer modes used and number of channels actually used.

Note:
The control table must be aligned on a 1024 byte boundary.

The uDMA controller supports multiple channels. Each channel has a set of attribute flags to control certain uDMA features and channel behavior. The attribute flags are set with the function ROM_uDMAChannelAttributeEnable() and cleared with ROM_uDMAChannelAttributeDisable(). The setting of the channel attribute flags can be queried by using the function ROM_uDMAChannelAttributeGet().

Next, the control parameters of the DMA transfer must be set. These parameters control the size and address increment of the data items to be transferred. The function ROM_uDMAChannelControlSet() is used to set up these control parameters.

All of the functions mentioned so far are used only once or infrequently to set up the uDMA channel and transfer. In order to set the transfer addresses, transfer size, and transfer mode, use the function ROM_uDMAChannelTransferSet(). This function must be called for each new transfer. Once everything is set up, then channel is enabled by calling ROM_uDMAChannelEnable(), which must be done before each new transfer. The uDMA controller will automatically disable the channel at the completion of a transfer. A channel can be manually disabled by using ROM_uDMAChannelDisable().

There are additional functions that can be used to query the status of a channel, either from an interrupt handler or in polling fashion. The function ROM_uDMAChannelSizeGet() is used to find the amount of data remaining to transfer on a channel. This will be zero when a transfer is complete. The function ROM_uDMAChannelModeGet() can be used to find the transfer mode of a uDMA channel. This is usually used to see if the mode indicates stopped which means that a transfer has completed on a channel that was previously running. The function ROM_uDMAChannelIsEnabled() can be used to determine if a particular channel is enabled.

The uDMA interrupt handler is only for software initiated transfers or errors. uDMA interrupts for a peripheral occur on the peripheral's dedicated interrupt channel, and should be handled by the peripheral interrupt handler. It is not necessary to acknowledge or clear uDMA interrupt sources. They are cleared automatically when they are serviced.

The uDMA interrupt handler should use the function ROM_uDMAErrorStatusGet() to test if a uDMA error occurred. If so, the interrupt must be cleared by calling ROM_uDMAErrorStatusClear().

Note:

Many of the API functions take a channel parameter that includes the logical OR of one of the values **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose the primary or alternate control structure. For Basic and Auto transfer modes, only the primary control structure is needed. The alternate control structure is only needed for complex transfer modes of Ping-pong or Scatter/gather. Refer to the device data sheet for detailed information about transfer modes.

23.2 Functions

Functions

- void ROM_uDMAChannelAssign (unsigned long ulMapping)
- void ROM_uDMAChannelAttributeDisable (unsigned long ulChannelNum, unsigned long ulAttr)
- void ROM_uDMAChannelAttributeEnable (unsigned long ulChannelNum, unsigned long ulAttr)
- unsigned long ROM_uDMAChannelAttributeGet (unsigned long ulChannelNum)
- void ROM_uDMAChannelControlSet (unsigned long ulChannelStructIndex, unsigned long ulControl)
- void ROM_uDMAChannelDisable (unsigned long ulChannelNum)
- void ROM_uDMAChannelEnable (unsigned long ulChannelNum)
- tBoolean ROM_uDMAChannelIsEnabled (unsigned long ulChannelNum)
- unsigned long ROM_uDMAChannelModeGet (unsigned long ulChannelStructIndex)
- void ROM_uDMAChannelRequest (unsigned long ulChannelNum)
- void ROM_uDMAChannelScatterGatherSet (unsigned long ulChannelNum, unsigned ulTaskCount, void *pvTaskList, unsigned long ulIsPeriphSG)
- void ROM_uDMAChannelSelectDefault (unsigned long ulDefPeriphs)
- void ROM_uDMAChannelSelectSecondary (unsigned long ulSecPeriphs)
- unsigned long ROM_uDMAChannelSizeGet (unsigned long ulChannelStructIndex)
- void ROM_uDMAChannelTransferSet (unsigned long ulChannelStructIndex, unsigned long ulMode, void *pvSrcAddr, void *pvDstAddr, unsigned long ulTransferSize)
- void * ROM_uDMAControlAlternateBaseGet (void)

- void * ROM_uDMAControlBaseGet (void)
- void ROM_uDMAControlBaseSet (void *pControlTable)
- void ROM_uDMADisable (void)
- void ROM_uDMAEnable (void)
- void ROM_uDMAErrorStatusClear (void)
- unsigned long ROM_uDMAErrorStatusGet (void)
- void ROM_uDMAIntClear (unsigned long ulChanMask)
- unsigned long ROM_uDMAIntStatus (void)

23.2.1 Function Documentation

23.2.1.1 ROM_uDMAChannelAssign

Assigns a peripheral mapping for a uDMA channel.

Prototype:

```
void
ROM_uDMAChannelAssign(unsigned long ulMapping)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelAssign` is a function pointer located at `ROM_UDMATABLE[23]`.

Parameters:

ulMapping is a macro specifying the peripheral assignment for a channel

Description:

This function assigns a peripheral mapping to a uDMA channel. It is used to select which peripheral is used for a uDMA channel. The parameter *ulMapping* should be one of the macros named **UDMA_CHn_tttt** from the header file *udma.h*. For example, to assign uDMA channel 0 to the UART2 RX channel, the parameter should be the macro **UDMA_CH0_UART2RX**.

Returns:

None.

23.2.1.2 ROM_uDMAChannelAttributeDisable

Disables attributes of a uDMA channel.

Prototype:

```
void
ROM_uDMAChannelAttributeDisable(unsigned long ulChannelNum,
                                unsigned long ulAttr)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelAttributeDisable` is a function pointer located at `ROM_UDMATABLE[12]`.

Parameters:

ulChannelNum is the channel to configure.

ulAttr is a combination of attributes for the channel.

Description:

This function is used to disable attributes of a uDMA channel.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**

- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

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

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:
None.

23.2.1.3 ROM_uDMAChannelAttributeEnable

Enables attributes of a uDMA channel.

Prototype:

```
void
ROM_uDMAChannelAttributeEnable(unsigned long ulChannelNum,
                               unsigned long ulAttr)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelAttributeEnable` is a function pointer located at `ROM_UDMATABLE[11]`.

Parameters:
ulChannelNum is the channel to configure.
ulAttr is a combination of attributes for the channel.

Description:
This function is used to enable attributes of a uDMA channel.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**

- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**
- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

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

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel (it is very unlikely that this flag should be used).
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.

- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:
None.

23.2.1.4 ROM_uDMAChannelAttributeGet

Gets the enabled attributes of a uDMA channel.

Prototype:
```
unsigned long
ROM_uDMAChannelAttributeGet(unsigned long ulChannelNum)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelAttributeGet` is a function pointer located at `ROM_UDMATABLE[13]`.

Parameters:
ulChannelNum is the channel to configure.

Description:
This function returns a combination of flags representing the attributes of the uDMA channel.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**

- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**
- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

Returns:

Returns the logical OR of the attributes of the uDMA channel, which can be any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

23.2.1.5 ROM_uDMAChannelControlSet

Sets the control parameters for a uDMA channel control structure.

Prototype:

```
void
ROM_uDMAChannelControlSet(unsigned long ulChannelStructIndex,
                          unsigned long ulControl)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelControlSet` is a function pointer located at `ROM_UDMATABLE[14]`.

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ulControl is logical OR of several control values to set the control parameters for the channel.

Description:

This function is used to set control parameters for a uDMA transfer. These are typically parameters that are not changed often.

The *ulChannelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *ulControl* parameter is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. The choices available for each of these values is described below.

Choose the data size from one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32** to select a data size of 8, 16, or 32 bits.

Choose the source address increment from one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

Choose the destination address increment from one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_DST_INC_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

The arbitration size determines how many items are transferred before the uDMA controller re-arbitrates for the bus. Choose the arbitration size from one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, **UDMA_ARB_8**, through **UDMA_ARB_1024** to select the arbitration size from 1 to 1024 items, in powers of 2.

The value **UDMA_NEXT_USEBURST** is used to force the channel to only respond to burst requests at the tail end of a scatter-gather transfer.

Note:

The address increment cannot be smaller than the data size.

Returns:

None.

23.2.1.6 ROM_uDMAChannelDisable

Disables a uDMA channel for operation.

Prototype:

```
void
ROM_uDMAChannelDisable(unsigned long ulChannelNum)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelDisable` is a function pointer located at `ROM_UDMATABLE[6]`.

Parameters:

ulChannelNum is the channel number to disable.

Description:

This function disables a specific uDMA channel. Once disabled, a channel will not respond to uDMA transfer requests until re-enabled via ROM_uDMAChannelEnable().

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**

- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

Returns:
None.

23.2.1.7 ROM_uDMAChannelEnable

Enables a uDMA channel for operation.

Prototype:
```
void
ROM_uDMAChannelEnable(unsigned long ulChannelNum)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelEnable` is a function pointer located at `ROM_UDMATABLE[5]`.

Parameters:
ulChannelNum is the channel number to enable.

Description:
This function enables a specific uDMA channel for use. This function must be used to enable a channel before it can be used to perform a uDMA transfer.

When a uDMA transfer is completed, the channel is automatically disabled by the uDMA controller. Therefore, this function should be called prior to starting up any new transfer.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**

- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**
- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

Returns:
None.

23.2.1.8 ROM_uDMAChannelIsEnabled

Checks if a uDMA channel is enabled for operation.

Prototype:

```
tBoolean
ROM_uDMAChannelIsEnabled(unsigned long ulChannelNum)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelIsEnabled` is a function pointer located at `ROM_UDMATABLE[7]`.

Parameters:

ulChannelNum is the channel number to check.

Description:

This function checks to see if a specific uDMA channel is enabled. This can be used to check the status of a transfer, since the channel will be automatically disabled at the end of a transfer.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**
- **UDMA_CHANNEL_USBEP1RX**

- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

Returns:

Returns **true** if the channel is enabled, **false** if disabled.

23.2.1.9 ROM_uDMAChannelModeGet

Gets the transfer mode for a uDMA channel control structure.

Prototype:

```
unsigned long
ROM_uDMAChannelModeGet(unsigned long ulChannelStructIndex)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelModeGet` is a function pointer located at `ROM_UDMATABLE[16]`.

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the transfer mode for the uDMA channel. It can be used to query the status of a transfer on a channel. When the transfer is complete the mode is **UDMA_MODE_STOP**.

Returns:

Returns the transfer mode of the specified channel and control structure, which is one of the following values: **UDMA_MODE_STOP**, **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_PINGPONG**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**.

23.2.1.10 ROM_uDMAChannelRequest

Requests a uDMA channel to start a transfer.

Prototype:

```
void
ROM_uDMAChannelRequest(unsigned long ulChannelNum)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelRequest` is a function pointer located at `ROM_UDMATABLE[10]`.

Parameters:

ulChannelNum is the channel number on which to request a uDMA transfer.

Description:

This function allows software to request a uDMA channel to begin a transfer. This could be used for performing a memory to memory transfer, or if for some reason a transfer needs to be initiated by software instead of the peripheral associated with that channel.

The *ulChannelNum* parameter must be only one of the following values:

- **UDMA_CHANNEL_ADC0**
- **UDMA_CHANNEL_ADC1**
- **UDMA_CHANNEL_ADC2**
- **UDMA_CHANNEL_ADC3**
- **UDMA_SEC_CHANNEL_ADC10**
- **UDMA_SEC_CHANNEL_ADC11**
- **UDMA_SEC_CHANNEL_ADC12**
- **UDMA_SEC_CHANNEL_ADC13**
- **UDMA_CHANNEL_SSI0RX**
- **UDMA_CHANNEL_SSI0TX**
- **UDMA_CHANNEL_SSI1RX**
- **UDMA_CHANNEL_SSI1TX**
- **UDMA_SEC_CHANNEL_SSI1RX**
- **UDMA_SEC_CHANNEL_SSI1TX**
- **UDMA_CHANNEL_TMR0A**
- **UDMA_CHANNEL_TMR0B**
- **UDMA_CHANNEL_TMR1A**
- **UDMA_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR1A**
- **UDMA_SEC_CHANNEL_TMR1B**
- **UDMA_SEC_CHANNEL_TMR2A_4**
- **UDMA_SEC_CHANNEL_TMR2B_5**
- **UDMA_SEC_CHANNEL_TMR2A_6**
- **UDMA_SEC_CHANNEL_TMR2B_7**
- **UDMA_SEC_CHANNEL_TMR2A_14**
- **UDMA_SEC_CHANNEL_TMR2B_15**
- **UDMA_SEC_CHANNEL_TMR3A**
- **UDMA_SEC_CHANNEL_TMR3B**
- **UDMA_CHANNEL_UART0RX**
- **UDMA_CHANNEL_UART0TX**
- **UDMA_CHANNEL_UART1RX**
- **UDMA_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART1RX**
- **UDMA_SEC_CHANNEL_UART1TX**
- **UDMA_SEC_CHANNEL_UART2RX_0**
- **UDMA_SEC_CHANNEL_UART2TX_1**
- **UDMA_SEC_CHANNEL_UART2RX_12**
- **UDMA_SEC_CHANNEL_UART2TX_13**

- **UDMA_CHANNEL_USBEP1RX**
- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**
- **UDMA_CHANNEL_SW**
- **UDMA_SEC_CHANNEL_SW**

Note:
If the channel is **UDMA_CHANNEL_SW** and interrupts are used, then the completion is signaled on the uDMA dedicated interrupt. If a peripheral channel is used, then the completion is signaled on the peripheral's interrupt.

Returns:
None.

23.2.1.11 ROM_uDMAChannelScatterGatherSet

Configures a uDMA channel for scatter-gather mode.

Prototype:
```
void
ROM_uDMAChannelScatterGatherSet(unsigned long ulChannelNum,
                                unsigned ulTaskCount,
                                void *pvTaskList,
                                unsigned long ulIsPeriphSG)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelScatterGatherSet` is a function pointer located at `ROM_UDMATABLE[22]`.

Parameters:
ulChannelNum is the uDMA channel number.
ulTaskCount is the number of scatter-gather tasks to execute.
pvTaskList is a pointer to the beginning of the scatter-gather task list.
ulIsPeriphSG is a flag to indicate it is a peripheral scatter-gather transfer (else it is memory scatter-gather transfer)

Description:
This function is used to configure a channel for scatter-gather mode. The caller must have already set up a task list, and pass a pointer to the start of the task list as the *pvTaskList* parameter. The *ulTaskCount* parameter is the count of tasks in the task list, not the size of the task list. The flag *bIsPeriphSG* should be used to indicate if the scatter-gather should be configured for a peripheral or memory scatter-gather operation.

Returns:
None.

23.2.1.12 ROM_uDMAChannelSelectDefault

Selects the default peripheral for a set of uDMA channels.

Prototype:

```
void
ROM_uDMAChannelSelectDefault(unsigned long ulDefPeriphs)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelSelectDefault` is a function pointer located at `ROM_UDMATABLE[18]`.

Parameters:

ulDefPeriphs is the logical or of the uDMA channels for which to use the default peripheral, instead of the secondary peripheral.

Description:

This function is used to select the default peripheral assignment for a set of uDMA channels.

The parameter *ulDefPeriphs* can be the logical OR of any of the following macros. If one of the macros below is in the list passed to this function, then the default peripheral (marked as **_DEF_**) is selected.

- **UDMA_DEF_USBEP1RX_SEC_UART2RX**
- **UDMA_DEF_USBEP1TX_SEC_UART2TX**
- **UDMA_DEF_USBEP2RX_SEC_TMR3A**
- **UDMA_DEF_USBEP2TX_SEC_TMR3B**
- **UDMA_DEF_USBEP3RX_SEC_TMR2A**
- **UDMA_DEF_USBEP3TX_SEC_TMR2B**
- **UDMA_DEF_UART0RX_SEC_UART1RX**
- **UDMA_DEF_UART0TX_SEC_UART1TX**
- **UDMA_DEF_SSI0RX_SEC_SSI1RX**
- **UDMA_DEF_SSI0TX_SEC_SSI1TX**
- **UDMA_DEF_ADC00_SEC_TMR2A**
- **UDMA_DEF_ADC01_SEC_TMR2B**
- **UDMA_DEF_ADC02_SEC_RESERVED**
- **UDMA_DEF_ADC03_SEC_RESERVED**
- **UDMA_DEF_TMR0A_SEC_TMR1A**
- **UDMA_DEF_TMR0B_SEC_TMR1B**
- **UDMA_DEF_TMR1A_SEC_EPI0RX**
- **UDMA_DEF_TMR1B_SEC_EPI0TX**
- **UDMA_DEF_UART1RX_SEC_RESERVED**
- **UDMA_DEF_UART1TX_SEC_RESERVED**
- **UDMA_DEF_SSI1RX_SEC_ADC10**
- **UDMA_DEF_SSI1TX_SEC_ADC11**

Returns:

None.

23.2.1.13 ROM_uDMAChannelSelectSecondary

Selects the secondary peripheral for a set of uDMA channels.

Prototype:

```
void
ROM_uDMAChannelSelectSecondary(unsigned long ulSecPeriphs)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelSelectSecondary` is a function pointer located at `ROM_UDMATABLE[17]`.

Parameters:

ulSecPeriphs is the logical or of the uDMA channels for which to use the secondary peripheral, instead of the default peripheral.

Description:

This function is used to select the secondary peripheral assignment for a set of uDMA channels. By selecting the secondary peripheral assignment for a channel, the default peripheral assignment is no longer available for that channel.

The parameter *ulSecPeriphs* can be the logical OR of any of the following macros. If one of the macros below is in the list passed to this function, then the secondary peripheral (marked as **_SEC_**) is selected.

- **UDMA_DEF_USBEP1RX_SEC_UART2RX**
- **UDMA_DEF_USBEP1TX_SEC_UART2TX**
- **UDMA_DEF_USBEP2RX_SEC_TMR3A**
- **UDMA_DEF_USBEP2TX_SEC_TMR3B**
- **UDMA_DEF_USBEP3RX_SEC_TMR2A**
- **UDMA_DEF_USBEP3TX_SEC_TMR2B**
- **UDMA_DEF_ETH0RX_SEC_TMR2A**
- **UDMA_DEF_ETH0TX_SEC_TMR2B**
- **UDMA_DEF_UART0RX_SEC_UART1RX**
- **UDMA_DEF_UART0TX_SEC_UART1TX**
- **UDMA_DEF_SSI0RX_SEC_SSI1RX**
- **UDMA_DEF_SSI0TX_SEC_SSI1TX**
- **UDMA_DEF_RESERVED_SEC_UART2RX**
- **UDMA_DEF_RESERVED_SEC_UART2TX**
- **UDMA_DEF_ADC00_SEC_TMR2A**
- **UDMA_DEF_ADC01_SEC_TMR2B**
- **UDMA_DEF_TMR0A_SEC_TMR1A**
- **UDMA_DEF_TMR0B_SEC_TMR1B**
- **UDMA_DEF_SSI1RX_SEC_ADC10**
- **UDMA_DEF_SSI1TX_SEC_ADC11**
- **UDMA_DEF_RESERVED_SEC_ADC12**
- **UDMA_DEF_RESERVED_SEC_ADC13**

Returns:

None.

23.2.1.14 ROM_uDMAChannelSizeGet

Gets the current transfer size for a uDMA channel control structure.

Prototype:

```
unsigned long
ROM_uDMAChannelSizeGet(unsigned long ulChannelStructIndex)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelSizeGet` is a function pointer located at `ROM_UDMATABLE[15]`.

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the uDMA transfer size for a channel. The transfer size is the number of items to transfer, where the size of an item might be 8, 16, or 32 bits. If a partial transfer has already occurred, then the number of remaining items is returned. If the transfer is complete, then 0 is returned.

Returns:

Returns the number of items remaining to transfer.

23.2.1.15 ROM_uDMAChannelTransferSet

Sets the transfer parameters for a uDMA channel control structure.

Prototype:

```
void
ROM_uDMAChannelTransferSet(unsigned long ulChannelStructIndex,
                           unsigned long ulMode,
                           void *pvSrcAddr,
                           void *pvDstAddr,
                           unsigned long ulTransferSize)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAChannelTransferSet` is a function pointer located at `ROM_UDMATABLE[0]`.

Parameters:

ulChannelStructIndex is the logical OR of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ulMode is the type of uDMA transfer.

pvSrcAddr is the source address for the transfer.

pvDstAddr is the destination address for the transfer.

ulTransferSize is the number of data items to transfer.

Description:

This function is used to set the parameters for a uDMA transfer. These are typically parameters that are changed often. The function ROM_uDMAChannelControlSet() MUST be called at least once for this channel prior to calling this function.

The *ulChannelStructIndex* parameter should be the logical OR of the channel number with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *ulMode* parameter should be one of the following values:

- **UDMA_MODE_STOP** stops the uDMA transfer. The controller sets the mode to this value at the end of a transfer.
- **UDMA_MODE_BASIC** to perform a basic transfer based on request.
- **UDMA_MODE_AUTO** to perform a transfer that will always complete once started even if request is removed.
- **UDMA_MODE_PINGPONG** to set up a transfer that switches between the primary and alternate control structures for the channel. This allows use of ping-pong buffering for uDMA transfers.
- **UDMA_MODE_MEM_SCATTER_GATHER** to set up a memory scatter-gather transfer.
- **UDMA_MODE_PER_SCATTER_GATHER** to set up a peripheral scatter-gather transfer.

The *pvSrcAddr* and *pvDstAddr* parameters are pointers to the first location of the data to be transferred. These addresses should be aligned according to the item size. The compiler will take care of this if the pointers are pointing to storage of the appropriate data type.

The *ulTransferSize* parameter is the number of data items, not the number of bytes.

The two scatter/gather modes, memory and peripheral, are actually different depending on whether the primary or alternate control structure is selected. This function will look for the **UDMA_PRI_SELECT** and **UDMA_ALT_SELECT** flag along with the channel number and will set the scatter/gather mode as appropriate for the primary or alternate control structure.

The channel must also be enabled using ROM_uDMAChannelEnable() after calling this function. The transfer will not begin until the channel has been set up and enabled. Note that the channel is automatically disabled after the transfer is completed, meaning that ROM_uDMAChannelEnable() must be called again after setting up the next transfer.

Note:

Great care must be taken to not modify a channel control structure that is in use or else the results are unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the ROM_uDMAChannelModeGet() returns **UDMA_MODE_STOP**. For PINGPONG or one of the SCATTER_GATHER modes, it is safe to modify the primary or alternate control structure only when the other is being used. The ROM_uDMAChannelModeGet() function will return **UDMA_MODE_STOP** when a channel control structure is inactive and safe to modify.

Returns:

None.

23.2.1.16 ROM_uDMAControlAlternateBaseGet

Gets the base address for the channel control table alternate structures.

Prototype:

```
void *
ROM_uDMAControlAlternateBaseGet(void)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_UDMATABLE is an array of pointers located at ROM_APITABLE[17].
ROM_uDMAControlAlternateBaseGet is a function pointer located at ROM_UDMATABLE[21].

Description:

This function gets the base address of the second half of the channel control table that holds the alternate control structures for each channel.

Returns:

Returns a pointer to the base address of the second half of the channel control table.

23.2.1.17 ROM_uDMAControlBaseGet

Gets the base address for the channel control table.

Prototype:

```
void *
ROM_uDMAControlBaseGet(void)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_UDMATABLE is an array of pointers located at ROM_APITABLE[17].
ROM_uDMAControlBaseGet is a function pointer located at ROM_UDMATABLE[9].

Description:

This function gets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel.

Returns:

Returns a pointer to the base address of the channel control table.

23.2.1.18 ROM_uDMAControlBaseSet

Sets the base address for the channel control table.

Prototype:

```
void
ROM_uDMAControlBaseSet(void *pControlTable)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_UDMATABLE is an array of pointers located at ROM_APITABLE[17].
ROM_uDMAControlBaseSet is a function pointer located at ROM_UDMATABLE[8].

Parameters:

pControlTable is a pointer to the 1024 byte aligned base address of the uDMA channel control table.

Description:

This function sets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel. The table must be aligned on a 1024 byte boundary. The base address must be set before any of the channel functions can be used.

The size of the channel control table depends on the number of uDMA channels, and which transfer modes are used. Refer to the introductory text and the microcontroller data sheet for more information about the channel control table.

Returns:

None.

23.2.1.19 ROM_uDMADisable

Disables the uDMA controller for use.

Prototype:

```
void
ROM_uDMADisable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMADisable` is a function pointer located at `ROM_UDMATABLE[2]`.

Description:

This function disables the uDMA controller. Once disabled, the uDMA controller will not operate until re-enabled with ROM_uDMAEnable().

Returns:

None.

23.2.1.20 ROM_uDMAEnable

Enables the uDMA controller for use.

Prototype:

```
void
ROM_uDMAEnable(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAEnable` is a function pointer located at `ROM_UDMATABLE[1]`.

Description:

This function enables the uDMA controller. The uDMA controller must be enabled before it can be configured and used.

Returns:

None.

23.2.1.21 ROM_uDMAErrorStatusClear

Clears the uDMA error interrupt.

Prototype:

```
void
ROM_uDMAErrorStatusClear(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAErrorStatusClear` is a function pointer located at `ROM_UDMATABLE[4]`.

Description:

This function clears a pending uDMA error interrupt. It should be called from within the uDMA error interrupt handler to clear the interrupt.

Returns:

None.

23.2.1.22 ROM_uDMAErrorStatusGet

Gets the uDMA error status.

Prototype:

```
unsigned long
ROM_uDMAErrorStatusGet(void)
```

ROM Location:

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

Description:

This function returns the uDMA error status. It should be called from within the uDMA error interrupt handler to determine if a uDMA error occurred.

Returns:

Returns non-zero if a uDMA error is pending.

23.2.1.23 ROM_uDMAIntClear

Clears uDMA interrupt status.

Prototype:

```
void
ROM_uDMAIntClear(unsigned long ulChanMask)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAIntClear` is a function pointer located at `ROM_UDMATABLE[20]`.

Parameters:

ulChanMask is a 32-bit mask with one bit for each uDMA channel.

Description:

Clears bits in the uDMA interrupt status register according to which bits are set in *ulChanMask*. There is one bit for each channel. If a a bit is set in *ulChanMask*, then that corresponding channel's interrupt status is cleared (if it was set).

Returns:

None.

23.2.1.24 ROM_uDMAIntStatus

Gets the uDMA controller channel interrupt status.

Prototype:

```
unsigned long
ROM_uDMAIntStatus(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UDMATABLE` is an array of pointers located at `ROM_APITABLE[17]`.
`ROM_uDMAIntStatus` is a function pointer located at `ROM_UDMATABLE[19]`.

Description:

This function is used to get the interrupt status of the uDMA controller. The returned value is a 32-bit bit mask that indicates which channels are requesting an interrupt. This function can be used from within an interrupt handler to determine or confirm which uDMA channel has requested an interrupt.

Returns:

Returns a 32-bit mask which indicates requesting uDMA channels. There is a bit for each channel, and a 1 in a bit indicates that channel is requesting an interrupt. Multiple bits can be set.

24 USB Controller

Introduction 313
Using uDMA with USB 314
Functions 318

24.1 Introduction

The USB APIs provide a set of functions that are used to access the Stellaris USB device or host controllers. The APIs are split into groups according to the functionality provided by the USB controller present in the microcontroller. Because of this, the driver has to handle microcontrollers that have only a USB device interface, a host and/or device interface, or microcontrollers that have an OTG interface, The groups are the following: USBDev, USBHost, USBOTG, USBEndpoint, and USBFIFO. The APIs in the USBDev group are only used with microcontrollers that have a USB device controller. The APIs in the USBHost group can only be used with microcontrollers that have a USB host controller. The USBOTG APIs are used by microcontrollers with an OTG interface. With USB OTG controllers, once the mode of the USB controller is configured, the device or host APIs should be used. The remainder of the APIs are used for both USB host and USB device controllers. The USBEndpoint APIs are used to configure and access the endpoints while the USBFIFO APIs are used to configure the size and location of the FIFOs.

The USB APIs abstract the IN/OUT nature of endpoints based on the type of USB controller that is in use. Any API that uses the IN/OUT terminology will comply with the standard USB interpretation of these terms. For example, an OUT endpoint on a microcontroller that has only a device interface will actually receive data on this endpoint, while a microcontroller that has a host interface will actually transmit data on an OUT endpoint.

Another important fact to understand is that all endpoints in the USB controller, whether host or device, have two “sides” to them. This allows each endpoint to both transmit and receive data. An application can use a single endpoint for both and IN and OUT transactions. For example: In device mode, endpoint 1 could be configured to have BULK IN and BULK OUT handled by endpoint 1. It is important to note that the endpoint number used is the endpoint number reported to the host. For microcontrollers with host controllers, the application can use an endpoint communicate with both IN and OUT endpoints of different types as well. For example: Endpoint 2 could be used to communicate with one device’s interrupt IN endpoint and another device’s bulk OUT endpoint at the same time. This effectively gives the application one dedicated control endpoint for IN or OUT control transactions on endpoint 0, and three IN endpoints and three OUT endpoints.

The USB controller has a configurable FIFOs in devices that have a USB device controller as well as those that have a host controller. The overall size of the FIFO RAM is 4096 bytes. It is important to note that the first 64 bytes of this memory are dedicated to endpoint 0 for control transactions. The remaining 4032 bytes are configurable however the application desires. The FIFO configuration is usually set at the beginning of the application and not modified once the USB controller is in use. The FIFO configuration uses the ROM_USBFIFOConfigSet() API to set the starting address and the size of the FIFOs that are dedicated to each endpoint.

Example: FIFO Configuration

```
//
// 0-64        - endpoint 0 IN/OUT (64 bytes).
//
```

```
// 64-576      - endpoint 1 IN     (512 bytes).
//
// 576-1088    - endpoint 1 OUT    (512 bytes).
//
// 1088-1600   - endpoint 2 IN     (512 bytes).
//

//
// FIFO for endpoint 1 IN starts at address 64 and is 512 bytes in size.
//
ROM_USBFIFOConfigSet(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_512,
                     USB_EP_DEV_IN);

//
// FIFO for endpoint 1 OUT starts at address 576 and is 512 bytes in size.
//
ROM_USBFIFOConfigSet(USB0_BASE, USB_EP_1, 576, USB_FIFO_SZ_512,
                     USB_EP_DEV_OUT);

//
// FIFO for endpoint 2 IN starts at address 1088 and is 512 bytes in size.
//
ROM_USBFIFOConfigSet(USB0_BASE, USB_EP_2, 1088, USB_FIFO_SZ_512,
                     USB_EP_DEV_IN);
```

24.2 Using USB with the uDMA Controller

The USB controller can be used with the uDMA for either sending or receiving data with both host and device controllers. The uDMA controller cannot be used to access endpoint 0, however all other endpoints are capable of using the uDMA controller. The uDMA channel numbers for USB are defined by the following values:

- **UDMA_CHANNEL_USBEP1RX**
- **UDMA_CHANNEL_USBEP1TX**
- **UDMA_CHANNEL_USBEP2RX**
- **UDMA_CHANNEL_USBEP2TX**
- **UDMA_CHANNEL_USBEP3RX**
- **UDMA_CHANNEL_USBEP3TX**

Since the uDMA controller views transfers as either transmit or receive, and the USB controller operates on IN/OUT transactions, some care must be taken to use the correct uDMA channel with the correct endpoint. USB host IN and USB device OUT endpoints both use receive uDMA channels while USB host OUT and USB device IN endpoints will use transmit uDMA channels.

When configuring the endpoint there are additional DMA settings needed. When calling ROM_USBDevEndpointConfigSet() for an endpoint that will use uDMA, extra flags need to be added to the *ulFlags* parameter. These flags are one of **USB_EP_DMA_MODE_0** or **USB_EP_DMA_MODE_1** to control the mode of the DMA transaction, and likely **USB_EP_AUTO_SET** to allow the data to be transmitted automatically once a packet is ready. **USB_EP_DMA_MODE_0** will generate an interrupt whenever there is more space available in the FIFO. This allows the application code to perform operations between each packet. **USB_EP_DMA_MODE_1** will only interrupt when the DMA transfer complete or there is some type of error condition. This can be used for larger transmissions that require no interaction between packets. **USB_EP_AUTO_SET** should normally be specified when using uDMA to prevent the need for application code to start the actual transfer of data.

Example: Endpoint configuration for a device IN endpoint:

```
//
// Endpoint 1 is a device mode BULK IN endpoint using uDMA.
//
ROM_USBDevEndpointConfigSet(USB0_BASE, USB_EP_1, 64,
                            (USB_EP_MODE_BULK | USB_EP_DEV_IN |
                             USB_EP_DMA_MODE_0 | USB_EP_AUTO_SET));
```

The application must provide the configuration of the actual uDMA controller. First, to clear out any previous settings, the application should call ROM_uDMAChannelAttributeDisable(). Then the application should call ROM_uDMAChannelAttributeEnable() for the uDMA channel that corresponds to the endpoint, and specify the **UDMA_ATTR_USEBURST** flag.

Note:
All uDMA transfers used by the USB controller must enable burst mode.

The application needs to indicate the size of each uDMA transactions, combined with the source and destination increments and the arbitration level for the uDMA controller.

Example: Configure endpoint 1 transmit channel.

```
//
// Set up the DMA for USB transmit.
//
ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_USBEP1TX, UDMA_ATTR_ALL);

//
// Enable uDMA burst mode.
//
ROM_uDMAChannelAttributeEnable(UDMA_CHANNEL_USBEP1TX, UDMA_ATTR_USEBURST);

//
// Data size is 8 bits and the source has a one byte increment.
// Destination has no increment as it is a FIFO.
//
ROM_uDMAChannelControlSet(UDMA_CHANNEL_USBEP1TX,
                          (UDMA_SIZE_8 | UDMA_SRC_INC_8 |
                           UDMA_DST_INC_NONE | UDMA_ARB_64));
```

The next step is to actually start the uDMA transfer once the data is ready to be sent. There are the only two calls that the application needs to call to start a new transfer. Normally all of the previous uDMA configuration can stay the same. The first call, ROM_uDMAChannelTransferSet(), resets the source and destination addresses for the DMA transfer and specifies how much data will be sent. The next call, ROM_uDMAChannelEnable() actually allows the uDMA controller to begin requesting data.

Example: Start the transfer of data on endpoint 1.

```
//
// Configure the address and size of the data to transfer.
//
ROM_uDMAChannelTransferSet(UDMA_CHANNEL_USBEP1TX, UDMA_MODE_BASIC, pData,
                           (void *)ROM_USBFIFOAddr(USB0_BASE, USB_EP_1),
                           64);
//
// Start the transfer.
//
ROM_uDMAChannelEnable(UDMA_CHANNEL_USBEP1TX);
```

Because the uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, the application must perform an extra check to determine what was the actual source of the interrupt. It is important to note that this DMA interrupt does not mean that the USB transfer is complete, but that the data has been transferred to the USB controller's FIFO. There will also be an interrupt indicating that the USB transfer is complete. However, both events need to be handled in the same interrupt routine. This because if other code in the system holds off the USB interrupt routine, both the uDMA complete and transfer complete can occur before the USB interrupt handler is called. The USB has no status bit indicating that the interrupt was due to a DMA complete, which means that the application must remember if a uDMA transaction was in progress. The example below shows the g_ulFlags global variable being used to remember that a uDMA transfer was pending.

Example: Interrupt handling with uDMA.

```
if((g_ulFlags & EP1_DMA_IN_PEND) &&
   (ROM_uDMAChannelModeGet(UDMA_CHANNEL_USBEP1TX) == UDMA_MODE_STOP))
{
    //
    // Handle the uDMA complete case.
    //
    ...
}

//
// Get the interrupt status.
//
ulStatus = ROM_USBIntStatus(USB0_BASE);

if(ulStatus & USB_INT_DEV_IN_EP1)
{
    //
    // Handler the transfer complete case.
    //
    ...
}
```

To use the USB device controller with an OUT endpoint, the application must use a receive uDMA channel. When calling ROM_USBDevEndpointConfigSet() for an endpoint that uses uDMA, the application must set extra flags in the *ulFlags* parameter. The **USB_EP_DMA_MODE_0** and **USB_EP_DMA_MODE_1** control the mode of the transaction, **USB_EP_AUTO_CLEAR** allows the data to be received automatically without needing to manually acknowledge that the data has been read. **USB_EP_DMA_MODE_0** will not generate an interrupt when each packet is sent over USB and will only interrupt when the uDMA transfer is complete. **USB_EP_DMA_MODE_1** will interrupt when the uDMA transfer complete or a short packet is received. This is useful for BULK endpoints that may not have prior knowledge of how much data is being received. **USB_EP_AUTO_CLEAR** should normally be specified when using uDMA to prevent the need for application code to acknowledge that the data has been read from the FIFO. The example below configures endpoint 1 as a Device mode Bulk OUT endpoint using DMA mode 1 with a max packet size of 64 bytes.

Example: Configure endpoint 1 receive channel:

```
//
// Endpoint 1 is a device mode BULK OUT endpoint using uDMA.
//
ROM_USBDevEndpointConfigSet(USB0_BASE, USB_EP_1, 64,
                            (USB_EP_DEV_OUT | USB_EP_MODE_BULK |
                             USB_EP_DMA_MODE_1 | USB_EP_AUTO_CLEAR));
```

Next the configuration of the actual uDMA controller is needed. Like the transmit case, the first a call to ROM_uDMAChannelAttributeDisable() is made to clear any previous settings. This is followed

by a call to ROM_uDMAChannelAttributeEnable() with the **DMA_ATTR_USEBURST** value.

Note:
All uDMA transfers used by the USB controller must use burst mode.

The final call sets the read access size to 8 bits wide, the source address increment to 0, the destination address increment to 8 bits and the uDMA arbitration size to 64 bytes.

Example: Configure endpoint 1 transmit channel.

```
//
// Clear out any uDMA settings.
//
ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_USBEP1RX, UDMA_ATTR_ALL);

ROM_uDMAChannelAttributeEnable(UDMA_CHANNEL_USBEP1RX, UDMA_ATTR_USEBURST);

ROM_uDMAChannelControlSet(UDMA_CHANNEL_USBEP1RX,
                          (UDMA_SIZE_8 | UDMA_SRC_INC_NONE |
                           UDMA_DST_INC_8 | UDMA_ARB_64));
```

The next step is to actually start the uDMA transfer. Unlike the transfer side, if the application is ready, this can be set up right away to wait for incoming data. Like the transmit case, these are the only calls needed to start a new transfer, normally all of the previous uDMA configuration can remain the same.

Example: Start requesting of data on endpoint 1.

```
//
// Configure the address and size of the data to transfer.  The transfer
// is from the USB FIFO for endpoint 0 to g_DataBufferIn.
//
ROM_uDMAChannelTransferSet(UDMA_CHANNEL_USBEP1RX, UDMA_MODE_BASIC,
                           (void *)ROM_USBFIFOAddr(USB0_BASE, USB_EP_1),
                           g_DataBufferIn, 64);

//
// Enable the uDMA channel and wait for data.
//
ROM_uDMAChannelEnable(UDMA_CHANNEL_USBEP1RX);
```

The uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, this means that the application needs to check to see what was the actual source of the interrupt. It is possible that the USB interrupt does not indicate that the USB transfer was complete. The interrupt could also have been caused by a short packet, error, or even a transmit complete. This requires that the application check both receive cases to determine if this is related to receiving data on the endpoint. Because the USB has no status bit indicating that the interrupt was due to a uDMA complete, the application must remember if a uDMA transaction was in progress.

Example: Interrupt handling with uDMA.

```
//
// Get the current interrupt status.
//
ulStatus = ROM_USBIntStatus(USB0_BASE);

if(ulStatus & USB_INT_DEV_OUT_EP1)
{
    //
```

```
    // Handle a short packet.
    //
    ...
}
else if((g_ulFlags & EP1_DMA_OUT_PEND) &&
        (ROM_uDMAChannelModeGet(UDMA_CHANNEL_USBEP1RX) == UDMA_MODE_STOP))
{
    //
    // Handle the uDMA complete case.
    //
    ...

    //
    // Restart receive uDMA if desired.
    //
    ...
}
```

24.3 Functions

Functions

- void ROM_UpdateUSB (unsigned char *pucUSBBootROMInfo)
- unsigned long ROM_USBDevAddrGet (unsigned long ulBase)
- void ROM_USBDevAddrSet (unsigned long ulBase, unsigned long ulAddress)
- void ROM_USBDevConnect (unsigned long ulBase)
- void ROM_USBDevDisconnect (unsigned long ulBase)
- void ROM_USBDevEndpointConfigGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long *pulMaxPacketSize, unsigned long *pulFlags)
- void ROM_USBDevEndpointConfigSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPacketSize, unsigned long ulFlags)
- void ROM_USBDevEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bIsLastPacket)
- void ROM_USBDevEndpointStall (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBDevEndpointStallClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBDevEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBDevMode (unsigned long ulBase)
- unsigned long ROM_USBEndpointDataAvail (unsigned long ulBase, unsigned long ulEndpoint)
- long ROM_USBEndpointDataGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long *pulSize)
- long ROM_USBEndpointDataPut (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long ulSize)
- long ROM_USBEndpointDataSend (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulTransType)
- void ROM_USBEndpointDataToggleClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)

- void ROM_USBEndpointDMAChannel (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulChannel)
- void ROM_USBEndpointDMADisable (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBEndpointDMAEnable (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long ROM_USBEndpointStatus (unsigned long ulBase, unsigned long ulEndpoint)
- unsigned long ROM_USBFIFOAddrGet (unsigned long ulBase, unsigned long ulEndpoint)
- void ROM_USBFIFOConfigGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long *pulFIFOAddress, unsigned long *pulFIFOSize, unsigned long ulFlags)
- void ROM_USBFIFOConfigSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFIFOAddress, unsigned long ulFIFOSize, unsigned long ulFlags)
- void ROM_USBFIFOFlush (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long ROM_USBFrameNumberGet (unsigned long ulBase)
- unsigned long ROM_USBHostAddrGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBHostAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)
- void ROM_USBHostEndpointConfig (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPayload, unsigned long ulNAKPollInterval, unsigned long ulTargetEndpoint, unsigned long ulFlags)
- void ROM_USBHostEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint)
- void ROM_USBHostEndpointDataToggle (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bDataToggle, unsigned long ulFlags)
- void ROM_USBHostEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long ROM_USBHostHubAddrGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void ROM_USBHostHubAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)
- void ROM_USBHostMode (unsigned long ulBase)
- void ROM_USBHostPwrConfig (unsigned long ulBase, unsigned long ulFlags)
- void ROM_USBHostPwrDisable (unsigned long ulBase)
- void ROM_USBHostPwrEnable (unsigned long ulBase)
- void ROM_USBHostPwrFaultDisable (unsigned long ulBase)
- void ROM_USBHostPwrFaultEnable (unsigned long ulBase)
- void ROM_USBHostRequestIN (unsigned long ulBase, unsigned long ulEndpoint)
- void ROM_USBHostRequestStatus (unsigned long ulBase)
- void ROM_USBHostReset (unsigned long ulBase, tBoolean bStart)
- void ROM_USBHostResume (unsigned long ulBase, tBoolean bStart)
- unsigned long ROM_USBHostSpeedGet (unsigned long ulBase)
- void ROM_USBHostSuspend (unsigned long ulBase)
- void ROM_USBIntDisable (unsigned long ulBase, unsigned long ulFlags)
- void ROM_USBIntDisableControl (unsigned long ulBase, unsigned long ulFlags)
- void ROM_USBIntDisableEndpoint (unsigned long ulBase, unsigned long ulFlags)
- void ROM_USBIntEnable (unsigned long ulBase, unsigned long ulFlags)
- void ROM_USBIntEnableControl (unsigned long ulBase, unsigned long ulFlags)

- void ROM_USBIntEnableEndpoint (unsigned long ulBase, unsigned long ulFlags)
- unsigned long ROM_USBIntStatus (unsigned long ulBase)
- unsigned long ROM_USBIntStatusControl (unsigned long ulBase)
- unsigned long ROM_USBIntStatusEndpoint (unsigned long ulBase)
- unsigned long ROM_USBModeGet (unsigned long ulBase)
- void ROM_USBOTGHostRequest (unsigned long ulBase, tBoolean bStart)
- void ROM_USBOTGMode (unsigned long ulBase)
- void ROM_USBPHYPowerOff (unsigned long ulBase)
- void ROM_USBPHYPowerOn (unsigned long ulBase)

24.3.1 Function Documentation

24.3.1.1 ROM_UpdateUSB

Starts an update over the USB interface.

Prototype:

```
void
ROM_UpdateUSB(unsigned char *pucUSBBootROMInfo)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_UpdateUSB` is a function pointer located at `ROM_USBTABLE[58]`.

Parameters:

pucUSBBootROMInfo is a pointer to an array containing the values that are used to customize the USB interface.

Description:

Calling this function commences an update of the firmware via the USB interface. This function assumes that the USB interface has already been configured and the device is being clocked by the PLL. By using the *pucUSBBootROMInfo*, the vendor ID, proudct ID, bus- versus self-powered, maximum power, device version, and USB strings can be customized.

Returns:

Never returns.

24.3.1.2 ROM_USBDevAddrGet

Returns the current device address in device mode.

Prototype:

```
unsigned long
ROM_USBDevAddrGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevAddrGet` is a function pointer located at `ROM_USBTABLE[1]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will return the current device address. This address was set by a call to ROM_USBDevAddrSet().

Note:
This function should only be called in device mode.

Returns:
The current device address.

24.3.1.3 ROM_USBDevAddrSet

Sets the address in device mode.

Prototype:
```
void
ROM_USBDevAddrSet(unsigned long ulBase,
                  unsigned long ulAddress)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevAddrSet` is a function pointer located at `ROM_USBTABLE[2]`.

Parameters:
ulBase specifies the USB module base address.
ulAddress is the address to use for a device.

Description:
This function will set the device address on the USB bus. This address was likely received via a SET ADDRESS command from the host controller.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.4 ROM_USBDevConnect

Connects the USB controller to the bus in device mode.

Prototype:
```
void
ROM_USBDevConnect(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevConnect` is a function pointer located at `ROM_USBTABLE[3]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will cause the soft connect feature of the USB controller to be enabled. Call ROM_USBDevDisconnect() to remove the USB device from the bus.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.5 ROM_USBDevDisconnect

Removes the USB controller from the bus in device mode.

Prototype:
```
void
ROM_USBDevDisconnect(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevDisconnect` is a function pointer located at `ROM_USBTABLE[4]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will cause the soft connect feature of the USB controller to remove the device from the USB bus. A call to ROM_USBDevConnect() is needed to reconnect to the bus.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.6 ROM_USBDevEndpointConfigGet

Gets the current configuration for an endpoint.

Prototype:
```
void
ROM_USBDevEndpointConfigGet(unsigned long ulBase,
                            unsigned long ulEndpoint,
                            unsigned long *pulMaxPacketSize,
                            unsigned long *pulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevEndpointConfigGet` is a function pointer located at `ROM_USBTABLE[41]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pulMaxPacketSize is a pointer which is written with the maximum packet size for this endpoint.

pulFlags is a pointer which is written with the current endpoint settings. On entry to the function, this pointer must contain either **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** to indicate whether the IN or OUT endpoint is to be queried.

Description:

This function will return the basic configuration for an endpoint in device mode. The values returned in ∗*pulMaxPacketSize* and ∗*pulFlags* are equivalent to the *ulMaxPacketSize* and *ulFlags* previously passed to ROM_USBDevEndpointConfigSet() for this endpoint.

Note:

This function should only be called in device mode.

Returns:

None.

24.3.1.7 ROM_USBDevEndpointConfigSet

Sets the configuration for an endpoint.

Prototype:

```
void
ROM_USBDevEndpointConfigSet(unsigned long ulBase,
                            unsigned long ulEndpoint,
                            unsigned long ulMaxPacketSize,
                            unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevEndpointConfigSet` is a function pointer located at `ROM_USBTABLE[5]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulMaxPacketSize is the maximum packet size for this endpoint.

ulFlags are used to configure other endpoint settings.

Description:

This function will set the basic configuration for an endpoint in device mode. Endpoint zero does not have a dynamic configuration, so this function should not be called for endpoint zero. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest.

The **USB_EP_MODE_** flags define what the type is for the given endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The **USB_EP_DMA_MODE_** flags determines the type of DMA access to the endpoint data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" section for more information on DMA configuration.

When configuring an IN endpoint, the **USB_EP_AUTO_SET** bit can be specified to cause the automatic transmission of data on the USB bus as soon as *ulMaxPacketSize* bytes of data are written into the FIFO for this endpoint. This is commonly used with DMA as no interaction is required to start the transmission of data.

When configuring an OUT endpoint, the **USB_EP_AUTO_REQUEST** bit is specified to trigger the request for more data once the FIFO has been drained enough to receive *ulMaxPacketSize* more bytes of data. Also for OUT endpoints, the **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this is not used, this flag must be manually cleared via a call to ROM_USBDevEndpointStatusClear(). Both of these settings can be used to remove the need for extra calls when using the controller in DMA mode.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.8 ROM_USBDevEndpointDataAck

Acknowledge that data was read from the given endpoint's FIFO in device mode.

Prototype:
```
void
ROM_USBDevEndpointDataAck(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          tBoolean bIsLastPacket)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevEndpointDataAck` is a function pointer located at `ROM_USBTABLE[6]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.
bIsLastPacket indicates if this is the last packet.

Description:
This function acknowledges that the data was read from the endpoint's FIFO. The *bIsLastPacket* parameter is set to a **true** value if this is the last in a series of data packets on endpoint

zero. The *bIsLastPacket* parameter is not used for endpoints other than endpoint zero. This call can be used if processing is required between reading the data and acknowledging that the data has been read.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.9 ROM_USBDevEndpointStall

Stalls the specified endpoint in device mode.

Prototype:
```
void
ROM_USBDevEndpointStall(unsigned long ulBase,
                        unsigned long ulEndpoint,
                        unsigned long ulFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevEndpointStall` is a function pointer located at `ROM_USBTABLE[7]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint specifies the endpoint to stall.
ulFlags specifies whether to stall the IN or OUT endpoint.

Description:
This function will cause to endpoint number passed in to go into a stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN** then the stall is issued on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT** then the stall is issued on the OUT portion of this endpoint.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.1.10 ROM_USBDevEndpointStallClear

Clears the stall condition on the specified endpoint in device mode.

Prototype:
```
void
ROM_USBDevEndpointStallClear(unsigned long ulBase,
                             unsigned long ulEndpoint,
                             unsigned long ulFlags)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_USBTABLE is an array of pointers located at ROM_APITABLE[16].
ROM_USBDevEndpointStallClear is a function pointer located at ROM_USBTABLE[8].

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies which endpoint to remove the stall condition.

ulFlags specifies whether to remove the stall condition from the IN or the OUT portion of this endpoint.

Description:

This function will cause the endpoint number passed in to exit the stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN** then the stall is cleared on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT** then the stall is cleared on the OUT portion of this endpoint.

Note:

This function should only be called in device mode.

Returns:

None.

24.3.1.11 ROM_USBDevEndpointStatusClear

Clears the status bits in this endpoint in device mode.

Prototype:

```
void
ROM_USBDevEndpointStatusClear(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              unsigned long ulFlags)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_USBTABLE is an array of pointers located at ROM_APITABLE[16].
ROM_USBDevEndpointStatusClear is a function pointer located at ROM_USBTABLE[9].

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags are the status bits that should be cleared.

Description:

This function will clear the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the ROM_USBEndpointStatus() call.

Note:

This function should only be called in device mode.

Returns:

None.

24.3.1.12 ROM_USBDevMode

Change the mode of the USB controller to device.

Prototype:

```
void
ROM_USBDevMode(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBDevMode` is a function pointer located at `ROM_USBTABLE[55]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function changes the mode of the USB controller to device mode.

Returns:

None.

24.3.1.13 ROM_USBEndpointDataAvail

Determine the number of bytes of data available in a given endpoint's FIFO.

Prototype:

```
unsigned long
ROM_USBEndpointDataAvail(unsigned long ulBase,
                         unsigned long ulEndpoint)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDataAvail` is a function pointer located at `ROM_USBTABLE[44]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

Description:

This function will return the number of bytes of data currently available in the FIFO for the given receive (OUT) endpoint. It may be used prior to calling ROM_USBEndpointDataGet() to determine the size of buffer required to hold the newly-received packet.

Returns:

This call will return the number of bytes available in a given endpoint FIFO.

24.3.1.14 ROM_USBEndpointDataGet

Retrieves data from the given endpoint's FIFO.

Prototype:

```
long
ROM_USBEndpointDataGet(unsigned long ulBase,
                       unsigned long ulEndpoint,
                       unsigned char *pucData,
                       unsigned long *pulSize)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDataGet` is a function pointer located at `ROM_USBTABLE[10]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pucData is a pointer to the data area used to return the data from the FIFO.

pulSize is initially the size of the buffer passed into this call via the *pucData* parameter. It is set to the amount of data returned in the buffer.

Description:

This function will return the data from the FIFO for the given endpoint. The *pulSize* parameter should indicate the size of the buffer passed in the *pulData* parameter. The data in the *pulSize* parameter is changed to match the amount of data returned in the *pucData* parameter. If a zero byte packet was received this call will not return a error but will instead just return a zero in the *pulSize* parameter. The only error case occurs when there is no data packet available.

Returns:

This call will return 0, or -1 if no packet was received.

24.3.1.15 ROM_USBEndpointDataPut

Puts data into the given endpoint's FIFO.

Prototype:

```
long
ROM_USBEndpointDataPut(unsigned long ulBase,
                       unsigned long ulEndpoint,
                       unsigned char *pucData,
                       unsigned long ulSize)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDataPut` is a function pointer located at `ROM_USBTABLE[11]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pucData is a pointer to the data area used as the source for the data to put into the FIFO.

ulSize is the amount of data to put into the FIFO.

Description:

This function will put the data from the *pucData* parameter into the FIFO for this endpoint. If a packet is already pending for transmission then this call will not put any of the data into the FIFO and will return -1. Care should be taken to not write more data than can fit into the FIFO allocated by the call to ROM_USBFIFOConfigSet().

Returns:

This call will return 0 on success, or -1 to indicate that the FIFO is in use and cannot be written.

24.3.1.16 ROM_USBEndpointDataSend

Starts the transfer of data from an endpoint's FIFO.

Prototype:

```
long
ROM_USBEndpointDataSend(unsigned long ulBase,
                        unsigned long ulEndpoint,
                        unsigned long ulTransType)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDataSend` is a function pointer located at `ROM_USBTABLE[12]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulTransType is set to indicate what type of data is being sent.

Description:

This function will start the transfer of data from the FIFO for a given endpoint. This is necessary if the **USB_EP_AUTO_SET** bit was not enabled for the endpoint. Setting the *ulTransType* parameter will allow the appropriate signaling on the USB bus for the type of transaction being requested. The *ulTransType* parameter should be one of the following:

- USB_TRANS_OUT for OUT transaction on any endpoint in host mode.
- USB_TRANS_IN for IN transaction on any endpoint in device mode.
- USB_TRANS_IN_LAST for the last IN transactions on endpoint zero in a sequence of IN transactions.
- USB_TRANS_SETUP for setup transactions on endpoint zero.
- USB_TRANS_STATUS for status results on endpoint zero.

Returns:

This call will return 0 on success, or -1 if a transmission is already in progress.

24.3.1.17 ROM_USBEndpointDataToggleClear

Sets the Data toggle on an endpoint to zero.

Prototype:

```
void
ROM_USBEndpointDataToggleClear(unsigned long ulBase,
                               unsigned long ulEndpoint,
                               unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDataToggleClear` is a function pointer located at `ROM_USBTABLE[13]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies the endpoint to reset the data toggle.

ulFlags specifies whether to access the IN or OUT endpoint.

Description:

This function will cause the controller to clear the data toggle for an endpoint. This call is not valid for endpoint zero and can be made with host or device controllers.

The *ulFlags* parameter should be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

24.3.1.18 ROM_USBEndpointDMAChannel

Sets the DMA channel to use for a given endpoint.

Prototype:

```
void
ROM_USBEndpointDMAChannel(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          unsigned long ulChannel)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDMAChannel` is a function pointer located at `ROM_USBTABLE[47]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies which endpoint's FIFO address to return.

ulChannel specifies which DMA channel to use for which endpoint.

Description:

This function is used to configure which DMA channel to use with a given endpoint. Receive DMA channels can only be used with receive endpoints and transmit DMA channels can only

be used with transmit endpoints. This allows the 3 receive and 3 transmit DMA channels to be mapped to any endpoint other than 0. The values that should be passed into the *ulChannel* value are the UDMA_CHANNEL_USBEP$*$ values defined in udma.h.

Note:
This function only has an effect on microcontrollers that have the ability to change the DMA channel for an endpoint. Calling this function on other devices will have no effect.

Returns:
None.

24.3.1.19 ROM_USBEndpointDMADisable

Disable DMA on a given endpoint.

Prototype:
```
void
ROM_USBEndpointDMADisable(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          unsigned long ulFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDMADisable` is a function pointer located at `ROM_USBTABLE[43]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.
ulFlags specifies which direction to disable.

Description:
This function will disable DMA on a given end point to allow non-DMA USB transactions to generate interrupts normally. The ulFlags should be **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** all other bits are ignored.

Returns:
None.

24.3.1.20 ROM_USBEndpointDMAEnable

Enable DMA on a given endpoint.

Prototype:
```
void
ROM_USBEndpointDMAEnable(unsigned long ulBase,
                         unsigned long ulEndpoint,
                         unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointDMAEnable` is a function pointer located at `ROM_USBTABLE[42]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags specifies which direction and what mode to use when enabling DMA.

Description:

This function will enable DMA on a given endpoint and set the mode according to the values in the *ulFlags* parameter. The *ulFlags* parameter should have **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** set.

Returns:

None.

24.3.1.21 ROM_USBEndpointStatus

Returns the current status of an endpoint.

Prototype:

```
unsigned long
ROM_USBEndpointStatus(unsigned long ulBase,
                      unsigned long ulEndpoint)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBEndpointStatus` is a function pointer located at `ROM_USBTABLE[14]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

Description:

This function will return the status of a given endpoint. If any of these status bits need to be cleared, then these these values must be cleared by calling the ROM_USBDevEndpointStatusClear() or ROM_USBHostEndpointStatusClear() functions.

The following are the status flags for host mode:

- **USB_HOST_IN_PID_ERROR** - PID error on the given endpoint.
- **USB_HOST_IN_NOT_COMP** - The device failed to respond to an IN request.
- **USB_HOST_IN_STALL** - A stall was received on an IN endpoint.
- **USB_HOST_IN_DATA_ERROR** - There was a CRC or bit-stuff error on an IN endpoint in Isochronous mode.
- **USB_HOST_IN_NAK_TO** - NAKs received on this IN endpoint for more than the specified timeout period.
- **USB_HOST_IN_ERROR** - Failed to communicate with a device using this IN endpoint.
- **USB_HOST_IN_FIFO_FULL** - This IN endpoint's FIFO is full.

- **USB_HOST_IN_PKTRDY** - Data packet ready on this IN endpoint.
- **USB_HOST_OUT_NAK_TO** - NAKs received on this OUT endpoint for more than the specified timeout period.
- **USB_HOST_OUT_NOT_COMP** - The device failed to respond to an OUT request.
- **USB_HOST_OUT_STALL** - A stall was received on this OUT endpoint.
- **USB_HOST_OUT_ERROR** - Failed to communicate with a device using this OUT endpoint.
- **USB_HOST_OUT_FIFO_NE** - This endpoint's OUT FIFO is not empty.
- **USB_HOST_OUT_PKTPEND** - The data transfer on this OUT endpoint has not completed.
- **USB_HOST_EP0_NAK_TO** - NAKs received on endpoint zero for more than the specified timeout period.
- **USB_HOST_EP0_ERROR** - The device failed to respond to a request on endpoint zero.
- **USB_HOST_EP0_IN_STALL** - A stall was received on endpoint zero for an IN transaction.
- **USB_HOST_EP0_IN_PKTRDY** - Data packet ready on endpoint zero for an IN transaction.

The following are the status flags for device mode:

- **USB_DEV_OUT_SENT_STALL** - A stall was sent on this OUT endpoint.
- **USB_DEV_OUT_DATA_ERROR** - There was a CRC or bit-stuff error on an OUT endpoint.
- **USB_DEV_OUT_OVERRUN** - An OUT packet was not loaded due to a full FIFO.
- **USB_DEV_OUT_FIFO_FULL** - The OUT endpoint's FIFO is full.
- **USB_DEV_OUT_PKTRDY** - There is a data packet ready in the OUT endpoint's FIFO.
- **USB_DEV_IN_NOT_COMP** - A larger packet was split up, more data to come.
- **USB_DEV_IN_SENT_STALL** - A stall was sent on this IN endpoint.
- **USB_DEV_IN_UNDERRUN** - Data was requested on the IN endpoint and no data was ready.
- **USB_DEV_IN_FIFO_NE** - The IN endpoint's FIFO is not empty.
- **USB_DEV_IN_PKTPEND** - The data transfer on this IN endpoint has not completed.
- **USB_DEV_EP0_SETUP_END** - A control transaction ended before Data End condition was sent.
- **USB_DEV_EP0_SENT_STALL** - A stall was sent on endpoint zero.
- **USB_DEV_EP0_IN_PKTPEND** - The data transfer on endpoint zero has not completed.
- **USB_DEV_EP0_OUT_PKTRDY** - There is a data packet ready in endpoint zero's OUT FIFO.

Returns:

The current status flags for the endpoint depending on mode.

24.3.1.22 ROM_USBFIFOAddrGet

Returns the absolute FIFO address for a given endpoint.

Prototype:

```
unsigned long
ROM_USBFIFOAddrGet(unsigned long ulBase,
                   unsigned long ulEndpoint)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_USBTABLE is an array of pointers located at ROM_APITABLE[16].
ROM_USBFIFOAddrGet is a function pointer located at ROM_USBTABLE[15].

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies which endpoint's FIFO address to return.

Description:

This function returns the actual physical address of the FIFO. This is needed when the USB is going to be used with the uDMA controller and the source or destination address needs to be set to the physical FIFO address for a given endpoint.

Returns:

None.

24.3.1.23 ROM_USBFIFOConfigGet

Returns the FIFO configuration for an endpoint.

Prototype:

```
void
ROM_USBFIFOConfigGet(unsigned long ulBase,
                     unsigned long ulEndpoint,
                     unsigned long *pulFIFOAddress,
                     unsigned long *pulFIFOSize,
                     unsigned long ulFlags)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_USBTABLE is an array of pointers located at ROM_APITABLE[16].
ROM_USBFIFOConfigGet is a function pointer located at ROM_USBTABLE[16].

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pulFIFOAddress is the starting address for the FIFO.

pulFIFOSize is the size of the FIFO in bytes.

ulFlags specifies what information to retrieve from the FIFO configuration.

Description:

This function will return the starting address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO so this function should not be called for endpoint zero. The *ulFlags* parameter specifies whether the endpoint's OUT or IN FIFO should be read. If in host mode, the *ulFlags* parameter should be **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode the *ulFlags* parameter should be either **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

24.3.1.24 ROM_USBFIFOConfigSet

Sets the FIFO configuration for an endpoint.

Prototype:

```
void
ROM_USBFIFOConfigSet(unsigned long ulBase,
                     unsigned long ulEndpoint,
                     unsigned long ulFIFOAddress,
                     unsigned long ulFIFOSize,
                     unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBFIFOConfigSet` is a function pointer located at `ROM_USBTABLE[17]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFIFOAddress is the starting address for the FIFO.

ulFIFOSize is the size of the FIFO in bytes.

ulFlags specifies what information to set in the FIFO configuration.

Description:

This function will set the starting FIFO RAM address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO so this function should not be called for endpoint zero. The *ulFIFOSize* parameter should be one of the values in the **USB_FIFO_SZ_** values. If the endpoint is going to use double buffering it should use the values with the **_DB** at the end of the value. For example, use **USB_FIFO_SZ_16_DB** to configure an endpoint to have a 16 byte double buffered FIFO. If a double buffered FIFO is used, then the actual size of the FIFO is twice the size indicated by the *ulFIFOSize* parameter. This means that the **USB_FIFO_SZ_16_DB** value will use 32 bytes of the USB controller's FIFO memory.

The *ulFIFOAddress* value should be a multiple of 8 bytes and directly indicates the starting address in the USB controller's FIFO RAM. For example, a value of 64 indicates that the FIFO should start 64 bytes into the USB controller's FIFO memory. The *ulFlags* value specifies whether the endpoint's OUT or IN FIFO should be configured. If in host mode, use **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode use **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

24.3.1.25 ROM_USBFIFOFlush

Forces a flush of an endpoint's FIFO.

Prototype:

```
void
ROM_USBFIFOFlush(unsigned long ulBase,
```

```
                    unsigned long ulEndpoint,
                    unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBFIFOFlush` is a function pointer located at `ROM_USBTABLE[18]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags specifies if the IN or OUT endpoint should be accessed.

Description:

This function will force the controller to flush out the data in the FIFO. The function can be called with either host or device controllers and requires the *ulFlags* parameter be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

24.3.1.26 ROM_USBFrameNumberGet

Get the current frame number.

Prototype:

```
unsigned long
ROM_USBFrameNumberGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBFrameNumberGet` is a function pointer located at `ROM_USBTABLE[19]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function returns the last frame number received.

Returns:

The last frame number received.

24.3.1.27 ROM_USBHostAddrGet

Gets the current functional device address for an endpoint.

Prototype:

```
unsigned long
ROM_USBHostAddrGet(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostAddrGet` is a function pointer located at `ROM_USBTABLE[20]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function returns the current functional address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the IN or OUT endpoint's device address is returned.

Note:

This function should only be called in host mode.

Returns:

Returns the current function address being used by an endpoint.

24.3.1.28 ROM_USBHostAddrSet

Sets the functional address for the device that is connected to an endpoint in host mode.

Prototype:

```
void
ROM_USBHostAddrSet(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned long ulAddr,
                   unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostAddrSet` is a function pointer located at `ROM_USBTABLE[21]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulAddr is the functional address for the controller to use for this endpoint.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function will set the functional address for a device that is using this endpoint for communication. This *ulAddr* parameter is the address of the target device that this endpoint is used to communicate with. The *ulFlags* parameter indicates if the IN or OUT endpoint should be set.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.1.29 ROM_USBHostEndpointConfig

Sets the base configuration for a host endpoint.

Prototype:

```
void
ROM_USBHostEndpointConfig(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          unsigned long ulMaxPayload,
                          unsigned long ulNAKPollInterval,
                          unsigned long ulTargetEndpoint,
                          unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostEndpointConfig` is a function pointer located at `ROM_USBTABLE[22]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulMaxPayload is the maximum payload for this endpoint.

ulNAKPollInterval is the either the NAK timeout limit or the polling interval depending on the type of endpoint.

ulTargetEndpoint is the endpoint that the host endpoint is targeting.

ulFlags are used to configure other endpoint settings.

Description:

This function will set the basic configuration for the transmit or receive portion of an endpoint in host mode. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest. The *ulFlags* parameter determines whether this is an IN endpoint (USB_EP_HOST_IN or USB_EP_DEV_IN) or an OUT endpoint (USB_EP_HOST_OUT or USB_EP_DEV_OUT), whether this is a Full speed endpoint (USB_EP_SPEED_FULL) or a Low speed endpoint (USB_EP_SPEED_LOW).

The **USB_EP_MODE_** flags control the type of the endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The *ulNAKPollInterval* parameter has different meanings based on the **USB_EP_MODE** value and whether or not this call is being made for endpoint zero or another endpoint. For endpoint zero or any Bulk endpoints, this value always indicates the number of frames to allow a device to NAK before considering it a timeout. If this endpoint is an isochronous or interrupt endpoint, this value is the polling interval for this endpoint.

For interrupt endpoints the polling interval is simply the number of frames between polling an interrupt endpoint. For isochronous endpoints this value represents a polling interval of 2 ^ (*ulNAKPollInterval* - 1) frames. When used as a NAK timeout, the *ulNAKPollInterval* value specifies 2 ^ (*ulNAKPollInterval* - 1) frames before issuing a time out. There are two special time out values that can be specified when setting the *ulNAKPollInterval* value. The first is

MAX_NAK_LIMIT which is the maximum value that can be passed in this variable. The other is **DISABLE_NAK_LIMIT** which indicates that there should be no limit on the number of NAKs.

The **USB_EP_DMA_MODE_** flags enables the type of DMA used to access the endpoint's data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" section for more information on DMA configuration.

When configuring the OUT portion of an endpoint, the **USB_EP_AUTO_SET** bit is specified to cause the transmission of data on the USB bus to start as soon as the number of bytes specified by *ulMaxPayload* have been written into the OUT FIFO for this endpoint.

When configuring the IN portion of an endpoint, the **USB_EP_AUTO_REQUEST** bit can be specified to trigger the request for more data once the FIFO has been drained enough to fit *ulMaxPayload* bytes. The **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this is not used, this flag must be manually cleared via a call to ROM_USBDevEndpointStatusClear() or ROM_USBHostEndpointStatusClear().

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.30 ROM_USBHostEndpointDataAck

Acknowledge that data was read from the given endpoint's FIFO in host mode.

Prototype:
```
void
ROM_USBHostEndpointDataAck(unsigned long ulBase,
                           unsigned long ulEndpoint)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostEndpointDataAck` is a function pointer located at `ROM_USBTABLE[23]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.

Description:
This function acknowledges that the data was read from the endpoint's FIFO. This call is used if processing is required between reading the data and acknowledging that the data has been read.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.31 ROM_USBHostEndpointDataToggle

Sets the value data toggle on an endpoint in host mode.

Prototype:

```
void
ROM_USBHostEndpointDataToggle(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              tBoolean bDataToggle,
                              unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostEndpointDataToggle` is a function pointer located at `ROM_USBTABLE[24]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies the endpoint to reset the data toggle.

bDataToggle specifies whether to set the state to DATA0 or DATA1.

ulFlags specifies whether to set the IN or OUT endpoint.

Description:

This function is used to force the state of the data toggle in host mode. If the value passed in the *bDataToggle* parameter is **false**, then the data toggle is set to the DATA0 state, and if it is **true** it is set to the DATA1 state. The *ulFlags* parameter can be **USB_EP_HOST_IN** or **USB_EP_HOST_OUT** to access the desired portion of this endpoint. The *ulFlags* parameter is ignored for endpoint zero.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.1.32 ROM_USBHostEndpointStatusClear

Clears the status bits in this endpoint in host mode.

Prototype:

```
void
ROM_USBHostEndpointStatusClear(unsigned long ulBase,
                               unsigned long ulEndpoint,
                               unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostEndpointStatusClear` is a function pointer located at `ROM_USBTABLE[25]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.
ulFlags are the status bits that should be cleared.

Description:
This function will clear the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the ROM_USBEndpointStatus() call.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.33 ROM_USBHostHubAddrGet

Get the current device hub address for this endpoint.

Prototype:
```
unsigned long
ROM_USBHostHubAddrGet(unsigned long ulBase,
                      unsigned long ulEndpoint,
                      unsigned long ulFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostHubAddrGet` is a function pointer located at `ROM_USBTABLE[26]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.
ulFlags determines if this is an IN or an OUT endpoint.

Description:
This function will return the current hub address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the device address for the IN or OUT endpoint is returned.

Note:
This function should only be called in host mode.

Returns:
This function returns the current hub address being used by an endpoint.

24.3.1.34 ROM_USBHostHubAddrSet

Set the hub address for the device that is connected to an endpoint.

Prototype:
```
void
ROM_USBHostHubAddrSet(unsigned long ulBase,
```

```
                                unsigned long ulEndpoint,
                                unsigned long ulAddr,
                                unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostHubAddrSet` is a function pointer located at `ROM_USBTABLE[27]`.

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulAddr is the hub address for the device using this endpoint.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function will set the hub address for a device that is using this endpoint for communication. The *ulFlags* parameter determines if the device address for the IN or the OUT endpoint is set by this call.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.1.35 ROM_USBHostMode

Change the mode of the USB controller to host.

Prototype:

```
void
ROM_USBHostMode(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostMode` is a function pointer located at `ROM_USBTABLE[54]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function changes the mode of the USB controller to host mode.

Returns:

None.

24.3.1.36 ROM_USBHostPwrConfig

Sets the configuration for USB power fault.

Prototype:

```
void
ROM_USBHostPwrConfig(unsigned long ulBase,
                     unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostPwrConfig` is a function pointer located at `ROM_USBTABLE[30]`.

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies the configuration of the power fault.

Description:

This function controls how the USB controller uses its external power control pins (USBnPFTL and USBnEPEN). The flags specify the power fault level sensitivity, the power fault action, and the power enable level and source.

One of the following can be selected as the power fault level sensitivity:

- **USB_HOST_PWRFLT_LOW** - An external power fault is indicated by the pin being driven low.
- **USB_HOST_PWRFLT_HIGH** - An external power fault is indicated by the pin being driven high.

One of the following can be selected as the power fault action:

- **USB_HOST_PWRFLT_EP_NONE** - No automatic action when power fault detected.
- **USB_HOST_PWRFLT_EP_TRI** - Automatically Tri-state the USBnEPEN pin on a power fault.
- **USB_HOST_PWRFLT_EP_LOW** - Automatically drive USBnEPEN pin low on a power fault.
- **USB_HOST_PWRFLT_EP_HIGH** - Automatically drive USBnEPEN pin high on a power fault.

One of the following can be selected as the power enable level and source:

- **USB_HOST_PWREN_MAN_LOW** - USBEPEN is driven low by the USB controller when ROM_USBHostPwrEnable() is called.
- **USB_HOST_PWREN_MAN_HIGH** - USBEPEN is driven high by the USB controller when ROM_USBHostPwrEnable() is called.
- **USB_HOST_PWREN_AUTOLOW** - USBEPEN is driven low by the USB controller automatically if USBOTGSessionRequest() has enabled a session.
- **USB_HOST_PWREN_AUTOHIGH** - USBEPEN is driven high by the USB controller automatically if USBOTGSessionRequest() has enabled a session.

The **USB_HOST_PWREN_FILTER** flag can be added to enable the VBUS glitch filter, which ignores small, short drops in VBUS level caused by high power consumption. This is mainly used to avoid causing VBUS errors caused by devices with high in-rush current.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.37 ROM_USBHostPwrDisable

Disables the external power pin.

Prototype:
```
void
ROM_USBHostPwrDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostPwrDisable` is a function pointer located at `ROM_USBTABLE[28]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function disables the USBEPEN signal to disable an external power supply in host mode operation.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.38 ROM_USBHostPwrEnable

Enables the external power pin.

Prototype:
```
void
ROM_USBHostPwrEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostPwrEnable` is a function pointer located at `ROM_USBTABLE[29]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function enables the USBEPEN signal to enable an external power supply in host mode operation.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.39 ROM_USBHostPwrFaultDisable

Disables power fault detection.

Prototype:
```
void
ROM_USBHostPwrFaultDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostPwrFaultDisable` is a function pointer located at `ROM_USBTABLE[31]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function disables power fault detection in the USB controller.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.40 ROM_USBHostPwrFaultEnable

Enables power fault detection.

Prototype:
```
void
ROM_USBHostPwrFaultEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostPwrFaultEnable` is a function pointer located at `ROM_USBTABLE[32]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function enables power fault detection in the USB controller. If the USBPFLT pin is not in use this function should not be used.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.41 ROM_USBHostRequestIN

Schedules a request for an IN transaction on an endpoint in host mode.

Prototype:
```
void
ROM_USBHostRequestIN(unsigned long ulBase,
                     unsigned long ulEndpoint)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostRequestIN` is a function pointer located at `ROM_USBTABLE[33]`.

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.

Description:
This function will schedule a request for an IN transaction. When the USB device being communicated with responds the data, the data can be retrieved by calling ROM_USBEndpointDataGet() or via a DMA transfer.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.42 ROM_USBHostRequestStatus

Issues a request for a status IN transaction on endpoint zero.

Prototype:
```
void
ROM_USBHostRequestStatus(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostRequestStatus` is a function pointer located at `ROM_USBTABLE[34]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function is used to cause a request for an status IN transaction from a device on endpoint zero. This function can only be used with endpoint zero as that is the only control endpoint that supports this ability. This is used to complete the last phase of a control transaction to a device and an interrupt is signaled when the status packet has been received.

Returns:
None.

24.3.1.43 ROM_USBHostReset

Handles the USB bus reset condition.

Prototype:
```
void
ROM_USBHostReset(unsigned long ulBase,
                 tBoolean bStart)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostReset` is a function pointer located at `ROM_USBTABLE[35]`.

Parameters:
ulBase specifies the USB module base address.
bStart specifies whether to start or stop signaling reset on the USB bus.

Description:
When this function is called with the *bStart* parameter set to **true**, this function will cause the start of a reset condition on the USB bus. The caller should then delay at least 20ms before calling this function again with the *bStart* parameter set to **false**.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.44 ROM_USBHostResume

Handles the USB bus resume condition.

Prototype:
```
void
ROM_USBHostResume(unsigned long ulBase,
                  tBoolean bStart)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostResume` is a function pointer located at `ROM_USBTABLE[36]`.

Parameters:

ulBase specifies the USB module base address.

bStart specifies if the USB controller is entering or leaving the resume signaling state.

Description:

When in device mode this function will bring the USB controller out of the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The device application should then delay at least 10ms but not more than 15ms before calling this function with the *bStart* parameter set to **false**.

When in host mode this function will signal devices to leave the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The host application should then delay at least 20ms before calling this function with the *bStart* parameter set to **false**. This will cause the controller to complete the resume signaling on the USB bus.

Returns:

None.

24.3.1.45 ROM_USBHostSpeedGet

Returns the current speed of the USB device connected.

Prototype:

```
unsigned long
ROM_USBHostSpeedGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostSpeedGet` is a function pointer located at `ROM_USBTABLE[37]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function will return the current speed of the USB bus.

Note:

This function should only be called in host mode.

Returns:

Returns either **USB_LOW_SPEED**, **USB_FULL_SPEED**, or **USB_UNDEF_SPEED**.

24.3.1.46 ROM_USBHostSuspend

Puts the USB bus in a suspended state.

Prototype:

```
void
ROM_USBHostSuspend(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBHostSuspend` is a function pointer located at `ROM_USBTABLE[38]`.

Parameters:
ulBase specifies the USB module base address.

Description:
When used in host mode, this function will put the USB bus in the suspended state.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.1.47 ROM_USBIntDisable

Disables the sources for USB interrupts.

Prototype:
```
void
ROM_USBIntDisable(unsigned long ulBase,
                  unsigned long ulFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntDisable` is a function pointer located at `ROM_USBTABLE[39]`.

Parameters:
ulBase specifies the USB module base address.
ulFlags specifies which interrupts to disable.

Description:
This function will disable the USB controller from generating the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_INT_STATUS**. If **USB_INT_ALL** is specified then all interrupts are disabled.

Note:
WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so ROM_USBIntDisableControl() or ROM_USBIntDisableEndpoint() should be used instead.

Returns:
None.

24.3.1.48 ROM_USBIntDisableControl

Disables control interrupts on a given USB controller.

Prototype:

```
void
ROM_USBIntDisableControl(unsigned long ulBase,
                         unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntDisableControl` is a function pointer located at `ROM_USBTABLE[48]`.

Parameters:

ulBase specifies the USB module base address.
ulFlags specifies which control interrupts to disable.

Description:

This function will disable the control interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which control interrupts to disable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTCTRL_***∗* and not any other **USB_INT** flags.

Returns:

None.

24.3.1.49 ROM_USBIntDisableEndpoint

Disables endpoint interrupts on a given USB controller.

Prototype:

```
void
ROM_USBIntDisableEndpoint(unsigned long ulBase,
                          unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntDisableEndpoint` is a function pointer located at `ROM_USBTABLE[51]`.

Parameters:

ulBase specifies the USB module base address.
ulFlags specifies which endpoint interrupts to disable.

Description:

This function will disable endpoint interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which endpoint interrupts to disable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTEP_***∗* and not any other **USB_INT** flags.

Returns:

None.

24.3.1.50 ROM_USBIntEnable

Enables the sources for USB interrupts.

Prototype:

```
void
ROM_USBIntEnable(unsigned long ulBase,
                 unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntEnable` is a function pointer located at `ROM_USBTABLE[40]`.

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies which interrupts to enable.

Description:

This function will enable the USB controller's ability to generate the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_STATUS**. If **USB_INT_ALL** is specified then all interrupts are enabled.

Note:

A call must be made to enable the interrupt in the main interrupt controller to receive interrupts. The USBIntRegister() API performs this controller level interrupt enable. However if static interrupt handlers are used then then a call to ROM_IntEnable() must be made in order to allow any USB interrupts to occur.

WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so ROM_USBIntEnableControl() or ROM_USBIntEnableEndpoint() should be used instead.

Returns:

None.

24.3.1.51 ROM_USBIntEnableControl

Enables control interrupts on a given USB controller.

Prototype:

```
void
ROM_USBIntEnableControl(unsigned long ulBase,
                        unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntEnableControl` is a function pointer located at `ROM_USBTABLE[49]`.

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies which control interrupts to enable.

Description:

This function will enable the control interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which control interrupts to enable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTCTRL_***∗* and not any other **USB_INT** flags.

Returns:

None.

24.3.1.52 ROM_USBIntEnableEndpoint

Enables endpoint interrupts on a given USB controller.

Prototype:

```
void
ROM_USBIntEnableEndpoint(unsigned long ulBase,
                         unsigned long ulFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntEnableEndpoint` is a function pointer located at `ROM_USBTABLE[52]`.

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies which endpoint interrupts to enable.

Description:

This function will enable endpoint interrupts for the USB controller specified by the *ulBase* parameter. The *ulFlags* parameter specifies which endpoint interrupts to enable. The flags passed in the *ulFlags* parameters should be the definitions that start with **USB_INTEP_***∗* and not any other **USB_INT** flags.

Returns:

None.

24.3.1.53 ROM_USBIntStatus

Returns the status of the USB interrupts.

Prototype:

```
unsigned long
ROM_USBIntStatus(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntStatus` is a function pointer located at `ROM_USBTABLE[0]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function will read the source of the interrupt for the USB controller. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes. This call will return the current status for all of these interrupts. The bit values returned should be compared against the **USB_HOST_IN**, **USB_HOST_OUT**, **USB_HOST_EP0**, **USB_DEV_IN**, **USB_DEV_OUT**, and **USB_DEV_EP0** values.

Note:

This call will clear the source of all of the general status interrupts.

WARNING: This API cannot be used on endpoint numbers greater than endpoint 3 so ROM_USBIntStatusControl() or ROM_USBIntStatusEndpoint() should be used instead.

Returns:

Returns the status of the sources for the USB controller's interrupt.

24.3.1.54 ROM_USBIntStatusControl

Returns the control interrupt status on a given USB controller.

Prototype:

```
unsigned long
ROM_USBIntStatusControl(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntStatusControl` is a function pointer located at `ROM_USBTABLE[50]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function will read control interrupt status for a USB controller. This call will return the current status for control interrupts only, the endpoint interrupt status is retrieved by calling ROM_USBIntStatusEndpoint(). The bit values returned should be compared against the **USB_INTCTRL_**$*$ values.

The following are the meanings of all **USB_INCTRL_** flags and the modes for which they are valid. These values apply to any calls to ROM_USBIntStatusControl(), ROM_USBIntEnableControl(), and ROM_USBIntDisableControl(). Some of these flags are only valid in the following modes as indicated in the parenthesis: Host, Device, and OTG.

- **USB_INTCTRL_ALL** - A full mask of all control interrupt sources.
- **USB_INTCTRL_VBUS_ERR** - A VBUS error has occurred (Host Only).
- **USB_INTCTRL_SESSION** - Session Start Detected on A-side of cable (OTG Only).
- **USB_INTCTRL_SESSION_END** - Session End Detected (Device Only)
- **USB_INTCTRL_DISCONNECT** - Device Disconnect Detected (Host Only)
- **USB_INTCTRL_CONNECT** - Device Connect Detected (Host Only)
- **USB_INTCTRL_SOF** - Start of Frame Detected.

- **USB_INTCTRL_BABBLE** - USB controller detected a device signaling past the end of a frame. (Host Only)
- **USB_INTCTRL_RESET** - Reset signaling detected by device. (Device Only)
- **USB_INTCTRL_RESUME** - Resume signaling detected.
- **USB_INTCTRL_SUSPEND** - Suspend signaling detected by device (Device Only)
- **USB_INTCTRL_MODE_DETECT** - OTG cable mode detection has completed (OTG Only)
- **USB_INTCTRL_POWER_FAULT** - Power Fault detected. (Host Only)

Note:
This call will clear the source of all of the control status interrupts.

Returns:
Returns the status of the control interrupts for a USB controller.

24.3.1.55 ROM_USBIntStatusEndpoint

Returns the endpoint interrupt status on a given USB controller.

Prototype:
```
unsigned long
ROM_USBIntStatusEndpoint(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBIntStatusEndpoint` is a function pointer located at `ROM_USBTABLE[53]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will read endpoint interrupt status for a USB controller. This call will return the current status for endpoint interrupts only, the control interrupt status is retrieved by calling ROM_USBIntStatusControl(). The bit values returned should be compared against the **USB_INTEP_**$*$ values. These are grouped into classes for **USB_INTEP_HOST_**$*$ and **USB_INTEP_DEV_**$*$ values to handle both host and device modes with all endpoints.

Note:
This call will clear the source of all of the endpoint interrupts.

Returns:
Returns the status of the endpoint interrupts for a USB controller.

24.3.1.56 ROM_USBModeGet

Returns the current operating mode of the controller.

Prototype:
```
unsigned long
ROM_USBModeGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBModeGet` is a function pointer located at `ROM_USBTABLE[46]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function returns the current operating mode on USB controllers with OTG or Dual mode functionality.

For OTG controllers:

The function will return on of the following values on OTG controllers: **USB_OTG_MODE_ASIDE_HOST**, **USB_OTG_MODE_ASIDE_DEV**, **USB_OTG_MODE_BSIDE_HOST**, **USB_OTG_MODE_BSIDE_DEV**, **USB_OTG_MODE_NONE**.

USB_OTG_MODE_ASIDE_HOST indicates that the controller is in host mode on the A-side of the cable.

USB_OTG_MODE_ASIDE_DEV indicates that the controller is in device mode on the A-side of the cable.

USB_OTG_MODE_BSIDE_HOST indicates that the controller is in host mode on the B-side of the cable.

USB_OTG_MODE_BSIDE_DEV indicates that the controller is in device mode on the B-side of the cable. If and OTG session request is started with no cable in place this is the default mode for the controller.

USB_OTG_MODE_NONE indicates that the controller is not attempting to determine its role in the system.

For Dual Mode controllers:

The function will return on of the following values: **USB_DUAL_MODE_HOST**, **USB_DUAL_MODE_DEVICE**, or **USB_DUAL_MODE_NONE**.

USB_DUAL_MODE_HOST indicates that the controller is acting as a host.

USB_DUAL_MODE_DEVICE indicates that the controller acting as a device.

USB_DUAL_MODE_NONE indicates that the controller is not active as either a host or device.

Returns:

Returns **USB_OTG_MODE_ASIDE_HOST**, **USB_OTG_MODE_ASIDE_DEV**, **USB_OTG_MODE_BSIDE_HOST**, **USB_OTG_MODE_BSIDE_DEV**, **USB_OTG_MODE_NONE**, **USB_DUAL_MODE_HOST**, **USB_DUAL_MODE_DEVICE**, or **USB_DUAL_MODE_NONE**.

24.3.1.57 ROM_USBOTGHostRequest

This function will enable host negotiation protocol when in device mode.

Prototype:

```
void
```

```
ROM_USBOTGHostRequest(unsigned long ulBase,
                      tBoolean bStart)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBOTGHostRequest` is a function pointer located at `ROM_USBTABLE[45]`.

Parameters:

ulBase specifies the USB module base address.

bStart specifies if this call starts or ends a session.

Description:

This function is used in OTG mode when the USB controller is on the B-Side of the cable and it needs to become the host during a session. If the *bHNP* parameter is set to **true**, then this will enable the USB controller to initiate the Host Negotiation Protocol(HNP) and if it is set to **false** it will disable HNP. Enabling the HNP sequence will allow the HNP protocol to start the next time the USB controller sees a suspend condition on the bus. If the sequence is successful, the USB controller will generate a **USB_INTCTRL_CONNECT** interrupt. The USB controller will also automatically generate a reset condition on the bus.

Note:

The application code should wait at least 20ms after receiving the **USB_INTCTRL_CONNECT** interrupt before clearing the reset condition with a call to ROM_USBHostReset().

In order to leave host mode due to a successful HNP sequence the USB controller must put the bus into suspend via a call to ROM_USBHostSuspend(). This signals the A-Side of the cable to resume host operation.

Returns:

None.

24.3.1.58 ROM_USBOTGMode

Change the mode of the USB controller to OTG.

Prototype:

```
void
ROM_USBOTGMode(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBOTGMode` is a function pointer located at `ROM_USBTABLE[59]`.

Parameters:

ulBase specifies the USB module base address.

Description:

This function changes the mode of the USB controller to OTG mode. This is only valid on microcontrollers that have the OTG capabilities.

Returns:
None.

24.3.1.59 ROM_USBPHYPowerOff

Powers off the USB PHY.

Prototype:
```
void
ROM_USBPHYPowerOff(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBPHYPowerOff` is a function pointer located at `ROM_USBTABLE[56]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will power off the USB PHY, reducing the current consuption of the device. While in the powered off state, the USB controller is unable to operate.

Returns:
None.

24.3.1.60 ROM_USBPHYPowerOn

Powers on the USB PHY.

Prototype:
```
void
ROM_USBPHYPowerOn(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_USBTABLE` is an array of pointers located at `ROM_APITABLE[16]`.
`ROM_USBPHYPowerOn` is a function pointer located at `ROM_USBTABLE[57]`.

Parameters:
ulBase specifies the USB module base address.

Description:
This function will power on the USB PHY, enabling it return to normal operation. By default, the PHY is powered on, so this function only needs to be called if ROM_USBPHYPowerOff() has previously been called.

Returns:
None.

25 Watchdog Timer

Introduction 359
Functions 359

25.1 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.

25.2 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_WatchdogIntTypeSet (unsigned long ulBase, unsigned long ulType)
- 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_WatchdogStallDisable (unsigned long ulBase)
- void ROM_WatchdogStallEnable (unsigned long ulBase)

- void ROM_WatchdogUnlock (unsigned long ulBase)
- unsigned long ROM_WatchdogValueGet (unsigned long ulBase)

25.2.1 Function Documentation

25.2.1.1 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.

25.2.1.2 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:
Because there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:
None.

25.2.1.3 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.

25.2.1.4 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:

Returns the current interrupt status, where a 1 indicates that the watchdog interrupt is active, and a 0 indicates that it is not active.

25.2.1.5 ROM_WatchdogIntTypeSet

Sets the type of interrupt generated by the watchdog.

Prototype:

```
void
ROM_WatchdogIntTypeSet(unsigned long ulBase,
                       unsigned long ulType)
```

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_WatchdogIntTypeSet` is a function pointer located at `ROM_WATCHDOGTABLE[15]`.

Parameters:

ulBase is the base address of the watchdog timer module.

ulType is the type of interrupt to generate.

Description:

This function sets the type of interrupt that is generated if the watchdog timer expires. *ulType* can be either **WATCHDOG_INT_TYPE_INT** to generate a standard interrupt (the default) or **WATCHDOG_INT_TYPE_NMI** to generate a non-maskable interrupt (NMI).

When configured to generate an NMI, the watchdog interrupt must still be enabled with ROM_WatchdogIntEnable(), and it must still be cleared inside the NMI handler with ROM_WatchdogIntClear().

Returns:

None.

25.2.1.6 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.

25.2.1.7 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.

25.2.1.8 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.

25.2.1.9 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 is 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.

25.2.1.10 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.

25.2.1.11 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.

25.2.1.12 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.

25.2.1.13 ROM_WatchdogStallDisable

Disables stalling of the watchdog timer during debug events.

Prototype:

```
void
ROM_WatchdogStallDisable(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_WatchdogStallDisable` is a function pointer located at `ROM_WATCHDOGTABLE[14]`.

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

This function disables the debug mode stall of the watchdog timer. By doing so, the watchdog timer continues to count regardless of the processor debug state.

Returns:

None.

25.2.1.14 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.

25.2.1.15 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.

25.2.1.16 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		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
TI E2E Community Home Page	e2e.ti.com		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2011-2013, Texas Instruments Incorporated